Text Processing *with* Java 6

Text Processing *with* Java 6

Bob Carpenter
Mitzi Morris
Breck Baldwin

LingPipe Publishing
New York
2011

Pending: *Library of Congress Cataloging-in-Publication Data*

Carpenter, Bob.
Text Processing with Java 6 / Bob Carpenter, Mitzi Morris, Breck Baldwin
p. cm.
Includes bibliographical references and index.
ISBN X-XXX-XXXXX-X (pbk.)

1. Text Processing 2. Java (computer program language) 3. Unicode I. Carpenter, Bob II. Morris, Mitzi III. Baldwin, Breck IV. Title

QAXXX.XXXXXXXX 2011

XXX/.XX'X-xxXX XXXXXXXX

Contents

Preface

Java provides an extensive set of tools for manipulating character-based data, including byte-to-character conversion, input and output streams including sockets and compression, memory-mapping files, and high-level methods for manipulating sequences of characters like buffers and regular expressions. We also cover the use of the Apache Lucene search library.

As a preliminary, we provide a warmup chapter on using the various tools required for text editors to version control to the Ant build system for Java.

This book explains the basics of text processing in Java. It assumes you are familiar with basic coding in a language like Java, but that you may not be familiar with low-level representations of bytes and integers, the representation and manipulation of strings, how to access objects from the classpath as resources, or how to write non-greedy regular expressions based on Unicode.

The presentation here will be hands on. You should be comfortable reading short and relatively simple Java programs. Java programming idioms like loop boundaries being inclusive/exclusive and higher-level design patterns like visitors will also be presupposed. More specific aspects of Java coding relating to text processing, such as streaming I/O, character decoding, string representations, and regular expression processing will be discussed in more depth. We will also go into some detail on collections, XML/HTML parsing with SAX, and serialization patterns.

This book grew out of work on its companion volume, *Natural Language Processing with LingPipe*. We found that the background material on Java was growing to the point where it made more sense to pull it out into its own standalone package. We have included a few examples of basic text processing using LingPipe.

Other open-source software dependencies include the International Components for Unicode (ICU) library and Apache Lucene search library.

We hope you have as much fun with text processing in Java as we've had. We'd love to hear from you, so feel free to drop us a line at `info@lingpipe.com` with any kind of question or comment you might have.

Bob Carpenter
Mitzi Morris
Breck Baldwin

New York
July 6, 2012

Text Processing *with* Java 6

Chapter 1

Getting Started

This book is intended to provide a relatively concise, yet comprehensive overview of text processing in the Java programming language.

Before beginning with a "hello world" example that will verify that everything's in place for compiling and executing the code in the book, we'll take some time to explain the tools we'll be using or that you might want to use. All of these tools are open source and freely available, though under a perplexing array of licensing terms and conditions. After describing and motivating our choice of each tool, we'll provide downloading and installation instructions.

1.1 Tools of the Trade

In this section, we go over the various tools we use for program development. Not all of these tools are needed to run every example, and in many cases, we discuss several alternatives among a large set of examples.

1.1.1 Unix Shell Tools

We present examples as run from a unix-like command-line shell. One reason to use the shell is that it's easy to script operations so that they may be repeated.

The main problem with scripts is that they are not portable across operating systems or even types of command-line shells. In this book, we use the so-called "Bourne Again Shell" (bash) for Unix-style commands. Bash is the default shell in Cygwin on Windows, for Mac OS X, and for Linux, and is also available for Solaris.

If you're working in Windows (XP, Vista or 7), we recommend the Cygwin suite of Unix command-line tools for Windows.

Cygwin is released under version 2 of the GNU Public License (GPLv2), with terms outlined here:

```
http://cygwin.com/license.html
```

and also available with a commercial license.

Cygwin can be downloaded and installed through Cygwin's home page, `http://www.cygwin.com/`

The `setup.exe` program is small. When you run it, it goes out over the internet to find the packages from registered mirrors. It then lists all the packages available. You can install some or all of them. You can just run the setup program again to update or install new packages; it'll show you what version of what's already installed.

It might also be easier to list what Cygwin doesn't support. We use it mainly for running Unix-like commands in Windows, including pipes, find, grep, (bash) shell support, tar and gzip, wget, aspell, which, and so on. We also use its implementation of the Subversion and CVS version control systems. We do not install emacs or TeX through Cygwin. These packages are indexed under various categories in Cygwin, such as Web, Net, Devel, and Base.

Although it's possible to get Python, Perl, Emacs, $\TeX$ and $\LaTeX$, and so on, we typically install these packages directly rather than as part of Cygwin.

Archive and Compression Tools

In order to unpack data and library distributions, you need to be able to run the `tar` archiving tool, as well as the unpacking commands `unzip` and `gunizp`. These may be installed as part of Cygwin on Windows.

1.1.2 Version Control

If you don't live in some kind of version control environment, you should. Not only can you keep track of your own code across multiple sites and/or users, you can also keep up to date with projects with version control, such as this book, the LingPipe sandbox, and projects hosted by open source hosting services such as SourceForge or Google Code.

We are currently using Subversion (SVN) for LingPipe and this book. You can install a shell-based version of Subversion, the command-name for which is `svn`. Subversion itself requires a secure shell (SSH) client over which to run, at least for most installations. Both SVN and SSH can be installed through Cygwin for Windows users.

There are graphical interfaces for subversion. Web-SVN, which as its name implies, runs as a server and is accessible through a web browser,

WebSVN: `http://websvn.tigris.org/`

and Tortoise SVN, which integrates with the Windows Explorer,

Tortoise SVN: `http://tortoisesvn.net/`

Other popular version control systems include the older Concurrent Version System (CVS), as well as the increasingly popular Git system, which is used by the Linux developers.

The best reference for Subversion of which we are aware is the official guide by the authors, available onlie at

`http://svnbook.red-bean.com/`

1.1.3 Text Editors

In order to generate code, HTML, and reports, you will need to be able to edit text. We like to work in the emacs text editor, because of its configurability. It's as close as you'll get to an IDE in a simple text editor.

Spaces, Not Tabs

To make code portable, we highly recommend using spaces instead of tabs. Yes, it takes up a bit more space, but it's worth it. We follow Sun's coding standard for Java, so we use four spaces for a tab. This is a bit easier to read, but wastes more horizontal space.

(GNU) Emacs

We use the GNU Emacs distribution of emacs, which is available from its home page,

```
http://www.gnu.org/software/emacs/
```

It's standard on most Unix and Linux distributions; for Windows, there is a zipped binary distribution in a subdirectory of the main distribution that only needs to be unpacked in order to be run. We've had problems with the Cygwin-based installations in terms of their font integration. And we've also not been able to configure XEmacs for Unicode, which is the main reason for our preference for GNU Emacs.

We like to work with the Lucida Console font, which is distributed with Windows; it's the font used for code examples in this book. It also supports a wide range of non-Roman fonts. You can change the font by pulling down the `Options` menu and selecting the `Set Default Font...` item to pop up a dialog box. Then use the `Save Options` item in the same menu to save it. It'll show you where it saved a file called `.emacs` which you will need to edit for the next customizations.

In order to configure GNU Emacs to run UTF-8, you need to add the following text to your `.emacs` file:[1]

```
(prefer-coding-system 'utf-8)
(set-default-coding-systems 'utf-8)
(set-terminal-coding-system 'utf-8)
(set-keyboard-coding-system 'utf-8)
(setq default-buffer-file-coding-system 'utf-8)
(setq x-select-request-type
    '(UTF8_STRING COMPOUND_TEXT TEXT STRING))
(set-clipboard-coding-system 'utf-16le-dos)
```

The requisite commands to force tabs to be replaced with spaces in Java files are:

[1]The UTF-8 instructions are from the Sacrificial Rabbit blog entry `http://blog.jonnay.net/archives/820-Emacs-and-UTF-8-Encoding.html`, downloaded 4 August 2010.

```
(defun java-mode-untabify ()
  (save-excursion
    (goto-char (point-min))
    (if (search-forward "t" nil t)
        (untabify (1- (point)) (point-max))))
  nil)

(add-hook 'java-mode-hook
      '(lambda ()
          (make-local-variable 'write-contents-hooks)
          (add-hook 'write-contents-hooks 'java-mode-untabify)))

(setq indent-tabs-mode nil)
```

1.1.4 Java Standard Edition 6

We chose Java as the basis for LingPipe because we felt it provided the best tradeoff among efficiency, usability, portability, and library availability.

The presentation here assumes the reader has a basic working knowledge of the Java programming language. We will focus on a few aspects of Java that are particularly crucial for processing textual language data, such as character and string representations, input and output streams and character encodings, regular expressions, parsing HTML and XML markup, etc. In explaining LingPipe's design, we will also delve into greater detail on general features of Java such as concurrency, generics, floating point representations, and the collection package.

This book is based on the latest currently supported standard edition of the Java platform (Java SE), which is version 6. You will need the Java development kit (JDK) in order to compile Java programs. A java virtual machine (JVM) is required to execute compiled Java programs. A Java runtime environment (JRE) contains platform-specific support and integration for a JVM and often interfaces to web browsers for applet support.

Java is available in 32-bit and 64-bit versions. The 64-bit version is required to allocate JVMs with heaps larger than 1.5 or 2 gigabytes (the exact maximum for 32-bit Java depends on the platform).

Licensing terms for the JRE are at

    http://www.java.com/en/download/license.jsp

and for the JDK at

    http://java.sun.com/javase/6/jdk-6u20-license.txt

Java is available for the Windows, Linux, and Solaris operating systems in both 32-bit and 64-bit versions from

    http://java.sun.com/javase/downloads/index.jsp

The Java JDK and JRE are included as part of Mac OS X. Updates are available through

```
http://developer.apple.com/java/download/
```

Java is updated regularly and it's worth having the latest version. Updates include bug fixes and often include performance enhancements, some of which can be quite substantial.

Java must be added to the operating system's executables path so that the shell commands and other packages like Ant can find it.

We have to manage multiple versions of Java, so typically we will define an environment variable `JAVA_HOME`, and add `${JAVA_HOME}/bin` (Unix) or `%JAVA_HOME%\bin` (Windows) to the `PATH` environment variable (or its equivalent for your operating system). We then set `JAVA_HOME` to either `JAVA_1_5`, `JAVA_1_6`, or `JAVA_1_7` depending on use case. Note that `JAVA_HOME` is one level above Java's `bin` directory containing the executable Java commands.

You can test whether you can run Java with the following command, which should produce similar results.

```
> java -version

java version "1.6.0_18"
Java(TM) SE Runtime Environment (build 1.6.0_18-b07)
Java HotSpot(TM) 64-Bit Server VM (build 16.0-b13, mixed mode)
```

Similarly, you can test for the Java compiler version, using

```
> javac -version

javac 1.6.0_20
```

Java Source

You can download the Java source files from the same location as the JDK. The source is subject to yet anoher license, available at

```
http://java.sun.com/javase/6/scsl_6-license.txt
```

The source provides a fantastic set of examples of how to design, code, and document Java programs. We especially recommend studying the source for strings, I/O and collections. Further, like all source, it provides a definitive answer to the questions of what a particular piece of code does and how it does it. This can be useful for understanding deeper issues like thread safety, equality and hashing, efficiency and memory usage.

1.1.5 Ant

We present examples for compilation and for running programs using the Apache Ant build tool. Ant has three key features that make it well suited for expository purposes. First, it's portable across all the platforms that support Java. Second, it provides clear XML representations for core Java concepts such as classpaths and command-line arguments. Third, invoking a target in Ant directly executes the dependent targets and then all of the commands in the target. Thus we find Ant builds easier to follow than those using the classic Unix build tool Make or its modern incarnation Apache Maven, both of which attempt to resolve dependencies among targets and determine if targets are up to date before executing them.

Although we're not attempting to teach Ant, we'll walk through a basic Ant build file later in this chapter to give you at least a reading knowledge of Ant. If you're using an IDE like Eclipse or NetBeans, you can import Ant build files directly to create a project.

Ant is is an Apache project, and as such, is subject to the Apache license,

```
http://ant.apache.org/license.html
```

Ant is available from

```
http://ant.apache.org/
```

You only need one of the binary distributions, which will look like `apache-ant-`*`version`*`-bin.tar.gz`.

First, you need to unpack the distribution. We like directory structures with release names, which is how ant unpacks, using top-level directory names like `apache-ant-1.8.1`. Then, you need to put the `bin` subdirectory of the top-level directory into the `PATH` environment variable so that Ant may be executed from the command line.

Ant requires the `JAVA_HOME` environment variable to be set to the path above the `bin` directory containing the Java executables. Ant's installation instructions suggest setting the `ANT_HOME` directory in the same way, and then adding but it's not necessary unless you will be scripting calls to Ant.

Ant build files may be imported directly into either the Eclipse or NetBeans IDEs (see below for a description of these).

You can test whether Ant is installed with

```
> ant -version

Apache Ant version 1.8.1 compiled on April 30 2010
```

1.1.6 Integrated Development Environment

Many pepeople prefer to write (and compile and debug) code in an integrated development environment (IDE). IDEs offer advantages such as automatic method, class and constant completion, easily configurable text formatting, stepwise debuggers, and automated tools for code generation and refactoring.

LingPipe development may be carried out through an IDE. The two most popular IDEs for Java are Eclipse and NetBeans.

Eclipse IDE

Eclipse provides a full range of code checking, auto-completion and code generation, and debugging facilities. Eclipse is an open-source project with a wide range of additional tools available as plugins. It also has modules for languages other than Java, such as C++ and PHP.

The full set of Eclipse downloads is listed on the following page,

```
http://download.eclipse.org/eclipse/downloads/
```

You'll want to make sure you choose the one compatible with the JDK you are using. Though originally a Windows-based system, Eclipse has been ported to Mac OS X (though Carbon) and Linux.

Eclipse is released under the Eclipse Public License (EPL), a slightly modified version of the Common Public License (CPL) from IBM, the full text of which is available from

```
http://www.eclipse.org/legal/epl-v10.html
```

NetBeans IDE

Unlike Eclipse, the NetBeans IDE is written entirely in Java. Thus it's possible to run it under Windows, Linux, Solaris Unix, and Mac OS X. There are also a wide range of plug-ins available for NetBeans.

Netbeans is free, and may be downloaded from its home page,

```
http://netbeans.org/
```

Its licensing is rather complex, being released under a dual license consisting of the Common Development and Distribution License (CDDL) and version 2 of the GNU Public License version 2 (GPLv2). Full details are at

```
http://netbeans.org/about/legal/license.html
```

1.1.7 Statistical Computing Environment

Although not strictly necessary, if you want to draw nice plots of your data or results, or compute p values for some of the statistics LingPipe reports, it helps to have a statistical computing language on hand. Such languages are typically interpreted, supported interactive data analysis and plotting.

In some cases, we will provide R code for operations not supported in LingPipe. All of the graphs drawn in the book were rendered as PDFs in R and included as vector graphics.

R Project for Statistical Computing

The most popular open-source statistical computing language is still the R Project for Statistical Computing (R). R runs under all the major operating systems, though the Windows installers and compatibility with libraries seem to be a bit more advanced than those for Linux or Mac OS X.

R implements a dialect of the S programming language for matrices and statistics. It has good built-in functionality for standard distributions and matrix operations. It also allows linking to C or Fortran back-end implementations. There are also plug ins (of varying quality) for just about everything else you can imagine. The officially sanctioned plug ins are available from the Comprehensive R Archive Network (CRAN).

R is available from its home page,

```
http://www.r-project.org/
```

The CRAN mirrors are also linked from the page and usually the first web search results for statistics-related queries in R.

R is distributed under the GNU Public License version 2 (GPLv2).

SciPy and NumPy

The Python programming/scripting language is becoming more and more popular for both preprocessing data and analyzing statistics. The library NumPy provides a matrix library on top of Python and SciPy a scientific library including statistics and graphing. The place to get started is the home page,

```
http://numpy.scipy.org/
```

NumPy and SciPy and Python itself are distributed under the much more flexible BSD License.

Like R, Python is interpreted and not very fast at operations like looping. NumPy helps by allowing some operations to be vectorized (as does R). Like R, Python also allows linking C and Fortran back-end packages for efficiency.

1.1.8 Full LingPipe Distribution

LingPipe is distributed under both commercial licenses and under a royalty-free license. A copy of the royalty free license is available at:

```
http://alias-i.com/lingpipe/licenses/lingpipe-license-1.txt
```

LingPipe may be downloaded with full source from its web site:

```
http://alias-i.com/lingpipe/
```

Other than unpacking the gzipped tarball, there is nothing required for installation. Downloading LingPipe is not technically necessary for running the examples in this book because the LingPipe library jar is included with this book's source download.

1.1.9 Book Source and Libraries

The code samples from this book are available via anonymous subversion checkout from the LingPipe sandbox. Specifically, the entire content of the book, including code and text, may be checked out anonymously using,

```
> svn co https://aliasi.devguard.com/svn/sandbox/lpbook
```

The distribution contains all of the code and Ant build files for running it. It also contains the LaTeX source for the book itself as well as the library we created to extract text from the source for automatic inclusion in the book. This last feature is actually critical to ensure the code in the distribution and in the book match.

1.2 Hello World Example

In this section, we provide a very simple hello world program. If you can run it, you'll be all set to jump into the next chapter and get started with the real examples. We'll also take this opportunity to walk through a simple Ant build file.

1.2.1 Running the Example

To the extent we are able, we'll start each discussion with an example of how to run examples. This is pretty easy for the hello-world example. As with all of our examples, we begin by changing directories into the directory containing the example. We suppose that `$BOOK/` is the directory in which the code for the book was unpacked. We then execute the change directory (`cd`) command to move into the subdirectory `/src/intro`,

```
> cd $BOOK/src/intro
```

Note that we have italicized the commands issued by the user. We then run the demo using Ant. In this case, the target is `hello`, invoked by

```
> ant hello
```

which produces the following output

```
Buildfile: c:\lpb\src\intro\build.xml

jar:
    [mkdir] Created dir: c:\lpb\src\intro\build\classes
    [javac] Compiling 1 source file to c:\lpb\src\intro\build\classes
      [jar] Building jar: c:\lpb\src\intro\build\lp-book-intro-4.0.jar

hello:
     [java] Hello World

BUILD SUCCESSFUL
Total time: 1 second
```

First, Ant echoes the name of the build file, here `c:\lpb\src\intro\build.xml`. When Ant is invoked without specifying a particular build file, it uses the `build.xml` in the directory from which it was called.

Reading down the left side of the output, you see the targets that are invoked. The first target invoked is the `jar` target, and the second target is the `hello` target. The targets are left aligned and followed by semicolons. A target consists of an ordered list of dependent targets to invoke before the current target, and an ordered list of tasks to execute after the dependent targets are invoked.

Under the targets, you see the particular tasks that the target executes. These are indented, square bracketed, and right aligned. The `jar` target invokes three tasks, `mkdir`, `javac`, and `jar`. The `hello` target invokes one task, `java`. All of the output for each task is shown after the task name. If there is more than one line of output, the name of the task is repeated.

In this example, the `mkdir` task makes a new directory, here the `build\classes` directory. The `javac` task runs the Java compiler, here compiling a single source file into the newly created directory. The `jar` task builds the Java library into the build subdirectory `build` in file `lp-book-intro-4.0.jar`. Moving onto the `hello` target, the `java` task runs a command-line Java program, in this case printing the output of the hello world program.

The reason the `jar` target was invoked was because the `hello` target was defined to depend on the `jar` target. Ant invokes all dependent targets (recursively) before the invoked target.

Finally, Ant reports that the build was successful and reports the amount of time taken. In the future, we will usually only show the output of the Java program executed and not all of the output from the compilation steps. To save more space, we also remove the `[java]` tag on the task. Under this scheme, the `hello` target invocation would be displayed in this book as

```
> ant hello

Hello World
```

1.2.2 Hello with Arguments

Often we need to supply arguments to programs from the command line. The easiest way to do this with Ant is by setting the properties on the command line. Our second demo uses this facility to provide a customized greeting.

```
> ant -Dfirst=Bobby -Dlast=C hello-name

Hello, Bobby C.
```

Each argument to a program corresponds to a named property, here `first` and `last`. In general, properties are specified `-D`*`key`*`=`*`val`*.[2]

1.2.3 Code Walkthrough

The code for the hello world program can be found in `$BOOK/src/intro/src/com/lingpipe/book/intro/HelloWorld.java`. Throughout the book, the files are organized this way, under the top-level `src` directory, then the name of the chapter (here `intro`), followed by `src`, followed by the path to the actual program. We follow the Java convention of placing files in a directory structure that mirrors their package structure, so the remaining part of the path is `com/lingpipe/book/intro/HelloWorld.java`. The contents of the `HelloWorld.java` program file is as follows.

[2]Depending on your shell, this may also require varying amounts of quoting and/or escaping special characters. For example, keys or values with spaces typically require double quotes on Windows.

```
package com.lingpipe.book.intro;

public class HelloWorld {

     public static void main(String[] args) {
         System.out.println("Hello World");
     }

}
```

As usual for Java programs, the first thing in the file is the package declaration, here `com.lingpipe.book.intro`; note how it matches the end of the path to the file, `com/lingpipe/book/intro`. Next is the class definition, `HelloWorld`, which matches the name of the file, `HelloWorld.java`.

When a Java class is invoked from the command line (or equivalently through Ant), it must implement a static method with a signature like the one shown. Specifically, the method must be named `main`, must take a single string array argument of type `String[]`, must not have a return value (i.e., return type `void`), and must be declared to be both public and static using the `public` and `static` modifiers. The method may optionally throw any kind of exception.

The body of the program here just prints `Hello World` to the system output. As usual, Java provides a handle to the system output through the public static variable `out` of type `java.io.PrintStream` in the `java.lang.System` class.

The more complex program `HelloName` that takes two command-line arguments varies in the body of its `main()` method,

```
public class HelloName {

     public static void main(String[] args) {
         String first = args[0];
         String last = args[1];
         System.out.printf("Hello, %s %s.\n",first,last);
```

As a convention, we assign the command-line arguments read from the array `args` to a sequence of named variables. We then use these same names in the Ant properties (see Section 1.3.3).

1.3 Introduction to Ant

Although we will typically skip discussing the Ant build files, as they're almost all the same, we will go over the one for the hello world program in some detail.

The build file for hello world is located at `$BOOK/src/intro/build.xml`, where `$BOOK/` is the root of the directory in which this book's files have been unpacked. In general, the build files will be broken out by chapter and/or section. The goal is to make each of them relatively self contained so that they may be used as a minimal basis for further development.

1.3.1 XML Declaration

The first thing in any Ant build file is the XML declaration, here

```
<?xml version="1.0" encoding="ASCII"?>
...
```

This just tells Ant's XML parser that what it's looking at is an XML file and that the ASCII character encoding is being used for subsequent data. We chose ASCII because we didn't anticipate using any non-ASCII characters; we could have chosen Latin1 or UTF-8 or even Big5 and written the build file in Chinese. Most XML parsers are robust enough to infer the character encoding, but it's always good practice to make it explicit.

The ellipses (`...`) indicate ellided material that will be filled in (or not) in continuing discussion.

1.3.2 Top-Level Project Element

The top-level element in the XML file is the project declaration, which is just

```
<project>
...
</project>
```

Given our convention for ellipses, the file actually looks as follows,

```
<?xml version="1.0" encoding="ASCII"?>
<project>
...
</project>
```

with the remaining ellipses to be filled in below.

The project element's tag is `project`, and there are no required attributes. The project declaration may optionally be given a name as the value of attribute `name`, a default target to invoke as the value of attribute `default`, and a base directory from which to run as the value of `basedir`. The base directory defaults to the directory containing the build file, which is where we set everything up to run from. We will not need a default target as we will specify all targets explicitly for clarity. The name doesn't really have a function.

1.3.3 Ant Properties

We organize Ant build files in the conventional way starting with properties, which are declared first in the project element's content. The properties define constant values for reuse. Here, we have

```
  <property name="version"
            value="4.0"/>

  <property name="jar"
            value="build/lp-book-intro-${version}.jar"/>
```

The first property definition defines a property `version` with value `4.0` (all values are strings). The second property is named `jar`, with a value, defined in terms of the first property, of `build/lpb-intro-4.0.jar`. note that the value of the property `version` has been substituted for the substring `${version}`. In general, properties are accessed by `${...}` with the property filled in for the ellipses.

Properties may be overridden from the command line by declaring an environment variable for the command. For example,

```
> ant -Djar=foo.jar jar
```

calls the build file, setting the value of the `jar` property to be the string `foo.jar` (which would create a library archive called `foo.jar`). The value of a property in Ant is always the first value to which it is set; further attempts to set it (as in the body of the Ant file) will be ignored.

1.3.4 Ant Targets

Next, we have a sequence of targets, each of which groups together a sequence of tasks. The order of targets is not important.

Clean Target

The first target we have performs cleanup.

```
<target name="clean">
  <delete dir="build"/>
</target>
...
```

This is a conventional clean-up target, given the obvious name of `clean`. Each target has a sequence of tasks that will be executed whenever the target is invoked. Here, the task is a delete task, which deletes the directory named `build`.

It is conventional to have a clean task that cleans up all of the automatically generated content in a project. Here, the `.class` files generated by the compiler (typically in a deletable subdirectory `build`), and the `.jar` file produced by the Java archiver (typically in the top-level directory).

Compiliation/Archive Target

The next target is the compilation target, named `jar`,

```
<target name="jar">
  <mkdir dir="build/classes"/>
  <javac debug="yes"
         debuglevel="source,lines,vars"
         destdir="build/classes"
         includeantruntime="false">
    <compilerarg value="-Xlint:all"/>
```

```
      <src path="src/"/>
    </javac>
    <jar destfile="${jar}">
      <fileset dir="build/classes"
               includes="**/*.class"/>
    </jar>
  </target>
  ...
```

Invoking the `jar` target executes three tasks, a make-directory (`mkdir`), java compilation (`javac`), and Java archive creation (`jar`). Note that, as we exploit here, it is allowable to have a target with the same name as a task, because Ant keeps the namespaces separate.

The first task is the make-directory task, `mkdir`, takes the path for a directory and creates that directory and all of its necessary parent directories. Here, it builds the directory `build/classes`. All files are relative to the base directory, which is by default the directory containing the `build.xml` file, here `$BOOK/src/intro`.

The second task is the Java compilation task, `javac`, does the actual compilation. Here we have supplied the task element with four attributes. The first two, `debug` and `debuglevel` are required to insert debugging information into the compiled code so that we can get stack traces with line numbers when processes crash. These can be removed to save space, but they can be very helpful for deployed systems, so we don't recommend it. The `debug` element says to turn debugging on, and the `debuglevel` says to debug the source down to lines and variable names, which is the maximal amount of debugging information available. The `javac` task may specify the character encoding used for the Java program using the attribute `encoding`.

The destination directory attribute, `destdir` indicates where the compiled classes will be stored. Here, the path is `build/classes`. Note that this is in the directory we first created with the make-directory task. Further recall that the `build` directory will be removed by the clean target.

Finally, we have a flag with attribute `includeantruntime` that says the Ant runtime libraries should not be included in the classpath for the compilation. In our opinion, this should have defaulted to `false` rather than `true` and saved us a line in all the build files. If you don't specify this attribute, Ant gripes and tells you what it's defaulting to.

The java compilation task here has content, startign with a compiler argument. The element `compilerarg` passes the value of the attribute `value` to the underlying call to the `javac` executable in the JDK. Here, we specified a value of `-Xlint:all`. The `-X` options to `javac` are documented by running the command `javac -X`. This `-X` option specifies `lint:all`, which says to turn all lint detection on. This provides extended warnings for form, including deprecation, unchecked casts, lack of serial version IDs, lack or extra override specifications, and so on. These can be turned on and off one by one, but we prefer to leave them all on and produce lint-free programs. This often involves suppressing warnings that would otherwise be unavoidable, such as casts after object I/O or internal uses of deprecated public methods.

When compilation requires external libraries, we may add the classpath specifications either as attributes or elements in the `javac` task. For instance, we can have a classpath specification, which is much like a property definition,

```
<path id="classpath">
  <pathelement location="${jar}"/>
  <pathelement location="../../lib/icu4j-4_4_1.jar"/>
</path>
```

And then we'd add the element

```
<classpath refid="classpath"/>
```

as content in the `javac` target.

The java compilation task's content continues a source element with tag `src`. This says where to find the source code to compile. Here we specify the value of attribute for the path to the source, `path`, as `src/`. As usual, this path is interpreted relative to the base directory, which by default is the one holding the `build.xml` file, even if it's executed from elsewhere.

The third task is the Java archive task, `jar`, which packages up the compiled code into a single compressed file, conventionally suffixed with the string `.jar`. The file created is specified as the value of the `destfile` attribute, here given as `${jar}`, meaning the value of the `jar` property will be substituted, here `build/lpb-intro-4.0.jar`. As ever, this is interpreted relative to the base project directory. Note that the jar is being created in the `build` directory, which will be cleaned by the clean target.

Java Execution Task

The final target is the one that'll run Java,

```
<target name="hello"
        depends="jar">
  <java classname="com.lingpipe.book.intro.HelloWorld"
        classpath="${jar}"
        fork="true">
  </java>
</target>
```

Unlike the previous targets, this target has a dependency, as indicated by the `depends` attribute on the target element. Here, the value is `jar`, meaning that the `jar` target is invoked before the tasks in the `hello` target are executed. This means that the compilation and archive tasks are always executed before the `hello` target's task.

It may seem at this point that Ant is using some notion of targets being up to date. In fact, it's Java's compiler, `javac`, and Java's `jar` command which are doing the checking. In particular, if there is a compiled class in the compilation location that has a later date than the source file, it is not recompiled.[3] Similarly, the `jar` command will not rebuild the archive if all the files from which it were built are older than the archive itself.

In general, there can be multiple dependencies specified as target names separated by commas, which will be invoked in order before the target declaring the dependencies. Furthermore, if the targets invoked through `depends` themselves have dependencies, these will be invoked recursively.

The `hello` target specifies a single task as content. The task is a run-Java task, with element tag `java`. The attribute `classname` has a value indicating which class is executed. This must be a fully specified class with all of its package qualifiers separated by periods (`.`).

To invoke a Java program, we must also have a classpath indicating where to find the compiled code to run. In Ant, this is specified with the `classpath` attribute on the `java` task. The value here is `${jar}`, for which the value of the Java archive for the project will be substituted. In general, there can be multiple archives or directories containing compiled classes on the classpath, and the classpath may be specifeid with a nested element as well as an attribute. Ant contains an entire syntax for specifying path-like structures.

Finally, there is a flag indicated by attribute `fork` being set to value `true`, which tells Ant to fork a new process with a new JVM in which to run the indicated class.

The target `hello-name` that we used for the hello program with two arguments consists of the following Java task.

```
<java classname="com.lingpipe.book.intro.HelloName"
      classpath="${jar}"
      fork="true">
  <arg value="${first}"/>
  <arg value="${last}"/>
</java>
```

This time, the element tagged `java` for the Java task has content, specifically two argument elements. Each argument element is tagged `arg` and has a single attribute `value`, the value of which is passed to the named Java programs `main()` program as arguments. As a convention, our programs will all create string variables of the same name as their first action, just as in our hello program with arguments, which we repeat here,

```
public class HelloName {

    public static void main(String[] args) {
        String first = args[0];
        String last = args[1];
        System.out.printf("Hello, %s %s.\n",first,last);
```

[3]This leads to a confusing situation for statics. Static constants are compiled by value, rather than by reference if the value can be computed at compile time. These values are only recomputed when a file containing the static constant is recompiled. If you're changing the definition of classes on which static constants depend, you need to recompile the file with the constants. Just clean first.

In more elaborate uses of the `java` task, we can also specify arguments to the Java virtual machine such as the maximum amount of memory to use or to use ordinary object pointer compression, set Java system properties, and so on.

1.3.5 Property Files

Properties accessed in an Ant build file may also be specified in an external Java properties file. This is particular useful in a setting where many users access the same build file in different environments. Typically, the build file itself is checked into version control. If the build file specifies a properties file, each user may have their own version of the properties file.

Properties may be loaded directly from the build file, or they may be specified on the command line. From within an Ant build file, the `file` attribute in the `property` element may be used to load properties. For example,

```
<property file="build.properties"/>
```

From the command line, a properties file may be specified with

```
> ant -propertyfile build.properties ...
```

The properties file is interpreted as a Java properties file. For instance, we have supplied a demo properties file `demo.properties` in this chapter's directory we can use with the named greeting program. The contents of this file is

```
first: Jacky
last: R
```

We invoke the Ant target `hello-name` to run the demo, specifying that the properties from the file `demo.properties` be read in.

```
> ant -propertyfile demo.properties hello-name
Hello, Jacky R.
```

Any command-line specifications using `-D` override properties. For example, we can override the value of `first`,

```
> ant -Dfirst=Brooks -propertyfile demo.properties hello-name
Hello, Brooks R.
```

Parsing Properties Files

The parser for properties file is line oriented, allowing the Unix (`\n`), Macintosh (`\r\n` Windows (`\r`) line-termination sequences. Lines may be continued like on the shell with a backslash character.

Lines beginning with the hash sign # or exclamation point (`!`) are taken to be comments and ignored. Blank lines are also ignored.

Each line is interpreted as a key followed by a value. The characters equal sign (=), colon (`:`) or a whitespace character (there are no line terminators within lines; line parsing has already been done).

Character escapes, including Unicode escapes, are parsed pretty much as in Java string literals with a bit more liberal syntax, a few additional escapes (notably for the colon and equals sign) and some excluded characters (notably backspace).

1.3.6 Precedence of Properties

In Ant, whichever property is read first survives. The command line properties precede any other properties, followed by properties read in from a command-line properties file, followed by the properties read in from Ant build files in the order they appear.

Chapter 2

Java Basics

In this appendix, we go over some of the basics of Java that are particularly relevant for text and numerical processing.

2.1 Numbers

In this section, we explain different numerical bases, including decimal, octal, hexadecimal and binary.

2.1.1 Digits and Decimal Numbers

Typically we write numbers in the Arabic form (as opposed to, say, the Roman form), using decimal notation. That is, we employ sequences of the ten digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

A number such as 23 may be decomposed into a 2 in the "tens place" and a 3 in the "ones place". What that means is that $23 = (2 \times 10) + (3 \times 1)$; similarly $4700 = (4 \times 1000) + (7 \times 100)$. We can write these equivalently as $23 = 2 \times 10^1 + 3 \times 10^0$ and $4700 = (4 \times 10^3) + (7 \times 10^2)$. Because of the base of the exponent, decimal notation is also called "base 10".

The number 0 is special. It's the additive identity, meaning that for any number x, $0 + x = x + 0 = x$.

We also conventionally use negative numbers and negation. For instance, -22 is read as "negative 22". We have to add 22 to it to get back to zero. That is, negation is the additive inverse, so that $x + (-x) = (-x) + x = 0$.

The number 1 is also special. 1 is the multiplicative identity, so that $1 \times x = x \times 1 = x$. Division is multiplicative inverse, so that for all numbers x other than 0, $\frac{x}{x} = 1$.

We also use fractions, written after the decimal place. Fractions are defined using negative exponents. For instance $.2 = 2 \times 10^{-1} = \frac{2}{10^1}$, and $.85 = .8 \times 10^{-1} + .5 \times 10^{-2}$.

For really large or really small numbers, we use scientific notation, which decomposes a value into a number times an exponent of 10. For instance, we write 4700 as 4.7×10^3 and 0.0047 as 4.7×10^{-3}. In computer languages, 4700 and 0.0047 are typically written as `4.7E+3` and `4.7E-3`. Java's floating point numbers may be input and output in scientific notation.

2.1.2 Bits and Binary Numbers

Rather than the decimal system, computers work in the binary system, where there are only two values. In binary arithmetic, bits play the role that digits play in the decimal system. A *bit* can have the value 0 or 1.

A number in *binary notation* consists of a sequence of bits (0s and 1s). Bits in binary binary numbers play the same role as digits in decimal numbers; the only difference is that the base is 2 rather than 10. For instance, in binary, 101 = $(1 \times 2^3) + (0 \times 2^2) + (1 \times 2^0)$, which is 7 in decimal notation. Fractions can be handled the same way as in decimal numbers. Scientific notation is not generally used for binary numbers.

2.1.3 Octal Notation

Two other schemes for representing numbers are common in computer languages, octal and hexadecimal. As may be gathered from its name, *octal*octal is base 8, conventionally written using the digits 0–7. Numbers are read in the usual way, so that octal 43 is expanded as $(4 \times 8^1) + (3 \times 8^0)$, or 35 in decimal notation.

In Java (and many other computer languages), octal notation is very confusing. Prefixing a numeric literal with a `0` (that's a zero, not a capital o) leads to it being interpreted as octal. For instance, the Java literal `043` is interpreted as the decimal 35.

2.1.4 Hexadecimal Notation

Hexadecimal is base 16, and thus we need some additional symbols. The first 16 numbers in hex are conventionally written

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F.

In hexadecimal, the value of A is what we'd write as 10 in decimal notation. Similarly, C has the value 12 and F the value 15. We read off compound numbers in the usual way, so that in hex, 93 = $(9 \times 16^1) + (3 \times 16^0)$, or 138 in decimal notation. Similarly, in hex, the number B2F = $(11 \times 16^2) + (2 \times 16^1) + (15 \times 16^0)$, or 2863 in decimal notation.

In Java (and other languages), hexadecimal numbers are distinguished by prefixing them with `0x`. For example, `0xB2F` is the hexadecimal equivalent of the decimal 2863.

2.1.5 Bytes WTF?

The basic unit of organization in contemporary computing systems is the byte. By this we mean it's the smallest chunk of data that may be accessed programatically, though in point of fact, hardware often groups bytes together into larger groups that it operates on all at once. For instance, 32-bit architectures often operate on a sequence of 4 bytes at once and 64-bit architectures on 8 bytes at once. The term "word" is ambiguously applied to a sequence of two bytes, or to the size of the sequence of bytes at which a particular piece of hardware operates.

A *byte*byte is a sequence of 8 bits, and is sometimes called an octet for that reason. Thus there are 256 distinct bytes, ranging from `00000000` to `11111111`. The bits are read from the high (left) end to the low (right end).

In Java, the `byte` primitive type is signed. Any number between `00000000` and `01111111` is interpreted as a binary number with decimal value between 0 and 127 (inclusive). If the high order bit is 1, the value is interpreted as negative. The negative value is value of the least significant 7 bits minus 128. For instance, `10000011` is interpreted as $3 - 128 = -125$, because `00000011` (the result of setting the high bit to `0`) is interpreted as 3.

The unsigned value of a byte `b` is returned by `(b < 0 ? (b + 256) : b)`.

The primitive data type for computers is a sequence of bytes. For instance, the contents of a file is a sequence of bytes, and so is the response from a web server to an HTTP request for a web page. Most importantly for using LingPipe, sequences of characters are represented by sequences of bytes.

2.1.6 Code Example: Integral Number Bases

There is a simple program in the code directory for this appendix

```
src/java/src/com/lingpipe/book/appjava/ByteTable.java
```

for displaying bytes, their corresponding unsigned value, and the conversion of the unsigned value to octal, hexadecimal and binary notations. The work is done by the loop

```
for (int i = 0; i < 256; ++i) {
     byte b = (byte) i; // overflows if i > 127
     int k = b < 0 ? (b + 256) : b;
     System.out.printf("%3d %5d %3s %2s %8s\n",
               k, b, Integer.toOctalString(k),
               Integer.toHexString(k),
               Integer.toBinaryString(k));
}
```

This code may be run from Ant by first changing into this chapter's directory and then invoking the `byte-table` target,

```
> cd $BOOK/src/java
```

```
> ant byte-table
```

The output, after trimming the filler generated by Ant, looks like

```
BASES
 10   -10   8  16        2
```

```
  0      0    0   0         0
  1      1    1   1         1
  2      2    2   2        10
...
  9      9   11   9      1001
 10     10   12   a      1010
 11     11   13   b      1011
...
127    127  177  7f   1111111
128   -128  200  80  10000000
129   -127  201  81  10000001
...
254     -2  376  fe  11111110
255     -1  377  ff  11111111
```

2.1.7 Other Primitive Numbers

In addition to bytes, Java provides three other primitive integer types. Each has a fixed width in bytes. Values of type `short` occupy 2 bytes, `int` 4 bytes, and `long` 8 bytes. All of them use the same signed notation as `byte`, only with more bits.

2.1.8 Floating Point

Java has two floating point types, `float` and `double`. A `float` is represented with 4 bytes and said to be single precision, whereas a `double` is represented with 8 bytes and said to be double precision.

In addition to finite values, Java follows the IEEE 754 floating point standard in providing three additional values. There is positive infinity, conventionally ∞ in mathematical notation, and referenced for floating point values by the static constant `Float.POSITIVE_INFINITY` in Java. There is also negative infinity, $-\infty$, referenced by the constant `Float.NEGATIVE_INFINITY`. There is also an "undefined" value, picked out by the constant `Float.NaN`. There are corresponding constants in `Double` with the same names.

If any value in an expression is `NaN`, the result is `NaN`. A `NaN` result is also returned if you try to divide 0 by 0, subtract an infinite number from itself (or equivalently add a positive and negative infinite number), divide one infinite number by another, or multiple an infinite number by 0.

Both `n/Double.POSITIVE_INFINITY` and `n/Double.NEGATIVE_INFINITY` evaluate to 0 if `n` is finite and non-negative. Conversely, `n/0` evaluates to ∞ if `n` is positive and $-\infty$ if `n` is negative. The result of multiplying two infinite number is ∞ if they are both positive or both negative, and $-\infty$ otherwise. If `Double.POSITIVE_INFINITY` is added to itself, the result is itself, and the same for `Double.NEGATIVE_INFINITY`. If one is added to the other, the result is `NaN`. The negation of an infinite number is the infinite number of opposite sign.

Monotonic transcendental operations like exponentiation and logarithms also play nicely with infinite numbers. In particular, the log of a negative number is

NaN, the log of 0 is negative infinity, and the log of positive infinity is positive infinity. The exponent of positive infinity is positive infinity and the exponent of negative infinity is zero.

2.2 Objects

Every object in Java inherits from the base class Object. There are several kinds of methods defined for Object, which are thus defined on every Java object.

2.2.1 Equality

Java distinguishes reference equality, written ==, from object equality. Reference equality requires that the two variables refer to the exact same object. Some objects may be equal, even if they are not identical. To handle this possibility, the Object class defines a method equals(Object) returning a boolean result.

In the Object class itself, the equality method delegates to reference equality. Thus subclasses that do not override Object's implementation of equality will have their equality conditions defined by reference equality.

Subclasses that define their own notion of equality, such as String, should do so consistently. Specifically, equality should form an equivalence relation. First, equality should be reflexive, so that every object is equal to itself; x.equals(x) is always true for non-null x. Second, equality should be symmetric, so if x is equal to y, y is equal to x; x.equals(y) is true if and only if y.equals(x) is true. Third, equality should be transitive, so that if x is equal to y and y is equal to z, then x is equal to z; x.equals(y) and y.equals(z) both being true imply x.equals(z) is true.

2.2.2 Hash Codes

The Object class also defines a hashCode() method, returning an int value. This value is typically used in collections as a first approximation to equality. Thus the specification requires consistency with equality in that if two objects are equal as defined by equals(), they must have the same hash code.

In order for collections and other hash-code dependent objects to function properly, hash codes should be defined

Although the implementation of hashCode() is not determined for Object by the language specification, it is required to be consistent with equality. Thus any two objects that are reference equal must have the same hash code.

2.2.3 String Representations

The Object class defines a toString() method returning a String. The default behavior is to return the name of the class followed by a hexadecimal representation of its hash code. This method should be overridden in subclasses primarily to help with debugging. Some subclasses use toString() for real work, too, as we will see with StringBuilder.

2.2.4 Other Object Methods

Determining an Object's Class

The runtime class of an object is returned by the `getClass()` method. This is particularly useful for deserialized objects and other objects passed in whose class might not be known. It is neither very clear nor very efficient to exploit `getClass()` for branching logic. Note that because generics are erased at runtime, there is no generic information in the return.

Finalization

At some point after there are no more references to an object left alive in a program, it will be finalized by a call to `finalize()` by the garbage collector. Using finalization is tricky and we will not have cause to do so in this book. For some reason, it was defined to throw arbitrary throwables (the superinterface of exceptions).

Threading

Java supports multithreaded applications as part of its language specification. The primitives for synchronization are built on top of the `Object` class. Each object may be used as a lock to synchronize arbitrary blocks of code using the `synchronized` keyword. Methods may also be marked with `synchornized`, which is equivalent to explicitly marking their code block as synchronized.

In addition, there are three `wait()` methods that cause the current thread to pause and wait for a notification, and two `notify()` methods that wake up (one or all) threads waiting for this object.

Rather than dealing with Java concurrency directly, we strongly recommend the `java.util.concurrent` library for handling threading of any complexity beyond exclusive synchronization.

2.2.5 Object Size

There's no getting around it – Java objects are heavy. The lack of a C-like `struct` type presents a serious obstacle to even medium-performance computing. To get around this problem, in places where you might declare an array of `struct` type in C, LingPipe employs parallel arrays rather than arrays of objects.

The language specification does not describe how objects are represented on disk, and different JVMs may handle things differently. We'll discuss how the 32-bit and 64-bit Sun/Oracle JVMs work.

Header

Objects consist of a header and the values of their member variables. The header itself consists of two references. The first establishes the object's identity. This identity reference is used for reference equality (==) and thus for locking, and also as a return value for the `hashCode()` implementation in `Object`. Even if an object is relocated in memory, its reference identity and locking properties remain fixed. The second piece of information is a reference to the class definition.

A reference-sized chunk of memory is used for each piece of information. In a a 32-bit architecture, references are four bytes. In 64-bit architectures, they are 8 bytes by default.

As of build 14 of Java SE 6, the `-XX:+UseCompressedOops` option may be used as an argument to `java` on 64-bit JVMs to instruct them to compress their ordinary object pointers (OOP) to 32-bit representations. This saves an enormous amount of space, cutting the size of a simple `Object` in half. It does limit the heap size to around four billion objects, which should be sufficient for most applications.

Member Values

Additional storage is required for member variables. Each primitive member variable requires at least the amount of storage required for the primitive (e.g., 1 byte for `byte`, 2 bytes for `short`, and 8 bytes for `double`). References will be 4 bytes in a 32-bit JVM and a 64-bit JVM using compressed pointers, and 8 bytes in the default 64-bit JVM.

To help with word alignment, values are ordered in decreasing order of size, with doubles and longs coming first, then references, then ints and floats, down to bytes.

Finally, the object size is rounded up so that its overall size is a multiple of 8 bytes. This is so all the internal 8-byte objects inside of an object line up on 8-byte word boundaries.

Static variables are stored at the class level, not at the individual object level.

Subclasses

Subclasses reference their own class definition, which will in turn reference the superclass definition. They lay their superclass member variables out the same way as their superclass. They then lay out their own member variables. If the superclass needed to be padded, some space is recoverable by reordering the subclass member variables slightly.

2.2.6 Number Objects

For each of the primitive number types, there is a corrsponding class for representing the object. Specifically, the number classes for integers are `Byte`, `Short`, `Integer`, and `Long`. For floating point, there is `Float` and `Double`. These objects are all in the base package `java.lang`, so they do not need to be explicitly imported before they are used. Objects are useful because many of the underlying Java libraries, particular the collection framework in `java.util`, operate over objects, rather than over primitives.[1]

[1] Open source libraries are available for primitive collections. The Jakarta Commons Primitives and Carrot Search's High Performance Primitive Collections are released under the Apache license, and GNU Trove under the Lessger GNU Public License (LGPL).

Each of these wrapper classes holds an immutable reference to an object of the underlying type. Thus the class is said to “box” the underlying primitive (as in “put in a box”). Each object has a corresponding method to return the underlying object. For instance, to get the underlying byte from a `Byte` object, use `byteValue()`.

Two numerical objects are equal if and only if they are of the same type and refer to the same primitive value. Each of the numerical object classes is both serializable and comparable, with comparison being defined numerically.

Constructing Numerical Objects

Number objects may be created using one-argument constructors. For instance, we can construct an object for the integer 42 using `new Integer(42)`.

The problem with construction is that a fresh object is allocated for each call to `new`. Because the underlying references are immutable, we only need a single instance of each object for each underlying primitive value. The current JVM implementations are smart enough to save and reuse numerical objects rather than creating new ones. The preferred way to acquire a reference to a numerical object is to use the `valueOf()` static factory method from the approrpiate class. For example, `Integer.valueOf(42)` returns an object that is equal to the result of `new Integer(42)`, but is not guaranteed to be a fresh instance, and thus may be reference equal (==) to other `Integer` objects.

Autoboxing

Autoboxing automatically provides an object wrapper for a primitive type when a primitive expression is used where an object is expected. For instance, it is legal to write

```
Integer i = 7;
```

Here we require an `Integer` object to assign, but the expression `7` returns an `int` primitive. The compiler automatically boxes the primitive, rendering the above code equivalent to

```
Integer i = Integer.valueOf(7);
```

Autoboxing also applies in other contexts, such as loops, where we may write

```
int[] ns = new int[] { 1, 2, 3 };
for (Integer n : ns) { ... }
```

Each member of the array visited in the loop body is autoboxed. Boxing is relatively expensive in time and memory, and should be avoided where possible.

The Java compiler also carries out auto-unboxing, which allows things like

```
int n = Integer.valueOf(15);
```

This is equivalent to using the `intValue()` method,

```
Integer nI = Integer.valueOf(15);
int n = nI.intValue();
```

Number Base Class

Conveniently, all of the numerical classes extend the abstract class `Number`. The class `Number` simply defines all of the get-value methods, `byteValue()`, `shortValue()`, ..., `doubleValue()`, returning the corresponding primitive type. Thus we can call `doubleValue()` on an `Integer` object; the return value is equivalent to casting the `int` returned by `intValue()` to a `double`.

2.3 Arrays

Arrays in Java are objects. They have a fixed size that is determined when they are constructed. Attempts to set or get objects out of range raises an `IndexOutOfBoundsException`. This is much better behaved than C, where out of bounds indexes may point into some other part of memory that may then be inadvertently read or corrupted.

In addition to the usual object header, arrays also store an integer length in 4 bytes. After that, they store the elements one after the other. If the values are `double` or `long` values or uncompressed references, there is an additional 4 bytes of padding after the length so that they start on 8-byte boundaries.

2.4 Synchronization

LingPipe follows to fairly simple approaches to synchronization, which we describe in the next two sections. For more information on synchronization, see Goetz et al.'s definitive book *Java Concurrency in Practice* (see Appendix B.2.2 for a full reference).

2.4.1 Immutable Objects

Wherever possible, instances of LingPipe classes are immutable. After they are constructed, immutable objects are completely thread safe in the sense of allowing arbitrary concurrent calls to its methods.

(Effectively) Final Member Variables

In LingPipe's immutable objects, almost all member variabls are declared to be final or implemented to behave as if they are declared to be final.

Final objects can never change. Once they are set in the constructor, they always have the same value.

LingPipe's effectively final variables never change value. In this way, they display the same behavior as the hash code variable in the implementation of Java's `String`. In the source code for `java.lang.String` in Java Standard Edition version 1.6.18, authored by Lee Boynton and Arthur van Hoff, we have the following four member variables:

```
private final char value[];
private final int offset;
private final int count;
private int hash;
```

The final member variables value, offset, and count represent the characte slice on which the string is based. This is not enough to guarantee true immutability, because the values in the value array may change. The design of the String class guarantees that they don't. Whenever a string is constructed, its value array is derived from another string, or is copied. Thus no references to value ever escape to client code.

The hash variable is not final. It is initialized lazily when the hashCode() method is called if it is has not already been set.

```
public int hashCode() {
    int h = hash;
    if (h == 0) {
        int off = offset;
        char val[] = value;
        int len = count;
        for (int i = 0; i < len; i++) {
            h = 31*h + val[off++];
        }
        hash = h;
    }
    return h;
}
```

Note that there is no synchronization. The lack of synchronization and public nature of hashCode() exposes a race condition. Specifically, two threads may concurrently call hashCode() and each see the hash code variable hash with value 0. They would then both enter the loop to compute and set the variable. This is safe because the value computed for hash only depends on the truly final variables offset, value, and count.

Safe Construction

The immutable objects follow the safe-construction practice of not releasing references to the object being constructed to escape during construction. Specifically, we do not supply the this variable to other objects within constructors. Not only must we avoid sending a direct reference through this, but we can also not let inner classes or other data structures containing implicit or explicit references to this to escape.

2.4.2 Read-Write Synchronization

Objects which are not immutable in LingPipe require read-write synchronization. A read method is one that does not change the underlying state of an object. In an immutable object, all methods are read methods. A write method may change the state of an object.

Read-write synchronization allows concurrent read operations, but ensures that writes do not execute concurrently with reads or other writes.

Java's Concurrency Utilities

Java's built-in `synchronized` keyword and block declarations use objects for simple mutex-like locks. Non-static methods declared to be synchronized use their object instance for synchronization, whereas static methods use the object for the class.

In the built-in `java.util.concurrent` library, there are a number of lock interfaces and implementations in the subpackage `locks`.

The `util.concurrent` base interface *`Lock`* generalizes Java's built-in `synchronized` blocks to support more fine-grained forms of lock acquisition and relaese, allowing operations like acquisitions that time out.

The subinterface `ReentrantLock` adds reentrancy to the basic lock mechanism, allowing the same thread to acquire the same lock more than once without blocking.

The `util.concurrent` library also supplies an interface *`ReadWriteLock`* for read-write locks. It contains two methods, `readLock()` and `writeLock()`, which return instances of the interface *`Lock`*.

There is only a single implementation of *`ReadWriteLock`*, in the class `ReentrantReadWriteLock`. Both of the locks in the implementation are declared to be reentrant (recall that a subclass may declare more specific return types than its superclass or the interfaces it implements). More specifically, they are declared to return instances of the static classes `ReadLock` and `WriteLock`, which are nested in ReentrantReadWriteLock.

The basic usage of a read-write lock is straightforward, though configuring and tuning its behavior is more involved. We illustrate the basic usage with the simplest possible case, the implementation of a paired integer data structure. The class has two methods, a paired set and a paired get, that deal with two values at once.

Construction simply stores a new read write lock, which itself never changes, so may be declared final.

```
private final ReadWriteLock mRwLock;
private int mX, mY;

public SynchedPair(boolean isFair) {
    mRwLock = new ReentrantReadWriteLock(isFair);
}
```

The set method acquires the write lock before setting both values.

```
public void set(int x, int y) {
    try {
        mRwLock.writeLock().lock();
        mX = x;
        mY = y;
    } finally {
        mRwLock.writeLock().unlock();
    }
}
```

Note that it releases the lock in a `finally` block. This is the only way to guarantee, in general, that locks are released even if the work done by the method raises an exception.

The get method is implemented with the same idiom, but using the read lock.

```
public int[] xy() {
    try {
        mRwLock.readLock().lock();
        return new int[] { mX, mY };
    } finally {
        mRwLock.readLock().unlock();
    }
}
```

Note that it, too, does all of its work after the lock is acquired, returning an array consisting of both values. If we'd tried to have two separate methods, one returning the first integer and one returning the second, there would be no way to actually get both values at once in a synchronized way without exposing the read-write lock itself. If we'd had two set and two get methods, one for each component, we'd have to use the read-write lock on the outside to guarantee the two operations always happened together.

We can test the method with the following command-line program.

```
public static void main(String[] args) throws
    InterruptedException {
    final SynchedPair pair = new SynchedPair(true);
    Thread[] threads = new Thread[32];
    for (int i = 0; i < threads.length; ++i)
        threads[i] = new Thread(new Runnable() {
                public void run() {
                    for (int i = 0; i < 2000; ++i) {
                        int[] xy = pair.xy();
                        if (xy[0] != xy[1])
                            System.out.println("error");
                        pair.set(i,i);
                        Thread.yield();
                    }
                }
            });
    for (int i = 0; i < threads.length; ++i)
        threads[i].start();
    for (int i = 0; i < threads.length; ++i)
        threads[i].join();
    System.out.println("OK");
}
```

It constructs a `SynchedPair` and assigns it to the variable `pair`, which is delcared to be final so as to allow its usage in the anonymous inner class runnable. It then creates an array of threads, setting each one's runnable to do a sequence of paired sets where the two values are the same. It tests first that the values it reads are coherent and writes out an error message if they are not. The runnables all yield after their operations to allow more interleaving of the threads in the test. After constructing the threads with anonymous inner class `Runnable` implementations, we start each of the threads and join them so the message won't print and JVM won't exit until they're ll done.

The Ant target `synched-pair` runs the demo.

```
> ant synched-pair

OK
```

The program takes about 5 seconds to run on our four-core notebook.

Chapter 3

Characters and Strings

Following Java, LingPipe uses Unicode character representations. In this chapter, we describe the Unicode character set, discuss character encodings, and how strings and other character sequences are created and used in Java.

3.1 Character Encodings

For processing natural language text, we are primarily concerned with the representation of characters in natural languages. A set of (abstract) characters is known as a character set. These range from the relatively limited set of 26 characters (52 if you count upper and lower case characters) used in English to the tens of thousands of characters used in Chinese.

3.1.1 What is a Character?

Precise definitions of characters in human writing systems is a notoriously tricky business.

Types of Character Sets

Characters are used in language to represent sounds at the level of phonemic segments (e.g., Latin), syllables (e.g., Japanese Hirigana and Katakana scripts, Linear B, and Cherokee), or words or morphemes, also known as logograms (e.g., Chinese Han characters, Egyptian hieroglyphics).

These distinctions are blurry. For instance, both Chinese characters and Egyptian hieroglyphs are typically pronounced as single syllables and words may require multiple characters.

Abstraction over Visualization

One problem with dealing with characters is the issue of visual representation versus the abstraction. Character sets abstract away from the visual representations of characters.

A sequence of characters is also an abstract concept. When visualizing a sequence of characters, different writing systems lay them out differently on the page. When visualizing a sequence of characters, we must make a choice in how to lay them out on a page. English, for instance, traditionally lays out a sequence of characters in left to right order horizontally across a page. Arabic, on the other hand, lays out a sequence of characters on a page horizontally from right to left. In contast, Japanese was traditionally set from top to bottom with the following character being under the preceding characters.

English stacks lines of text from top to bottom, whereas traditional Japanese orders its columns of text from right to left.

When combining multiple pages, English has pages that "turn" from right to left, whereas Arabic and traditional Japanese conventionally order their pages in the opposite direction. Obviously, page order isn't relevant for digital documents rendered a page at a time, such as on e-book readers.

Even with conventional layouts, it is not uncommon to see caligraphic writing or signage laid out in different directions. It's not uncommon to see English written from top to bottom or diagonally from upper left to lower right.

Compound versus Primitive Characters

Characters may be simple or compound. For instance, the character *ö* used in German is composed of a plain Latin *o* character with an umlaut diacritic on top. Diacritics are visual representations added to other characters, and there is a broad range of them in European languages.

Hebrew and Arabic are typically written without vowels. In very formal writing, vowels may be indicated in these languages with diacritics on the consonants. Devanagari, a script used to write languages including Hindi and Nepali, includes pitch accent marks.

The Hangul script used for Korean is a compound phonemic system often involving multiple glyphs overlaid to form syllabic units.

3.1.2 Coded Sets and Encoding Schemes

Eventually, we need to represent sequences of characters as sequences of bytes. There are two components to a full character encoding.

The first step is to define a coded character set, which assigns each character in the set a unique non-negative integer code point. For instance, the English character 'A' (capital A) might be assigned the code point 65 and 'B' to code point 66, and so on. Code points are conventionally written using hexadecimal notation, so we would typically use the hexadecmial notation 0x41 instead of the decimal notation 65 for capital A.

The second step is to define a character encoding scheme that maps each code point to a sequence of bytes (bytes are often called octets in this context).

Translating a sequence of characters in the character set to a sequence of bytes thus consists of first translating to the sequence of code points, then encoding these code points into a sequence of bytes.

3.1.3 Legacy Character Encodings

We consider in this section some widely used character encodings, but because there are literally hundreds of them in use, we can't be exhaustive.

The characters in these and many other character sets are part of Unicode. Because LingPipe and Java both work with Unicode, typically the only thing you need to know about a character encoding in order to convert data in that encoding to Unicode is the name of the character encoding.

ASCII

ASCII is a character encoding consisting of a small character set of 128 code points. Being designed by computer scientists, the numbering starts at 0, so the code points are in the range 0–127 (inclusive). Each code point is then encoded as the corresponding single unsigned byte with the same value. It includes characters for the standard keys of an American typewriter, and also some "characters" representing abstract typesetting notions such as line feeds and carriage returns as well as terminal controls like ringing the "bell".

The term "ASCII" is sometimes used informally to refer to character data in general, rather than the specific ASCII character encoding.

Latin1 and the Other Latins

Given that ASCII only uses 7 bits and bytes are 8 bits, the natural extension is to add another 128 characters to ASCII. Latin1 (officially named ISO-8859-1) did just this, adding common accented European characters to ASCII, as well as punctuation like upside down exclaimation points and question marks, French quote symbols, section symbols, a copyright symbol, and so on.

There are 256 code points in Latin1, and the encoding just uses the corresponding unsigned byte to encode each of the 256 characters numbered 0–255.

The Latin1 character set includes the entire ASCII character set. Conveniently, Latin1 uses the same code points as ASCII for the ASCII characters (code points 0–127). This means that every ASCII encoding is also a Latin1 encoding of exactly the same characters.

Latin1 also introduced ambiguity in coding, containing both an 'a' character, the '¨' umlaut character, and their combination 'ä'. It also includes all of 'a', 'e', and the compound 'æ', as well as '1', '/', '2' and the compound $\frac{1}{2}$.

With only 256 characters, Latin1 couldn't render all of the characters in use in Indo-European languages, so a slew of similar standards, such as Latin2 (ISO-8859-2) for (some) Eastern European languages, Latin3 (ISO-8859-1) for Turkish, Maltese and Esperanto, up through Latin16 (ISO-8859-16) for other Eastern European languages, new styles of Western European languages and Irish Gaelic.

Windows 1252

Microsoft Windows, insisting on doing things its own way, uses a character set known as Windows-1252, which is almost identical to Latin1, but uses code points 128–159 as graphical characters, such as curly quotes, the euro and yen symbols, and so on.

Big5 and GB(K)

The Big5 character encoding is used for traditional Chinese scripts and the GB (more recently GBK) for simplified Chinese script. Because there are so many characters in Chinese, both were designed to use a fixed-width two-byte encoding of each character. Using two bytes allows up to $2^{16} = 65,536$ characters, which is enough for most of the commonly used Chinese characters.

3.1.4 Unicode

Unicode is the de facto standard for character sets of all kinds. The latest version is Unicode 5.2, and it contains over 1,000,000 distinct characters drawn from hundreds of languages (technically, it assumes all code points are in the range 0x0 to 0x10FFFF (0 to 1,114,111 in decimal notation).

One reason Unicode has been so widely adopted is that it contains almost all of the characters in almost all of the widely used character sets in the world. It also happens to have done so in a thoughful and well-defined manner.

Conveniently, Unicode's first 256 code points (0–255) exactly match those of Latin1, and hence Unicode's first 128 code points (0–127) exactly match those of ASCII.

Unicode code points are conventionally displayed in text using a hexadecimal representation of their code point padded to at least four digits with initial zeroes, prefixed by `U+`. For instance, code point 65 (hex 0x41), which represents capital A in ASCII, Latin1 and Unicode, is conventionally written U+0041 for Unicode.

The problem with having over a million characters is that it would require three bytes to store each character ($2^{24} = 16,777,216$). This is very wasteful for encoding languages based on Latin characters like Spanish or English.

Unicode Transformation Formats

There are three standard encodings for Unicode code points that are specified in the Unicode standard, UTF-8, UTF-16, and UTF-32 ("UTF" stands for "Unicode transformation format").[1] The numbers represent the coding size; UTF-8 uses single bytes, UTF-16 pairs of bytes, and UTF-32 quadruples of bytes.

UTF-32

The UTF-32 encoding is fixed-width, meaning each character occupies the same number of bytes (in this case, 4). As with ASCII and Latin1, the code points are encoded directly as bits in base 2.

[1]There are also non-standard encodings of (subsets of) Unicode, like Apache Lucene's and the one in Java's `DataOutput.writeUTF()` method.

There are actually two UTF-32 encodings, UTF-32BE and UTF-32LE, depending on the order of the four bytes making up the 32 bit blocks. In the big-endian (BE) scheme, bytes are ordered from left to right from most significant to least significant digits (like for bits in a byte).

In the little-endian (LE) scheme, they are ordered in the opposite direction. For instance, in UTF-32BE, U+0041 (capital A), is encoded as 0x00000041, indicating the four byte sequence 0x00, 0x00, 0x00, 0x41. In UTF32-LE, it's encoded as 0x41000000, corresponding to the byte sequence 0x41, 0x00, 0x00, 0x00.

There is also an unmarked encoding scheme, UTF-32, which tries to infer which order the bytes come in. Due to restrictions on the coding scheme, this is usually possible. In the simplest case, 0x41000000 is only legal in little endian, because it'd be out of range in big-endian notation.

Different computer processors use different endian-ness internally. For example, Intel and AMD x86 architecture is little endian, Motorola's PowerPC is big endian. Some hardware, such as Sun Sparc, Intel Itanium and the ARM, is called bi-endian, because endianness may be switched.

UTF-16

The UTF-16 encoding is variable length, meaning that different code points use different numbers of bytes. For UTF-16, each character is represented using either two bytes (16 bits) or four bytes (32 bits).

Because there are two bytes involved, their order must be defined. As with UTF-32, UTF-16BE is the big-endian order and UTF-16LE the little-endian order.

In UTF-16, code points below U+10000 are represented using a pair of bytes using the natural unsigned encodings. For instance, our old friend U+0041 (capital A) is represented in big endian as the sequence of bytes `0x00`, `0x41`.

Code points at or above U+10000 are represented using four bytes arranged in two pairs. Calculating the two pairs of bytes involves bit-twiddling. For instance, given an integer code point `codepoint`, the following code calculates the four bytes. [2]

```
int LEAD_OFFSET = 0xD800 - (0x10000 >> 10);
int SURROGATE_OFFSET = 0x10000 - (0xD800 « 10) - 0xDC00;

int lead = LEAD_OFFSET + (codepoint >> 10);
int trail = 0xDC00 + (codepoint & 0x3FF);

int byte1 = lead >>> 8;
int byte2 = lead & 0xFF;
int byte3 = trail >>> 8;
int byte4 = trail & 0xFF;
```

Going the other way around, given the four bytes (as integer variables), we can calculate the code point with the following code.

[2]This code example and the following one for decoding were adapted from the example code supplied by the Unicode Consortium in their FAQ at

`http://unicode.org/faq/utf_bom.html`

```
int lead = byte1 « 8 | byte2;
int trail = byte3 « 8 | byte4;
int codepoint = (lead « 10) + trail + SURROGATE_OFFSET;
```

Another way to view this transformation is through the following table.[3]

Code Point Bits				*UTF-16 Bytes*
xxxxxxxx xxxxxxxx			xxxxxxxx	xxxxxxxx
000uuuuuxxxxxxxxxxxxxxxx	110110ww	wwxxxxxx	110111xx	xxxxxxxx

Here wwww = uuuuu - 1 (interpreted as numbers then recoded as bits). The first line indicates is that if the value fits in 16 bits, those 16 bits are used as is. Note that this representation is only possible because not all code points between 0 and $2^{16} - 1$ are legal. The second line indicates that if we have a code point above U+10000, we strip off the high-order five bits uuuuu and subtract one from it to get four bits wwww; again, this is only possible because of the size restriction on uuuuu. Then, distribute these bits in order across the four bytes shown on the right. The constants 110110 and 110111 are used to indicate what are known as surrogate pairs.

Any 16-bit sequence starting with 110110 is the leading half of a surrogate pair, and any sequence starting with 110111 is the trailing half of a surrogate pair. Any other sequence of initial bits means a 16-bit encoding of the code point. Thus it is possible to determine by looking at a pair of bytes whether we have a whole character, the first half of a character represented as a surrogate pair, or the second half of a character represented as a pair.

UTF-8

UTF-8 works in much the same way as UTF-16 (see the previous section), only using single bytes as the minimum code size. We use the table-based visualization of the encoding scheme.[4]

Code Point Bits				*UTF-8 Bytes*
0xxxxxxxx				0xxxxxxx
00000yyyyyxxxxxx			110yyyyy	10xxxxxx
zzzzyyyyyyxxxxxx		1110zzzz	10yyyyyy	10xxxxxx
000uuuuuzzzzyyyyyyxxxxxx	11110uuu	10uuzzzz	10yyyyyy	10xxxxxx

Thus 7-bit ASCII values, which have values from U+0000 to U+007F, are encoded directly in a single byte. Values from U+0080 to U+07FF are represented with two bytes, values between U+0800 and U+FFFF with three bytes, and values between U+10000 and U+10FFFF with four bytes.

Non-Overlap Principle

The UTF-8 and UTF-16 encoding schemes obey what the Unicode Consortium calls the "non-overlap principle." Technically, the leading, continuing and trailing

[3]Adapted from *Unicode Standard Version 5.2*, Table 3-5.
[4]Adapted from *Unicode Standard Version 5.2*, Table 3-6.

code units (bytes in UTF-8, pairs of bytes in UTF-16) overlap. Take UTF-8 for example. Bytes starting with 0 are singletons, encoding a single character in the range U+0000 to U+007F. Bytes starting with 110 are the leading byte in a two-byte sequence representing a character in the range U+0080 to U+07FF. Similarly, bytes starting with 1110 represent the leading byte in a three-byte sequence, and 11110 the leading byte in a four-byte sequence. Any bytes starting with 10 are continuation bytes.

What this means in practice is that it is possible to reconstruct characters locally, without going back to the beginning of a file. At most, if a byte starts with 10 you have to look back up to three bytes to find the first byte in the sequence.

Furthermore, the corruption of a byte is localized so that the rest of the stream doesn't get corrupted if one byte is corrupted.

Another advantage is that the sequence of bytes encoding one character is never a subsequence of the bytes making up another character. This makes applications like search more robust.

Byte Order Marks

In the encoding schemes which are not explicitly marked as being little or big endian, namely UTF-32, UTF-16 and UTF-8, it is also legal, according to the Unicode standard, to prefix the sequence of bytes with a byte-order mark (BOM). It is not legal to have byte-order marks preceding the explicitly marked encodings, UTF-32BE, UTF-32LE, UTF-16BE, UTF-16LE.

For UTF-32, the byte-order mark is a sequence of four bytes indicating whether the following encoding is little endian or big endian. the sequence 0x00, 0x00, 0xFE, 0xFF indicates a big-endian encoding and the reverse, 0xFF, 0xFE, 0x00, 0x00 indicates little endian.

For UTF-16, two bytes are used, 0xFE, 0xFF for big endian, and the reverse, 0xFF, 0xFE for little endian.

Although superfluous in UTF-8, the three byte sequence 0xEF, 0xBB, 0xBF is a "byte order mark", though there is no byte order to mark. As a result, all three UTF schemes may be distinguished by inspecting their initial bytes.

Any text processor dealing with Unicode needs to handle the byte order marks. Some text editing packages automatically insert byte-order marks for UTF encodings and some don't.

Character Types and Categories

The Unicode specification defines a number of character types and general categories of character. These are useful in Java applications because characters may be examined programatically for their class membership and the classes may be used in regular expressions.

For example, category "Me" is the general category for enclosing marks, and "Pf" the general category for final quote punctuation, whereas "Pd" is the general category for dash-like punctuation, and "Sc" the category of currency symbols.

Unicode supplies a notion of case, with the category "Lu" used for uppercase characters and "Ll" for lowercase.

Unicode marks the directionality in which characters are typically wirtten using character types like "R" for right-to-left and "L" for left-to-right.

We provide a sample program in Section 3.4.2 for exploring the types of properties in Java.

Normalization Forms

In order to aid text processing efforts, such as search, Unicode defines several notions of normalization. These are defined in great detail in Unicode Standard Annex #15, *The Unicode Normalization Forms*.

First, we consider decomposition of precomposed characters. For example, consider the three Unicode characters: U+00E0, LATIN SMALL LETTER A WITH GRAVE, rendered *à*, U+0061, LATIN SMALL LETTER A, rendered as *a*, and U+0300, COMBINING GRAVE ACCENT, which is typically rendered by combining it as an accent over the previous character. For most purposes, the combination U+00E0 means the same thing in text as U+0061 followed by U+0300.

There are four basic normalization forms, reproduced here from Table 1 in the normalization annex document.

Normalization Form	*Description*
NFD	canonical decomposition
NFC	canonical decomposition, followed by canonical composition
NFKD	compatibility decomposition
NFKC	compatibility decomposition, followed by canonical composition

The first two forms, NFD and NFC, first perform a canonical decomposition of character sequences. This breaks compound characters into a base character plus accent marks, sorting multiple accent marks into a canonical order. For instance, the canonical decomposition of the sequence U+0061, U+0300 (small a followed by grave accent) is the same as that of the single character U+00E0 (small a with grave accent), namely the sequence U+0061, U+0300.

In general, a full canonical decomposition continues to decompose characters into their components until a sequence of non-decomposible characters is produced. Furthermore, if there is more than one nonspacing mark (like U+0300), then they are sorted into a canonical order.

NFC differs from NFD in that it further puts the characters back together into their canonical composed forms, whereas NFD leaves them as a sequence. Applying NFC normalization to either the sequence U+0061, U+0300 or the precomposed character U+00E0 results in the canonical composed character U+00E0.

The second two forms, NFKD and NFKC, replace the canonical decomposition with a compatibility decomposition. This reduces different Unicode code points that are in some sense the same character to a common representation. As an example, the character U+2075, SUPERSCRIPT FIVE, is typically written as a superscripted version of U+0035, DIGIT FIVE, namely as ^{5}. Both involve the underlying character *5*. With canonical decomposition, U+2075 is normalized to U+0035. Similarly, the so-called “full width” forms of ASCII characters, U+FF00 to U+FF5E, are often used in Chinese; each has a corresponding ASCII character in U+0021 to U+007E to which it is compatibility normalized.

There are special behaviors for multiple combining marks. When recombining, only one mark may wind up being combined, as the character with multiple marks may not have its own precomposed code point.

There is also special behavior for composites like the ligature character *ﬁ*, which in Unicode gets its own precomposed code point, U+FB01, LATIN SMALL LIGATURE FI. Only the compatibility version decomposes the ligature into U+0066, LATIN SMALL F, followed by U+0069, LATIN SMALL I.

Two strings are said to be canonical equivalent if they have the same canonical decomposition, and compatibility equivalent if they have the same compatibility decomposition.

The normalizations are nice in that if strings are canonical equivalents, then their NFC normalizations are the same. If two strings are canonical equivalents, their NFD normalizations are also the same. The NFD normalization and NFC normalization of a single string are not necessarily the same. The same properties hold for compatibility equivalents and NFKC and NFKD normalizations. Another nice property is that the transformations are stable in that applying a normalization twice (in the sense of applying it once, then applying it again to the output), produces the same result as applying it once (i.e., its an idemopotent transformation).

We show how to convert character sequences to their canonical forms programatically using the ICU package in Section 3.10.1.

3.2 Encoding Java Programs

Java has native support for many encodings, as well as the ability to plug in additional encodings as the need arises (see Section 3.9.3). Java programs themselves may be written in any supported character encoding.

If you use characters other than ASCII characters in your Java programs, you should provide the `javac` command with a specification of which character set you used. If it’s not specified, the platform default is used, which is typically Windows-1252 on Windows systems and Latin1 on other systems, but can be modified as part of the install.

The command line to write your programs in Chinese using Big5 encoding is

```
javac -encoding Big5 ...
```

For Ant, you want to use the `encoding` attribute on the `javac` task, as in

```
<javac encoding="Big5" ...
```

3.2.1 Unicode Characters in Java Programs

Even if a Java program is encoded in a small character set encoding like ASCII, it has the ability to express arbitrary Unicode characters by using escapes. In a Java program, the sequence `\u`*xxxx* behaves like the character U+*xxxx*, where *xxxx* is a hexadecimal encoded value padded to four characters with leading zeroes.

For instance, instead of writing

```
int n = 1;
```

we could write

```
\u0069\u006E\u0074\u0020\u006E\u0020\u003D\u0020\u0031\u003B
```

and it would behave exactly the same way, because *i* is U+0069, *n* is U+006E, *t* is U+0074, the space character is U+0020, and so on, up through the semicolon character (;), which is U+003B.

As popular as Unicode is, it's not widely enough supported in components like text editors and not widely enough understood among engineers. Writing your programs in any character encoding other than ASCII is thus highly error prone. Our recommendation, which we follow in LingPipe, is to write your Java programs in ASCII, using the full `\u`*xxxx* form of non-ASCII characters.

3.3 char Primitive Type

Java's primitive `char` data type is essentially a 16-bit (two byte) unsigned integer. Thus a `char` can hold values between 0 and $2^1 6 - 1 = 65535$. Furthermore, arithmetic over `char` types works like an unsigned 16-bit representation, including casting a `char` to a numerical primitive type (i.e., `byte`, `short`, `int`, or `long`).

Character Literals

In typical uses, instances of `char` will be used to model text characters. For this reason, there is a special syntax for character literals. Any character wrapped in single quotes is treated as a character in a program. For instance, we can assign a character to a variable with

```
char c = 'a';
```

Given the availability of arbitrary Unicode characters in Java programs using the syntax `\u`*xxxx*, it's easy to assign arbitrary Unicode characters. For instance, to assign the variable `c` to the character *à*, U+00E0, use

```
char c = '\u00E0';
```

Equivalently, because values of type `char` act like unsigned short integers, it's possible to directly assign integer values to them, as in

```
char c = 0x00E0;
```

The expression \0x00E0 is an integer literal in hexadecimal notation. In general, the compiler will infer which kind of integer is intended.

Because the Java char data type is only 16 bits, the Unicode escape notation only goes up to U+FFFF.

Several characters may not be used directly within character (or string) literals: new lines, returns, form feeds, backslash, or a single quote character. We can't just drop in a Unicode escape like \u000A for a newline, because that behaves just like a newline itself. To get around this problem, Java provides some special escape sequences which may be used in character and string literals.

Escape	*Code Point*	*Description*
\n	U+000A	LINE FEED
\t	U+0009	CHARACTER TABULATION
\b	U+0008	BACKSPACE
\r	U+000D	CARRIAGE RETURN
\f	U+000C	FORM FEED
\\	U+005C	REVERSE SOLIDUS
\'	U+0027	APOSTROPHE
\"	U+0022	QUOTATION MARK

For instance, we'd write

```
char c = '\n';
```

to assign the newline character to the variable c.

Character Arithmetic

A character expression may be assigned to an integer result, as in

```
int n = 'a';
```

After this statement is executed, the value of the variable n will be 97. This is often helpful for debugging, because it allows the code points for characters to be printed or otherwise examined.

It's also possible, though not recommended, to do arithmetic with Java character types. For instance, the expression 'a'+1 is equal to the expression 'b'.

Character data types behave differently in the context of string concatenation than integers. The expressions 'a' + "bc" and \u0061 + "bc" are identical because the compiler treats \u0061 and 'a' identically, because the Unicode code point for *a* is U+0061. Both expressions evaluate to a string equal to "abc".

The expression 0x0061 + "bc" evaluates to "97bc", because 0x0061 is taken as a hexadecimal integer literal, which when concatenated to a string, first gets converted to its string-based decimal representation, "97". There are static methods in java.lang.Integer to convert integers to string-based representations. The method Integer.toString(int) may be used to convert an integer to a string-based decimal notation; Integer.toHexString(int) does the same for hex.

Characters Interpreted as UTF-16

Java made the questionable decision to use 2-byte representations for characters. This is both too wide and too narrow. It's wasteful for representing text in Western European languages, where most characters have single-byte representations. It's also too narrow, in that any code point above U+FFFF requires two characters to represent in Java; to represent code points would require an integer (primitive `int` type, which is 4 bytes). [5]

When `char` primitives are used in strings, they are interpreted as UTF-16. For any code point that uses only two bytes in UTF-16, this is just the unsigned integer representation of that code point. This is why UTF-16 seemed so natural for representing characters in the original Java language design. Unfortunately, for code points requiring four bytes in UTF-16, Java requires two characters, so now we have all the waste of 2-bytes per character and all the disadvantages of having Unicode code points that require multiple characters.

3.4 `Character` Class

Just like for the numerical types, there is a class, `Character`, used to box an underlying primitive of type `char` with an immutable reference. As for numbers, the preferred way to acquire an instance of `Character` is with the static factory method `Character.valueOf(char)`. Autoboxing and unboxing works the same way as for numerical values.

Like the numerical classes, the `Character` class is also serializable and comparable, with comparison being carried out based on interpreting the underlying `char` as an unsigned integer. There are also utilities in Java to sort sequences of `char` values (such as strings) in a locale-specific way, because not every or dialect of a language uses the same lexicographic sorting.

Equality and hash codes are also defined similarly, so that two character objects are equal if and only if they reference the same underlying character.

3.4.1 Static Utilities

The `Character` class supports a wide range of Unicode-sensitive static utility constants and methods in addition to the factory method `valueOf()`.

For each of the Unicode categories, there is a constant, represented using a byte, because the class predates enums. For instance, the general category "Pe" in the Unicode specification is represented by the constant `END_PUNCTUATION` and the general category of mathematical symbols, "Sm" in Unicode, is represented by the constant `MATH_SYMBOL`.

For many of these constants, there are corresponding methods. For instance, the method `getDirectionality(char)` returns the directionality of a character as a byte value, which can then be tested against the possible values represented

[5] The designers' choice is more understandable given that there were no code points that used more than a single pair of bytes for UTF-16 when Java was designed, though Unicode all along advertised that it would continue to add characters and was not imposing a 16-bit upper limit.

as static constants. There are also useful methods for determining the category of a characer, such as `isLetter(char)` and `isWhitespace(char)`.

Because multiple Java `char` instances might be needed to represent a code point, there are also methods operating on code points directly using integer (primitive `int`) arguments. For instance, `isLetter(int)` is the same as `isLetter(char)` but generalized to arbitrary code points. The method `charCount(int)` returns the number of `char` values required to represent the code point (so the value will be 1 or 2, reflecting 2 or 4 bytes in UTF-16). The methods `isLowSurrogate(char)` and `isHighSurrogate(char)` determine if the `char` represents the first or second half of the UTF-16 representation of a code point above U+FFFF.

The method `charPointAt(char[],int)` determines the code point that starts at the specified index int he array of `char`. The method `charPointCount(char[],int,int)` returns the number of code points encoded by the specified `char` array slice.

3.4.2 Demo: Exploring Unicode Types

The full set of character categories recognized by Java is detailed in the constant documentation for `java.lang.Character`. There's a missing component to the implementation, which makes it hard to take a given `char` value (or code point) and find its Unicode properties. To that end, we've written a utility class `UnicodeProperties` that displays properties of Unicode. The code just enumerates the constants in `Character`, indexes them by name, writes out a key to the values produced by `Character.getType(char)` along with their standard abbreviations (as used in regular expressions) and names.

The code may be run from the Ant target `unicode-properties`. This target will print the key to the types, then the type of each character in a range specified by the properites `low.hex` and `high.hex` (both inclusive).

```
> ant -Dlow.hex=0 -Dhigh.hex=FF unicode-properties

ID CD TYPE                        ID CD TYPE
 0 Cn UNASSIGNED                  15 Cc CONTROL
                                  16 Cf FORMAT
 1 Lu UPPERCASE_LETTER            18 Co PRIVATE_USE
 2 Ll LOWERCASE_LETTER            19 Cs SURROGATE
 3 Lt TITLECASE_LETTER
 4 Lm MODIFIER_LETTER             20 Pd DASH_PUNCTUATION
 5 Lo OTHER_LETTER                21 Ps START_PUNCTUATION
                                  22 Pe END_PUNCTUATION
 6 Mn NON_SPACING_MARK            23 Pc CONNECTOR_PUNCTUATION
 7 Me ENCLOSING_MARK              24 Po OTHER_PUNCTUATION
 8 Mc COMBINING_SPACING_MARK      29 Pi INITIAL_QUOTE_PUNCTUATION
                                  30 Pf FINAL_QUOTE_PUNCTUATION
 9 Nd DECIMAL_DIGIT_NUMBER
10 Nl LETTER_NUMBER               25 Sm MATH_SYMBOL
11 No OTHER_NUMBER                26 Sc CURRENCY_SYMBOL
```

```
                                  27 Sk MODIFIER_SYMBOL
12 Zs SPACE_SEPARATOR             28 So OTHER_SYMBOL
13 Zl LINE_SEPARATOR
14 Zp PARAGRAPH_SEPARATOR
```

For display, we have organized the results based on primary type. The initial character of a type, which is capitalized, indicates the main category, and the second character, which is lowercase, indicates the subtype. We've added extra space in the output to pick out the groupings. For instance, there are five letter types, all of which begin with L, and three number types, which begin with N.

After displaying all types, the program prints out all of the characters in range (in hex) followed by their type. We specified the entire Latin1 range in the Ant invocation, but only display the first instance of each type rather all 256 characters.

```
CH   CD TYPE                      CH   CD TYPE
 0   Cc CONTROL                   41   Lu UPPERCASE_LETTER
20   Zs SPACE_SEPARATOR           5E   Sk MODIFIER_SYMBOL
21   Po OTHER_PUNCTUATION         5F   Pc CONNECTOR_PUNCTUATION
24   Sc CURRENCY_SYMBOL           61   Ll LOWERCASE_LETTER
28   Ps START_PUNCTUATION         A6   So OTHER_SYMBOL
29   Pe END_PUNCTUATION           AB   Pi INITIAL_QUOTE_PUNCTUATION
2B   Sm MATH_SYMBOL               AD   Cf FORMAT
2D   Pd DASH_PUNCTUATION          B2   No OTHER_NUMBER
30   Nd DECIMAL_DIGIT_NUMBER      BB   Pf FINAL_QUOTE_PUNCTUATION
```

Letter Characters

U+0041, LATIN CAPITAL LETTER A (*A*), and U+0061, LATIN SMALL LETTER A (*a*), are typed as uppercase and lowercase letters (Lu and Ll), respectively. All the other Latin 1 letters, even the ones with accents, are typed as letters.

Number Characters

U+0030, DIGIT ZERO (*0*), is marked as a decimal digit number (Nd), as are the other digits.

U+00B2, SUPERSCRIPT TWO (^{2}), is assigned to the other number type (No).

Separator Characters

U+0020, SPACE, the ordinary (ASCII) space character, is typed as a space separator (Zs). There are also types for line separators (Zl) and paragraph separators (Zp), but there are no characters of these types in Latin1.

Punctuation Characters

U+0021, EXCLAMATION MARK (*!*) is classified as other punctuation (Po), as is the question mark (*?*) and period (*.*), as well as the pound sign (*#*), asterisk (***), percentage sign (*%*), at-sign (*@*).

Characters U+0028 and U+0029, LEFT PARENTHESIS (*(*) and RIGHT PARENTHESIS (*)*), the round parentheses, are assigned start and end punctuation types (Ps and Pe), respectively. Other brackets such as the ASCII curly and square varieties, are also assigned start and end punctuation types.

U+002D, HYPHEN-MINUS (-), used for hyphenation and for arithmetical subtraction, is typed as a dash punctuation (Pd). It can also be used as an arithmetic operation, presenting a problem for a typology based on unique types for characters. The problem with whitespace (typed as a separator) and newline (typed as a control character) is similar.

U+005F, LOW LINE (_), the ASCII underscore character, is typed as connector punctuation (Pc).

U+00AB, LEFT-POINTING DOUBLE ANGLE QUOTATION MARK («), and U+00BB, RIGHT-POINTING DOUBLE ANGLE QUOTATION MARK (»), traditionally known as the left and right guillemet, and used in French like open and close quotation marks in English, are initial quote and final quote punctuation categories, (Pi and Pf), respectively. The "curly" quotes, U+201C (“), and U+201D (”), are also divided into open and initial and final quote punctuation.

In contrast, U+0022, QUOTATION MARK ("), and U+0027, APOSTROPHE ('), the standard ASCII undirected quotation mark and apostrophe, are marked as other punctuation (Po). This is problematic because they are also used for quotations in ASCII-formatted text, which is particularly prevalent on the web.

Symbol Characters

U+0024, DOLLAR SIGN (*$*), is marked as a currency symbol (Sc), as are the signs for the euro, the yen, and other currencies.

U+002B, PLUS SIGN (+), the symbol for arithmetic addition, is characterized as a math symbol (Sm).

U+005E, CIRCUMFLEX ACCENT (^), is typed as a modifier symbol (Sm). This is a modifier in the sense that it is typically rendered as a circumflex on the following character.

U+00A6, the Latin1 BROKEN BAR, which is usually displayed as a broken horizontal bar, is typed as an other symbol (So).

Control Characters

U+0000, <CONTROL> NULL, is the null control character, and is typed as a control character (Cc). Backspaces, carriage returns, linefeeds, and other control characters get the same type. This can be confusing for newlines and their ilk, which aren't treated as space characters in the Unicode typology, but are in contexts like regular expressions.

Although it is usually rendered visually, U+00AD, SOFT HYPHEN, is typed as a format character (Cf). It's used in typesetting for continuation hyphens used to split words across lines. This allows a word split with a hyphen across lines to be put back together. This character shows up in conversions from PDF documents generated by LaTeX, such as the one for this book.

3.5 CharSequence Interface

Java provides an interface for dealing with sequences of characters aptly named CharSequence. It resides in the package java.lang, so it's automatically imported. Implementations include strings and string builders, which we describe in the next sections.

The character sequence interface specifies a method length() that returns the number of characters in the sequence.

The method charAt(int) may be used to return the character at a specified position in the string. Like arrays, positions are numbered starting from zero. Thus the method must be passed an integer in the range from zero (inclusive) to the length of the string (exclusive). Passing an index that's out of range raises an IndexOutOfBoundsException, which is also in the java.lang package.

Indexing from position zero (inclusive) to length (exclusive) supports the conventional for loop for iterating over items in a sequence. For instance, if cs is a variable of type CharSequence, we can loop

```
for (int i = 0; i < cs.length(); ++i) {
    char c = cs.charAt(i);
    // do something
```

Because lengths are instances of int, are only allowed to be as long as the longest positive int value, $2^{32-1} - 1$, which is approximately 2 billion. Any longer sequence of characters needs to be streamed (e.g., with a Reader or Writer) or scanned (e.g., with a RandomAccessFile).

Creating Subsequences

Given a character sequence, a subsequence may be generated using subSequence(int,int), where the two indexes are start position (inclusive) and end position (exclusive). As for charAt, the indexes must be in range (start greater than or equal to zero and less than or equal to end, and end less than or equal to the sequence length). One nice feature of this inclusive-start/exclusive-end encoding is that the length of the subsequence is the end minus the start.

The mutable implementations of CharSequence approach subsequencing differently. The StringBuffer and StringBuilder classes return instances of String, which are immutable. Thus changes to the buffer or builder won't affect any subsequences generated. The CharBuffer abstract class in java.nio returns a view into itself rather than an independent copy. Thus when the underlying buffer changes, so does the subsequence.

Converting Character Sequences to Strings

The last method in CharSequence is toString(), which is specified to return a string that has the same sequence of characters as the character sequence.

Equality and Hashing

The `CharSequence` interface does not place any constraints on the behavior of equality or hash codes. Thus two character sequences may contain the same sequence of characters while not being equal and having different hash codes. We'll discuss equality and hash codes for particular implementations of `CharSequence` in the next few sections.

3.6 `String` Class

Strings are so fundamental to Java that they have special literals and operators built into the Java language. Functionally, the class `String`, in the package `java.lang`, provides an immutable implementation of `CharSequence`. Sequence immutability, along with a careful lazy initialization of the hash code, makes strings thread safe.

Because `String` is declared to be final, there can be no subclasses defined that might potentially override useful invariants such as equality, comparability, and serializability. This is particularly useful because strings are so widely used as constants and method inputs and outputs.

3.6.1 Constructing a String

Strings may be constructed from arrays of integer Unicode code points, arrays of characters, or arrays of bytes. In all cases, a new array of characters is allocated to hold the underlying `char` values in the string. Thus the string becomes immutable and subsequent changes to the array from which it was constructed will not affect it.

From Unicode Code Points

When constructing a string from an array of Unicode code points, the code points are encoded using UTF-16 and the resulting sequence converted to a sequence of `char` values. Not all integers are valid Unicode code points. This constructor will throw an `IllegalArgumentException` if there are illegal code points in the sequence.

From `char` Values

When constructing a string from an array of characters or another string or character sequence, the characters are just copied into the new array. There is no checking to ensure that the sequence of `char` values is a legal UTF-16 sequence. Thus we never know if a string is well-formed Unicode.

A string may also be constructed from another string. In this case, a deep copy is created with its own fresh underlying array. this may seem useless, but consider the behavior of `substring()`, as defined below; in this case, constructing a new copy may actually save space. Strings may also be constructed by copying the current contents of a string builder or string buffer (see the next section for more about these classes, which implement the builder pattern for strings).

From Bytes

When constructing a string from bytes, a character encoding is used to convert the bytes into `char` values. Typically character encodings are specified by name. Any supported encoding or one of its aliases may be named; the list of available encodings may be accessed programatically and listed as shown in Section 3.9.3.

If a character encoding is not specified, the platform's default is used, resulting in highly non-portable behavior.

A character encoding is represented as an instance of `CharSet`, which is in the `java.nio.charset` package. A `CharSet` may be provided directly to the constructor for `String`, or they may be specified by name. Instances of `CharSet` may be retrieved by name using the static method `CharSet.forName(String)`.

If a byte sequence can't be converted to Unicode code points, its behavior is determined by the underlying `CharSetDecoder` referenced by the `CharSet` instance. This decoder will typically replace illegal sequences of bytes with a replacement string determined by `CharSetDecoder.replacement()`, typically a question mark character.

If you need more control over the behavior in the face of illegal byte sequences, you may retrieve the `CharSet` by name and then use its `newDecoder()` method to get a `CharSetDecoder`.

3.6.2 String Literals

In Java code, strings may be represented as characters surrounded by double quotes. For instance,

```
String name = "Fred";
```

The characters between the quotes can be any characters in the Java program. Specifically, they can be Unicode escapes, so we could've written the above replacing `F` and `e` with Unicode escapes, as in

```
String name = "\u0046r\u0065d";
```

In practice, string literals essentially create static constants for the strings. Because strings are immutable, the value of string contstants including literals, will be shared. Thus `"abc" == "abc"` will evaluate to true because the two literals will denote reference identical strings. This is carried out as if `intern()` had been called on constructed strings, as described in Section 3.6.11.

String literals may contain the same character escape sequences as character literals (see Section 3.3).

3.6.3 Contents of a String

Strings are very heavy objects in Java. First, they are full-fledged objects, as opposed to the C-style strings consisting only of a pointer into an array. Second, they contain references to an array of characters, char[], holding the actual characters, as well as an integer start index into that array and an integer length. They also store their hash code as an integer. Thus an empty string will be 36 bytes in the 32-bit Sun/Oracle JVM and 60 bytes in the 64-bit JVM.[6]

3.6.4 String Equality and Comparison

Two strings are equal if they have the same length and the same character at each position. Strings are not equal to objects of any other runtime class, even other implementations of CharSequence.

Strings implement Comparable<String> in a way that is consistent with equality. The order they use is lexicographic sort over the underlying char values. This is exactly the way words are sorted in dictionaries. The easiest way to define lexicographic sort is with code, here comparing strings s and t.

```
for (int i = 0; i < Math.min(s.length(),t.length()); ++i)
     if (s.charAt(i) != t.charAt(i))
         return s.charAt(i) - t.charAt(i);
return s.length() - t.length();
```

We walk over both strings from the beginning until we find a position where the characters are different, at which point we return the difference.[7] For instance, "abc" is less than "abde" "ae", and "e". If we finish either string, we return the difference in lengths. Thus "a" is less than "ab". If we finish one of the strings and the strings are the same length, we return zero, because they are equal.

3.6.5 Hash Codes

The hash code for a string s is computed as if by

```
int hashCode = 0;
for (int i = 0; i < s.length(); ++i)
     hashCode = 31 * hashCode + s.charAt(i);
```

This definition is consistent with equals in that two equal strings have the same character sequence, and hence the same hash code.

Because the hash code is expensive to compute, requiring an assignment, add and multiply for each character, it is lazily initialized. What this means is that when hash code is called, if there is not a value already, it will be computed and stored. This operation is thread safe without synchronization because integer assignment is atomic and the sequence of characters is immutable.

[6]The JVM option -XX:+UseCompressedOops will reduce the memory footprint of 64-bit objects to the same size as 32-bit objects in many cases.

[7]Conveniently, char values are converted to integers for arithmetic, allowing negative results for the difference of two values. Also, because string lengths are non-negative, there can't be underflow or overflow in the difference.

3.6.6 Substrings and Subsequences

The `substring(int,int)` and method return subsequences of a string specified by a start position (inclusive) and an end position (exclusive). The `subSequence(int,int)` method just delegates to the substring method.

When a substring is generated from a string, it shares the same array as the string from which it was generated. Thus if a large string is created and a substring generated, the substring will actually hold a reference to a large array, potentially wasting a great deal of memory. Alternatively, if a large string is created and a large number of substrings are created, this implementation potentially saves space.

A copy of the character array underlying the string is returned by `toCharArray()`.

3.6.7 Simple Pattern Matching

Simple text processing may be carried out using methods supplied by `String`. For instance, `endsWith(String)` and `startsWith(String)` return boolean values based on whether the string has the specified exact suffix or prefix. These comparisons are carried out based on the underlying `char[]` array. There is also a multiple argument `regionMatches()` method that compares a fixed number the characters starting at a given position to be matched against the characters starting at a different position in a second string.

There is also a method `contains(CharSequence)`, which returns true if the string contains the characters specified by the character sequence argument as a substring. And there is a method `contentEquals(CharSequence)` that compares the character content of the string with a character sequence.

The `indexOf(char,int)` method returns the index of the first appearance of a character after the specified position, and `lastIndexOf(char,int)` does the same thing in reverse, returning the last instance of the character before the specified position. There are also string-based methods, `indexOf(String,int)` and `lastIndexOf(String,int)`.

3.6.8 Manipulating Strings

Several useful string manipulation methods are provided, such as `trim()`, which returns a copy of the string with no whitespace at the front or end of the string. The methods `toLowerCase(Locale)` and `toUpperCase(Locale)` return a copy of the string in the specified locale.

3.6.9 Unicode Code Points

The string class also defines methods for manipulating Unicode code points, which may be coded by either one or two underlying `char` values. The method `codePointAt(int)` returns the integer code point starting at the specified index. To find out the length of a string in unicode code points, there is the method `codePointCount(int,int)` which returns the number of Unicode code points coded by the specified range of underlying characters. It is up to the user to get the boundaries correct.

3.6.10 Testing String Validity

LingPipe provides a static utility method `isValidUTF16(CharSequence)` in the `Strings` class in the `com.aliasi.util` package. This method checks that every high surrogate UTF-16 `char` value is followed by a low surrogate UTF-16 value. Surrogacy is defined as by Java's `Character.isHighSurrogate()` and `Character.isLowSurrogate()` methods (see Section 3.4.1).

3.6.11 Interning Canonical Representations

The JVM keeps an underlying pool of string constants. A call to `intern()` on a string returns a string in the constant pool. The method `intern()` returns a string that is equal to the string on which it is called, and is furthermore reference identical to any other equal string that has also been interned.

For example, suppose we call

```
String s = new String("abc");
String t = new String("abc");
```

The expression `s.equals(t)` will be true, but `s == t` will be false. The constructor `new` always constructs new objects. Next, consider calling

```
String sIn = s.intern();
String tIn = t.intern();
```

Afterwards, `sIn == tIn` will evaluate to true, as will `sIn.equals(tIn)`. The interned strings are equal to their originals, so that `sIn.equals(s)` and `tIn.equals(t)` also return true.

3.6.12 Utility Methods

There are a number of convenience methods in the string class that reproduce behavior available elsewhere.

Regular Expression Utilities

The string class contains various split and replace methods based on regular expressions. For instance, `split(String)` interprets its argument as a regular expression and returns an array of strings containing the sequences of text between matches of the specified pattern. The method `replaceFirst(String,String)` replaces the first match of a regular expression (the first argument) with a given string (the second argument). In both cases, matching is defined as in the regular expression method `Matcher.find()`.

String Representations of Primitives

The string class contains a range of static `valueOf()` methods for converting primitives, objects, and arrays of characters to strings. For instance, `valueOf(double)` converts a double-precision floating point value to a string using the same method as `Double.toString(double)`.

3.6.13 Example Conversions

We wrote a simple program to illustrate encoding and decoding between strings and byte arrays.

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

    System.setOut(new PrintStream(System.out,true,"UTF-8"));

    String s = "D\u00E9j\u00E0 vu";
    System.out.println(s);

    String encode = args[0];
    String decode = args[1];

    byte[] bs = s.getBytes(encode);
    String t = new String(bs,decode);
```

First, the string *Déjà vu* is assigned to variable `s`. The string literal uses two Unicode escapes, `\u00E9` for *é* and `\u00E0` for *à*. The name of the character encoding to use for encoding and decoding are read from the command line.

Then we convert the string `s` to an array of bytes `bs` using the character encoding `encode`. Next, we reconstitute a string using the bytes we just created using the character encoding `decode`. Either of these methods may raise an exception if the specified encoding is not supported, so the `main()` method is declared to throw an `UnsupportedEncodingException`, imported from `java.io`. The rest of the code just prints the relevant characters and bytes.

There is an Ant target `byte-to-string` that's configured to take two arguments consisting of the names of the character encodings. The first is used to convert a string to bytes and the second to convert the bytes back to a string. For instance, to use UTF-8 to encode and Latin1 (ISO-8859-1) to decode, the program prints

```
> ant -Dencode=UTF-8 -Ddecode=Latin1 byte-to-string
```

```
char[] from string
  44    e9    6a    e0    20    76    75
byte[] from encoding with UTF-8
  44    c3    a9    6a    c3    a0    20    76    75
char[] from decoding with Latin1
  44    c3    a9    6a    c3    a0    20    76    75
```

The characters are printed using hexadecimal notation. The initial line of characters is just what we'd expect; for instance the `char` value 44 is the code point and UTF-16 representation of U+0044, LATIN CAPITAL LETTER D, `char` e9 picks out U+00E9, LATIN SMALL LETTER E WITH ACUTE, and so on.

We have printed out the bytes using unsigned hexadecimal notation (see Section 2.1.5 for more information about signs and bytes). Note that there are more bytes than `char` values. This is because characters above U+0080 require two bytes or more to encode in UTF-8 (see Section 3.1.4). For instance, the sequence of bytes 0xC3, 0xA9 is the UTF-8 encoding of U+00E9, LATIN SMALL LETTER E WITH ACUTE.

We use Latin1 to decode the bytes back into a sequence of `char` values. Latin1 simply translates the bytes one-for-one into unsigned characters (see Section 3.1.3. This is fine for the first byte, 0x44, which corresponds to U+0044, but things go wrong for the second character. Rather than recover the single `char` value U+00E9, we have 0xC3 and 0xA9, which are U+00C3, LATIN CAPITAL LETTER A WITH TILDE, and U+00A9, COPYRIGHT SIGN. Further down, we see 0xA0, which represents U+00A0, NO-BREAK SPACE. Thus instead of getting *Déjà vu* back, we get *D©Ãj© vu*.

If we use `-Dencode=UTF-16` when invoking the Ant target `byte-to-string`, we get the bytes

```
byte[] from encoding with UTF-16
  fe   ff    0   44    0   e9    0   6a    0   ...
```

The first two bytes are the byte-order marks U+00FE and U+00FF; in this order, they indicate big-endian (see Section 3.1.4). The two bytes after the byte-order marks are 0x0 and 0x44, which is the UTF-16 big-endian representation of the character U+0044, LATIN CAPITAL LETTER D. Similarly, 0x0 and 0xE9 are the UTF-16 big-endian representation of U+00E9, LATIN SMALL LETTER E WITH ACUTE. The UTF-16 encoding is twice as long as the Latin1 encoding, plus an additional two bytes for the byte-order marks.

3.7 `StringBuilder` Class

The `StringBuilder` class is essentially a mutable implementation of the `CharSequence` interface. It contains an underlying array of `char` values, which it resizes as necessary to support a range of `append()` methods. It implements a standard builder pattern for strings, where after adding content, `toString()` is used to construct a string based on the buffered values.

A typical use would be to create a string by concatenating the members of an array, as in the following, where `xs` is an array of strings of type `String[]`,

```
StringBuilder sb = new StringBuilder();
for (String x : xs)
     sb.append(x);
String s = sb.toString();
```

First a string builder is created, then values are appended to it, then it is converted to a string. When a string builder is converted to a string, a new array is constructed and the buffered contents of the builder copied into it.

The string builder class lets you append any type of argument, with the effect being the same as if appending the string resulting from the matching `String.valueOf()` method. For instance, non-null objects get converted using their `toString()` methods and null objects are converted to the string *null*. Primitive numbers converted to their decimal notation, using scientific notation if necessary, and booleans to either *true* or *false*.

The underlying array for buffering characters starts at length 16 by default, though may be set explicitly in the constructor. Resizing adds one to the length then doubles it, leading to roughly log (base 2) resizings in the length of the buffer (e.g., 16, 34, 70, 142, 286, 574, ...). Because string builders are rarely used in locations that pose a computational bottleneck, this is usually an acceptable behavior.

In cases where character sequence arguments suffice, this allows us to bypass the construction of a string altogether. Thus many of LingPipe's methods accept character sequence inputs without first converting to strings.

3.7.1 Unicode Support

The append method `appendCodePoint(int)` appends the characters representing the UTF-16 encoding of the specified Unicode code point. The builder also supports a `codePointAt()`, `codePointBefore()`, and `codePointCount()` methods with behavior like those for strings.

3.7.2 Modifying the Contents

String builders are mutable, and hence support operations like `setCharAt(int,char)`, which replaces a single character, and `insert(int,String)` and the longer form `replace(int,int,String)`, which splice the characters of a string into the buffer in place of the specified character slice. There is also a `setLength(int)` method which changes the length of the charactr sequence, either shrinking or extending the underlying buffer as necessary.

3.7.3 Reversing Strings

The method `reverse()` method in `StringBuilder` reverses the `char` values in a string, except valid high-low surrogate pairs, which maintain the same ordering. Thus if the string builder holds a well-formed UTF-16 sequence representing a sequence of code points, the output will be the well-formed UTF-16 sequence representing the reversed sequence of code points. For instance, the String literal `"\uDC00\uD800"` constructs a string whose `length()` returns 2. Because `\uDC00` is a high-surrogate UTF-16 `char` value and `\uD800` is a low-surrogate, together they pick out the single code point U+10000, the first code point beyond the basic multilingual plane. We can verify this by evaluating the expression `"\uDC00\uD800".codePointAt(0)` and verifying the result is 0x10000.

3.7.4 Chaining and String Concatenation

Like many builder implementations, string builders let you chain arguments. Thus it's legal to write

```
int n = 7; String s = "ABC"; boolean b = true;
char c = 'C'; Object z = null;
String t = new StringBuilder().append(n).append(s).append(b)
     .append(c).append(z).toString();
```

with the reslt that string t has a value equal to the literal "7ABCtrueCnull".

In fact, this is exactly the behavior underlying Java's heavily overloaded addition (+) operator. When one of the arguments to addition is a string, the other argument is converted to a string as if by the appropriate String.valueOf() method. For example, the variables u and v will have equivalent values after executing the following pair of statements.

```
String u = 7 + "ABC";
String v = String.valueOf(7) + "ABC";
```

3.7.5 Equality and Hash Codes

The StringBuilder class does not override the definitions of equality or hash codes in Object. Thus two string builders are equal only if they are reference equal. In particular, string builders are never equal to strings. There is a utility method on strings, String.contentEquals(CharSequence) method which returns true if the char sequence is the same length as the string and contains the same character as the string at each position.

3.7.6 String Buffers

The class StringBuffer, which predates StringBuilder, is essentially a string builder with synchronization. Because it is rarely desirable to write to a string buffer concurrently, StringBuffer has all but disappeared from contemporary Java programs.

3.8 CharBuffer Class

The CharBuffer class resides in the java.nio package along with corresponding classes for other primitive types, such as ByteBuffer and IntBuffer. These buffers all hold sequences of their corresponding primitive types and provide efficient means to bulk load or bulk dump sequences of the primitive values they hold.

3.8.1 Basics of Buffer Positions

The buffers all extend the Buffer class, also from java.nio. Buffers may be created backed by an array that holds the appropriate primitive values, though array backing is not required.

Every buffer has four important integer values that determine how it behaves, its capacity, position, mark and limit. Buffers may thus only have a capacity of up to `Integer.MAX_VALUE` (about 2 billion) primitive values.

The capacity of a buffer is available through the `capacity()` method. This is the maximum number of items the buffer can hold.

The position of a buffer is available through the `position()` method and settable with `setPosition(int)`. The position is the index of the next value to be accessed. The position must always be between zero (inclusive) and the buffer's capacity (inclusive).

Relative read and write operations start at the current position and increment the position by the number of primitive values read or written. Underflow or overflow exceptions will be thrown if there are not enough values to read or write respectively. There are also absolute read and write operations which require an explicit index.

The limit for a buffer indicates the first element that should not be read or written to. The limit is always between the position (inclusive) and the capacity (inclusive).

Three general methods are used for resetting these values. The `clear()` method sets the limit to the capacity and the position to zero. The `flip()` method sets the limit to the current position and the position to zero. The `rewind()` method just sets the position to zero.

All of the buffers in `java.nio` also implement the `compact()` operation, which moves the characters from between the current position and the limit to the start of the buffer, resetting the position to be just past the moved characters (limit minus original position plus one).

Finally, the mark for a buffer indicates the position the buffer will have after a call to `reset()`. It may be set to the current position using `mark()`. The mark must always be between the zero (inclusive) and the position (inclusive).

3.8.2 Equality and Comparison

Equality and comparison work on the remaining characters, which is the sequence of characters between the current position and the limit. Two `CharBuffer` instances are equal if they have the same number of (remaining) elements and all these elements are the same. Thus the value of `hashCode()` only depends on the remaining characters. Similarly, comparison is done lexicographically using the remaining characters.

Because hash codes for buffers may change as their elements change, they should not be put into collections based on their hash code (i.e., `HashSet` or `HashMap`) or their natural order (i.e., `TreeSet` or `TreeMap`).

The `length()` and `charAt(int)` methods from `CharSequence` are also defined relative to the remaining characters.

3.8.3 Creating a `CharBuffer`

A `CharBuffer` of a given capacity may be created along with a new backing array using `CharBuffer.allocate(int)`; a buffer may also be created from an existing array using `CharBuffer.wrap(char[])` or from a slice of an existing array. Wrapped buffers are backed by the specified arrays, so changes to them affect the buffer and vice-versa.

3.8.4 Absolute Sets and Gets

Values may be added or retrieved as if it were an array, using `get(int)`, which returns a `char`, and `set(int,char)`, which sets the specified index to the specified `char` value.

3.8.5 Relative Sets and Gets

There are also relative set and get operations, whose behavior is based on the position of the buffer. The `get()` method returns the character at the current position and increments the position. The `put(char)` method sets the character at the current position to the specified value and increments the position.

In addition to single characters, entire `CharSequence` or `char[]` values or slices thereof may be put into the array, with overflow exceptions raised if not all values fit in the buffer. Similarly, `get(char[])` fills the specified character array starting from the current position and increments the position, throwing an exception if there are not enough characters left in the buffer to fill the specified array. The portion of the array to fill may also be specified by slice.

The idiom for bulk copying between arrays is to fill the first array using relative puts, flip it, then copy to another array using relative gets. For instance, to use a `CharBuffer` to concatenate the values in an array, we might use[8]

```
String[] xs = new String[] { "a", "b", "c" };
CharBuffer cb = CharBuffer.allocate(1000);
for (String s : xs)
    cb.put(s);
cb.flip();
String s = cb.toString();
```

After the `put()` operations in the loop, the position is after the last character. If we were to dump to a string at this point, we would get nothing. So we call `flip()`, which sets the position back to zero and the limit to the current position. Then when we call `toString()`, we get the values between zero and the limit, namely all the characters we appended. The call to `toString()` does not modify the buffer.

3.8.6 Thread Safety

Buffers maintain state and are not synchronized for thread safety. Read-write synchronization would be sufficient.

[8]Actually, a `StringBuilder` is better for this job because its size doesn't need to be set in advance like a `CharBuffer`'s.

3.9 `Charset` Class

Conversions between sequences of bytes and sequences of characters are carried out by three related classes in the `java.nio.charset` package. A character encoding is represented by the confusingly named `Charset` class. Encoding characters as bytes is carried out by an instance of `CharsetDecoder` and decoding by an instance of `CharsetEncoder`. All characters are represented as usual in Java with sequences of `char` values representing UTF-16 encodings of Unicode.

Encoders read from character buffers and write to byte buffers and decoders work in the opposite direction. Specifically, they use the `java.nio` classes `CharBuffer` and `ByteBuffer`. The class `CharBuffer` implements `CharSequence`.

3.9.1 Creating a `Charset`

Typically, instances of `Charset` and their corresponding encoders and decoders are specified by name and accessed behind the scenes. Examples include constructing strings from bytes, converting strings to bytes, and constructing `char` streams from `byte` streams.

Java's built-in implementations of `Charset` may be accessed by name with the static factory method `CharSet.forName(String)`, which returns the character encoding with a specified name.

It's also possible to implement subclasses of `Charset` by implementing the abstract `newEncoder()` and `newDecoder()` methods. Decoders and encoders need to define behavior in the face of malformed input and unmappable characters, possibly defining replacement values. So if you must have Morse code, it's possible to support it directly.

3.9.2 Decoding and Encoding with a `Charset`

Once a `Charset` is obtained, its methods `newEncoder()` and `newDecoder()` may be used to get fresh instances of `CharsetEncoder` and `CharsetDecoder` as needed. These classes provide methods for encoding `char` buffers as `byte` buffers and decoding `byte` buffers to `char` buffers. The `Charset` class also provides convenience methods for decoding and encoding characters.

The basic encoding method is `encode(CharBuffer,ByteBuffer,boolean)`, which maps the `byte` values in the `ByteBuffer` to `char` values in the `CharBuffer`. These buffer classes are in `java.nio`. The third argument is a flag indicating whether or not the bytes are all the bytes that will ever be coded. It's also possible to use the `canEncode(char)` or `canEncode(CharSequence)` methods to test for encodability.

The `CharsetEncoder` determines the behavior in the face of unmappable characters. The options are determined by the values of class `CodingErrorAction` (also in `java.nio.charset`).[9]The three actions allowed are to ignore unmappable characters (static constant `IGNORE` in

[9]The error action class is the way type-safe enumerations used to be encoded before `enums` were added diretly to the language, with a private constructor and a set of static constant implementations.

CodingErrorAction), replace them with a specified sequence of bytes (REPLACE), or to report the error, either by raising an exception or through a return value (REPORT). Whether an exception is raised or an exceptional return value provided depends on the calling method. For the replacement option, the bytes used to replace an unmappable character may be set on the CharsetEncoder using the replaceWith(byte[]) method.

The CharsetDecoder class is similar to the encoder class, only moving from a CharBuffer back to a ByteBuffer. The way the decoder handles malformed sequences of bytes is similar to the encoder's handling of unmappable characters, only allowing a sequence of char as a replacement rather than a sequence of byte values.

The CharsetEncoder and CharsetDecoder classes maintain buffered state, so are not thread safe. New instances should be created for each encoding and decoding job, though they may be reused after a call to reset().

3.9.3 Supported Encodings in Java

Each Java installation supports a range of character encodings, with one character encoding serving as the default encoding. Every compliant Java installation supports all of the UTF encodings of Unicode. Other encodings, as specified by Charset implementations, may be added at the installation level, or implemented programatically within a JVM instance.

The supported encodings determine what characters can be used to write Java programs, as well as which character encoding objects are available to convert sequences of bytes to sequences of characters.

We provide a class com.lingpipe.book.char.SupportedEncodings to display the default encoding and the complete range of other encodings supported. The first part of the main pulls out the default encoding using a static method of java.nio.charset.Charset,

```
Charset defaultEncoding = Charset.defaultCharset();
```

The code to pull out the full set of encodings along with their aliases is

```
Map<String,Charset> encodings = Charset.availableCharsets();
for (Charset encoding : encodings.values()) {
     Set<String> aliases = encoding.aliases();
```

The result of running the program is

```
> ant available-charsets

Default Encoding=windows-1252


Big5
     csBig5
...
x-windows-iso2022jp
    windows-iso2022jp
```

The program first writes out the default encoding. This is `windows-1252` on my Windows install. It's typically `ISO-8859-1` on Linux installs in the United States. After that, the program writes all of the encodings, with the official name followed by a list of aliases that are also recognized. We ellided most of the output, because there are dozens of encodings supported.

3.10 International Components for Unicode

The International Components for Unicode (ICU) provide a range of useful open source tools, licensed under the X license (seeSection C.5), for dealing with Unicode and internationalization (I18N). Mark Davis, the co-developer of Unicode and president of the Unicode Consortium, designed most of ICU including the Java implementation, and is still active on its project management committee. The original implementation was from IBM, so the class paths still reflect that for backward compatibility, though it is now managed independently.

The home page for the package is

```
http://icu-project.org/
```

The package has been implemented in both Java and C, with ICU4J being the Java version. There is javadoc, and the user guide section of the site contains extensive tutorial material that goes far beyond what we cover in this section. It is distributed as a Java archive (jar) file, `icu4j-`*`Version`*`.jar`, which we include in the `$BOOK/lib` directory of this book's distribution. There are additional jar files available on the site to deal with even more character encodings and locale implementations.

We are primarily interested in the Unicode utilities for normalization of character sequences to canonical representations and for auto-detecting character encodings from sequences of bytes. ICU also contains a range of locale-specific formatting utilities, time calculations and collation (sorting). It also contains deeper linguistic processing, including boundary detection for words, sentences and paragraphs as well as transliteration between texts in different languages.

3.10.1 Unicode Normalization

ICU implements all the Unicode normalization schemes (see Section 3.1.4). The following code shows how to use NFKC, the normalization form that uses compatibility decomposition, followed by a canonical composition.

```
Normalizer2 normalizer
    = Normalizer2.getInstance(null,"nfkc",Mode.COMPOSE);
String s1 = "\u00E0"; // a with grave
String s2 = "a\u0300"; // a, grave

String n1 = normalizer.normalize(s1);
String n2 = normalizer.normalize(s2);
```

The ICU class `Normalizer2` is defined in the package `com.ibm.icu.text`. There is no public constructor, so we use the rather overloaded and confusing static factory method `getInstance()` to create a normalizer instance. The first argument is `null`, which causes the normalizer to use ICU's definitions of normalization rather than importing definitions through an input stream. The second argument, `"nfkc"` is an unfortunately named string contstant indicating that we'll use NFKC or NFKD normalization. The third argument is the static constant `COMPOSE`, from the static nested class `Normalizer2.Mode`, which says that we want to compose the output, hence producing NFKC. Had we used the constant `DECOMPOSE` for the third argument, we'd get an NFKD normalizer. Finally, it allows case folding after the NFKC normalization by specifying the second argument as `"nfkc_cf"`.

Once we've constructed the normalizer, using it is straightforward. We just call its `normalize()` method, which takes a `CharSequence` argument. We first created two sequences of Unicode characters, U+00E0 (small a with grave accent), and U+0061, U+0300 (small a followed by grave accent). After that, we just print out the `char` values from the strings.

```
> ant normalize-unicode

s1 char values=  e0
s2 char values=  61, 300
n1 char values=  e0
n2 char values=  e0
```

Clearly the initial strings `s1` and `s2` are different. Their normalized forms, `n1` and `n2`, are both the same as `s1`, because we chose the composing normalization scheme that recomposes compound characters to the extent possible.

3.10.2 Encoding Detection

Unfortunately, we are often faced with texts with unknown character encodings. Even when character encodings are specified, say for an XML document, they can be wrong. ICU does a good job at detecting the character encoding from a sequence of bytes, although it only supports a limited range of character encodings, not including the pesky Windows-1252, which is highly confusible with Latin1 and just as confuisble with UTF-8 as Latin1. It's also problematic for Java use, because the default Java install on Windows machines uses Windows-1252 as the default character encoding.

We created a sample class `$BOOK/DetectEncoding`, which begins its `main()` method by extracting (and then displaying) the detectable character encodings,

```
String[] detectableCharsets
    = CharsetDetector.getAllDetectableCharsets();
```

These are the only encodings that will ever show up as return values from detection.

The next part of the program does the actual detection

```
String declared = "ISO-8859-1";
String s = "D\u00E9j\u00E0 vu.";
```

```
String[] encodings = { "UTF-8", "UTF-16", "ISO-8859-1" };
for (String encoding : encodings) {
     byte[] bs = s.getBytes(encoding);
     CharsetDetector detector = new CharsetDetector();
     detector.setDeclaredEncoding(declared);
     detector.setText(bs);
     CharsetMatch[] matches = detector.detectAll();
```

First, it sets a variable that we'll use for the declared encoding, ISO-8859-1, a variable for the string we'll decode, *Déjà vu*, and an array of encodings we'll test. For each of those encodings, we set the byte array `bs` to result of getting the bytes using the current encoding. We then create an instance of `CharsetDetector` (in `com.ibm.icu.text`) using the no-arg constructor, set its declared encoding, and set its text to the byte array. We then generate an array of matches using the `detecAll()` method on the detector. These matches are instances of `CharsetMatch`, in the same pacakge.

The rest of our the code just iterates through the matches and prints them.

```
     for (CharsetMatch match : matches) {
         String name = match.getName();
         int conf = match.getConfidence();
         String lang = match.getLanguage();
         String text = match.getString();
```

For each match, we grab the name name of the character encoding, the integer confidence value, which will range from 0 to 100 with higher numbers being better, and the inferred language, which will be a 3-letter ISO code. We also get the text that is produced by using the specified character encoding to convert the string. These are then printed, but instead of printing the text, we just print whether its equal to our original string `s`.

The result of running the code is

```
> ant detect-encoding
```

```
Detectable Charsets=[UTF-8, UTF-16BE, UTF-16LE, UTF-32BE,
UTF-32LE, Shift_JIS, ISO-2022-JP, ISO-2022-CN, ISO-2022-KR,
GB18030, EUC-JP, EUC-KR, Big5, ISO-8859-1, ISO-8859-2,
ISO-8859-5, ISO-8859-6, ISO-8859-7, ISO-8859-8, windows-1251,
windows-1256, KOI8-R, ISO-8859-9, IBM424_rtl, IBM424_ltr,
IBM420_rtl, IBM420_ltr]

encoding=UTF-8 # matches=3
     guess=UTF-8 conf=80 lang=null
         chars= 44 e9 6a e0 20 76 75 2e
     guess=Big5 conf=10 lang=zh
         chars= 44 77c7 6a fffd 20 76 75 2e
     ...

encoding=UTF-16 # matches=4
```

```
    guess=UTF-16BE conf=100 lang=null
        chars= feff 44 e9 6a e0 20 76 75 2e
    guess=ISO-8859-2 conf=20 lang=cs
        chars= 163 2d9 0 44 0 e9 0 6a 0 155 0 20 0 ...
    ...

encoding=ISO-8859-1 # matches=0
```

Overall, it doesn't do such a good job on short strings like this. The only fully correct answer is for UTF-8, which retrieves the correct sequence of `char` values.

The UTF-16BE encoding is the right guess, but it messes up the conversion to text by concatenating the two byte order marks `FE` and `FF` indicating big-endian into a single `char` with value `0xFEFF`. The detector completely misses the Latin1 encoding, which seems like it should be easy.

The language detection is very weak, being linked to the character encoding. For instance, if the guessed encoding is Big5, the language is going to be Chinese (ISO code "zh").

Confidence is fairly low all around other than UTF-16BE, but this is mainly because we have a short string. In part, the character set detector is operating statistically by weighing how much evidence it has for a decision.

3.10.3 General Transliterations

ICU provides a wide range of transforms over character sequences. For instance, it's possible to transliterate Latin to Greek, translate unicode to hexadecimal, and to remove accents from outputs. Many of the transforms are reversible. Because transforms are mappings from strings to strings, they may be composed. So we can convert Greek to Latin, then then remove the accents, then convert to hexadecimal escapes. It's also possible to implement and load your own transforms.

Using transliteration on strings is as easy as the other interfaces in ICU. Just create the transliterator and apply it to a string.

Built-In Transliterations

We first need to know the available transliterations, the identifiers of which are avaialble programatically. We implement a `main()` method in `ShowTransliterations` as follows

```
Enumeration<String> idEnum = Transliterator.getAvailableIDs();
while (idEnum.hasMoreElements())
    System.out.println(idEnum.nextElement());
```

The ICU `Transliterator` class is imported from the package. `com.ibm.icu.text`. We use its static method `getAvailableIDs()` to produce an enumeration of strings, which we then iterate through and print out.[10] An abbreviated list of outputs (converted to two columns) is

[10]The `Enumeration` interface in the package `java.util` from Java 1.0 has been all but deprecated in favor of the `Iterator` interface in the same package, which adds a remove method and shortens the method names, but is otherwise identical.

```
> ant show-transliterations

Accents-Any                   Hiragana-Katakana
Amharic-Latin/BGN             Hiragana-Latin
Any-Accents                   ...
Any-Publishing                Persian-Latin/BGN
Arabic-Latin                  Pinyin-NumericPinyin
...                           ...
Latin-Cyrillic                Any-Null
Latin-Devanagari              Any-Remove
Latin-Georgian                Any-Hex/Unicode
Latin-Greek                   Any-Hex/Java
...                           ...
Han-Latin                     Any-NFC
Hangul-Latin                  Any-NFD
Hebrew-Latin                  Any-NFKC
Hebrew-Latin/BGN              ...
```

The transliterations are indexed by script, not by language. Thus there is no French or English, just Latin, because that's the script. There is only so much that can be done at the script level, but the transliterations provided by ICU are surprisingly good.

In addition to the transliteration among scripts like Arabic and Hiragana, there are also transforms based on Unicode itself. For instance, the `Any-Remove` transliteration removes combining characters, and thus may be used to convert accented characters to their deaccented forms across all the scripts in Unicode. There are also conversions to hexadecimal representation; for instance `Any-Hex/Java` converts to hexadecimal `char` escapes suitable for inclusion in a Java program. The normalizations are also avaialble through `Any-NFC` and so on.

Transliterating

The second demo class is `Transliterate`, which takes two command line arguments, a string to transliterate and a transliteration scheme, and prints the output as a string and as a list of `char` values. The body of the `main()` method performs the transliteration as follows.

```
String text = args[0];
String scheme = args[1];
Transliterator trans = Transliterator.getInstance(scheme);
String out = trans.transliterate(text);
```

First we collect the text and transliteration scheme from the two command-line arguments. Then, we create a transliterator with the specified scheme using the static factory method `getInstance()`. Finally, we apply the newly created transliterator's `transliterate()` method to the input to produce the output.

We use the Ant target `transliterate` which uses the properties `text` and `scheme` for the text to transliterate and coding scheme respectively.

```
> ant -Dtext="taxi cab" -Dscheme=Latin-Greek transliterate
```

```
Scheme=Latin-Greek
Input=taxi cab
Output=???? ???
Output char values
  3c4  3b1  3be  3b9   20  3ba  3b1  3b2
```

The reason the output looks like `???? ???` is that it was cut and pasted from the Windows DOS shell, which uses the Windows default Java character encoding, Windows-1252, which cannot represent the Greek letters (see Section 5.17 for information on how to change the encoding for the standard output). For convenience, we have thus dumped the hexadecimal representations of the characters. For instance, U+03c4, GREEK SMALL LETTER TAU, is the character τ and U+03b1, GREEK SMALL LETTER ALPHA, is α. The entire string is $\tau\alpha\xi\iota\ \kappa\alpha\beta$, which if you sound it out, is roughly the same as *taxi cab* in Latin.

To see how transliteration can be used for dumping out hex character values, consider

```
> ant -Dtext=abe -Dscheme=Any-Hex/Java transliterate
```

```
Input=abe
Output=\u0061\u0062\u0065
```

The output may be quoted as a string and fed into Java. Translations are also reversible, so we can also do it the other way around,

```
> ant -Dtext=\u0061\u0062\u0065 -Dscheme=Hex/Java-Any transliterate
```

```
Input=\u0061\u0062\u0065
Output=abe
```

Filtering and Composing Transliterations

The scope of transforms may be filtered. For instance, if we write `[aeiou] Upper`, the lower-case ASCII letters are transformed to upper case. For example, using our demo program, we have

```
> ant -Dtext="abe" -Dscheme="[aeiou] Upper" transliterate
```

```
Scheme=[aeiou] Upper
Input=abe
Output=AbE
```

Transforms may also be composed. For any transforms `A` and `B`, the transform `A; B` first applies the transform `A`, then the transform `B`. Filters are especially useful when combined with composition. For example, the transform `NFD; [:Nonspacing Mark:] Remove; NFC` composes three transforms, `NFD`, `[:Nonspacing Mark:] Remove`, and `NFC`. The first transform performs canonical character decomposition (NFD), the second removes the non-spacing marks, such as combining characters, and the third performs a canonical composition (NFC). We can demo this with the string `déjà`, which we encode using Java escapes as `\u0064\u00E9\u006a\u00E0`, by first prefixing the transform with `Hex/Java-Any` to convert the escapes to Unicode. The output is

```
Scheme=Hex/Java-Any; NFD; [:Nonspacing Mark:] Remove; NFC
Input=\u0064\u00E9\u006a\u00E0
Output=deja
```

Chapter 4

Regular Expressions

Generalized regular expressions, as implemented in Java and all the scripting languages like Perl and Python, provide a general means for describing spans of Unicode text. Given a regular expression (regex) and some text, basic regex functionality allows us to test whether a string matches the regex or find non-overlapping substrings of a string matching the regex.

4.1 Matching and Finding

Despite the rich functionality of regexes, Java's `java.util.regex` package contains only two classes, `Pattern` and `Matcher`. An instance of `Pattern` provides an immutable representation of a regular expression. An instance of `Matcher` represents the state of the matching of a regular expression against a string.

4.1.1 Matching Entire Strings

It's easiest to start with an example of using a regular expression for matching, which we wrote as a `main()` method in the class `RegexMatch`, the work of which is done by

```
String regex = args[0];
String text = args[1];

Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(text);
boolean matches = matcher.matches();
```

First, we read the regex from the first command-line argument, then the text from the second argument. We then use the regular expression to compile a pattern, using the static factory method `pattern.compile()`. This pattern is reusable. We next create a matcher instance, using the method `matcher()` on the pattern we just created. Finally, we assign a boolean variable `matches` the value of calling the method `matches()` on the matcher. And then we print out the result.

Regular expressions may consist of strings, in which case they simply carry out exact string matching. For example, the regex `aab` does not match the string *aabb*. There is an Ant target `regex-match` which feeds the command-line arguments to our program. For the example at hand, we have

```
> ant -Dregex="aab" -Dtext="aabb" regex-match

Regex=|aab|    Text=|aabb|    Matches=false
```

On the other hand, the regex `abc` does match the string *abc*.

```
> ant -Dregex="abc" -Dtext="abc" regex-match

Regex=|abc|    Text=|abc|    Matches=true
```

Note that we have used the vertical bar to mark the boundaries of the regular expression and text in our output. These vertical bars are not part of the regular expression or the text. This is a useful trick in situations where space may appear as the prefix or suffix of a string. It may get confusing if there is a vertical bar within the string, but the outer vertical bars are always the ones dropped.

4.1.2 Finding Matching Substrings

The second main application of regular expressions is to find substrings of a string that match the regular expression. The main method in our class `RegexFind` illustrates this. We read in two command-line aguments into string variables `regex` and `text` as we did for `RegexMatch` in the previous section. We begin by compling the pattern and creating a matcher for the text just as in `RegexFind`.

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(text);
while (matcher.find()) {
     String found = matcher.group();
     int start = matcher.start();
     int end = matcher.end();
```

The first call of the `find()` method on a matcher returns `true` if there is a substring of the text that matches the pattern. If `find()` returned `true`, then the are method `group()` returns the substring that matched, and the methods `start()` and `end()` return the span of the match, which is from the start (inclusive) to the end (exclusive).

Subsequent calls to `find()` return `true` if there is a match starting on or after the end position of the previous calls. Thus the loop structure in the program above is the standard idiom for enumerating all the matches.

As an example, we use the Ant target `regex-find`, which takes the same arguments as `regex-match`.

```
> ant -Dregex=aa -Dtext=aaaaab regex-find

Found |aa| at (0,2)    Found |aa| at (2,4)
```

As before, the vertical bars are delimeters, not part of the matching substring. The string *aa* actually shows up in four distinct locations in *aaaaab*, at spans (0,2), (1,3), (2,4), and (3,5). Running find only returns two of them. The matcher works from the start to the end of the string, returning the first match it finds after the first call to `find()`. In this case, that's the substring of *aaaaab* spanning (0,2). The second call to `find()` starts looking for a match at position 2, succeeding with the span (2,4). Next, it starts looking at position 4, but there is no substring starting on or after position 4 that matches *aa*.

4.2 Character Regexes

The most basic regular expressions describe single characters. Some characters have special meanings in regexes and thus need to be escaped for use in regular expresions.

4.2.1 Characters as Regexes

A single character may be used a regular expression. A regular expression consisting of a single character only matches that single character. For instance, `Pattern.compile("a").matcher("a").matches()` would evaluate to `true`. Using our Ant match target,

```
> ant -Dregex=b -Dtext=b regex-match
```

```
Regex=|b|    Text=|b|    Matches=true
```

```
> ant -Dregex=b -Dtext=c regex-match
```

```
Regex=|b|    Text=|c|    Matches=false
```

Within a Java program, Java character literals may be used within the string denoting the regex. For instance, we may write `Pattern.compile("\u00E9")` to compile the regular expression that matches the character U+00E9, LATIN SMALL E WITH ACUTE, which would be written as *é*.

4.2.2 Unicode Escapes

Arbitrary unicode characters may be escaped in the same way as in Java. That is, `\u`*hhhh* is a regular expression matching the character with Unicode code point U+*hhhh*. For example,

```
> ant -Dregex=\u0041 -Dtext=A regex-match
```

```
Regex=|\u002E|    Text=|.|    Matches=true
```

Note that we did not have to escape the backslash for the shell because the following character, `u`, is not an escape. Unicode escapes are useful for for matching the period character, U+002E, FULL STOP, which is a reserved character in regexes representing wildcards (see Section 4.3.1).

```
> ant -Dregex="\u002E" -Dtext="." regex-match

Regex=|\u002E|    Text=|.|    Matches=true
```

In a Java program, we'd still have to write `Pattern.compile("\\u002E")`.

Unicode escapes in regexes are confusing, because they use the same syntax as Java unicode literals. Further complicating matters, the literal for backslashes in Java programs is itself an escape sequence, \\. There is a world of difference between the patterns produced by `Pattern.compile("\u002E")` and `Pattern.compile("\\u002E")`. The former is the same as `Patten.compile(".")`, and matches any character, whereas the latter uses a regex escape, and only matches the period character.

4.2.3 Other Escapes

There are also other built-in escapes, such as `\n` for newline and \\ for a backslash character. Again, we have to be careful about distinguishing Java string escapes, which are used to create a regex, which itself may contain regex escapes.

To confuse matters, Java itself also uses a backslash for escapes, so that the character for backslash is written `'\\'` and so the regex `\n` must be written as the string literal `"\\n"`. To create the regex \\, we need to use two backslash escapes in our string literal, writing `Pattern.compile("\\\\")`.

This is all further confused by the command-line shell, which has its own escape sequences. Our sample program reads a command from the command line, which is being supplied by Ant and set as a property on the command-line call to Ant. For instance, consider this puzzler,

```
> ant -Dregex="\\\\" -Dtext="\\" regex-match

Regex=|\\|    Text=|\|    Matches=true
```

On the command line, backslash is escaped as \\.[1] Our Java program gets a value for `regex` consisting of a length-two string made up of two backslashes, and value for text consisting of a single backslash. The match succeeds because the regex \\ is the escaped backslash character, which matches a single backslash.

4.3 Character Classes

Regular expressions provide a range of built-in character classes based on ASCII or Unicode, as well as the ability to define new classes. Each character class matches a set of characters. The reason to use classes rather than disjunctions is that they have

[1]Usually backslash doesn't need to be escaped because the following character isn't a valid escape; here, the following character is a quote ("), and \" is a valid escape sequence for the shell.

4.3.1 WildCards

A singe period (`.`) is a regular expression that matches any single characer. We may think of it as the universal character class. For instance, the regex `.` matches the string *a*, and `a.c` matches *abc*.

```
> ant -Dregex=a.c -Dtext=abc regex-match
Regex=|a.c|     Text=|abc|     Matches=true
```

Whether or not a wildcard matches an end-of-line sequence depends on whether or not the `DOTALL` flag is set (see Section 4.12); if it is set, the wildcard matches end-of-line sequences. The following two expressions evaluate to `true`,

```
Pattern.compile(".").matcher("A").matches();
Pattern.compile(".",DOTALL).matcher("\n").matches();
```

whereas the following expression evaluates to `false`,

```
Pattern.compile(".").matcher("\n").matches();
```

4.3.2 Unicode Classes

Unicode defines a range of categories for characters, such as the category `Lu` of uppercase letters (see Section 3.1.4 for more information). The regular expression `\p{`*X*`}` matches any unicode character belonging to the Unicode category *X*. For example, the class `Lu` picks out uppercase letters, so we have

```
> ant -Dregex=\p{Lu} -Dtext=A regex-match
Regex=|\p{Lu}|     Text=|A|     Matches=true
```

Using a capital letter complements the category; for instance, `\p{Lu}` matches uppercase letters, whereas `\P{Lu}` matches any character except an uppercase letter.

4.3.3 ASCII Classes

There are character classes built in for ASCII characters. For instance, `\d` is for digits, `\s` for whitespace, and `\w` for alphanumeric characters. For example,

```
> ant -Dregex=\d -Dtext=7 regex-match
Regex=|\d|     Text=|7|     Matches=true
```

For these three ASCII classes, the capitalized forms match ASCII characters that don't match the lowercase forms. So `\D` matches non-digits, `\S` non whitespace, and `\W` non-alphanumeric characters.

There are a range of built-in ASCII classes from the POSIX standard built in. They use the same syntax as the Unicode classes described in the previous section. For example, `\p{ASCII}` matches any ASCII character and `\p{Punct}` for ASCII punctuation characters.

The ASCII characters must be used with care, because they will not have their described behavior when interpreted over all of Unicode. For example, there are whitespaces in Unicode that don't match `\s` and digits that don't match `\d`.

4.3.4 Compound Character Classes

Character classes may be built up from single characters and/or other character classes using set operations like union, intersection, and negation.

The syntax for compound character classes uses brackets around a character class expression. The atomic character class expressions are single characters like `a`, and the character class escapes like `\p{Lu}` matches any uppercase Unicode letter character.

Character classes may be unioned, which confusingly uses a concatenation syntax. For instance, `[aeiou]` is the character class picking out the ASCII vowels. It is composed of the union of character class expressions `a`, `e`, ..., `u`.

```
> ant -Dregex=[aeiou] -Dtext=i regex-match

Regex=|[aeiou]|    Text=|i|    Matches=true
```

Class escapes may be used, so that `[\p{Lu}\p{N}]` picks out upper case letters and numbers.

The use of range notation may be used as shorthand for unions. For instance, `[0-9]` picks out the ASCII digits *0*, *1*, ..., *9*. For example,

```
> ant -Dregex="[I-P]" -Dtext=J regex-match

Regex=|[I-P]|    Text=|J|    Matches=true
```

Character classes may be complemented. The syntax involves a caret, with `[ˆ`*A*`]` picking out the class of characters not in class *A*. The expression *A* must be either a range or a sequence of character class primitives. For instance, `[ˆz]` represents the class of every character other than *z*.

```
> ant -Dregex="[ˆz]" -Dtext=z regex-match

Regex=|[^z]|    Text=|z|    Matches=false

> ant -Dregex="[ˆz]" -Dtext=a regex-match

Regex=|[^z]|    Text=|a|    Matches=true
```

Unlike the usual rules governing the logical operations disjunction, conjunction, and negation, the character class complementation operator binds more weakly than union. Thus `[ˆaeiou]` picks out every character that is not an ASCII vowel.

Character classes may also be intersected. The syntax is the same as for logical conjunction, with `[`*A*`&&`*B*`]` picking out the characters that are in the classes denoted by both *A* and *B*. For instance, we could write `[[a-z]&&[ˆaeiou]]` to pick out all lowercase ASCII characters other than the vowels,

```
> ant -Dregex="[[a-z]&&[ˆaeiou]]" -Dtext=i regex-match

Regex=|[[a-z]&&[^aeiou]]|    Text=|i|    Matches=false
```

As described in the Javadoc for `Pattern`, the order of attachment for character class operations is

Order	*Expression*	*Example*
1	Literal Escape	`\x`
2	Grouping	`[...]`
3	Range	`0-9`
4	Union	`ab`, `[0-9][ab]`,
5	Intersection	`[[a-z]&&[^aeiou]]`

Thus the square brackets act like grouping parentheses for character classes. Complementaton must be treated separately. Complementation only applies to character groups or ranges, and must be closed within brackets before being combined with union or intersection or the scope may not be what you expect. For instance, `[^a[b-c]]` is not equivalent to `[^abc]`, but rather is equivalent to `[[^a][b-c]]`.

4.4 Concatenation

As we saw in our early examples, characters may be concatenated together to form a regular expression that matches the string. This is not the same operation as character classes, which confusingly uses the same notation.

4.4.1 Empty Regex

With a list of characters (or strings) being concatenated, the boundary case is the concatenation of an empty list, which is why we start this section with a discussion of the empty regex.

The simplest regular expression is the empty regular expression. It only matches the empty string. For example, the expression `Pattern.compile("").matcher("").matches()` evaluates to `true`. Using our test code from the command-line proves tricky, because we have to escape the quotes if we want to assign environment variables, as in

```
> ant -Dregex=\"\" -Dtext=\"\" regex-match
```

```
Regex=||    Text=||    Matches=true
```

```
> ant -Dregex=\"\" -Dtext=a regex-match
```

```
Regex=||    Text=|a|    Matches=false
```

4.4.2 Concatenation

Two regular expressions *x* and *y* may be concatenated to produce a compound regular expression *xy*. The regex *xy* matches any string that can be decomposed into a string that matches *x* followed by a string that matches *y*. For example, if we have a regex `[aeiou]` that matches vowels and a regex `0-9` that matches digits, we can put them together to produce a regex that matches a vowel followed by a digit.

```
> ant -Dregex=[aeiou][0-9] -Dtext=i7 regex-match
```

```
Regex=|[aeiou][0-9]|    Text=|i7|    Matches=true
```

Concatenation is associative, meaning that if we have three regexes *x*, *y*, and *z*, the regexes (*xy*)*z* and *x*(*yz*) match exactly the same set of strings. Because parentheses may be used for groups (see Section 4.10), the two regexes do not always behave exactly the same way.

4.5 Disjunction

If *X* and *Y* are regular expressions, the regular expression *x*|*y* matches any string that matches either *X* or *Y* (or both). For example,

```
> ant -Dregex="ab|cd" -Dtext=ab regex-match
```

```
Regex=|ab|cd|
Text=|ab|
Matches=true
```

```
> ant -Dregex="ab|cd" -Dtext=cd regex-match
```

```
Regex=|ab|cd|    Text=|cd|    Matches=true
```

```
> ant -Dregex="ab|cd" -Dtext=bc regex-match
```

```
Regex=|ab|cd|    Text=|bc|    Matches=false
```

Concatenation takes precedence over disjunction, so that ab|cd is read as (ab)|(cd), not a(b|c)d.

The order of disjunctions doesn't matter for matching (other than for efficiency), but it matters for the find operation. For finds, disjunctions are evaluated left-to-right and the first match returned.

```
> ant -Dregex="ab|abc" -Dtext=abc regex-find
```

```
Found |ab| at (0,2)
```

```
> ant -Dregex="abc|ab" -Dtext=abc regex-find
```

```
Found |abc| at (0,3)
```

Because complex disjunctions are executed through backtracking, disjunctions that have to explore many branches can quickly lead to inefficient regular expressions. Even a regex as short as (a|b)(c|d)(e|f) has $2^3 = 8$ possible search paths to explore. Each disjunction doubles the size of the search space, leading to exponential growth in the search space in the number of interacting disjunctions. In practice, disjunctions often fail earlier. For instance, if there is no match for a|b, further disjunctions are not explored.

A simple pathological case can be created by concatenating instances of the regex (|a) and then matching a long string of a characters. You can try it out yourself with

```
> ant -Dregex="(|a)(|a)...(|a)(|a)" -Dtext=aa...aa regex-find
```

where the elipses are meant to indicate further copies of a. This expression will explore every branch of every disjunction. Around 25 or 26 copies of (|a) there are tens of millions of paths to explore, and with 30, there's over a billion. You'll be waiting a while for an answer. But if the building block is (a|b), there's no problem, because each disjunction has to match a.

```
> ant -Dregex="(b|a)(b|a)...(b|a)(b|a)" -Dtext=aa...aa regex-find
```

4.6 Greedy Quantifiers

There are operators for optionality or multiple instances that go under the general heading of "greedy" because of they match as many times as they can. In the next section, we consider their "reluctant" counterparts that do the opposite.

4.6.1 Optionality

The simplest greedy quantifier is the optionality marker. The regex *A*? matches a string that matches *A* or the empty string. For example,

```
> ant -Dregex=a?  -Dtext=a regex-match

Regex=|a?|    Text=|a|    Matches=true

> ant -Dregex=a?  -Dtext=\"\" regex-match

Regex=|a?|    Text=||    Matches=true
```

The greediness of the basic optionality marker, which tries to match before trying to match the empty string, is illustrated using find(),

```
> ant -Dregex=a?  -Dtext=aa regex-find

Found |a| at (0,1)    Found |a| at (1,2)    Found || at (2,2)
```

The first two substrings found match a rather than the empty string. Only when we are at position 2, at the end of the string, do we match the empty string because we can no longer match a.

The greediness of optionality means that the *A*? behaves like the disjunction (*A*|) of *A* and the empty string. Because it is the first disjunct, a match against *A* is tried first.

4.6.2 Kleene Star

A regular expression *x* may have the Kleene-star operator[2] applied to it to produce the regular expression *A**. The regex *A** matches a string if the string is composed of a sequence of zero or more matches of *A*. For example, [01]* matches any sequence composed only of 0 and 1 characters,

[2]Named after Stephen Kleene, who invented regular expressionsn as a notation for his characterization of regular languages.

```
> ant -Dregex="[01]*" -Dtext=00100100 regex-match

Regex=|[01]*|    Text=|00100100|    Matches=true
```

It will also match the empty string, but will not match a string with any other character.

Using find, the standard Kleene-star operation is greedy in that it consumes as much as it can during a `find()` operation. For example, consider

```
> ant -Dregex="[01]*" -Dtext=001a0100 regex-find

Found |001| at (0,3)    Found || at (3,3)    Found |0100| at (4,8)
Found || at (8,8)
```

The first result of calling `find()` consumes the expression `[01]` as many times as possible, consuming the first three characters, `001`. After that, the expression matches the empty string spanning from 3 to 3. It then starts at position 4, finding `0100` and then the empty string again after that. Luckily, `find()` is implemented cleverly enough that it only returns the empty string once.

Kleene star may also be understood in terms of disjunction. The greedy Kleene star regex *A** behaves like the disjunction of *A*(*A**) and the empty regex. Thus it first tries to match *A* followed by another match of *A**, and only failing that tries to match the empty string. For instance, the two regexes in the following patterns match exactly the same strings.

```
Pattern p1 = Pattern.compile("(xyz)*");
Pattern p2 = Pattern.compile("(xyz(xyz)*)|");
```

Kleene star interacts with disjunction in the expected way. For example, matching consumes an entire string, as with

```
> ant -Dregex="(ab|abc)*" -Dtext=abcab regex-match

Regex=|(ab|abc)*|    Text=|abcab|    Matches=true
```

while finding returns only an initial match,

```
> ant -Dregex="(ab|abc)*" -Dtext=abcab regex-find

Found |ab| at (0,2)    Found || at (2,2)    Found |ab| at (3,5)
Found || at (5,5)
```

This is a result of the greediness of the Kleene-star operator and the evaluation order of disjunctions. If we reorder, we get the whole string,

```
> ant -Dregex="(abc|ab)*" -Dtext=abcab regex-find

Found |abcab| at (0,5)    Found || at (5,5)
```

because we first match `abc` against the disjunct, then continue trying to match `(abc|ab)*`, which matches `ab`.

4.6.3 Kleene Plus

The Kleene-plus operator is like the Kleene star, but requires at least one match. Thus *A*+ matches a string if the string is composed of a sequence of one or more matches of *A*. The Kleene-plus operator may be defined in terms of Kleene star and concatenation, with *A*+ behaving just like *AA*∗.

Use Kleene plus instead of star to remove those pesky empty string matches. For instance,

```
> ant -Dregex="(abc|ab)+" -Dtext=abcab regex-find
Found |abcab| at (0,5)
```

4.6.4 Match Count Ranges

We can specify an exact number or a range of possible number of matches. The regular expression *A*{*m*} matches any string that that may be decomposed into a sequence of *m* strings, each of which matches *A*.

```
> ant -Dregex="a{3}" -Dtext=aaa regex-match
Regex=|a{3}|    Text=|aaa|    Matches=true
```

There is similar notation for spans of counts. The regex *A*{*m*,*n*} matches any string that may be decomposed into a sequence of between *m* (inclusive) and *n* (inclusive) matches of *A*. The greediness comes in because it prefers to match as many instances as possible. For example,

```
> ant -Dregex="a{2,3}" -Dtext=aaaaa regex-find
Found |aaa| at (0,3)    Found |aa| at (3,5)
```

The first result found matches the regex a three times rather than twice.

There is also an open-ended variant. The regex *A*{*m*,} matches any string that can be decomposed into *m* or more strings each of which matches *A*.

4.7 Reluctant Quantifiers

There are a range of reluctant quantifiers that parallel the more typical greedy quantifiers. For matching, the relunctant quantifiers behave just like their greedy counterparts, whereas for find operations, reluctant quantifiers try to match as little as possible.

The reluctant quantifiers are written like the greedy quantifiers followed by a question mark. For instance, *A*∗? is the reluctant version of the Kleene star regex *A*∗, *A*{m,n}? is the reluctant variant of the range egex *A*{m,n}, and *A*?? is the reluctant variant of the optionality regex *A*?.

Reluctant quantifiers may be understood by reversing all the disjuncts in the definitions of the greedy quantifier equivalents. For instance, we can think of *A*?? as *(|A)* and *A*∗? as (|*AA*∗?).

Here we repeat some key examples of the previous section using reluctant quantifiers.

```
> ant -Dregex=a??  -Dtext=aa regex-find
Found || at (0,0)    Found || at (1,1)    Found || at (2,2)
```

```
> ant -Dregex="(abc|ab)+?" -Dtext=abcab regex-find
Found |abc| at (0,3)    Found |ab| at (3,5)
```

```
> ant -Dregex="a{2,3}?" -Dtext=aaaaa regex-find
Found |aa| at (0,2)    Found |aa| at (2,4)
```

4.8 Possessive Quantifiers

The third class of quantified expression is the possessive quantifier. Possessive quantifiers will not match a fewer number of instances of their pattern if they can match more. Procedurally, they commit to a match once they've made it and do not allow backtracking.

The syntax for a possessive quantifier follows the quantifier symbol with a plus sign (+); for instance, ?+ is possessive optionality and *+ is greedy Kleene star.

Consider the difference between a greedy Kleene star

```
> ant "-Dregex=[0-9]*1" -Dtext=1231 regex-match
Regex=|[0-9]*1|    Text=|1231|    Matches=true
```

which matches, and a possessive Kleene star,

```
> ant "-Dregex=[0-9]*+1" -Dtext=1231 regex-match
Regex=|[0-9]*+1|    Text=|1231|    Matches=false
```

which does not match. Even though `[0-9]*` indicates a greedy match, it is able to back off and only match the first three characters of the input, `123`. The possessive quantifier, once it has matched `1231`, does not let it go, and the overall match fails.

4.8.1 Independent Group Possessive Marker

If *A* is a regex, then `(?>`*A*`)` is a regex that matches the same thing as *A*, but groups the first match possessively. That is, once `(?>`*A*`)` finds a match for *A*, it does not allow backtracking. We can see this with a pair of examples.

```
> ant -Dregex="(?>x+)xy" -Dtext=xxy regex-match
Regex=|(?>x+)xy|    Text=|xxy|    Matches=false
```

This doesn't match, because the `x+` matches both instances of *x* in the text being matched and doesn't allow backtracking. Contrast this with

```
> ant -Dregex="(?>x+)y" -Dtext=xxy regex-match
Regex=|(?>x+)y|    Text=|xxy|    Matches=true
```

The parentheses in this construction do not count for grouping (see Section 4.10). There is a similar construction `(?:`*A*`)` which is not possessive, and also does not count the parentheses for grouping.

4.9 Non-Capturing Regexes

Some regexes match virtual positions in an input without consuming any text themselves.

4.9.1 Boundary Matchers

Boundary matchers match specific sequences, but do not consume any input themselves.

Begin- and End-of-Line Matchers

The caret (ˆ) is a regular expression that matches the end of a line and the dollar sign ($) a regular expression matching the beginning of a line. For example,

```
> ant -Dregex="ˆabc$" -Dtext=abc regex-match
```

```
Regex=|^abc$|    Text=|abc|    Matches=true
```

```
> ant -Dregex=ab$c -Dtext=abc regex-match
```

```
Regex=|ab$c|    Text=|abc|    Matches=false
```

The begin- and end-of-line regexes do not themselves consume any characters.

These begin-of-line and end-of-line regexes do not match new lines inside of strings unless the MULTILINE flag is set (see Section 4.12). By default, these regexes match new-line sequences for any platform (Unix, Macintosh, or Windows) and many more (see Section 4.9.4 for the full set of line terminators). If the UNIX_LINES flag is set, they only match Unix newlines (see Section 4.12).

Begin and End of Input

Because begin-of-line and end-of-line have variable behavior, there are regexes \A for the beginning of the input and \z for the end of the input. These will match the begin and end of the input no matter what the match mode is. For instance,

```
> ant -Dregex=\A(mnp)\z -Dtext=mnp regex-group
```

```
regex=|\A(mnp)\z|    text=|mnp|
Group  0=|mnp| at (0,3)    Group  1=|mnp| at (0,3)
```

Word Boundaries

The regex \b matches a word boundary and \B a non-word boundary, both without consuming any input. Word boundaries are the usual punctuation and spaces and the begin and end of input. For example,

```
> ant -Dregex=\b\w+\b -Dtext="John ran quickly." regex-find
```

```
Found |John| at (0,4)        Found |ran| at (5,8)
Found |quickly| at (9,16)
```

4.9.2 Positive and Negative Lookahead

Often you want to match something only if it occurs in a given context. If *A* is a regex, then `(?=A)` is the positive lookahead and `(?!A)` the negative lookahead regular expression. These match a context that either matches or doesn't match the specified regular expression. They must be used following other regular expressions. For instance, to find the instances of numbers preceding periods, we could use

```
> ant -Dregex="\d+(?=\.)" -Dtext=12.2 regex-find
```

```
Found |12| at (0,2)
```

and to find those not preceding periods,

```
> ant -Dregex="\d+(?!\.)" -Dtext=12.2 regex-find
```

```
Found |1| at (0,1)    Found |2| at (3,4)
```

In a real application, we would probably exclude numbers and periods in the negative lookahead, with a regex such as `\d+(?![\.\d])`.

One reason the lookahead is nice is that further finds can match the material that the lookahead matched.

4.9.3 Positive and Negative Lookbehind

There are backward-facing variants of the positive and negative lookahead constructs, `(?<=X)` and `(?<!X)`. These are more limited in that the regular expression *X* must expand to a simple sequence, and thus may not contain the Kleene-star or Kleene-plus operations.

For example, the following example finds sequences of digits that do not follow periods or other digits

```
> ant -Dregex="(?<![\.\d])\d+" -Dtext=12.32 regex-find
```

```
Found |12| at (0,2)
```

4.9.4 Line Terminators and Unix Lines

Code Point(s)	*Description*	*Java*
U+000A	LINE FEED	`\n`
U+000D	CARRIAGE RETURN	`\r`
U+000D, U+000A	CARRIAGE RETURN, LINE FEED	`\r\n`
U+0085	NEXT LINE	`\u0085`
U+2028	LINE SEPARATOR	`\u2028`
U+2029	PARAGRAPH SEPARATOR	`\u2029`

4.10 Parentheses for Grouping

Parentheses play the usual role of grouping in regular expressions, and may be used to disambiguate what would otherwise be misparsed. For instance, the expression `(a|b)c` is very different than `a|(bc)`.

Parentheses also play the role of identifying the subexpressions they wrap.

If *A* is a regular expression, (*A*) is a regular expression that matches exactly the same strings as *A*. Note that parentheses come in matched pairs, and each picks out the unique subexpression *A* it wraps.

The pairs of parentheses in a regular expression are numbered from left to right, beginning with 1. For example, we've written a very simple regex with the identifier of each parentheses group on the next line

```
 a(b(cd)(e))
0 1 2   3
```

There is an implicit pair of parentheses around the entire pattern, with group number 0. Group 1 is `b(cd)(e)`, group 2 is `cd`, and group 3 is `e`.

The identities of the groups are useful for pulling out substring matches (this section), using back references in regexes (see Section 4.11) and replace operations based on regexes (see Section 4.14).

After a match or a find operation, the substring of the match that matched each group is available from the matcher with a method `group(int)`, where the argument indicates which group's match is returned. The start and end position of a group may also be retrieved, using `start(int)` and `end(int)`. The total number of groups in the regex is available through the method `groupCount()`. The methods `group()`, `start()`, and `end()` we used earlier to retrieve the match of the entire string are just shorthand for `group(0)`, `start(0)`, and `end(0)`. If there hasn't been a match or the last find did not succeed, attempting to retrieve groups or positions raises an illegal state exception.

We provide a simple class `RegexGroup` to illustrate the workings of group extraction. The work is done in the `main()` method by the code

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(text);
if (matcher.matches()) {
     for (int i = 0; i <= matcher.groupCount(); ++i) {
          String group = matcher.group(i);
          int start = matcher.start(i);
          int end = matcher.end(i);
```

Here we call `matcher.match()`, which requires a match of the entire text.

We set up the Ant target `regex-group` to call it, and test it in the usual way, using the example whose groups are numbered above.

```
> ant -Dregex=a(b(cd)(e)) -Dtext=abcde regex-group

regex=|a(b(cd)(e))|    text=|abcde|
Group  0=|abcde| at (0,5)    Group  1=|bcde| at (1,5)
Group  2=|cd| at (2,4)       Group  3=|e| at (4,5)
```

A more typical linguistic example would be to pull out names after the word *Mr*, as in

```
> ant -Dregex=Mr((\s(\p{Lu}\p{L}*))+) -Dtext="Mr J Smith"
regex-group
```

```
Group  0=|Mr J Smith| at (0,10)    Group  1=| J Smith| at (2,10)
Group  2=| Smith| at (4,10)        Group  3=|Smith| at (5,10)
```

Group 1 corresponds to the name, but note that we have an extra leading space. We can get rid of that with a more complex regex, and we could also allow optional periods after *Mr* with the appropriate escape, and also deal with periods in names. Matching complicated patterns with regexes is a tricky business.

4.10.1 Non-Capturing Group

It is possible to use parentheses without capturing. If *X* is a regex, then `(?:`*X*`)` matches the same things as *X*, but the parentheses are not treated as capturing.

4.11 Back References

The regex `\`*n* is a back reference to the match of the *n*-th capturing group before it. We can use matching references to find pairs of duplicate words in text. For instance, consider

```
> ant -Dregex="(\d+).*(\1)" -Dtext="12 over 12" regex-group
```

```
Group  0=|12 over 12| at (0,10)    Group  1=|12| at (0,2)
Group  2=|12| at (8,10)
```

The regex \1 will match whatever matched group 1. Group 1 is `(\d+)` and group 2 is `(\1)`; both match the expression *12*, only at different places. You can see that the text *12 over 13* would not have matched.

If the match index is out of bounds for the number of groups, matching fails without raising an exception.

4.12 Pattern Match Flags

There are a set of flags, all represented as static integer constants in `Pattern`, which may be supplied to the pattern compiler factory method `Pattern.compile(String,int)` to control the matching behavior of the compiled pattern.

Constant	*Value*	*Description*
`DOTALL`	32	Allows the wild card (`.`) expression to match any character, including line terminators. If this mode is not set, the wild card does not match line terminators.
`MULTLINE`	8	Allows the begin-of-line (ˆ) and end-of-line ($) to match internal line terminators. If this is not set, they only match before the first and after the last character. The full Java notion of newline is used unless the flag `UNIX_LINES` is also set.
`UNIX_LINES`	1	Treat newlines as in Unix, with a single U+000A, LINE FEED, Java string escape `\n`, representing a new line. This affects the behavior of wildcard (`.`) and begin-of-line (ˆ) and end-of-line ($) expressions.
`CASE_INSENSITIVE`	2	Matching is insensitive to case. Case folding is only for the ASCII charset (U+0000 to U+007F) unless the flag `UNICODE_CASE` is also set, which uses the Unicode definitions of case folding.
`UNICODE_CASE`	64	If this flag and the `CASE_INSENSITIVE` flag is set, matching ignores case distinctions as defined by Unicode case folding rules.
`CANON_EQ`	128	Matches are based on canonical equivalence in Unicode (see Section 3.1.4).
`LITERAL`	16	Treats the pattern as a literal (i.e., no parsing).
`COMMENTS`	4	Allows comments in patterns and causes pattern-internal whitespace to be ignored.

The flags are defined using a bit-mask pattern whereby multiple flags are set by taking their bitwise-or value.[3] For instance, the expression `DOTALL | MULTILINE` allows the wildcard to match anything and the line-terminator expressions to match internal newlines. More than two flags may be set this way, so we could further restrict to Unix newlines with `DOTALL | MULTILINE | UNIX_LINES`.

Even stronger matching than canonical equivalence with unicode case folding can be achieved by writinger the regular expression using compatibility normal forms and using the ICU package to perform a compatibility decomposition on the input (see Section 3.1.4).

[3]Although bitwise or looks like addition for flags like these with only one bit on, it breaks down when you try to set flags twice. Thus `DOTALL + MULTILINE` evaluates to the same thing as `DOTALL | MULTINE`, `DOTALL + DOTALL` evaluates to the same thing as `UNICODE_CASE`, whereas `DOTALL | DOTALL` evaluates to the same thing as `DOTALL`.

4.13 Pattern Construction Exceptions

Attempting to compile a regular expression with a syntax error raises a runtime exception, `PatternSyntaxException` from the `java.util.regex` package. The parser tries to provide a helpful warning message. For instance, here's what we see if we inadvertently add a right parenthesis to our regex.

```
> ant -Dregex=aa) -Dtext=aaab regex-find
```

```
Exception in thread "main"
    java.util.regex.PatternSyntaxException:
    Unmatched closing ')' near index 1
```

4.14 Find-and-Replace Operations

A common usage pattern for regular expressions is to find instances of one pattern and replace them with another. We provide a sample program `RegexReplace`, which illustrates replacement in its `main()` method,

```
Pattern pattern = Pattern.compile(regex);
Matcher matcher = pattern.matcher(text);
String result = matcher.replaceAll(replacement);
```

The pattern and matcher are created as usual. Then, we create a string by replacing all the matches of the pattern in the text provided to the matcher with the specified replacement. We could also use `replaceFirst()` to return the result of only replacing the first match.

With the Ant target `regex-replace`, we can see this in action.

```
>      ant -Dregex=\d+ -Dreplacement=0 -Dtext="+1 (718) 290-9170"
regex-replace
```

```
regex=|\d+|    replacement=|0|
text=|+1 (718) 290-9170|    result=|+0 (0) 0-0|
```

We've replaced every sequence of digits with a zero.

A nice feature of replace operations is that they may reference pieces of the match by group number and use them in the replacement. An occurrence of `$`*n* in the replacement string represents whatever matched group *n* in the match.

For instance, we can extract the version number from a jar and redisplay it using

```
>    ant -Dregex=(\w+)\-([\d\.]+)\.jar -Dreplacement="$1.jar v$2"
-Dtext="foo-7.6.jar" regex-replace
```

```
regex=|(\w+)\-([\d\.]+)\.jar|    replacement=|$1.jar v$2|
text=|foo-7.6.jar|               result=|foo.jar v7.6|
```

Note that we had to escape the literal hyphen and period characters in the regex. The class `\w` is for word characters, which are alphanumerics and the underscore. The example regexes are very fragile, and it is quite difficult to write more robust regexes.

4.15 String Regex Utilities

There are many utility methods in the `String` class that are based on regular expressions.

The method `matches(String)` returns `true` if the string matches the regex specified by the argument. Given a regex `r`, `matches(r)` is just shorthand for `Pattern.compile(r).matcher(this).matches()`.

The method `replaceAll(String,String)`, which takes a string representing a regex and a replacement string, returns the result of replacing all the substrings in the string matching the regex with the replacement string. Given a regex `r` and replacement `s`, `replaceAll(r,s)` is just shorthand for `Pattern.compile(r).matcher(this).replaceAll(s)`. As such, the replacement may contain references to match groups. There is also a corresponding `replaceFirst()` method.

One of the more useful utilities is `split(String)`, which splits a string around the matches of a regular expression, returning an array as a result. For instance, `"foo12bar177baz.split(\d+)` splits on digits, returning the string array `{ "foo", "bar", "baz" }`. Note that the matching numbers are removed.

4.16 Thread Safety, Serialization, and Reuse

4.16.1 Thread Safety

Because instances of `Pattern` are immutable, it is safe to use them concurrently in multiple threads. Because instances of `Matcher` contain the state of the current match, it is not thread safe to use matchers concurrently without synchronization.

4.16.2 Reusing Matchers

Matchers may be reused. The method `reset()` may be called on a matcher to reset it to behave as if it had just been constructed with its given pattern and character sequence. The character sequence may be changed. The pattern may also be changed on the fly, without changing the current position. This may be useful if you alternately match first one pattern then another. Although matchers may be reused, the typical usage pattern is to construct them for a text, use them once, then discard them.

4.16.3 Serialization

Patterns are serializable. The string representing the pattern and the compilation flags are stored and the version read back in is an instance of `Pattern` behaving just like the original. Often patterns do not need to be serialized because they're effectively singletons (i.e., constants).

Chapter 5

Input and Output

Java provides a rich array of classes for dealing with input and output. At the basic level, input and output streams provide a means to transfer data consisting of a sequence of bytes. Data may be transferred among memory with memory-backed streams, the file system via file streams or memory mapped files, the terminal via standard input and output streams, and remote servers via sockets. Stream filters may be put in place to compress or decompress data.

Some streams block awaiting input or output. For instance, file (network) input streams will block while waiting for the file system (network), which is much slower than Java code. Read requests from a console representing standard input will block awaiting the next input from a user. To handle pipes, which allow an input stream to read from an output stream, Java uses threads to allow reads to wait until writes are ready or vice-versa.

Bytes may encode anything, but we are particularly interested in the case of natural language text data, with or without document structure. To support streams of `char` values, Java provides `char` streams as byte streams. In order to create a character stream from byte stream, the character encoding must be known.

We are also interested in sequences of bytes that encode Java objects through serialization. We use serialized objects to represent models and their components, such as tokenizers.

5.1 Files

Java provides the class `File` in the `java.io` package for representing files and directories on the local file system. Underlyingly, an instance of `File` is nothing more than a string representing a path coupled with a convenient set of utility methods, some of which access the underlying file system.

5.1.1 Relative vs. Absolute Paths

Files may be either relative or absolute. An absolute path name provides a full description of where to find a file on the system in question. A relative path picks out a file or directory starting from a root directory (the form of which will be operating system specific). Relative paths may pick out different resources in different situations depending on the base directory.

The static method `File.listRoots()` provides an array of all of the available root directories. We wrote a simple demo in the class `ListRoots`, the heart of which is

```
File[] roots = File.listRoots();
for (File file : roots)
```

Running it on my Windows machine returns

```
> ant list-roots

Root Directories
     C:\
     E:\
```

The root directories `C:\` and `E:\` correspond to my two drives, but they could also correspond to partitions of a single drive or a redundant array of independent disks (RAID), or even a section of memory acting like a disk drive.

Root of Relative Paths

In Java, relative paths are interpreted relative to the directory from which Java was called. The same goes for Ant, except when a `basedir` attribute is set on the top-level `project` element.

A consequence of this behavior is that the empty string (`""`) is a path that picks out the directory from which Java was called.

5.1.2 Windows vs. UNIX File Conventions

Due to the differing setup of the operating systems, `File` objects behave differently on different platforms. In all cases, they respect the conventions of the file system on which they are run, but it requires some care to create portable code or Ant tasks that involve files.

On either Windows or UNIX, files are organized hierarchically into directories under one or more root directories. Each file other than one of the root directories has a parent directory. The most prominent difference between the two operating system flavors is that Windows uses a backslash (\), whereas UNIX uses a forward slash (/) to separate directories and file names in a path.

In Unix, there is a single root directory for the file system, and absolute path names will start with a forward slash (/).

In Windows, there may be more than one root directory. Each root directory is written as a drive letter followed by a colon-backslash (`:\`), as in `C:\` for the root directory on the drive named `C`.

Windows also allows Universal Naming Convention (UNC) network paths, which begin with two backslashes (\\) followed by the name of the computer or device on the network. For instance, `\\MONTAGUE` is the root of the shared files on the network computer named `MONTAGUE`. Each networked Windows computer makes visible its top-level shared files.

The biggest difference is that in Windows, file names are not case sensitive. Thus `FOO`, `foo`, and `Foo` will all pick out the same file.

5.1.3 Constructing a `File`

There are four `File` constructors. The simplest, `File(String)`, simply takes a path name to construct a file object. The string may be anything, but separator characters will be parsed and normalized.

Conveniently, Java lets you use either forward (Unix) or backward (Windows) slashes interchangeably, as does Ant. Thus in Windows, I can use Unix-style forward slashes with the same effect as backward slashes. For example, `new File("c:/foo")`, constructs the same object as `new File("c:\foo")`.

There are also two-argument constructors taking a parent directory and a file name. The parent may be specified as a string using the constructor `File(String,String)` or as a file itself, using the constructor `File(File,String)`.

URI/URL to File Conversion

There is a fourth `File` constructor that takes a uniform resource identifier (URI) argument, `File(URI)` (see Section 5.18 for more information on URIs and URLs). The `URI` class is in the `java.net` package, which comes standard with Java.

URIs for files have the form `file://`*`host`*`/`*`path`*, where the (optional) *`host`* picks out a domain name and *`path`* a file path. If the *`host`* is not specified, the form devolves to `file:///`*`path`*, and the host is assumed to be the the so-called localhost, the machine on which Java is running.

It is also possible to convert an instance of `File` to a URI, using the method `toURI()` in `File`.

It is also possible to get a uniform resource locator (URL) from a file, by first converting to a URI and then calling the `URI` method `toURL()`, which returns an instance of `URL`, also in `java.net`.

5.1.4 Getting a File's Canonical Path

The method `getCanonicalFile()` returns a `File` based on an absolute path with a conventionalized name. For instance, if I happen to be running in directory `c:\lpb` (on Windows), then the following two expressions return equivalent files

```
new File("c:\\lpb\\Foo").getCanonicalPath()

new File("foo").getCanonicalPath();
```

Note that this is not truly a unique normal form. The resulting files will have different capitalizations for `foo` in this case, but will be equivalent running under Windows. Further note that the canonical path is not used for equality (see Section 5.1.8).

5.1.5 Exploring File Properties

There are many methods on File objects which explore the type of a file. The method exists() tests whether it exists, isAbsolute() tests whether it denotes an absolute path, isDirectory() whether it's directory, isFile() whether it's an ordinary file, and isHidden() tests whether it's hidden (starts with a period (.) in Unix or is so marked by the operating system in Windows).

What permissions an application has for a file can be probed with canExecute(),canRead(), and canWrite(), all of which return boolean values.

The time it was last modified, available through lastModified(), returns the time as a long denoting milliseconds since the epoch (the standard measure of time on computers, and the basis of Java's date and calendar representations).

The number of bytes in a file is returned as a long by length().

The name of the file itself (everything but the parent), is returned by getName(). The methods getParent() and getParentFile() return the parent directory as a string or as a file, returning null if no parent was specified.

5.1.6 Listing the Files in a Directory

If a file is a directory, the method listFiles() returns an array of files that the directory contains; it returns null if the file's not a directory. The files may also be listed by name using list().

File listings may be filtered with an implementation of FileFilter, an interface in java.io specifying a single method accept(File) returning true if the file is acceptable. An older interface, FilenameFilter, is also available for file name listings.

LingPipe's FileExtensionFilter

LingPipe provides a convenience class FileExtensionFilter in package com.aliasi.io, which implements Java's interface FileFilter based on the extension of a file. A file's extension is here defined to mean the string after the final period (.) in the file's name. This is purely a convention for naming files, which is especially useful in Windows or web browsers, where each extension may be associated with a program to open or view files with that extesnion.

A file extension filter takes a single string or multiple strings specified as an array or using variable length arguments. The filter will match files ending with any of the extensions.

A file extension filter is configured with a flag indicating whether it should automatically accept directories or not, which is convenient for recursive descents through a file system. The default for the unmarked constructors is to accept directories.

File extension filters work on File objects through Java's FileFilter interface. File extension filters do not implement the Java interface FilenameFilter because it produces an ambiguity when calling the methods on Java File objects listFiles(FilenameFilter) and listFiles(FileFilter).[1] Although it's not

[1]In general, if there are methods foo(A) and foo(B) in a class taking arguments specified by

usually necessary, if you need to filter files based on name, file extension filters implement a method `fileNameFilter()` which returns the corresponding file name filter implementing `FilenameFilter`.

There are two static utility methods in the class `Files` in `com.aliasi.util` that break file names apart into their suffix and their base name. The method `Files.extension(File)` returns the extesnion of a file, consisting of the text after the final period (`.`), or `null` if there is no period. The method `baseName(File)` returns the base name of a file, which excludes the last period and extension. For instance, a file named `io.tex` has extension `tex` and base name `io`, whereas `foo.bar.baz` has extension `baz` and base name `foo.bar`.

5.1.7 Exploring Partitions

Each file in an operating system exists in a partition, which is a top-level root directory. This is of interest because a file can't be created in a partition that's larger than the available space in that partition.

There are methods in `File` for exploring partitions. The methods `getFreeSpace()`, `getTotalSpace()`, and `getUsableSpace()` return the respective amounts of space on the partition as a `long` value.

5.1.8 Comparison, Equality, and Hash Codes

Equality of `File` objects is based on the underlying platform's notion of equality. For instance, on Windows, a file created with `new File("Foo")` creates a file that is equal to `new File("foo")`. For better or worse, this comparison is based on the platform's convention for file comparison, not by constructing a canonical path and comparing that. For instance, if I happen to be running from directory `c:\lpb`, then `new File("foo")` is not equal to `c:\lpb\foo`, even though their canonical paths are the same.

5.1.9 Example Program Listing File Properties

We wrote a simple program `FileProperties`, which given the name of a file as an argument, lists all of its properties, its canonical path, its URI form, and provides a listing if it is a directory. The code is just a litany of prints for all of the methods described above.

Calling it with the Ant target `file-properties`, it uses the `file.in` Ant property to specify an argument. We can call it on its own source file using a relative path,

```
>     ant -Dfile.in=src/com/lingpipe/book/io/FileProperties.java
file-properties
```

```
arg=|src/com/lingpipe/book/io/FileProperties.java|
toString()=src\com\lingpipe\book\io\FileProperties.java
getcanonicalFile()=C:\lpb\src\io\src\com\lingpipe\book\
```

interfaces `A` and `B`, and there is an object `x` that implements both `A` and `B`, then you cannot write `foo(x)`, because the compiler doesn't know which implementation of `foo()` to use.

```
     io\FileProperties.java
getName()=FileProperties.java
getParent()=src\com\lingpipe\book\io
toURI()=file:/c:/lpb/src/io/src/com/lingpipe/book/
     io/FileProperties.java
toURI().toURL()=file:/c:/lpb/src/io/src/com/lingpipe/
     book/io/FileProperties.java
exists()=true
isAbsolute()=false
isDirectory()=false
isFile()=true
isHidden()=false
hashCode()=-72025272
lastModified()=1279224002134
new Date(lastModified())=Thu Jul 15 16:00:02 EDT 2010
length()=2141
canRead()=true
canExecute()=true
canWrite()=true
getFreeSpace()=320493494272
getTotalSpace()=500105736192
getUsableSpace()=320493494272
listFiles()={    }
```

The line breaks plus indents were inserted by hand so it'd fit in the book.

5.1.10 Creating, Deleting, and Modifying Files and Directories

In addition to inspecting files, the `File` class lets you modify files, too.

Calling `createNewFile()` creates an empty file for the path specified by the `File` on which it is called, returning `true` if a new file was created and `false` if it already existed.

The `delete()` method attempts to delete a file, returning `true` if it's successful. If applied to a directory, `delete()` will only succeed if the directory is empty. To get around this limitation, LingPipe includes a static utility method `removeRecursive(File)` in the class `File` in package `com.aliasi.util`, which attempts to remove a file and all of its descendants.

The properties of a file may be modified using, for example, `setExecutable(boolean, boolean)` which attempts to make a file executable for everyone or just the owner of the file. Similar methods are available for readability, writability and last modification date.

Files may be ranemd using `renameTo(File)`, which returns `true` if it succeeds.

The method `mkdir()` creates the directory corresponding to the file. This method only creates the directory itself, so the parent must exist. The method `mkdirs()` recursively creates the directory and all of its parent directories.

5.1.11 Temporary Files

It is sometimes useful to create a temporary file. One of the properties used to create the Java virtual machine is a directory for “temporary” files.

This is typically a platform default directory, like `/tmp` in UNIX or `C:\WINNT\TEMP` on Windows, but it may be configured when the JVM starts up.

The static `File` method `createTempFile(String,String)` attempts to create a temporary file in the default temporary file with the specified prefix and suffix, and it returns the file that’s created. After the file is created, it will exist, but have length zero.

The temporariness of the file doesn’t mean it’ll be guaranteed to disappear. It is a matter of whether you delete the file yourself or whether the operating system occassionally cleans out the temp directory.

Invoking the method `deleteOnExit()` on a file tells the operating system that the file should be deleted when the JVM exits. Unfortunately, this method is buggy on Windows, so should not be taken as provided strong cross-platform guarantees. It’s always a good idea to make sure to use a `finally` block to clear out temporary files.

5.2 I/O Exceptions

Almost all of the I/O operations are declared to throw `IOException`, a checked exception in the `java.io` package. Like all exceptions, I/O exceptions should not be ignored. In most cases, methods should simply pass on exceptions raised by methods they call. Thus most methods using I/O operations will themselves be declared to throw `IOException`.

At some point, a method will have to step up and do something with an exception. In a server, this may mean logging an error and continuing on. In a short-lived command-line program, it may mean terminating the program and printing an informative error message.

It is important to clean up system resources in a long-running program. Thus it is not uncommon to see code in a `try` with a `finally` block to clean up even if an I/O exception is raised.

5.2.1 Catching Subclass Exceptions

Some methods are declared to throw exceptions that are instances of subclasses of `IOException`. For instance, the constructor `FileInputStream(File)` may throw a `FileNotFoundException`.

If you are passing on exceptions, don’t fall into the trap of passing just a general `IOException`. Declare your method to throw `FileNotFoundException` and to throw `IOException`. This provides more information to clients of the method, who then have the option to handle the exceptions generically at the `IOException` level, or more specifically, such as for a `FileNotFoundException`. It also helps readers of your Javadoc understand what specifically could go wrong with their method calls.

In general, it's a good idea to catch subclasses of exceptions separately so that error handling and error messages may be as specific as possible. If the methods called in the `try` block are declared to throw the more general `IOException`, you'll want to catch the `FileNotFoundException` first. For example, it's legal to write

```
try {
     in = new FileInputStream(file);
} catch (FileNotFoundException e) {
     // do something
} catch (IOException e) {
     // do something
}
```

In general, you have to order the catches from more specific to mroe general. If you try to catch the I/O exception before the file not found exception, the compiler will gripe that the file not found exception has already been caught.

5.3 Security and Security Exceptions

Almost all of the I/O operations that attempt to read or modify a file may also throw an unchecked `SecurityException` from the base package `java.lang`. Typically, such exceptions are thrown when a program attempts to overstep its permissions.

File security is managed through an instance of `SecurityManager`, also in `java.lang`. This is a very flexible interface that is beyond the scope of this book.

5.4 Input Streams

The abstract class `InputStream` in `java.io` provides the basic operations available to all byte input streams. Concrete input streams may be constructed in many ways, but they are all closed through the `Closeable` interface (see Section 5.6).

5.4.1 Single Byte Reads

A concrete extension of the `InputStream` class need only implement the top-level `read()` method. The `read()` method uses the C programming language pattern of returning an integer representing an unsigned byte value between 0 (inclusive) and 255 (inclusive) as the value read, or -1 if the end of stream has been reached.

5.4.2 Byte Array Reads

It is usually too inefficient to access data one byte at a time. Instead, it is usually accessed in blocks through an array. The `read(byte[])` method reads data into the specified byte array and returns the number of bytes that have been read. If the return value is -1, the end of the input stream has been reached. There is a similar method that reads into a slice of an array given an offset and length.

5.4.3 Available Bytes

The method `available()` returns an integer representing the number of bytes available for reading without blocking. This is not the same as the number of bytes available. For one thing, streams may represent terabytes of data, and `available()` only returns an integer. More importantly, some streams may block waiting for more input. For instance, the standard input (see Section 5.17 may block while waiting for user input and show no avaialable bytes, even if bytes may later become available.

Because no bytes may be available without blocking, it's possible for the block read method to return after reading no bytes. This does not mean there won't be more bytes to read later. The `read()` methods will return -1 when there are truly no more bytes to be read.

5.4.4 Mark and Reset

The method `markSupported()` returns `true` for markable input streams. If a stream is markable, calling `mark()` sets the mark at the current position in the stream. A later call to `reset()` resets the stream back to the marked position.

5.4.5 Skipping

There is also a method `skip(long)`, which attempts to skip past the specified number of bytes, returning the number of bytes it actually skipped. The only reason to use this method is that it is often faster than actually reading the bytes and then throwing them away, because it skipping bypasses all the byte assignments that would otherwise be required for a read.

5.5 Output Streams

The `OutputStream` abstract class in the package `java.io` provides a general interface for writing bytes. It is thus the complement to `InputStream`. Output streams implement the `Closeable` interface, and should be closed in the same way as other closeable objects (see Section 5.6).

`OutputStream` itself implements the array writers in terms of single byte writes, with the close and flush operations doing nothing. Subclasses override some or all of these methods for additional functionality or increased efficiency.

5.5.1 Single Byte Writes

A single abstract method needs to be implemented by concrete subclasses, `write(int)`, which writes a byte in unsigned notation to the output stream. The low-order byte in the integer is written and the three high-order bytes are simply ignored (see Section 2.1.5 for more on the byte organization of integers). Thus the value should be an unsigned representation of a byte between 0 (inclusive) and 255 (inclusive).

5.5.2 Byte Array Writes

Arrays of bytes may be written using the method `write(byte[])`, and a slice variant taking a byte array with a start position and length.

The behavior is slightly different than for reads. When a write method returns, all of the bytes in the array (or specified slice thereof) will have been written. Thus it is not uncommon for array writes to block if they need to wait for a resource such as a network, file system, or user console input.

5.5.3 Flushing Writes

The method `flush()` writes any bytes that may be buffered to their final destination. As noted in the documentation, the operating system may provide further buffering, for instance to the network or to disk operations.

The `OutputStream` class is defined to implement `Flushable`, the interface in `java.io` for flushable objects. Like the general `close` method from the `Closeable` interface, it may raise an `IOException`.

Most streams call the `flush()` method from their `close()` method so that no data is left in the buffer as a result of closing an object.

5.6 Closing Streams

Because most streams impose local or system overhead on open I/O streams, it is good practice to close a stream as soon as you are done with it.

5.6.1 The `Closeable` Interface

Conveniently, all the streams implement the handy `Closeable` interface in the package `java.io`. The `Closeable` interface specifies a single method, `close()`, which closes a stream and releases any resources it may be maintaining.

Care must be taken in closing streams, as the `close()` method itself may throw an `IOException`. Furthermore, if the stream wasn't successfully created, the variable to which it was being assigned may be null.

The typical idiom for closing a `Closeable` may be illustrated with an input stream

```
InputStream in = null;
File file = null;
try {
     in = new FileInputStream(file);
     // do something
} finally {
     if (in != null)
         in.close();
}
```

Sometimes, the close method itself is wrapped to suppress warnings, as in

```
try {
     in.close();
} catch (IOException e) {
     /* ignore exception */
}
```

Note that even if the close method is wrapped, the operations in the try block may throw I/O exceptions. To remove these, I/O exceptions may be caught in a block parallel to the `try` and `finally`.

5.6.2 LingPipe Quiet Close Utility

If a sequence of streams need to be closed, things get ugly quickly. LingPipe provides a utility method `Streams.closeQuietly(Closeable)` in `com.aliasi.util` that tests for null and swallows I/O exceptions for close operations. Basically, it's defined as follows.

```
static void close(Closeable c) {
     if (c == null) return;
     try {
         c.close();
     } catch (IOException e) {
         /* ignore exception */
     }
}
```

If you need to log stream closing exceptions, a similar method may be implemented with logging in the catch operation and a message to pass as an argument in the case of errors.

5.7 Readers

The `Reader` abstract base class is to `char` data as `InputStream` is to `byte` data. For instance, `read()` reads a single `char` coded as an `int`, with -1 indicating the end of the streamstream. There is a block reading method `read(char[])` that returns the number of values read, with -1 indicating end of stream, and a similar method taking a slice of an array of `char` values. There is also a `skip(int)` method that skips up to the specified number of `char` values, returning the number of values skipped. The `Reader` class implements the `Closeable` interface, and like streams, have a single `close()` method. All of these methods are declared to throw `IOException`.

Unlike input streams, concrete subclasses must implement the abstract methods `close()` and the slice-based bulk read method `read(char[],int,int)`.

Some readers are markable, as indicated by the same `markSupported()` method, with the same `mark()` method to set the mark at the current position and `reset()` method to return to the marked position.

5.7.1 The `Readable` Interface

The `Reader` class also implements the `Readable` interface in `java.lang`. This interface has a single method `read(CharBuffer)` that reads `char` values into a specified buffer, returning the number of values read or -1 for end of stream (see Section 3.8 for more on `CharBuffer`).

5.8 Writers

Writers are to readers as output streams are to input streams. The method `write(int)` writes a single character using the two low-order bytes of the specified integer as a `char`.

Arrays of characters may be written using `write(char[])` and there is a method to write only a slice of a character array. Strings and slices of strings may also be written.

Like other streams, `Writer` implements `Closeable` and is hence closed in the usual way using `close()`. Like other output streams, it implements `Flushable`, so calling `flush()` should try to write any buffered characters.

5.8.1 The `Appendable` Interface

The `Writer` class also implements the `Appendable` interface in `java.lang`. This is the same interface as is implemented by the mutable character sequence implementations `CharBuffer`, `StringBuilder`, and `StringBuffer`.

This interface specifies a method for appending a single character, `append(char)`, and for sequences, `append(CharSequence)`, as well as for slices of character sequences. The result specified in the `Appendable` interface is `Appendable`, so calls may be chained. The result specified by `Writer` is `Writer`; subclasses may declare more specific return types than their superclasses. Other implementations also declare their own types as their return types.

5.9 Converting Byte Streams to Characters

Resources are typically provided as byte streams, but programs manipulating language data require character streams. Given a character encoding, a byte stream may be converted to a character stream.

5.9.1 The `InputStreamReader` Class

Given an input stream and a character encoding, an instance of `InputStreamReader` may be constructed to read characters. The usual idiom given that encoding is an instance of String and in an instance of InputStream is

```
Reader reader = new InputStreamReader(in,encoding);
// do something
reader.close();
```

The operations ellided in the middle may then use the reader to retrieve character-based data. Instances may also be constructed from an instance of CharSet or with a CharSetDecoder, both of which are in java.nio.charset; the string-based method is just a shortcut that creates a decoder for a character set behind the scenes. This decoder is used to convert the sequence of bytes from the underlying input stream into the char values returned by the reader.

The question remains as to whether to attempt to close the input stream itself given that the reader was closed. When an InputStreamReader is closed, the input stream it wraps will be closed.[2] As long as the reader was constructed, there should be no problem. For more robustness, the stream may be nested in a try-finally block to ensure it's closed. The reader should be closed first, and if all goes well, the stream should already be closed when the close in the finally block is reached. This second call should not cause problems, because the Closeable interface specifies that redundant calls to close() should have no effect on a closed object.

5.9.2 The OutputStreamWriter Class

In the other direction, the class OutputStreamWriter does the conversion. The idiom for construction, use, and closure is the same, only based on a CharSetEncoder to convert sequences of characters to sequences of bytes for writing.

5.9.3 Illegal Sequences

The documentation for both of these classes indicate that their behavior in the face of illegal byte or char sequences. For instance, an InputStreamReader may be faced with a sequence of bytes that is not decodeable with the specified character encoding. Going the other way, an OutputStreamWriter may receive a sequence char values that is not a legal UTF-16 sequence, or that contains code points that are not encodable in the specified character encoding.

In practice, a InputStreamReader's behavior will be determined by its CharSetDecoder, and a OutputStreamWriter's by its CharSetEncoder. The encoders and decoders may be configured to allow exceptions to be raised for illegal sequences or particular substitutions to be carried out.

[2]This is not clear from the documentation, but is clear from the source code. Perhaps we're meant to infer the behavior from the general contract for close() in Closeable, which specifies that it should release all of the resources it is holding.

5.10 File Input and Output Streams

Streaming access to files is handled by the subclasses `FileInputStream` and `FileOutputStream`, which extned the obvious base classes.

5.10.1 Constructing File Streams

A file input stream may be constructed with either a file using the constructor `FileInputStream(File)` or the name of a file, using `FileInputStream(String)`.

Similarly, output streams may be constructed using `FileOuptutStream(File)` and `FileOuptutStream(String)`. If constructed this way, if the file exists, it will be overwritten and resized to the output produced by the output stream. To append to a file, construct an output stream using `FileOuptutStream(File,boolean)`, where the boolean flag specifies whether or not to append.

The constructors are declared to throw the checked exception `FileNotFoundException`, in package `java.io`, which is a subcclass of `IOException`.

5.10.2 Finalizers

Both the input and output file streams declare finalizer methods which may release any system resources when the related stream is no longer in scope. The problem with finalization is that it's not guaranteed to happen until the JVM on which the code was running is terminated.

In any case, whether it exits normally or not, the operating system will recover any process-specific file handles. Thus short-lived processes like simple commands do not need to be as careful about recovering file-based resources as long-running server processes. In any case, the best the code can do is close all streams as soon as they are no longer needed, even in the face of exceptions.

5.10.3 File Descriptors

The `FileDescriptor` class in `java.io` provides an encapsulation of a file input or output stream. Every time a file input or output stream is created, it has an associated file descriptor. The file descriptor may be retrieved from a file input stream or file output stream using the method `getFD()`. A new input or output stream may be constructed based on a file descriptor.

5.10.4 Examples of File Streams

Counting the Bytes in a File

We wrote a demo program `FileByteCount` that opens a file input stream and then reads all of its content, counting the number of instances of each byte. The work is done with

```
public static void main(String[] args)
     throws FileNotFoundException, IOException {

     File file = new File(args[0]);
     InputStream in = new FileInputStream(file);
     long[] counts = new long[256];
     int b;
     while ((b = in.read()) != -1)
         ++counts[b];
```

Note that the `main()` method is declared to throw both a `FileNotFoundException` and its superclass `IOException` (see Section 5.2 for more information on I/O exceptions and their handling).

The body of the method creates an instance of `File` using the command-line argument, then an input stream, based on the file. It then allocates an array of `long` values the size of the number of bytes, which is used to store the counts (recall that primitive numbers are always initialized to 0 in Java, including in arrays). Then we use the C-style read method to read each bye into the integer variable `b`. If the value is -1, indicating the end of stream (end of file here), the loop is exited. In the body of the read loop, the count for the byte just read is incremented.

We call the method using the Ant target `file-byte-count`, specifying the `file.in` property for the file to be counted. Here we call it on its own source code,

```
>       ant -Dfile.in=src/com/lingpipe/book/io/FileByteCount.java
file-byte-count
```

```
Dec  Hex      Count
 10    a         33
 32   20        189
 33   21          1
...
123   7b          2
125   7d          2
```

Because the file is encoded in ASCII, these bytes can be read as characters. For instance, character hexadecimal value 0A is the linefeed, 20 is a newline, 32 is the digit 2, up through 7D, which is the right curly bracket (}). We can infer that the file has 34 lines, because there are 33 line feeds.

Copying from One File to Another

Our next sample program, `CopyFile`, copies the bytes in one file into another file and illustrates the use of array-based reads and writes.[3] After reading in the command-line arguments, the work is carried out by

[3]The same functionality is available through the LingPipe static utility method `copyFile(File,File)` from the `Files` class in `com.aliasi.util`.

```
InputStream in = new FileInputStream(fileIn);
OutputStream out = new FileOutputStream(fileOut);
byte[] buf = new byte[8192];
int n;
while ((n = in.read(buf)) >= 0)
     out.write(buf,0,n);
out.close();
in.close();
```

We first create the file input and output streams. We've assigned them to variables of type `InputStream` and `OutputStream` rather than to their specific subclasses because we do not need any of the functionality specific to file streams. We then allocate an array `buf` of bytes to act as a buffer and declare a variable `n` to hold the count of the number of bytes read. The loop continually reads from the input stream into the buffer, assigning the number of bytes read to `n`, and exiting the loop if the number of bytes read is less than zero (signaling end of stream). In the body of the loop, we write the bytes in the buffer to the output stream, being careful to specify that we only want to write `n` bytes. Unlike reads, writes either succeed in writing all their bytes (perhaps having to block), or throw an exception. When we're done, we close both streams.[4]

Character Encoding Conversion

As a third example, we'll show how to wrap our file-based streams in readers and writers in order to carry out character encoding conversion. Here we use a character buffer to do the transfer,

```
InputStream in = new FileInputStream(fileIn);
InputStreamReader reader = new InputStreamReader(in,encIn);
OutputStream out = new FileOutputStream(fileOut);
OutputStreamWriter writer = new OutputStreamWriter(out,encOut);
char[] buf = new char[4096];
for (int n; (n = reader.read(buf)) >= 0; )
     writer.write(buf,0,n);
writer.close();
reader.close();
```

The program may be called with the Ant target `file-decode-encode` with arguments given by properties `file.in`, `enc.in`, `file.out`, and `enc.out`.

[4]We don't need elaborate try-finally logic to close the file streams because they are closed automatically when the command-line program exits. We could get into trouble if another program were to call this `main()` without exiting the JVM after the call. A more robust version of this program might catch I/O exceptions thrown by the write method and delete the output stream rather than leaving a partial copy. We've shown what this would look like in the source file for `CopyFile` in a parallel method named `main2()`.

5.11 Buffered Streams

Some sources of input, such as an HTTP connection or a file, are most efficiently dealt with in larger chunks than a single byte. In most cases, when using such streams, it is more efficient to buffer the input or output. Both byte and character streams may be buffered for both input and output. For byte streams, the classes are `BufferedInputStream` and `BufferedOutputStream`, and for character streams, `BufferedReader` and `BufferedWriter`

5.11.1 Constructing Buffered Streams

The constructor `BufferedInputStream(InputStream,int)` creates a buffered input stream wrapping the specified input stream with the specified buffer size (in bytes). The constructor `BufferedOutputStream(OutputStream,int)` performs the same role for output streams. There are also one-argument constructors for which the buffer sizes may be dropped, in which case a default size is used.

For example, we may created a buffered input stream from a file input stream given a `File` instance `file` with

```
InputStream in = new FileInputStream(file);
InputStream bufIn = new BufferedInputStream(in);
```

The variable `bufIn` is then used where the unbuffered stream `in` would've been used. By assigning all these streams of different types to a base `InputStream`, it makes it easier to change the code later.

The constructors only save a pointer to the underlying stream and create a buffer, so they don't throw I/O exceptions, because they're not doing any I/O.

5.11.2 Sizing the Buffer

There is often a measurable performance improvement from getting the buffer size right for the underlying platform and application, but performance measurement is a tricky business (see Section B.2 for recommended reading on performance tuning).

5.11.3 Closing a Buffered Stream

When a buffered stream is closed, it closes the contained stream. See the discussion in Section 5.9.1 concerning writing robust stream closing idioms. The close will also flush any bytes or characters currently being buffered.

5.11.4 Built-in Buffering

Many of the streams (or readers and writers) have built-in buffering. For instance, the GZIP input and output streams have built-in buffers. In these cases, there's no need to buffer twice for efficiency's sake.

If you are eventually going to wrap a byte stream in a buffered reader or writer, the byte stream doesn't itself need to be buffered. The buffering, though use of block reads and writes, carries itself through.

Similarly, if your own code does buffering, as in our example in `CopyFile` (see Section 5.10.4), there's no need to buffer the file input or output streams.

By this same line of reasoning, if a byte stream is converted to a character stream or vice versa, buffering is not required at both places. Usually, the buffer is placed later in the chain, around the reader or writer, as in the following examples.[5]

```
InputStream in = new FileInputStream(file);
Reader reader = new InputStreamReader(in,Strings.UTF8);
BufferedReader bufReader = new BufferedReader(reader);

OutputStream out = new FileOutputStream(file);
Writer writer = new OutputStreamWriter(out,Strings.UTF8);
BufferedWriter bufWriter = new BufferedWriter(writer);
```

5.11.5 Line-Based Reads

The `BufferedReader` class implements a method to read lines. The method `readLine()` returns a `String` corresponding to the next line of input read, not including the end-of-line markers. If the end of stream has been reached, this method will return null.

This class recognizes three end-of-line markers, a linefeed (`\n`), a carriage return (`\r`), or a linefeed followed by a carriage return (`\n\r`).

5.12 Array-Backed Input and Output Streams

The array-backed input streams and readers read from arrays of data held in memory. The array-backed output streams and writers buffer the bytes or characters written to them and provide access to them as an array at any time.

5.12.1 The `ByteArrayInputStream` Class

The class `ByteArrayInputStream` wraps a byte array to produce an in-memory input stream. The constructor `ByteArrayInputStream(byte[])` creates the input stream. The byte array is not copied, so changes to it will affect the input stream.

Byte array input streams support the `mark()` and `reset()` methods.

Closing a byte array input stream has no effect whatsoever.

[5]We use UTF-8 encoding so frequently that LingPipe provides the static final string constant `Strings.UTF8` to use to specify UTF-8 encodings.

5.12.2 The `CharArrayReader` Class

The class `CharArrayReader` performs the same service, creating a `Reader` from an array of characters. They support the same range of methods with almost all the same behaviors, only for `char` data rather than `byte` data. Inconsistently, `close()` behaves differently, releasing the reference to the underlying character array and causing all subsequent calls to read methods to throw I/O exceptions.

There is also a class `StringReader` that does the same thing as a character array reader for string input.

5.12.3 The `ByteArrayOutputStream` Class

A `ByteArrayOutputStream` provides an in-memory output stream that buffers any bytes written to it. Because it uses arrays, the maximum amount of data that can be written to an output stream is `Integer.MAX_VALUE` bytes.

There is a no-argument constructor and one that specifies the initial capacity of the buffer.

The method `size()` returns the number of bytes that are buffered.

All of the bytes written to a byte array output stream are available as a byte array using the method `toByteArray()`. The returned array is a copy of the bytes buffered in an underlying array, so changes to it do not affect the output stream. Calling this method does not remove the returned bytes from the buffer.

Calling `reset()` removes all of the currently buffered bytes, allowing a byte array output stream to be reused.

Closing a byte-array output stream has no effect. In particular, the bytes that have been buffered will still be available.

5.12.4 The `CharArrayWriter` Class

A `CharArrayWriter` does the same thing for characters, implementing methods of the same name as byte-array output streams.

5.13 Compressed Streams

In many cases, resources are compressed to save storage space or transfer bandwidth. The standard Java package `java.util.zip` supports a general pair of streams, `DeflaterInputStream` and `InflaterInputStream`, for compressing and decompressing sequences of bytes. Although new implementations may be written, Java builds in support for Gnu Zip (GZIP) compression and for the ZIP format for compressing archives consisting of multiple hierarchically named streams as an archive. Java's own Java archive (JAR) files are essentially zipped archives, and are also implemented.

5.13.1 GZIP Input and Output Streams

The GZIP input and output streams are constructed by wrapping anothr stream and optionally specifying a buffer size. After construction, they behave just like any other stream.

The sample program FileGZIP wraps a file output stream in a GZIP output stream in order to implement a simple compression command-line program. The work in the main() method is done by

```
InputStream in = new FileInputStream(fileIn);
OutputStream out = new FileOutputStream(fileOut);
OutputStream zipOut = new GZIPOutputStream(out);

byte[] bs = new byte[8192];
for (int n; (n = in.read(bs)) >= 0; )
     zipOut.write(bs,0,n);

in.close();
zipOut.close();
```

Aside from the creation of the compressed stream, the rest is just a buffered stream copy.

The sample program FileUnGZIP uncompresses one file into the other, with all the work being done in the creation of the streams,

```
InputStream in = new FileInputStream(fileIn);
InputStream zipIn = new GZIPInputStream(in);
OutputStream out = new FileOutputStream(fileOut);
```

After the streams are created, the input is copied to the output as before. Both programs print the input and output file lengths for convenience.

The Ant target file-gzip calls the program with arguments the value of properties file.uncompressed and file.gzipped. It's conventional to name a compressed file the same thing as the original file with a suffix denoting the type of compression, which for GZIP is gz. Here we compress the build file build.xml, which being XML, is highly redundant, and hence highly compressible.

```
>   ant -Dfile.uncompressed=build.xml -Dfile.gzipped=build.xml.gz
file-gzip

Original Size=7645          Compressed Size=1333
```

We can uncompress in the same way, calling the file-ungzip target, which takes as command-line arguments the values of properties file.gzipped and file.ungzipped.

```
> ant -Dfile.ungzipped=build.xml.copy -Dfile.gzipped=build.xml.gz
file-ungzip

Original Size=1333          Expanded Size=7645
```

The round trip results in build.xml.copy containing exactly the same bytes as build.xml. You may verify the files contain the same content using the diff program from the command line,

```
> diff build.xml build.xml.copy
```

which prints no output when the files being compared are the same. Listing the directory contents shows the sizes,

```
> ls -l

7645 Aug  5 16:54 build.xml
7645 Aug  5 16:58 build.xml.copy
1333 Aug  5 16:58 build.xml.gz
```

Along the same lines, you may verify that command-line versions of GZIP, such as `gunzip`, can handle the compressed file.[6][7]

If you try to compress a stream that isn't in GZIP format, the input stream will throw an I/O exception with an explanatory message.

5.13.2 ZIP Input and Output Streams

ZIP streams enable the representation of compressed archives. A ZIP archive consists of a sequence of entries. Each entry contains meta information and data. The data is accessed through an input stream for reading and output stream for writing. The pattern for using the ZIP streams is somewhat unusual, so we will provide an example.

The `ZIPEntry` Class

The meta-information for an entry is encapsulated in an instance of `ZIPEntry` in the `java.util.zip` package. Information about the entry including compressed size, CRC-32 checksum, time last modified, comments, and the entry's name are available through methods.

ZIP archives may be arranged hierarchically. Directory structure is described through entry names, with the forward slash (/) acting as the separator. The method `isDirectory()` is available to determine if an entry is a directory, which is signaled by its ending in a forward slash.

For the purpose of creating archives, the zip entry values may be set with setter methods corresponding to the get methods. This entails that the `ZIPEntry` class is mutable.

Creating a Zip Archive with Ant

To provide a zip file for experimental purposes, we've written an Ant target `example-zip` that builds a zipped archive using Ant's built-in `mkdir`, `echo`, and `zip` tasks. The Ant code is straightforward,

```
<property name="file.in.zip"
          value="build/demo-archive.zip"/>
```

[6]We recommend working in a temporary directory so as not to destroy the original `build.xml` file. The command-line compressors and uncompressors typically work by creating an output file which adds or removes the compression format's standard suffix.

[7]We've had trouble getting the compression programs to work from the DOS command line.

```
<target name="example-zip">
  <mkdir dir="build/archive/dir"/>
  <echo message="Hello."
        encoding="UTF-8"
        file="build/archive/hello.txt"/>
...
  <zip destfile="${file.in.zip}"
       basedir="build/archive"
       encoding="UTF-8"/>
</target>
```

It uses the property `file.in.zip` to define the zipped archive's location. The `mkdir` task makes a working directory in our build directory, which will be cleaned the next time the `clean` target is invoked. The `echo` task writes text to a file with a specified encoding. The ellided tasks write two more files. We then call the `zip` target, specifying the target file, the directory which is being zipped up, and the encoding to use for file names.

The demo file creation target needs to be run before the demo so there's something to unzip.

```
> ant example-zip
```

```
Building zip: c:\lpb\src\io\build\demo-archive.zip
```

Reading a ZIP Archive

A `ZIPInputStream` is created, and then the archive is read entry by entry. The method `getNextEntry()` returns the `ZIPEntry` object for the next entry to be read, or `null` if there ae no more entries. For each entry, we read its meta-information from the entry object and its data from the input stream.

We provide a sample program `FileZipView` that reads a ZIP archive from a file and displays its properties and contents. The work in the `main()` method is done by

```
InputStream in = new FileInputStream(fileZipIn);
ZipInputStream zipIn = new ZipInputStream(in);
byte[] bs = new byte[8192];
while (true) {
     ZipEntry entry = zipIn.getNextEntry();
     if (entry == null) break;
     printEntryProperties(entry);
     for (int n; (n = zipIn.read(bs)) >= 0; )
         System.out.write(bs,0,n);
}
zipIn.close();
```

To get going, the file input stream is wrapped in a `ZipInputStream`. The buffer `bs` is used for writing. This loop uses the while-true-break idiom to run until a condition discovered in the middle of the body is discovered. Here, we assign the entry variable to the next entry. If it's `null`, we break out of the loop. Otherwise, we print the properties of the entry using a subroutine with the obvious implementation, then read the bytes from the zip input stream and write them to standard output.

We have an Ant target `file-zip-view` that dumps out the contents of a Zip archive specified through the `file.in.zip` property.

```
> ant -Dfile.in.zip=build/demo-archive.zip file-zip-view

getName()=abc/     isDirectory()=true
getSize()=0     getCompressedSize()=0
new Date(getTime())=Tue Jul 20 18:21:36 EDT 2010
getCrc()=0     getComment()=null
---------------------------
getName()=abc/bye.txt     isDirectory()=false
getSize()=7     getCompressedSize()=9
new Date(getTime())=Tue Jul 20 18:21:36 EDT 2010
getCrc()=3258114536     getComment()=null
Goodbye
---------------------------
...
---------------------------
getName()=hello.txt     isDirectory()=false
...
Hello
```

We've moved lines together to save space and ellided the third entry and part of the fourth. Each division is for an entry, and the commands used to access the properties are printed. For instance, the first entry is a directory named `abc/`, the second entry is an ordinary entry named `abc/bye.txt`, implying containment in the first directory by naming. The final entry is another ordinary entry named `hello.txt`, which isn't in any directory. Note that the compressed size is larget than the uncompressed size, which is not unusual for a short (or random, or already compressed) sequence of bytes.

After printing out all the properties for each entry, the program prints the contents by simply writing the bytes read directly to the standard output. For example, the entry named `abc/bye.txt` contains the UTF-8 bytes for `Goodbye`.

Creating a Zip Archive

Zip archives may be created programmatically using `ZipOutputStream` by reversing the process by which they are read. That requires creating `ZipEntry` objects for each entry in the archive and setting their properties. The entry meta-information is then written to the archive using the `putNextEntry()` method on the Zip output stream, then writing the bytes to the stream. The method `setMethod(int)` may be used with argument constants `STORED` for uncompressed entries and `DEFLATED` for compressed entries.

5.14 Tape Archive (Tar) Streams

The archaically named Tape Archive (tar) format is popular, especially among Unix users, for distributing data with a directory structure. Tar files themselves are typically compressed using GZIP (see Section 5.13.1). The Ant build library contains classes for manipulating tar streams, so we will be importing the Ant jar for compiling and running the examples in this section.

5.14.1 Reading Tar Streams

The way in which data may be read out of a tar file is easy to see with an example.

Code Walkthrough

We provide a demo class `FileTarView` that reads from a tar file that has optionally been gzipped. The `main()` method is defined to throw an I/O exception and read a file `tarFile` and boolean flag `gzipped` from the first two command-line arguments. The code begins by constructing the tar input stream.

```
InputStream fileIn = new FileInputStream(tarFile);
InputStream in = gzipped
     ? new GZIPInputStream(fileIn)
     : fileIn;
TarInputStream tarIn = new TarInputStream(in);
```

We first create a file input stream. Then we wrap it in a GZIP input stream if the gzipped flag has been set. Then, we just wrap the input stream in a `TarInputStream`, from package `org.apache.tools.tar`.

We then use a read loop, inside which we access the properties of the entries and read the entry data.

```
while (true) {
     TarEntry entry = tarIn.getNextEntry();
     if (entry == null) break;

     boolean isDirectory = entry.isDirectory();
     String name = entry.getName();
     String userName = entry.getUserName();
     String groupName = entry.getGroupName();
     int mode = entry.getMode();
     Date date = entry.getModTime();
     long size = entry.getSize();

     byte[] bs = Streams.toByteArray(tarIn);
```

We use the `getNextEntry()` method of the tar input stream to get the next instance of `TarEntry`, also from `org.apache.tools.tar`. If the entry is null, the stream has ended and we break out of the loop.

If the tar entry exists, we then access some of its more prominent properties including whether or not it's a directory, the name of the entry, the name of the user and group for whom it was created, the mode (usually specified as octal, so we print it out in octal), the date the entry was last modified, and the entry's size.

Finally, we use LingPipe's `toByteArray()` utility, from class `Streams` in `com.aliasi.util`, to read the data in the entry. Note that like for zip input streams, this is done by reading to the end of the tar input stream. The tar input stream is then reset.

After we're done with the tar input stream, we close it through the closeable interface.

```
tarIn.close();
```

Tar input streams implement Java's `Closeable` interface, so we close it through the `close()` method. Although not documented, closing a tar input stream will close any contained stream other than `System.in`. Here, this closes the GZIP input stream, which in turn closes the file input stream.

Running the Demo

The Ant target `file-tar-view` runs the command, passing the value of properties `tar.file` and `gzipped` as command-line arguments. We use the gzipped 20 newsgroups distribution as an example (see Section A.2 for a description and link).

```
>          ant -Dtar.file=c:/carp/data/20news/20news-bydate.tar.gz
-Dgzipped=true file-tar-view

------------------------------------
name=20news-bydate-test/    isDirectory=true
userName=jrennie    groupName=    mode=755
date=Tue Mar 18 07:24:56 EST 2003
size=0    #bytes read=0
------------------------------------
name=20news-bydate-test/alt.atheism/    isDirectory=true
userName=jrennie    groupName=    mode=755
date=Tue Mar 18 07:23:48 EST 2003
size=0    #bytes read=0
------------------------------------
name=20news-bydate-test/alt.atheism/53265    isDirectory=false
userName=jrennie    groupName=    mode=644
date=Tue Mar 18 07:23:47 EST 2003
size=1994    #bytes read=1994
...
name=20news-bydate-train/    isDirectory=true
userName=jrennie    groupName=    mode=755
date=Tue Mar 18 07:24:55 EST 2003
size=0    #bytes read=0
------------------------------------
name=20news-bydate-train/alt.atheism/     isDirectory=true
...
```

Note that we first see the directory 20news-bydate-test. The user name is jrennie, there is no group name, and the mode indicating permissions is 755 (octal). The last-modified date is in 2003. The size of the directory is zero, and the number of bytes read is zero. The next directory looks similar, only its name is test/alt.atheism, indicating that it is nested in the previous entry. In this corpus, it also indicates that the directory contains postings from the alt.atheism newsgroup. The TarEntry object lets you access all the entries for the files or directories it contains, but that won't let you read the data.

After the two directories, we see a file, with name bydate-test/alt.atheism/53265 and mode 644 (octal). he reported size matches the number of bytes read, 1994.

After the files in the alt.atheism directory, there are 19 more test newsgroup directories. Then, we see the training data, which is organized the same way.

5.15 Accessing Resources from the Classpath

When distributing an application, it is convenient to package all the necessary files along with the compiled classes in a single Java archive (jar). This is particularly useful in the context of servlets, which are themselves packaged in web archives (war), a kind of jar with extra meta-information.

5.15.1 Adding Resources to a Jar

Resources that will be accessed programatically are added to an archive file just like any other file. We provide an Ant target example-resource-jar which may be used to create a jar at the path picked out by the property resource.jar.

```
<property name="resource.jar"
          value="build/demo-text.jar"/>

<target name="example-resource-jar">
  <mkdir dir="build/resources/com/lingpipe/book/io"/>
  <echo message="Hello from the jar."
        encoding="UTF-8"
        file="build/resources/com/lingpipe/book/io/hello.txt"/>
  ...
  <jar destfile="${resource.jar}">
    <fileset dir="build/resources"
             includes="**/*.txt"/>
  </jar>
</target>
```

We define the default for the property resource.jar to be a file of the same name in the build directory (which is subject to cleaning by the clean target). The target that builds the jar first makes a directory under build/resources into which files will be placed. These are created by echo tasks, for instance creating a file /com/lingpipe/book/io/hello.txt under build/resources. After the directory structure to be archived is created in build/resources, a jar task creates it, specifying that the archive should include all the files in build/resources that match the pattern **/*.txt, which is all files ending in .txt.

This command should be run before the demo in the next section.

```
> ant -Dresource.jar=build/demo-text.jar example-resource-jar

Building jar: c:\lpb\src\io\build\demo-text.jar
```

5.15.2 Resources as Input Streams

One of the ways in which an input stream may be accessed is by specifying a resource. The system will then look for the resource on the classpath in a number of prespecified locations based on its name.

The sample program ResourceInput reads the content of a resource from the classpath and writes it to the standard output. The part of the main() method that gets the resource as an input stream is

```
String resourceName = args[0];
InputStream in
     = ResourceInput.class
     .getResourceAsStream(resourceName);
```

The name of the resource is read in as the first command-line argument. Then the method getResourceAsStream(String) is called with the name of the resource. It is called on the Class instance for the ResourceInput class, specified by the static constant ResourceInput.class. The calling class is also important because the rules for locating the resource depend on the particular class loader used to load the class.

If the name begins with a forward slash (/), it is taken to be an absolute path name (absolute in the classpath, not in the file system). Otherwise, it is a relative path. Relative paths are interpreted relative to the package name of the class from which the get method was called. Specifically, the package name of the class using forward slashes as separators is used a prefix for the name.

The Ant target resource-input look for a resource with the name given by property resource.name, adding the archive named by property resource.jar to the classpath. The jar from which we are reading needs to be on the classpath. How it gets there doesn't matter: you could put it on the command line, set an environment variable read by the JVM, use a servlet container's classpath, etc. Here, we just add the jar named by property resource.jar to the classpath in the Ant target. Very often, we just bundle resources we want to read from the class path along with the compiled classes for an application. This is convenient for everything from document type definitions (DTDs) to compiled statistical models.

We run the program using the resource named hello.txt and the jar we created in the last section.

```
> ant -Dresource.jar=build/demo-text.jar -Dresource.name=hello.txt
resource-input

Hello from the jar.
```

Note that we would get the same result if we specified the `resource.name` property to be the absolute path `/com/lingpipe/book/io/hello.txt` (note the initial forward slash).

Once the stream is retrieved from the resource, it may be manipulated like any other stream. It may be buffered, converted to a reader or object input stream, interpreted as an image, etc.

5.16 Print Streams and Formatted Output

A print stream is a very flexible output stream that supports standard stream output, conversion of characters to bytes for output, and structured C-style formatting. This provides most of the functionality of a `Writer`'s methods, while extending the base class `InputStream`.[8]

5.16.1 Constructing a Print Stream

A print stream may be constructed by wrapping another input stream or providing a file to which output should be written. The character encoding and whether auto-flushing should be carried out may also be specified. We recommend restricting your attention to the constructors `PrintStream(File,String)` that writes to a file given a specified encoding, and `PrintStream(OutputStream,boolean,String)`, which prints to the specified output stream, optionally using auto-flushing, using a specified character encoding. The othe methods are shorthands that allow some parameters to be dropped.

5.16.2 Exception Buffering

The second unusual feature of print streams is that none of their methods is declared to throw an I/O exception. Instead, what it does is implement an exception buffering pattern, only without an actual buffer.[9] In cases where other streams would raise an I/O exception, the print stream methods exit quietly, while at the same time setting an error flag. The status of this flag is available through the method `checkError()`, which returns `true` if a method has caught an I/O exception.

[8]It's not possible in Java to be both a `Writer` and a `InputStream` because these are both abstract base classes, and a class may only extend one other class. In retrospect, it'd be much more convenient if the streams were all defined as interfaces.

[9]This pattern is sometimes implemented so that the actual exceptions that were raised are accessible as a list.

5.16.3 Formatted Output

One of the most convenient aspects of print streams is their implementation of C-style printf() methods for formatted printing. These methods implicitly use an instance of Formatter in java.util to convert a sequence of objects into a string.

Format syntax is very elaborate, and for the most part, we restrict our attention to a few operations. The method's signature is printf(Locale,String,Object...), where the string is the formatting string and the variable-length argument Object... is zero or more objects to format. How the objects are formatted depends on the format string and the Locale (from java.util). The locale is optional; if it is not supplied, the default locale will be used. The locale impacts the printing of things like dates and decimal numbers, which are conventionally printed differently in different places. Although we are obsessive about maintaining unadulterated character data and work in many languages, a discussion of the intricacies of application internationalization is beyond the scope of this book.

For example, the sample program FileByteCount (see Section 5.10.4) uses string, decimal and hexadecimal formatting for strings and numbers,

```
System.out.printf("%4s %4s %10s\n","Dec","Hex","Count");
for (int i = 0; i < counts.length; ++i)
    if (counts[i] > 0L)
        System.out.printf("%4d %4h %10d\n",i,i,counts[i]);
```

The first printf() has the format string "%4s %4s %10s\n", which says to write the first string using 4 characters, the second string using 4 characters, and the third string using 10 characters; if they are too short, they align to the right, and if they are too long, they run over their alloted space. The second printf() uses %4d for a four place integer (right aligned), and %4h for a four-digit hexadecimal representation. Note that auto-boxing (see Section 2.2.6) converts the integers i and counts[i] to instances of Integer to act as inputs to printf(). Formatted prints do not automatically occupy a line; we have manually added UNIX-style line boundary with the line feed character escape \n.

Finally, note that the two printf() statements intentionally use the same width so the output is formatted like a table with headings aligned above the columns.

5.17 Standard Input, Output, and Error Streams

In many contexts, the environment in which a program is executed provides streams for input, output, and error messages. These streams are known as standard input, standard output, and standard error. Standard UNIX pipelines direct these streams in two ways. They can direct the standard input to be read from a file or the standard output or error to be written to a file. Second, they can pipe processes together, so that the standard output from one process is used for the standard input of the next process. Shell scripts and scripting languages such as Python manage standard input and output from processes in a similar way to shells.[10]

5.17.1 Accessing the Streams

Java provides access to these streams through three static variables in the class `System` in the package `java.lang`. The variable `System.in` is of type `InputStream` and may be used to read bytes from the shell's standard input. The standard and error output streams, `System.out` and `System.err`, are provided constants of type `PrintStream` (see Section 5.16 for more information on print streams). The standard output and error print streams use the platform's default encoding (see below for how to reset this).

5.17.2 Redirecting the Streams

The standard input and output streams may be reset using the methods `setIn(InputStream)`, `setOut(PrintStream)` and `setErr(PrintStream)`. This is convenient for writing Java programs that act like Python or shell scripts.

The mutability of the output and error streams enables a Java program to reset their character encodings. For example, to reset standard output to use UTF-8, insert the following statement before any output is produced,

```
System.setOut(new PrintStream(System.out,true,Strings.UTF8));
```

The value `true` enables auto-flushing. The new standard output wraps the old standard output in a new print stream with the desired encoding.

The output streams may be similarly redirected to files by using a `PrintStream` constructed from a file in the same way. For example,

```
System.setErr(new PrintStream("stdout.utf8.txt","UTF-16BE"));
```

sends anyting the program writes to the standard error stream to the file `stdout.utf8.txt`, using the UTF-16BE character encoding to convert characters to bytes.

The standard input stream may be wrapped in an `InputStreamReader` or `BufferedReader` to interpret byte inputs as characters. Standard output and error may be wrapped in the same way, which is an alternative to resetting the streams.

5.17.3 UNIX-like Commands

Commands in UNIX are often defined to default to writing to the standard output if a file is not specified. This idiom is easy to reconstruct for Java commands. Suppose we have a variable `file` that is initialized to a file or `null` if the command is to write an output

[10]Java provides an implementation of Unix-style pipes through piped streams and readers and writers. These are based on the classes `PipedInputStream`, `PipedOutputStream`, `PipedReader`, and `PipedWriter` in the standard `java.io` package. These classes are multi-threaded so that they may asynchronously consume the output of one process to provide the input to the next, blocking if need be to wait for a read or write.

```
PrintStream out
     = (file != null)
     ? new PrintStream(new FileOutputStream(file))
     : System.out;
```

If the file is non-null, an output stream for it is wrapepd in a print stream, otherwise the standard output is used. The same can be done for input.

5.18 URIs, URLs and URNs

A uniform resource identifier (URI) picks out something on the web by name. This is a very general notion, but we will only need a few instances in this book. The specifications for these standards are quite complex, and beyond the scope of this book.

5.18.1 Uniform Resource Names

There are two (overlapping) subtypes of URIs, based on function. A uniform resource name (URN) is a URI that provides a name for something on the web, but may not say how or where to get it. A typical instance of a URN is the Digital Object Identifier (DOI) system, which is becoming more and more widely used to identify text resources such as journal articles. For example, the DOI `doi:10.1016/j.jbi.2003.10.001` picks out a paper on which my wife was a co-author. An alternative URN for the same paper is `PMID:15016385`, which uses the PubMed identifier (PMID) for the paper.

5.18.2 Uniform Resource Locators

A uniform resource locator (URL) provides a means of locating (a concrete instance of) an item on the web. A URL indicates how to get something by providing a scheme name, such as `file://` for files or `http://` for the hypertext transport protocol (HTTP). For example, the URL `http://dx.doi.org/10.1016/j.jbi.2003.10.001` is for the same paper as above, but specifies a web protocol, HTTP, and host, `dx.doi.org` from which to fetch a specific embodiment of the resource. There may be many URLs that point to the same object. For instance, `http://linkinghub.elsevier.com/retrieve/pii/S1532046403001126` is a URL for the same paper, which links to the publisher's site.

5.18.3 URIs and URLs in Java

The standard `java.net` package has a class `URI` for URIs and a class `URL` for URLs. Both of them may be constructed using a string representation of the resource. Once constructed, the classes provide a range of structured access to the fields making up the resource. For instance, both objects have methods such as `getHost()` and `getPort()`, which may return proper values or indicators like `null` or -1 that there is no host or port.

The demo program `UrlProperties` constructs a URL from a command-line argument and prints out its available properties. The beginning of the `main()` method is

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

     String url = args[0];
     URL u = new URL(url);
     System.out.println("getAuthority()=" + u.getAuthority());
```

The rest of the program just prints out the rest of the properties. The `main()` method throws a `MalformedURLException`, a subclass of `IOException`, if the URL specified on the command line is not well formed. This ensures that once a URL is constructed, it is well formed. There are no exceptions to catch for the get methods; they operate only on the string representations and are guaranteed to succeed if the URL was constructed.

The Ant target `url-properties` calls the command supplying as an argument the value of the property `url.in`.

```
>              ant -Durl.in="http://google.com/webhp?q=dirichlet"
url-properties

getAuthority()=google.com
getDefaultPort()=80
getFile()=/webhp?q=dirichlet
getHost()=google.com
getPath()=/webhp
getPort()=-1
getProtocol()=http
getQuery()=q=dirichlet
getRef()=null
getUserInfo()=null
toExternalForm()=http://google.com/webhp?q=dirichlet
toString()=http://google.com/webhp?q=dirichlet
```

Looking over the output shows how the URL is parsed into components such as host, path and query.

5.18.4 Reading from URLs

URLs know how to access resources. These resources may reside locally or remotely. Wherever they arise, and whatever their content, resources are all coded as byte streams. Java provides built-in support for protocols `http`, `https`, `ftp`, `file`, and `jar`.

Once a URL is created, the method `openStream()` returns an `InputStream` that may be used to read the bytes from the URL. Depending on the URL itself, these bytes may be from a local or networked file, a local or remote web server, or elsewhere. Opening the stream may throw an I/O exception, as a connection is opened to the URL. Once the input stream is opened, it may be used like any other stream.

The sample program ReadUrl shows how to read the bytes from an URL and relay them to standard output.

```
URL u = new URL(url);
InputStream in = u.openStream();

byte[] buf = new byte[8096];
int n;
while ((n = in.read(buf)) >= 0)
     System.out.write(buf,0,n);

in.close();
```

The code between the stream creation and its closing is just a generic buffered stream copy.

The Ant target read-url is configured to call the sample program using as URL the value of the property url.in.

```
> ant -Durl.in=http://lingpipe.com url-read

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html ...
<html ...
<head>
<title>LingPipe Home</title>
<meta http-equiv="Content-type"
      content="application/xhtml+xml; charset=utf-8"/>
...
pageTracker._trackPageview();
} catch(err) {}</script></body>
</html>
```

As with most pages these days, LingPipe's home page starts with a title, content type, and cascading stylesheet (CSS) declaration, and ends with some Javascript. Whether you can see all of the output of a page depends on your shell; you can save the output to a file and then view it in a browser.

A file URL may be used as well as an HTTP-based URL. This allows an abstraction over the location of the bytes being read.

5.18.5 URL Connections

The output stream method we used above, getOutputStream(), is just a convenience method for calling the openConnection() method to get an instance of URLConnection and calling its getOutputStream() method. URL connections allow various properties of the connection to be inspected and set.

Although the URL method openConnection() is declared to return a URLConenction, if the URL uses the HTTP protocol, the documentation specifies that a HttpURLConnection will be returned. Thus the URLConnection may be cast to `HttpURLConnection` at run time. The HTTP-based connection provides additional methods for manipulating HTTP headers and responses, such as content types, last-modified dates, character encodings, handling redirected web pages, fetching HTTP respose codes, managing timeouts and keep-alive times for the connection, and other intricacies of HTTP management.

A URL connection may also be used to write to a URL. This makes it possible to attach data for a POST-based request and thus implement general web services.

5.19 The Input Source Abstraction

The class `InputSource`, in standard Java package `org.xml.sax`, provides an input abs`tractionforXMLdocuments` with a rather awkward mutable implementation. An input source specifies the source of its input as a URL, an input stream, or a reader.

There is a constructor for each mode of input specification. `InputSource(String)` may be used with a system identifier in the form of a URL, and `InputSource(InputStream)` and `InputSource(Reader)` for stream data. Although input sources are not immutable, we recommend using a detailed constructor rather than a simple constructor followed by setters.

There is also a no-argument constructor `InputSource()` and setters for each of the input types. The setters `setSystemId(String)`, `setByteStream(InputStream)` and `setCharacterStream(Reader)` correspond to the three contentful constructors.

The system ID may be used to resolve relative paths within an XML document. Thus setting a system ID may be useful even when a byte stream or a character stream has been specified.

Before using an input source, a character encoding may be specified using the `setEncoding(String)` method. A further awkwardness arises because this method only makes sense in conjunction with an input stream or URL; readers supply already converted `char` values.

It is up to the consumer of the input source to gather data from whatever source is specified. Typically, consumers will try the reader, then the input stream, then the URL. Because of the mutability of the class through the setters, an input source may have an input stream and a reader and a URL specified.

5.20 File Channels and Memory Mapped Files

As an alternative to streaming I/O, memory mapping essentially provides a random access interface for files. The method is by providing an array-like access abstraction on top of files. It is then up to the operating system's file manager to deal with actual commits to and reads from disk.

Memory mapped files are great for random access to data on disk. For instance, databases and search engines make heavy use of memory mapping.

Both the file input and file output stream classes implement a `getChannel()` method which returns an instance of `FileChannel`. File channels are part of the "new" I/O, rooted at package name java.nio, specifically in the package java.nio.channels.

File channels support reading and writing into ByteBuffer instances. Byte buffers and other buffers are in the java.nio package. Byte buffers may be wrapped to act as character buffers or other primitive type buffers. See Section 3.8 for a basic overview of buffers focusing on character buffers.

File channels may also be used to produce a MappedByteBuffer, which creates a memory-mapped file accessible as a ByteBuffer. The other way to create a file channel which may be used to produce a memory mapped buffer is through the RandomAccessFile class in java.io. The random access file class implements the DataInput and DataOutput interfaces, which provide utilities to write arbitrary primitives and strings in addition to bytes.

5.21 Object and Data I/O

Java provides built-in support for writing objects to streams through the classes ObjectInputStream and ObjectOutputStream in the package java.io. Object input and output streams implement the ObjectInput and ObjectOutput interfaces, which extend the DataInput and DataOutput interfaces. The object streams deal with writing objects and the data streams with writing primitive types and strings. Only objects with a runtime class that implements the Serializable interface in package java.io may be written or read. We first provide an example then discuss how to implement the Serializable interface.

5.21.1 Example of Object and Data I/O

The sample program ObjectIo provides an example of using object input and output streams to write objects and primitive objects and then reconstitute them. The main() method begins by writing the objects and primitive types,

```
ByteArrayOutputStream bytesOut = new ByteArrayOutputStream();
ObjectOutput out = new ObjectOutputStream(bytesOut);

out.writeObject(new File("foo"));
out.writeUTF("bar");
out.writeInt(42);

out.close();
```

Note that a byte array output stream is created and assigned to the appropriate variable. We use a byte array for convenience; it could've been a file output stream, a compressed file output stream, or an HTTP write in either direction. The output stream is then wrapped in an object output stream and assigned to a variable of the interface type, ObjectOutput. The object output is then used to write a file object using the writeObject() method from the ObjectOutput interface. Next, a string is written using the writeUTF(String) method from the DataOutput intrface. This does not write the string as an object, but using an efficient UTF-8-like encoding scheme. Lastly, an integer is written; the DataOutput interface provides write methods for all of the primitive types. Finally, the output stream is closed in the usual way through the `Closeable` interface method `close()`.

We then reverse the process to read the object back in,

```
byte[] bytes = bytesOut.toByteArray();
InputStream bytesIn = new ByteArrayInputStream(bytes);
ObjectInput in = new ObjectInputStream(bytesIn);

@SuppressWarnings("unchecked")
File file = (File) in.readObject();
String s = in.readUTF();
int n = in.readInt();

in.close();
```

First, we get the bytes that were written from the byte array output stream and use them to create a byte array input stream.[11] Then we wrap that in an object input stream, which we use to read the object, string and primitive integer value. Then we close the input stream. The result of the object read must be upcast to `File` because the `readObject()` method is declared to return `Object`.[12] Because the class may not be found or the class loader may fail to load the class for the object, the `readObject()` method is also declared to throw a `ClassNotFoundException`, which we simply declare on the `main()` method rather than handle. The result of running the program is

```
> ant object-io

file.getCanonicalPath()=C:\lpb\src\io\foo    s=bar    n=42
```

5.21.2 The `Serializable` Interface

The `Serializable` interface in package `java.io` is what's known as a marker interface. The marker pattern involves an interface, such as `Serializable`, which declares no methods. The interface is used to treat classes that implement it differently in some contexts. The object input and output classes use the `Serializable` marker interface to determine if a class can be serialized or not.

When a class is serialized, its fully qualified path name is written out as if by the `writeUTF()` method. A 64-bit serial version identifier is also written. Then data representing the state of the class is written.

All that is strictly necessary to make a class serializable is to declare it to implement the `Serializable` interface, although it must also meet a few additional conditions described below. If nothing else is specified, the serial version identifier will be computed from the class's signature, including the signature of its parents and implemented interfaces. If nothing else is done, the state of the

[11]Note that we assign variables to the most general type possible; the input stream had to be assigned to `ByteArrayInputStream` because we needed to call the `toByteArray()` method on it.

[12]The `SuppressWarnings` annotation is used to suppress the unchecked warning that would otherwise be generated by the compiler for the runtime cast. Although the cast can't fail in this context, because we know what we wrote, it may fail in a runtime context if the expected input format is not received.

class is serialized by serializing each of its member objects and writing each of its primitive member variables, including all those inherited from parent classes.

The first condition for default serializability is that all of the class's member variables (including those of its superclasses) implement the `Serializable` interface or are declared to be `transient`. If that is the case, the object output method uses reflection to write out each object and primitive member variable in turn.

The second requirement on a class for default serializabiilty is that it must implement a public no-argument constructor. If that is the case, reconstituting an object will also happen by reflection, first calling the no-argument constructor, then reading and setting all the member variables.

The third requirement is that the same version of the class be used for reading and writing. There are two parts to this requirement. The first version requirement is that the signature of the class used to read in the serialized instance must be the same as that used to write the objects. If you change the member variables, method declarations, subclasses, etc., it will interfere with deserialization. The second version requirement is that the major version of Java being used to deserialize must be at least as great as that used to serialize. Thus it is not possible to serialize in Java version 6 and deserialize in version 5.

5.21.3 The Serialization Proxy Pattern

Java allows classes to take full control over their serialization and deserialization, which we take advantage of to overcome the limitations of default serialization, such as the restrictions on superclass behavior, backward compatibility with changing signatures, and the ability to create immutable objects without no-arg constructors.

Our recommended approach to serialization is to use the serialization proxy pattern, which we describe in this section by example, with an implementation in `SerialProxied` defined here. The class begins by declaring it implements `Serializable`, then definining two final member variables in a single two-argument constructor.

```
public class SerialProxied implements Serializable {

    private final String mS;
    private final int mCount;
    private final Object mNotSerializable = new Object();

    public SerialProxied(String s, int count) {
        mS = s;
        mCount = count;
    }
```

The finality of the variables mean that instances of the the class are immutable in the sense that once they are constructed they will never change. We use immutable classes wherever possible in LingPipe, checking during construction that they are well formed. Immutable classes are easy to reason about, especially in multi-threaded environments, because they are well formed when constructed and never change behavior.

The `writeReplace()` Method

Before applying default serialization, the object write methods first check, using reflection, if a class implements the `writeReplace()` method. If it does, the return result of calling it is serialized instead of the class itself.

The `SerialProxied` class implements the `writeReplace()` method by returning a new instance of the `Serializer` class, which we show below

```
private Object writeReplace() {
    return new Serializer(this);
}
```

Note that the method is declared to be private, so it will not appear in the documentation for the class and may not be called by clients. Reflection is still able to access the method even though it's private.

Serialization Proxy Implementation

The rest of the work's done in the class `Serializer`. We declare the class within the `SerialProxied` class as a static nested class, `Serializer`.

```
private static class Serializer implements Externalizable {

    SerialProxied mObj;

    public Serializer() { }

    Serializer(SerialProxied obj) {
        mObj = obj;
    }
```

The class itself is declared to be private and static. The full name of which would be `com.lingpipe.book.io.SerialProxied.Serializer`, but given its privacy, we only access it from within `SerialProxied` anyway.

As we will shortly see, the `Externalizable` interface it implements extends `Serializable` with the ability to take full control over what is written and read. The variables and methods within it are package protected, though they could also all be private. There are two constructors, a no-arg constructor which is required to be public for deserialization, and a one-argument constructor used by the `writeReplace()` method shown above for serialization.

The rest of the methods in the `Serializer` class are

```
public void writeExternal(ObjectOutput out)
    throws IOException {

    out.writeUTF(mObj.mS);
    out.writeInt(mObj.mCount);
}
```

```
public void readExternal(ObjectInput in) throws IOException {
     String s = in.readUTF();
     int count = in.readInt();
     mObj = new SerialProxied(s,count);
}

Object readResolve() {
     return mObj;
}
```

We describe them in turn below.

The `writeExternal()` Method

The first method is the `writeExternal()` method, which writes to the specified object output. In this case, it uses its within-class private access to retrieve and write the member variables of the `SerialProxied` object `mObj`.

The `readExternal()` Method

The next method is the `readExternal()` method. This method and its sister method, `writeExternal()` are specified in the `Externalizable` interface as public, so must be public here.[13]

The read method first reads the serialized string and integer from the specified object input. Then it uses these to construct a new `SerialProxied` instance based on the strnig and count and assigns it to the object variable `mObj`.

The `readResolve()` Method

Finally, the `readResolve()` method is used to return the object that was read in. This method is the deserialization counterpart to `writeReplace()`. After the `readExternal()` method is called, if `readResolve()` is defined, its return value is returned as the result of deserialization. Thus even though the `Serializer` is the one being serialized and deserialized, it is not visible at all to external clients.

Serial Version Variable

The final piece of the puzzle to take control over is the computation of the serial version identifier. This allows serialization to bypass the computation of this identifier by inspecting the serialized object's signature by reflection. This is not only more efficient, it also allows the signatures to be changed without affecting the ability to deserialize already serialized instances.

If a class has already been fielded without declaring a serial version ID, the command-line program `serialver`, which is shipped with the Sun/Oracle JDK, may be used to compute the value that would be computed by reflection during serialization. It only needs the classpath and full name of the class.

[13]The `readExternal()` method is also declared to throw a `ClassNotFoundException` to deal with cases when it fails to deserialize an object using the object input's `readObject()` method. We don't need it here because we are only using the data input methods `readUTF()` and `readInt()`.

```
> serialver -classpath build/lp-book-io-4.0.jar
com.lingpipe.book.io.SerialProxied
```

```
static final long serialVersionUID = -688378786294424932L;
```

The output is a piece of Java code declaring a constant `serialVersionUID` with the current value of the identifier as computed by serialization. We just insert that into the class `SerialProxied` class. Technically, we may use whatever number we like here; it's only checked for consistency during serialization.

We also need to insert the version ID constant into the nested static class `SerialProxied.Serializer`.

5.22 Lingpipe I/O Utilities

LingPipe provides a number of utilities for input and output, and we discuss the general-purpose ones in this section.

5.22.1 The `FileLineReader` Class

To deal with line-based file formats, LingPipe provides the class `FileLineReader`, in the `com.aliasi.io` package. This class extends `LineNumberReader` in `java.io`, which itself extends `BufferedReader`, so all the inherited methods are avaiable.

For convenience, the `FileLineReader` class implements the `Iterable<String>` interface, with `iterator()` returning an `Iterator<String>` that iterates over the lines produced by the `readLine()` method inherited from `BufferedReader`. Being iterable enables for-loop syntax, so that if `file` is an instance of `File`, the standard idiom for reading lines is

```
FileLineReader lines = new FileLineReader(file,"UTF-8");
for (String line : lines) {
     // do something
}
lines.close();
```

The ellided code block will then perform some operation on each line.

There are also static utility methods, `readLines()` and `readLinesArray()`, both taking a file and character encoding name like the constructor, to read all of the lines of a file into either an array of type `String[]` or a list of type `List<String>`.

5.22.2 The `Files` Utility Class

Some standard file operations are implemented as static methods in the `Files` class from the package `com.aliasi.util`.

File Name Parsing

The `baseName(File)` and `extension(File)` methods split a file's name into the part before the last period `(.)` and the part after the alst period. If there is no last period, the base name is the entire file name.

File Manipulation

The method `removeDescendants(File)` removes all the files contained in the specified directory recursively. The `removeRecursive(File)` method removes the descendants of a file and the file itself. The `copy(File,File)` method copies the content of one file into another.

Bulk Reads and Writes

There are methods to read bytes, characters, or a string from a file, and corresponding methods to write. The methods all pass on any I/O exceptions they encounter.

The method `readBytesFromFile(File)` returns the array of bytes corresponding to the file's contents, whereas `writeBytesToFile(byte[],File)` writes the specified bytes to the specified file.

The methods `readCharsFromFile(File,String)` and `writeCharsToFile(char[],File,String)` do the same thing for `char` values. These methods both require the character encoding to be specified as the final argument. There are corresponding methods for strings, `readFromFile(File,String)` and `writeStringToFile(String,File,String)`. Together, we can use these methods to transcode from one encoding to another,

```
char[] cs = Files.readCharsFromFile(fileIn,encIn);
Files.writeCharsToFile(cs,fileOut,encOut);
```

5.22.3 The `Streams` Utility Class

The class `Streams` in the package `com.aliasi.io` provides static utility methods for dealing with byte and character streams.

Quiet Close Methods

The method `closeQuietly(Closeable)` closes the specified closeable object, catching and ignoring any I/O exceptions raised by the close operation. If the specified object is null, the quiet close method returns.

The method `closeInputSource(InputSource)` closes the byte and character streams of the specified input source quietly, as if by the `closeQuietly()` method.

Bulk Reads

There are utility bulk read methods that buffer inputs and return them as an array. Special utility methods are needed because the amount of data available or read from a stream is unpredictable. Bulk writing, on the other hand, is built into output streams and writers themselves.

The bulk read method `toByteArray(InputStream)` reads all of the bytes from the input stream returning them as an array. The method `toCharArray(Reader)` does the same for `char` values.

The method `toCharArray(InputSource)` reads from an input source, using the reader if available, and if not, the input stream using either the specified character encoding or the platfrom default, and barring that, reads from the input source's system identifier (URL).

The method `toCharArray(InputStream,String)` reads an array of `char` values by reading bytes from the specified input stream and converts them to `char` values using the specified character encoding.

Copying Streams

The method `copy(InputStream,OutputStream)` reads bytes from the specified input stream until the end of stream, writing what it finds to the output stream. The `copy(Reader,Writer)` does the same thing for `char` value streams.

5.22.4 The `Compilable` Interface

LingPipe defines the interface `Compilable` in the package `com.aliasi.util` for classes that define a compiled representation.

The `Compilable` interface has a single method, `compileTo(ObjectOutput)`, used for compiling an object to an object output stream. Like all I/O interface methods, it's defined to throw an `IOException`. The method is used just like the `writeExternal(ObjectOutput)` method from Java's `Externalizable` interface. It is assumed that only whatever is written, deserialization will produce a single object that is the compiled form of the object that was compiled.

Many of LingPipe's statistical models, such as language models, classifiers and HMMs have implementations that allow training data to be added incrementally. The classes implementing these models are often declared to be both `Serializable` and `Compilable`. If they are serializable, the standard serialization followed by deserialization typically produces an object in the same state and of the same class as the one serialized. If a model's compilable, a compilation followed by deserialization typically produces an object implementing the same model more efficiently, but without the ability to be updated with new training data.

5.22.5 The `AbstractExternalizable` Class

LingPipe provides a dual-purpose class `AbstractExternalizable` in the package `com.aliasi.util`. This class provides static utility methods for dealing with serializable and compilable objects, as well as supporting the serialization proxy pattern.

Base Class for Serialization Proxies

The abstract class `AbstractExternalizable` in package `com.aliasi.util` provides a base class for implementing the serializaion proxy pattern, as described in Section 5.21.3. It is declared to implement `Externalizable`, and the `writeExternal(ObjectOutput)` method from that interface is declared to be abstract and thus must be implemented by a concrete subclass. The method `readExternal(ObjectInput)` is implemented by `AbstractExternalizable`, as is the serialization method `readResolve()`. The other abstract method that must be implemented is `read(ObjectInput)`, which returns an object. This method is called by `readExternal()` and the object it returns stored to be used as the return value for `readResolve()`.

To see how to use this class as the base class for serialization proxies, we provide an example that parallels the one we implemented directly in Section 5.21.3. This time, we use a nested inner class that extends `AbstractExternalizable`.

The class is constructed as follows.

```
public class NamedScoredList<E> implements Serializable {

     static final long serialVersionUID = -3215750891469163883L;

     private final List<E> mEs;
     private final String mName;
     private final double mScore;
     private final Object mNotSerializable = new Object();

     public NamedScoredList(List<E> es, String name,
                            double score) {
         mEs = new ArrayList<E>(es);
         mName = name;
         mScore = score;
     }
```

Note that the class is defined with a generic and declared to implement Java's `Serializable` interface. Also note the explicit declaration of the serial version UID. The constructor stores a copy of the input list, the string name, and the double score, as well as a new object which is not serializable itself.

The two features making this class immutable also prevent it from being serializable. First, there is no zero-argument constructor. Second, the member variables are declared to be final. Adding a public zero-arg constructor and removing the final declarations would allow the class to be serialized in place.

Instead, we use the `writeReplace()` method as before to return the object's serialization proxy, which is also declared to be generic and provided with the object itself through `this`.

```
     Object writeReplace() {
         return new Externalizer<E>(this);
     }
```

The class `Externalizer` is declared as a nested static class as follows.

```
static class Externalizer<F> extends AbstractExternalizable {
     static final long serialVersionUID = 1576954554681604249L;
     private final NamedScoredList<F> mAe;
     public Externalizer() { this(null); }
     public Externalizer(NamedScoredList<F> ae) {
         mAe = ae;
     }
```

Note that the generic argument is different; they can't be the same because this is a nested static class, not an inner class, so it doesn't have access to the containing class's generic. It is defined to extend the utility class `AbstractExternalizable`. It also defines its own serial version UID. It holds a final reference to a `NamedScoredList` for which it is the serialization proxy. There is the requisit public no-argument constructor, as well as a constructor that sets the member variable to its argument. The no-arg constructor is used for deserialization and the one-arg constructor for serialization.

The helper class requires two abstract methods to be overridden. The first is for writing the object, and gets called as soon as `writeReplace()` creates the proxy.

```
     @Override
     public void writeExternal(ObjectOutput out)
         throws IOException {

         out.writeObject(mAe.mEs);
         out.writeUTF(mAe.mName);
         out.writeDouble(mAe.mScore);
     }
```

This uses the contained instance `mAe` to access the data that needs to be written. Here, it's an object, a string, and a double, each with their own write methods. The class is defined to throw an I/O exception, and will almost always require this declaration because the write methods throw I/O exceptions.

The second overridden method is for reading. This just reverses the writing process, returning an object.

```
     @Override
     public Object read(ObjectInput in)
         throws IOException, ClassNotFoundException {

         @SuppressWarnings("unchecked")
         List<F> es = (List<F>) in.readObject();
         String s = in.readUTF();
         double x = in.readDouble();
         return new NamedScoredList<F>(es,s,x);
     }
}
```

We read in the list, using the generic on the class itself for casting purposes. We then read in the string and double with the matching read methods on object inputs. Finally, we return a new named scored list with the same generic. This object is what's finally returned on deserialization.

The `main()` method for the demo just creates an object and then serializes and deserializes it using the utility method.

```
List<String> xs = Arrays.asList("a","b","c");
NamedScoredList<String> p1
     = new NamedScoredList<String>(xs,"foo",1.4);

@SuppressWarnings("unchecked")
NamedScoredList<String> p2 = (NamedScoredList<String>)
     AbstractExternalizable.serializeDeserialize(p1);
```

This may be run using the Ant target `ae-demo`.

```
> ant ae-demo

p1=NSL([a, b, c], foo, 1.4)     p2=NSL([a, b, c], foo, 1.4)
```

Serializing Singletons

This class makes it particularly easy to implement serialization for a singleton. We provide an example in `SerializedSingleton`. Following the usual singleton pattern, we have a private constructor and a public static constant,

```
public class SerializedSingleton implements Serializable {

     private final String mKey;
     private final int mValue;

     private SerializedSingleton(String key, int value) {
         mKey = key;
         mValue = value;
     }

     public static final SerializedSingleton INSTANCE
         = new SerializedSingleton("foo",42);
```

We then have the usual definition of `writeReplace()`, and a particularly simple implementation of the serialization proxy itself,

```
     Object writeReplace() {
         return new Serializer();
     }

     static class Serializer extends AbstractExternalizable {

         public Serializer() { }

         @Override
         public Object read(ObjectInput in) {
             return INSTANCE;
```

```
        }

        @Override
        public void writeExternal(ObjectOutput out) { }

        static final long serialVersionUID
            = 8538000269877150506L;
    }
```

If the nested static class had been declared public, we wouldn't have needed to define the public no-argument constructor. The version ID is optional, but highly recommended for backward compatibility.

To test that it works, we have a simple `main()` method,

```
SerializedSingleton s = SerializedSingleton.INSTANCE;
Object deser = AbstractExternalizable.serializeDeserialize(s);
boolean same = (s == deser);
```

which serializes and deserializes the instance using the LingPipe utility in `AbstractExternalizable`, then tests for reference equality.

We can run this code using the Ant target `serialized-singleton`, which takes no arguments,

```
> ant serialized-singleton

same=true
```

Serialization Proxy Utilities

The `AbstractExternalizable` class defines a number of static utility methods supporting serialization of arrays. These methods write the number of members then each member of the array in turn. For instance, `writeInts(int[],ObjectOutput)` writes an array of integers to an object output and `readInts(ObjectInput)` reads and returns an integer array that was serialized using the `writeInts()` method. There are similar methods for `double`, `float`, and `String`.

The method `compileOrSerialize(Object,ObjectOutput)` will attempt to compile the specified object if it is compilable, otherwise it will try to serialize it to the specified object output. It will throw a `NotSerializableException` if the object is neither serializable nor compilable. The method `serializeOrCompile(Object,ObjectOutput)` tries the operations in the opposite order, compiling only if the object is nor serializable.

General Serialization Utilities

The `AbstractExternalizable` class defines a number of general-purpose methods for supporting Java's serialization and LingPipe's compilation.

There are in-memory versions of compilation and serialization. The method `compile(Compilable)` returns the result of compiling then deserializing an object. The method `serializeDeserialize(Serializable)` serializes and deserializes an object in memory. These methods are useful for testing serialization and compilation. These operations are carried out in memory, which requires enough capacity to store the original object, a byte array holding the serialized or compiled bytes, and the deserialized or compiled object itself. The same operation may be performed with a temporary file in much less space because the original object may be garbage collected before deserialization and the bytes reside out of main memory.

There are also utility methods for serializing or compiling to a file. The method `serializeTo(Serializable,File)` writes a serialized form of the object to the specified file. The `compileTo(Compilable,File)` does the same thing for compilable objects.

To read serialized or compiled objects back in, the method `readObject(File)` reads a serialized object from a file and returns it.

The method `readResourceObject(String)` reads an object from a resource with the specified absolute path name (see Section 5.15.2 for information on resources including how to create them and put them on the classpath). This method simply creates an input stream from the resource then wraps it in an object input stream for reading. The `readResourceObject(Class<?>,String)` method reads a (possibly relative) resource relative to the specified class.

5.22.6 Demos of Serialization and Compilation

In the demo classes `CompilationDemo` and `SerializationDemo`, we provide examples of how to serialize and deserialize a class. Both classes are based on a simple `main()` command-line method and both use the same object to serialize and compile.

Object Construction

Suppose we have an instance of a class that's compilable, like `TrainSpellChecker` in LingPipe's `spell` package. We will assume it is constructed using an appropriate language model `lm` and weighted edit distance `dist`, using the following code at the beginning of the main method of both demo classes.

```
TrainSpellChecker tsc
     = new TrainSpellChecker(lm,dist);
tsc.handle("The quick brown fox jumped.");
```

This code also provides it with a training example so it's not completely empty later (not that being empty would cause problems).

Serialization Demo

To serialize a file directly, we can use the following.

```
File fc1 = new File("one.CompiledSpellChecker");
OutputStream out = new FileOutputStream(fc1);
ObjectOutput objOut = new ObjectOutputStream(out);
```

```
objOut.writeObject(tsc);
objOut.close();

InputStream in = new FileInputStream(fc1);
ObjectInput objIn = new ObjectInputStream(in);
Object o1 = objIn.readObject();
TrainSpellChecker tsc1 = (TrainSpellChecker) o1;
objIn.close();
```

We first create a file input stream, then wrap it in an object output stream, in both cases assigning to more general variables. We then call `writeObject()` on the object output with the object to be serialized as an argument. This is how all serialization works in Java at base. If the object being passed in is not serializable, an `NotSerializableException` will be raised. We then close the object output stream to ensure all bytes are written.

The complementary set of calls reads the object back in. We create an input stream using the same file, wrap it in an object input, then call the `readObject()` method on the object input. Finally, we cast to the appropriate class, `TrainSpellChecker`, and then close the object input.

The same thing can be accomplished with the static utility methods `serializeTo()` and `readObject()`, both in `AbstractExternalizable`.

```
File fc2 = new File("two.CompiledSpellChecker");
AbstractExternalizable.serializeTo(tsc,fc2);

Object o2 = AbstractExternalizable.readObject(fc2);
TrainSpellChecker tsc2 = (TrainSpellChecker) o2;
```

It is also possible to serialize and deserialize in memory, which is particularly useful for testing purposes. The static utility method `serializeDeserialize()` carries this out, as follows.

```
Object o3 = AbstractExternalizable.serializeDeserialize(tsc);
TrainSpellChecker tsc3 = (TrainSpellChecker) o3;
```

We can run the demo using the Ant target `serialize-demo`, as follows.

```
> ant serialize-demo

tsc.getClass()=class com.aliasi.spell.TrainSpellChecker
o1.getClass()=class com.aliasi.spell.TrainSpellChecker
o2.getClass()=class com.aliasi.spell.TrainSpellChecker
o3.getClass()=class com.aliasi.spell.TrainSpellChecker
```

The output shows the result of calling the general object method `getClass()` on the varialbes of type `Object` read back in. This approach works in general for inspecting the class of a deserialized object.

Compilation Demo

Assuming we have constructed a trained spell checker in variable `tsc` as shown above, we can use the following code to compile it.

```
File fc1 = new File("one.CompiledSpellChecker");
OutputStream out = new FileOutputStream(fc1);
ObjectOutput objOut = new ObjectOutputStream(out);
tsc.compileTo(objOut);
objOut.close();

InputStream in = new FileInputStream(fc1);
ObjectInput objIn = new ObjectInputStream(in);
Object o1 = objIn.readObject();
CompiledSpellChecker csc1 = (CompiledSpellChecker) o1;
objIn.close();
```

This involves creating an object output stream wrapping a file output stream, then calling `compileTo()` on the object being compiled with the object output as argument. Make sure to close the stream so that everything's written.

After creating the object, we reverse the process by creating an object input that wraps a file input stream. We then use the `readObject()` method on the object input to read an object, which we assign to variable `o1` of type `Object`. We then cast the object to a compiled spell checker and make sure to close the object input.

Exactly the same pair of operations can be carried out using the static utility methods `compileTo()` and `readObject()` with the following code.

```
File fc2 = new File("two.CompiledSpellChecker");
AbstractExternalizable.compileTo(tsc,fc2);

Object o2 = AbstractExternalizable.readObject(fc2);
CompiledSpellChecker csc2 = (CompiledSpellChecker) o2;
```

If you look at the implementation of these methods, they closely match what was written above, with the addition of buffering and more robust closes in finally blocks.

If you don't need an on-disk representation, the single method `compile()` does the same thing in memory, though with more robust closes.

```
Object o3 = AbstractExternalizable.compile(tsc);
CompiledSpellChecker csc3 = (CompiledSpellChecker) o3;
```

If we run the demo, using the `compile-demo` Ant target, we see that the objects read back in are compiled spell checkers in each case.

```
> ant compile-demo

tsc.getClass()=class com.aliasi.spell.TrainSpellChecker
o1.getClass()=class com.aliasi.spell.CompiledSpellChecker
o2.getClass()=class com.aliasi.spell.CompiledSpellChecker
o3.getClass()=class com.aliasi.spell.CompiledSpellChecker
```

Chapter 6

The Lucene Search Library

Apache Lucene is a search library written in Java. Due to its performance, configurability and generous licensing terms, it has become very popular in both academic and commercial settings. In this section, we'll provide an overview of Lucene's components and how to use them, based on a single simple hello-world type example.

Lucene provides search over documents. A document is essentially a collection of fields, where a field supplies a field name and value. Lucene manages a dynamic document index, which supports adding documents to the index and retrieving documents from the index using a highly expressive search API.

Lucene does not in any way constrain document structures. An index may store a heterogeneous set of documents, with any number of different fields which may vary by document in arbitrary ways. Lucene can store numerical and binary data as well as text, but in this tutorial we will concentrate on text values.

What actually gets indexed is a set of terms. A term combines a field name with a token that may be used for search. For instance, a title field like *Molecular Biology, 2nd Edition* might yield the tokens *molecul*, *biolog*, *2*, and *edition* after case normalization, stemming and stoplisting. The index structure provides the reverse mapping from terms, consisting of field names and tokens, back to documents. To search this index, we construct a term composed of the field title and the tokens resulting from applying the same stemming and stoplisting to the text we are looking for.

A Lucene search takes a query and returns a set of documents that are ranked by relevancy with documents most similar to the query having the highest score. Lucene's search scoring algorithm weights results using TF-IDF, term frequency-inverse document frequency. *Term frequency* means that high frequency terms within a document have higher weight than do low-frequency terms. *Inverse document frequency* means that terms which occur frequently across many documents in a collection of documents are less likely to be meaningful descriptors of any given document in a corpus and are therefore down-weighted.

6.1 Fields

A document is a collection of fields. Search and indexing is carried out over these fields. All fields in Lucene are instances of the `Fieldable` interface in the package `org.apache.lucene.document`. This interface is implemented by the abstract class `AbstractField` and the two final classes `Field` and `NumericField` which inherit from it. In this tutorial we cover the use of the class `Field` to index and store text.

For example, a MEDLINE citation might be stored as a series of fields: one for the name of the article, another for name of the journal in which it was published, another field for the authors of the article, a pub-date field for the date of publication, a field for the text of the article's abstract, and another field for the list of topic keywords drawn from Medical Subject Headings (MeSH). Each of these fields would get a different name, and at search time, the client could specify that it was searching for authors or titles or both, potentially restricting to a date range and set of journals by constructing search terms for the appropriate fields and values.

6.1.1 Constructing Fields

A field requires all of its components to be specified in the constructor. Even so, fields are defined to be mutable so that their values, but not their field names, may be reset after construction.

The field constructor takes the field name, value, and a set of flags which specify how it will be saved in the index. These indexing flags are discussed in a subsequent section.

There are also several utility constructors that provide default values for the flags in addition to those taking the text value as a `Reader`. There is also a public constructor that takes a `TokenStream` (see Section 6.3) rather than a string.[1] An additional constructor takes a boolean flag controlling whether the field's name is interned or not (see Section 3.6.11), with the default setting being `true`.

6.1.2 Field Names and Values

Each constructor for a field requires a name for the field. At search time, the supplied field name restricts the search to particular fields.

Each constructor for a field requires a value for the field which may be supplied as a Java `String` or `Reader`.[2] The value for a binary field is supplied as a byte array slice.

[1]According to the javadoc, this is useful for pre-analyzed fields. Users must be careful to make sure the token stream provided is consistent with whatever analysis will happen at query time.

[2]We recommend not using a `Reader`, because the policy on closing such readers is confusing. It's up to the client to close, but the close can only be done after the document has been added to an index. Making fields stateful in this way introduces a lifecycle management problem that's easily avoided. Very rarely will documents be such that a file or network-based reader may be used as is in a field; usually such streams are parsed into fields before indexing, eliminating any performance advantage readers might have.

6.1.3 Indexing Flags

As of version 3.0 of Lucene, the constructor for a field has three arguments that control how a term is indexed and/or stored. These arguments are specified using nested enum instances in the `Field` class.

The `Field.Store` Enum

All fields are marked as to whether their raw value is stored in the index or not. Storing raw values allows you to retrieve them at search time, but may consume substantial space.

The enum `Field.Store` is used to mark whether or not to store the value of a field. Its two instances, `Store.YES` and `Store.NO`, have the obvious interpretations.

The `Field.Index` Enum

All fields are also marked as to whether they are indexed or not. A field must be indexed in order for it to be searchable. While it's possible to have a field that is indexed but not stored, or stored but not indexed, it's pointless to have a field that is neither stored nor indexed.

Whether a field is indexed is controlled by an instance of the `Field.Index` enum. The value `Index.NO` turns off indexing for a field. The other values all turn on indexing for a field. Where they vary is how the terms that are indexed are pulled out of the field value. The value `Index.ANALYZED` indexes a field with tokenization turned on (see Section 6.3). The value `Index.NOT_ANALYZED` causes the field value to be treated like a single token; it's primarily used for identifier fields that are not decomposed into searchable terms.

Lucene's default behavior is to compute term frequency the proportion of the number of times a term occurs as opposed to a simple frequency count. It does this by storing a normalizing factor for each field that is indexed. The values `Index.ANALYZED_NO_NORMS` and `Index.NOT_ANALYZED_NO_NORMS` disable storage of these normalizing factors. This results in less memory usage during search, but will affect search results. Furthermore, in order for this to be effective, this must be turned off during indexing for all documents in the index.

The `Field.TermVector` Enum

The final specification on a field is whether to store term vectors or not, and if they are stored, what specifically to store. A term vector stores a list of the document's terms and number of occurrences in that document. Term vectors can also store token position. They may be useful for downstream processing like results clustering, finding documents that are similar to a known document, or document highlighting.

Whether to use term vectors is controlled by an instance of the enum `Field.TermVector`. The default is not to store term vectors, corresponding to value `TermVector.NO`. Because we do not need term vectors for our simple demo, we use a constructor for `Field` which implicitly sets the term vector flag to `TermVector.NO`.

6.1.4 Field Getters

Once we have a field, we can access the components of it such as its name, value, whether its indexed, stored, or tokenized, and whether term vectors are stored. These methods are all specified in the `Fieldable` interface. For instance, `name()` returns a field's name, and `stringValue()` its value.[3]

There are convenience getters derived from the flag settings. For instance, `isIndexed()` indicates if the field is indexed, and `isTokenized()` indicates whether the indexing involved analysis of some sort. The method `isStored()` indicates if the value is stored in the index, and `isTermVectorStored()` whether the term vector is stored in the index.

6.2 Documents

In Lucene, documents are represented as instances of the final class `Document`, in package `org.apache.lucene.document`.

6.2.1 Constructing and Populating Documents

Documents are constructed using a zero-arg constructor `Document()`. Once a document is constructed, the method `add(Fieldable)` is used to add fields to the document.

Lucene does not in any way constrain document structures. An index may store a heterogeneous set of documents, with any number of different fields which may vary by document in arbitrary ways. It is up to the user to enforce consistency at the document collection level.

A document may have more than one field with the same name added to it. All of the fields with a given name will be searchable under that name (if the field is indexed, of course). The behavior is conceptually similar to what you'd get from concatenating all the field values; the main difference is that phrase searches don't work across the concatenated items.

6.2.2 Accessing Fields in Documents

The `Document` class provides a means of getting fields by name. The method `getFieldable(String)` returns the field for a specified name. If there's no field with the specified name, it returns `null` rather than raising an exception.

[3]For fields constructed with a `Reader` for a value, the method `stringValue()` returns `null`. Instead, the method `readerValue()` must be used. Similarly, the methods `tokenStreamValue()` and `binaryValue()` are used to retrieve values of fields constructed with token streams or byte array values. The problem with classes like this that that allow disjoint setters (or constructors) is that it complicates usage for clients, who now have to check where they can get their data. Another example of this anti-pattern is Java's built-in XML `InputSource` in package `org.xml.sax`.

The return type is Fieldable, but this interface provides nearly the same list of methods as Field itself, so there is rarely a need to cast a fieldable to a field.

If there is more than one field in a document with the same name, the simple method getFieldable(String) only returns the first one added. The method getFieldables(String) returns an array of all fields in a document with the given name. It's costly to construct arrays at run time (in both space and time for allocation and garbage collection), so if there is only a single value, the simpler method is preferable.

6.2.3 Document Demo

We provide a simple demo class, DocDemo, which illustrates the construction, setting and accessing of fields and documents.

Code Walkthrough

The main() method starts by constructing a document and populating it.

```
Document doc = new Document();
doc.add(new Field("title", "Fast and Accurate Read Alignment",
                  Store.YES,Index.ANALYZED));
doc.add(new Field("author", "Heng Li",
                  Store.YES,Index.ANALYZED));
doc.add(new Field("author", "Richard Durbin",
                  Store.YES,Index.ANALYZED));
doc.add(new Field("journal","Bioinformatics",
                  Store.YES,Index.ANALYZED));
doc.add(new Field("mesh","algorithms",
                  Store.YES,Index.ANALYZED));
doc.add(new Field("mesh","genomics/methods",
                  Store.YES,Index.ANALYZED));
doc.add(new Field("mesh","sequence alignment/methods",
                  Store.YES,Index.ANALYZED));
doc.add(new Field("pmid","20080505",
                  Store.YES,Index.NOT_ANALYZED));
```

After constructing the document, we add a sequence of fields, including a title field, two author fields, a field for the name of the journal, several fields storing mesh terms, and a field storing the document's PubMed identifier. These terms are all stored and analyzed other than the identifier, which is not analyzed.

After constructing the document, we loop over the fields and inspect them.

```
for (Fieldable f : doc.getFields()) {
     String name = f.name();
     String value = f.stringValue();
     boolean isIndexed = f.isIndexed();
     boolean isStored = f.isStored();
     boolean isTokenized = f.isTokenized();
     boolean isTermVectorStored = f.isTermVectorStored();
```

Note that the access is through the `Fieldable` interface. We include the calls to the relevant methods, but omit the actual print statements.

Running the Demo

The Ant target `doc-demo` runs the demo.

```
> ant doc-demo
```

```
name=title value=Fast and Accurate Read Alignment
     indexed=true store=true tok=true termVecs=false
name=author value=Heng Li
     indexed=true store=true tok=true termVecs=false
...
name=mesh value=genomics/methods
     indexed=true store=true tok=true termVecs=false
name=mesh value=sequence alignment/methods
     indexed=true store=true tok=true termVecs=false
name=pmid value=20080505
     indexed=true store=true tok=false termVecs=false
```

We've elided three fields, marked by ellipses.

6.3 Analysis and Token Streams

Lucene employs analyzers to convert the text value of a fields marked as analyzed to a stream of tokens. At indexing time, Lucene is supplied with an implementation of the abstract base class `Analyzer` in package `org.apache.lucene.analysis`. An analyzer maps a field name and text value to a `TokenStream`, also in the `analysis` package, from which the terms to be indexed are retrieved using an iterator-like pattern.

6.3.1 Token Streams and Attributes

Before version 3.0 of Lucene, token streams had a string-position oriented tokenization API, much like LingPipe's tokenizers. Version 3.0 generalized the interface for token streams and other basic objects using a very general design pattern based on attributes of other objects. [4]

Code Walkthrough

To see how the tokenization process works, we provide a sample class `LuceneAnalysis` that applies an analyzer to a field name and text input and prints out the resulting tokens. The work is done in a simple `main()` with two

[4]The benefit of this pattern is not in its use, which is less convenient than a direct implementation. The advantage of Lucene's attribute pattern is that it leaves enormous flexibility for the developers to add new features without breaking backward compatibility.

arguments, the field name, set as the string variable `fieldName`, and the text to be analyzed, set as the string variable `text`.

The first step is to create the analyzer.

```
StandardAnalyzer analyzer
     = new StandardAnalyzer(Version.LUCENE_36);
```

Here we've used Lucene's `StandardAnalyzer`, in package `org.apache.lucene.analysis.standard`, which applies case normalization and English stoplisting to the simple tokenizer, which pays attention to issues like periods and e-mail addresses.[5] Note that it's constructed with a constant for the Lucene version, as the behavior has changed over time.

The standard analyzer, like almost all of Lucene's built-in analyzers, ignores the name of the field that is passed in. Such analyzers essentially implement simple token stream factories, like LingPipe's tokenizer factories.[6]

The next step of the `main()` method constructs the token stream given the string values of the command-line arguments `fieldName` and `text`.

```
Reader textReader = new StringReader(text);

TokenStream tokenStream
     = analyzer.tokenStream(fieldName,textReader);

CharTermAttribute terms =
     tokenStream.addAttribute(CharTermAttribute.class);

OffsetAttribute offsets
     = tokenStream.addAttribute(OffsetAttribute.class);

PositionIncrementAttribute positions
     = tokenStream
     .addAttribute(PositionIncrementAttribute.class);
```

We first have to create a `Reader`, which we do by wrapping the input text string in a `StringReader` (from `java.io`).[7] Then we use the analyzer to create a token stream from the field name and text reader. The next three statements attach attributes to the token stream, specifically a term attribute,[8] offset attribute

[5]See the tokenization chapter in the companion volume, *Natural Language Processing in LingPipe* for an overview of natural language tokenization in LingPipe, as well as adapters between Lucene analyzers and LingPipe tokenizer factories).

[6]It is unfortunate that Lucene does not present a token stream factory interface to produce token streams from text inputs. Then it would be natural to construct an analyzer by associating token stream factories with field names. We follow this pattern in adapting LingPipe's tokenizer factories to analyzers in the companion volume, *Natural Language Processing with LingPipe*, in the section on adapting Lucene analyzers.

Lucene's sister package, Solr, which embeds Lucene in a client-server architecture, includes a token stream factory interface `TokenizerFactory`, which is very much like LingPipe's other than operating over readers rather than character sequences and providing Solr-specific initialization and configuration management.

[7]Unlike the case for documents, there is no alternative to using readers for analysis. It is common to use string readers because they do not maintain handles on resources other than their string reference.

[8]As of version 3.1, the `TermAttribute` class was renamed `CharTermAttribute` because it holds objects of type `CharSequence`, which can be printed by the `toString` method.

and position increment attribute. These are used to retrieve the text of a term, the span of the term in the original text, and the ordinal position of the term in the sequence of terms in the document. The position is given by an increment from the previous position, and Lucene uses these values for phrase-based search (i.e., searching for a fixed sequence of tokens in the given order without intervening material).

The last block of code in the `main()` method iterates through the token stream, printing the attributes of each token it finds.

```
while (tokenStream.incrementToken()) {
     int increment = positions.getPositionIncrement();
     int start = offsets.startOffset();
     int end = offsets.endOffset();
     String term = terms.toString();
```

The while loop continually calls `incrementToken()` on the token stream, which advances to the next token, returning `true` if there are more tokens. The body of the loop just pulls out the increment, start and end positions, and term for the token. The rest of the code, which isn't shown, just prints these values. this pattern of increment-then-get is particularly popular for tokenizers; LingPipe's tokenizers use a similar model.

Running the Demo

It may be run from the Ant target `lucene-analysis`, with the arguments provided by properties `field.name` and `text` respectively.

```
>    ant -Dfield.name=foo -Dtext="Mr. Sutton-Smith will pay $1.20
for the book." lucene-analysis
```

```
Mr. Sutton-Smith will pay $1.20 for the book.
012345678901234567890123456789012345678901234
0         1         2         3         4

 INCR (START,   END) TERM         INCR (START,   END) TERM
    1 (    0,     2) mr              2 (   22,    25) pay
    1 (    4,    10) sutton          1 (   27,    31) 1.20
    1 (   11,    16) smith           3 (   40,    44) book
```

The terms are all lowercased, and non-word-internal punctuation has been removed. The stop words *will*, *for* and *the* are also removed from the output. Unlike punctuation, when a stop word is removed, it causes the increment between terms to be larger. For instance, the increment between *smith* and *pay* is 2, because the stopword *will* was removed between them. The start (inclusive) and end (exclusive) positions of the extracted terms is also shown.

6.4 Directories

Lucene provides a storage abstraction on top of Java in the abstract base class `Directory` in the `org.apache.lucene.store` package. Directories provide an interface that's similar to an operating system's file system.

6.4.1 Types of Directory

The `FSDirectory` abstract base class, also in package `store`, extends `Directory` to support implementations based on a file system. This is the most common way to create a directory in Lucene. The implementation `RAMDirectory`, also in `store` supports in-memory directories, which are efficient, but less scalable than file-system directories. The package `org.apache.lucene.store` contains several specialized implementations.

6.4.2 Constructing File-System Directories

An instance of a file-system directory may be created using the factory method `FSDirectory.open(File)`, which returns an implementation of `FSDirectory`. As of Lucene 3.6, this method returns a specific `FSDirectory` implementation, based on your environment and the known limitations of each implementation.

At construction, all `FSDirectory` implementations are supplied with a File and a `LockFactory` object which specifies how the files on the file system will be locked. The `LockFactory` class is an abstract class. Several implementations are provided in the package `org.apache.lucene.store`. Convenience constructors supply a default `LockFactory`. As of Lucene 3.6, this is a `NativeFSLockFactory`.

6.5 Indexing

Lucene uses the `IndexWriter` class in `org.apache.lucene.index` to add documents to an index and optimize existing indexes. Documents do not all need to be added at once — documents may be added to or removed from an existing index. We consider deleting documents in Section 6.8.

6.5.1 Constructing an IndexWriter

An IndexWriter is constructed from a `lucene.store.Directory` and a `IndexWriterConfig` object which specifies the Lucene version of the index, the default Analyzer, and how the IndexWriter uses memory and processing resources during indexing. The `IndexWriterConfig` constructor takes the Lucene version and the default analyzer as arguments and sets its properties to default values accordingly. Getter and setter methods are used to query and update these properties.

6.5.2 Merging and Optimizing

Indexing maintains a small buffer of documents in memory, occasionally writing the data in that batch of documents out to disk. After enough such batches have been written to disk, Lucene automatically merges the individual batches into bigger batches. Then, after a number of larger chunks of index accumulate, these are merged. You can observe this behavior in your file browser if you're using a disk directory for indexing a large batch of documents. As more documents are added, small index segments are continually added. Then, at various points, these smaller indexes are merged into larger indexes.

It's possible to control the size of the in-memory buffer and the frequency of merges via setters on the `IndexWriterConfig`.

It's even possible to programmatically merge two indexes from different directories. Lucene's `IndexWriter` class provides a handy variable-argument-length method `addIndexes(IndexReader...)` which adds indexes maintained by the specified readers to the index writer. Then, when the writer is optimized, the indexes will all be merged into the writer's index directory.

Being able to merge indexes makes it easy to split the indexing job into multiple independent jobs and then either merge the results or use all of the indexes together without merging them (using a `MultiReader`).

6.5.3 Indexing Demo

We provide a demo class `LuceneIndexing` that shows how basic text indexing works.

Code Walkthrough

The work is all done in the `main()` method, whIch starts by constructing the index writer.

```
public static void main(String[] args)
     throws CorruptIndexException, LockObtainFailedException,
            IOException {

     File docDir = new File(args[0]);
     File indexDir = new File(args[1]);

     Directory fsDir = FSDirectory.open(indexDir);

     Analyzer stdAn
         = new StandardAnalyzer(Version.LUCENE_36);
     Analyzer ltcAn
         = new LimitTokenCountAnalyzer(stdAn,Integer.MAX_VALUE);

     IndexWriterConfig iwConf
         = new IndexWriterConfig(Version.LUCENE_36,ltcAn);
     iwConf.setOpenMode(IndexWriterConfig.OpenMode.CREATE);

     IndexWriter indexWriter
         = new IndexWriter(fsDir,iwConf);
```

The two arguments correspond to the directory from which documents to be indexed are read and the directory to which the Lucene index is written. We create a file-system-based directory using the index directory (see Section 6.4).

We then create a standard analyzer (see Section 6.3). In order to index all the text in a field, however long that field may be, we need to wrap the standard analyzer in a `LimitTokenCountAnalyzer` We set the maximum field length to `Integer.MAX_VALUE`, the largest possible value available.

Next we specify the config for the index writer. We call the `setOpenMode` method with the enum constant `IndexWriterConfig.OpenMode.CREATE` which causes the index writer to create a new index or overwrite an existing one. The other two possible open modes enums are `IndexWriterConfig.OpenMode.CREATE_OR_APPEND` which creates a new index if one does not exist, else opens an existing index and appends documents and `IndexWriterConfig.OpenMode.APPEND` which opens an existing index.

For both the standard analyzer and the index writer config we pass in a Lucene version constant.[9] Finally, we create an index writer from the directory and the index writer config.

Constructing the index may throw all three exceptions listed on the `main()` method. The first two exceptions are Lucene's, and both extend `IOException`. You may wish to catch them separately in some cases, as they clearly indicate what went wrong. A `CorruptIndexException` will be thrown if we attempt to open an index that is not well formed. A `LockObtainFailedException` will be thrown if the index writer could not obtain a file lock on the index directory. A plain-old Java `IOException` will be thrown if there is an underlying I/O error reading or writing from the files in the directory.

The second half of the `main()` method loops over the files in the specified document directory, converting them to documents and adding them to the index.

```
for (File f : docDir.listFiles()) {
     String fileName = f.getName();
     String text = Files.readFromFile(f,"ASCII");
     Document d = new Document();
     d.add(new Field("file",fileName,
                     Store.YES,Index.NOT_ANALYZED));
     d.add(new Field("text",text,
                     Store.YES,Index.ANALYZED));
     indexWriter.addDocument(d);
}
int numDocs = indexWriter.numDocs();

indexWriter.forceMerge(1);
indexWriter.commit();
indexWriter.close();
```

[9]The Lucene version constant supplied to components in an application can differ by component. For components used for both search and indexing it is critical that the Lucene version is the same in the code that is called at indexing time and the code that is called at search time.

We keep a count of the number of characters processed in the variable `numChars`. We then loop is over all the files in the specified document directory. For each file, we get its name and its text (using LingPipe's static `readFromFile()` utility method, which converts the bytes in the file to a string using the specified character encoding, here ASCII).

We then create a document and add the file name as an unanalyzed field and the text as an analyzed field. After creating the document, we call the `addDocument(Document)` method of the index writer to add it to the index.

After we've finished indexing all the files, we call the index writer's `forceMerge(int)` method, followed by `commit`. The argument to `forceMerge` is 1, so that the index will be merged down into a single segment, resulting in a smaller index with better search performance. This is a costly operation. The amount of free space required is two to three times the size of the directory that the index writer is opened on. When the merge completes, both the pre-merge and newly merged indexes exist. Because Lucene allows search to proceed independently of indexing, Lucene search components may have index readers open on the same directory. These search components will be operating on the index that existed when they were opened and will not see any changes made to the directory by the index writer until the call to `commit`, which syncs all referenced index files. At this point old indexes will be deleted, freeing up space.

We then close the index writer using the `close()` method, which may throw an `IOException`; the `IndexWriter` class is declared to implement Java's `Closeable` interface, so we could've used LingPipe's `Streams.closeSilently()` utility method to close it and swallow any I/O exceptions raised.

Finally, we get the number of documents that are in the current index using the method `numDocs()`; if documents were in the index when it was opened, these are included in the count. We also print out other counts, such as the number of characters, (print statements omitted from above code listing).

Running the Demo

The Ant target `lucene-index` runs the indexing demo. It supplies the values of properties `doc.dir` and `index.dir` to the program as the first two command-line arguments. In this example we will index the 85 *Federalist Papers*, and we will continue to use this index in subsequent examples in this tutorial.

The sample code includes the subdirectory `data/federalist-papers` which contains the Project Gutenberg distribution of the plain-text version of the*Federalist Papers* along with the shell script `get_papers.sh`. Running this script will create a directory called *texts* and populate it with the individual papers, one paper per file. Once unpacked, we use this directory as the `doc.dir` argument.

```
>                 ant -Ddoc.dir=../../data/federalist-papers/texts
-Dindex.dir=temp.idx lucene-index

Index Directory=/Users/mitzimorris/aliasi/lpbook/src/applucene/temp.idx
Doc Directory=/Users/mitzimorris/aliasi/lpbook/data/federalist-papers/texts
```

```
num docs=85
```

Lucene's very fast. On a workstation, it takes less than a second to run the demo, including forking a new JVM. On my wimpy notebook, it takes two seconds. The run indexed 85 documents consisting of approximately 1.1 million words total.

After indexing, we can look at the contents of the index directory, showing file size in kilobytes.

```
> export BLOCKSIZE=1024; ls -s temp.idx
```

```
total 1520
1144 _0.fdt
   4 _0.fdx
   4 _0.fnm
  84 _0.frq
   4 _0.nrm
 188 _0.prx
   4 _0.tii
  80 _0.tis
   4 segments.gen
   4 segments_1
```

These files contain binary representations of the index. The file segements.gen is a global file which contains the generation number of the index. The file segments_1 contains the per-commit list of segments.

The per-segment files begin with an underscore and have suffixes which identify the type of data they contain. The field info file, suffix `.fnm`, contains the field names and infos. The term dictionary files, `.tis` and `.tii`, are used to navigate the index to retrieve information for each term. Term frequencies are stored in the `.frq` file and term positions are stored in the `.prx` file. Normalization factors used for scoring are stored in the `.nrm` file. The files `.fdt` and `.fdx` contain the raw text data for the stored fields. These last two are not used by Lucene for index search and scoring, they are only for retrieval of search results.

The index takes up 1.5M total disk space. Most of this space is used for the stored fields file, 1.2M in size, which is slightly smaller than the raw text files themselves.

```
> du -sh ../../data/federalist-papers/texts
```

```
1.3M ../../data/federalist-papers/texts
```

6.5.4 Luke

Luke is an index browser for Lucene, also written in Java. Downloads are available from `http://code.google.com/p/luke/`. It provides a quick and easy way to explore the contents of the index. Figure 6.1 illustrates how Luke presents an overview of the contents of the index.

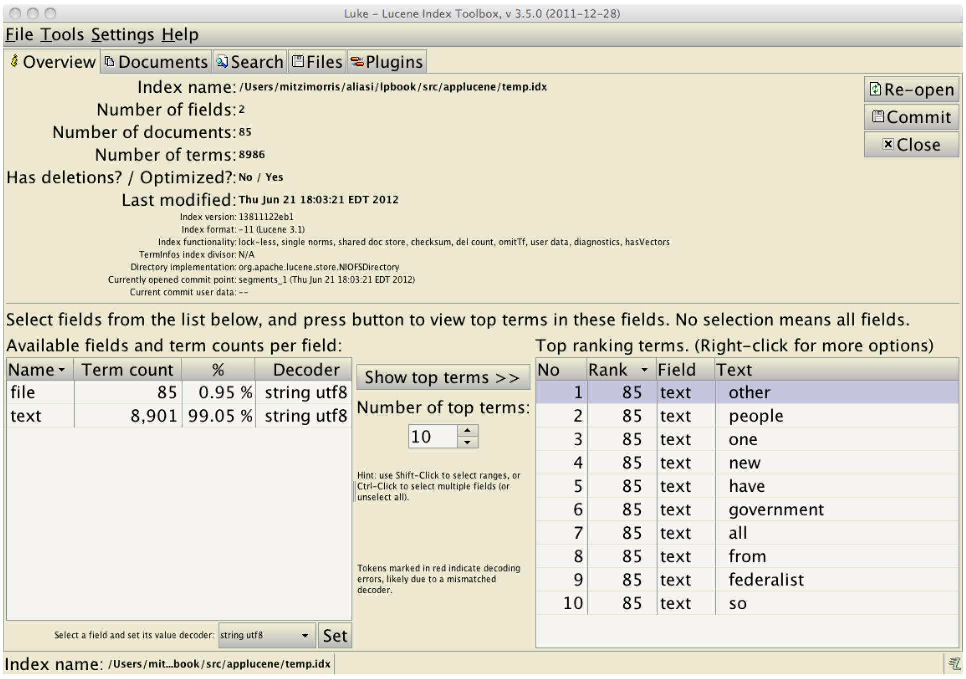
6.5.5 Duplicate Documents

If we were to run the demo program again, each of the documents would be added to the index a second time, and the number of documents reported will be 170 (twice the initial 85). Although a Lucene index provides identifiers for documents that are unique (though not necessarily stable over optimizations), nothing in the index enforces uniqueness of document contents. Lucene will happily create another document with the same fields and values as another document. It keeps them separate internally using its own identifiers.

6.6 Queries and Query Parsing

Lucene provides a highly configurable hybrid form of search that combines exact boolean searches with softer, more relevance-ranking-oriented vector-space search methods. All searches are field-specific, because Lucene indexes terms and a term is comprised of a field name and a token.[10]

[10]Given that search is carried out over terms, there's no way to easily have a query search over all fields. Instead, field-specific queries must be disjoined to achieve this effect. Scoring for this approach may be problematic because hits on a shorter field will have a higher score than hits on a longer field. Another approach is to denormalize the documents by creating synthetic fields that concatenate the value of other fields.


Luke – Lucene Index Toolbox, v 3.5.0 (2011-12-28)
File Tools Settings Help
Overview Documents Search Files Plugins
Re-open
Commit
Close
Show top terms >>
Set

Index name: /Users/mitzimorris/aliasi/lpbook/src/applucene/temp.idx
Number of fields: 2
Number of documents: 85
Number of terms: 8986
Has deletions? / Optimized?: No / Yes
Last modified: Thu Jun 21 18:03:21 EDT 2012
Index version: 13811122eb1
Index format: -11 (Lucene 3.1)
Index functionality: lock-less, single norms, shared doc store, checksum, del count, omitTf, user data, diagnostics, hasVectors
TermInfos index divisor: N/A
Directory implementation: org.apache.lucene.store.NIOFSDirectory
Currently opened commit point: segments_1 (Thu Jun 21 18:03:21 EDT 2012)
Current commit user data: --

Select fields from the list below, and press button to view top terms in these fields. No selection means all fields.

Available fields and term counts per field:

Name	Term count	%	Decoder
file	85	0.95 %	string utf8
text	8,901	99.05 %	string utf8

Number of top terms: 10

Hint: use Shift-Click to select ranges, or Ctrl-Click to select multiple fields (or unselect all).

Tokens marked in red indicate decoding errors, likely due to a mismatched decoder.

Select a field and set its value decoder: string utf8

Top ranking terms. (Right-click for more options)

No	Rank	Field	Text
1	85	text	other
2	85	text	people
3	85	text	one
4	85	text	new
5	85	text	have
6	85	text	government
7	85	text	all
8	85	text	from
9	85	text	federalist
10	85	text	so

Index name: /Users/mit...book/src/applucene/temp.idx

Fig. 6.1: *Screenshot of Luke index browser, showing overview of index temp.idx*

6.6.1 Constructing Queries Programmatically

Queries may be constructed programmatically using the dozen or so built-in implementations of the the `Query` abstract base class from the package `org.apache.lucene.search`.

The most basic query is over a single term in a single field. This form of query is implemented in Lucene's `TermQuery` class, also in the `search` package. A term query is constructed from a `Term`, which is found in package `org.apache.lucene.index`. A term is constructed from a field name and text for the term, both specified as strings.

The `BooleanQuery` class is very misleadingly named; it supports both hard boolean queries and relevance-ranked vector-space queries, as well as allowing them to be mixed.

A boolean query may be constructed with the no-argument constructor `BooleanQuery()` (there is also a constructor that provides extra control over similarity scoring by turning off the coordination component of scoring).

Other queries may then be added to the boolean query using the method `add(Query,BooleanClause.Occur)`. The second argument, an instance of the nested enum `BooleanClause.Occur` in package `search`, indicates whether the added query is to be treated as a hard boolean constraint or contribute to the relevance ranking of vector queries. Possible values are `BooleanClause.MUST`, `BooleanClause.MUST_NOT`, and `BooleanClause.SHOULD`. The first two are used for hard boolean queries, requiring the term to appear or not appear in any result. The last value, `SHOULD`, is used for vector-space queries. With this occurrence value, Lucene will prefer results that match the query, but may return results that do not match the query.

The recursive nature of the API and the overloading of queries to act as both hard boolean and vector-type relevance queries, leads to the situation where queries may mix hard and soft constraints. It appears that clauses constrained by hard boolean occurrence constraints, `MUST` or `MUST_NOT`, do not contribute to scoring. It's less clear what happens when one of these hybrid queries is nested inside another boolean query with its own occurrence specification. For instance, it's not clear what happens when we nest a query with must-occur and should-occur clauses as a must-occur clause in a larger query.

```
BooleanQuery bq1 = new BooleanQuery();
bq1.add(new TermQuery(new Term("text","biology")), Occur.MUST);
bq1.add(new TermQuery(new Term("text","cell")), Occur.SHOULD);

BooleanQuery bq2 = new BooleanQuery();
bq2.add(new TermQuery(new Term("text","micro")), Occur.SHOULD);
bq2.add(bq1,Occur.MUST);
```

6.6.2 Query Parsing

Lucene specifies a language in which queries may be expressed.

For instance, `[computer NOT java]`[11] produces a query that specifies the

[11]We display queries Q as `[`Q`]` to indicate the scope of the search without using quotes, which are

term *computer* must appear in the default field and the term *java* must not appear. Queries may specify fields, as in *text:java*, which requires the term *java* to appear in the `text` field of a document.

The full syntax specification is available from `http://lucene.apache.org/java/3_0_2/queryparsersyntax.html`. The syntax includes basic term and field specifications, modifiers for wildcard, fuzzy, proximity or range searches, and boolean operators for requiring a term to be present, absent, or for combining queries with logical operators. Finally, sub-queries may be boosted by providing numeric values to raise or lower their prominence relative to other parts of the query.

A query parser is constructed using an analyzer, default field, and Lucene version. The default field is used for queries that do not otherwise specify the field they search over. It may then be used to convert string-based queries into query objects for searching.

The query language in Lucene suffers from a confusion between queries over tokens and queries over terms. Complete queries, must of course, be over terms. But parts of queries are naturally constrained to be over tokens in the sense of not mentioning any field values. For instance, if *Q* is a well-formed query, then so is `foo:`*Q*. In proper usage, the query *Q* should be constrained to not mention any fields. In other words, *Q* should be a query over tokens, not a general query.

Query Language Syntax

In Figure 6.2, we provide an overview of the full syntax available through Lucene's query parser. The following characters must be escaped by preceding them with a backslash:

```
+  -  &  |  !  (  )  {  }  [  ]  ^  "  ~  *  ?  :  \
```

For example, `[foo:a\(c]` searches for the three-character token *a(c* in the field `foo`. Of course, if the queries are specified as Java string literals, further escaping is required (see Section 3.3).

6.6.3 Default Fields, Token Queries, and Term Queries

When we set up a query parser, we will be supplying a default field. Unmarked token queries will then be interpreted as if constrained to that field. For instance, if `title` is the default query field, then query `[cell]` is the same as the query `[title:cell]`.

Like the programmatic queries, Lucene's query language does not clearly separate the role of token-level queries, which match tokens or sequences of tokens, and term-level queries, which match tokens within a field. Thus it's possible to write out queries with rather unclear structure, such as `[text:(money AND file:12.ascii.txt)]`; this query will actually match (as you can try with the demo in the next section) because the embedded field `file` takes precedence over the top-level field `text`.

often part of the search itself.

Type	*Syntax*	*Description*
Token	`t`	Match token `t`
Phrase	`"cs"`	Match tokens in `cs` in exact order without gaps
Field	`f:Q`	Match query `Q` in field `f`
Wildcard, Char	`cs1?cs2`	Match tokens starting with `cs1`, ending with `cs2`, with any char between
Wildcard, Seq	`cs1*cs2`	Match tokens starting with `cs1`, ending with `cs2`, with any char sequence between
Fuzzy	`t~`	Match token `t` approximately
Fuzzy, Weighted	`t~d`	Match token `t` within minimum similarity `d`
Proximity	`P~n`	Match tokens in phrase `P` within distance `n`
Range, Inclusive	`f:[t1 TO t2]`	Match tokens lexicographically between tokens `t1` and `t2` inclusive
Range, Exclusive	`f:(t1 TO t2)`	Match tokens lexicographically between tokens `t1` and `t2` exclusive
Boosting	`P^d`	Match phrase `P`, boosting score by `d`
Disjunction	`Q1 OR Q2`	Match query `Q1` or query `Q2` (or both)
Conjunction	`Q1 AND Q2`	Match query `Q1` and match query `Q2`
Difference	`Q1 NOT Q2`	Match query `Q1` but not query `Q2`
Must	`+P`	Token or phrase `P` must appear
Mustn't	`-P`	Token or phrase `P` must not appear
Grouping	`(Q)`	Match query `Q` (disambiguates parsing)

Fig. 6.2: Lucene's Query Syntax. *In the table,* `t` *is a token made up of a sequence of characters,* `f` *is a field name made up of a sequence of characters,* `cs1` *is a non-empty sequence of characters,* `cs2` *is any sequences of characters,* `d` *is a decimal number,* `n` *is a natural number,* `Q` *is an arbitrary well-formed query, and* `P` *is a well-formed phrase query.*

6.6.4 The `QueryParser` Class

Lucene's `QueryParser` class, in package `org.apache.lucene.queryparser`, converts string-based queries which are well-formed according to Lucene's query syntax into `Query` objects.

The constructor `QueryParser(Version,String,Analyzer)` requires a Lucene version, a string picking out a default field for unmarked tokens, and an analyzer with which to break phrasal queries down into token sequences.

Query parsing is accomplished through the method `parse(String)`, which returns a `Query`. The parse method will throw a Lucene `ParseException`, also in package `queryparser`, if the query is not well formed.

6.6.5 Using Luke to Develop and Test Queries

The Luke index browser provides interactive search over an index via the search tab. The top part of this tab contains a set of controls. The controls on the right side specify the behavior of the analyzer and the searcher. On the top left side there is a text box in which to enter the query string. Below the text box is a display of the result of parsing this query string. The bottom half of this tab is given over to search results.

Figure 6.3 illustrates a simple search over the index temp.idx that we created in the previous section. In the top right, we specify the analyzer and default field used by the query parser to be 'text' and `StandardAnalyzer` respectively. In the top left, we entered the words *powers of the judiciary*. The parser treats each word as a search term. The standard analyzer stop-lists *of* and *the* and produces a query consisting of two terms: `[text:powers]` and `[text:judiciary]`. The search results are displayed in ranked order.

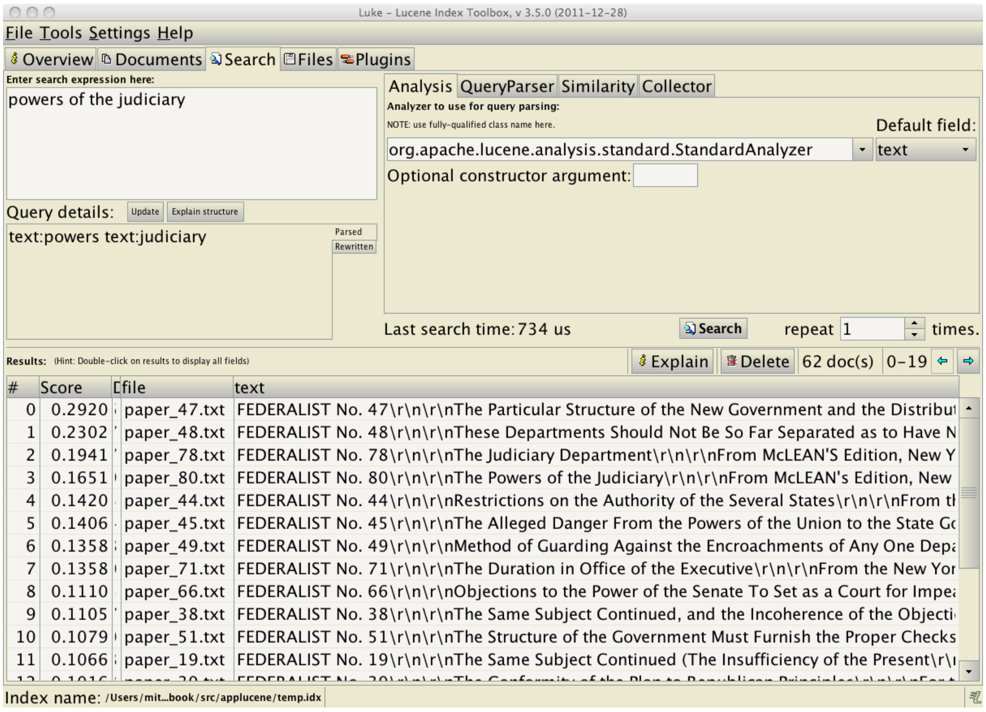
6.7 Search

6.7.1 Index Readers

Lucene uses instances of the aptly named `IndexReader` to read data from an index.

Distributed Readers

A convenient feature of the reader design in Lucene is that we may construct an index reader from multiple indexes, which will then combine their contents at search time. From the reader's client's perspective, the behavior is indistinguishable (other than in terms of speed) from a combined and optimized index. We can even distribute these indexes over multiple machines on a network using Java's Remote Method Invocation (RMI).


Luke - Lucene Index Toolbox, v 3.5.0 (2011-12-28)
File Tools Settings Help
Overview Documents Search Files Plugins
Analysis QueryParser Similarity Collector
Update Explain structure
Parsed
Rewritten
Search
Explain Delete

Enter search expression here:
powers of the judiciary

Query details:
text:powers text:judiciary

Analyzer to use for query parsing:
NOTE: use fully-qualified class name here.
org.apache.lucene.analysis.standard.StandardAnalyzer
Default field: text
Optional constructor argument:

Last search time: 734 us repeat 1 times.

Results: (Hint: Double-click on results to display all fields) 62 doc(s) 0-19

#	Score	file	text
0	0.2920	paper_47.txt	FEDERALIST No. 47\r\n\r\nThe Particular Structure of the New Government and the Distribu
1	0.2302	paper_48.txt	FEDERALIST No. 48\r\n\r\nThese Departments Should Not Be So Far Separated as to Have N
2	0.1941	paper_78.txt	FEDERALIST No. 78\r\n\r\nThe Judiciary Department\r\n\r\nFrom McLEAN'S Edition, New Y
3	0.1651	paper_80.txt	FEDERALIST No. 80\r\n\r\nThe Powers of the Judiciary\r\n\r\nFrom McLEAN's Edition, New
4	0.1420	paper_44.txt	FEDERALIST No. 44\r\n\r\nRestrictions on the Authority of the Several States\r\n\r\nFrom tl
5	0.1406	paper_45.txt	FEDERALIST No. 45\r\n\r\nThe Alleged Danger From the Powers of the Union to the State Gc
6	0.1358	paper_49.txt	FEDERALIST No. 49\r\n\r\nMethod of Guarding Against the Encroachments of Any One Depa
7	0.1358	paper_71.txt	FEDERALIST No. 71\r\n\r\nThe Duration in Office of the Executive\r\n\r\nFrom the New Yor
8	0.1110	paper_66.txt	FEDERALIST No. 66\r\n\r\nObjections to the Power of the Senate To Set as a Court for Impe
9	0.1105	paper_38.txt	FEDERALIST No. 38\r\n\r\nThe Same Subject Continued, and the Incoherence of the Objecti
10	0.1079	paper_51.txt	FEDERALIST No. 51\r\n\r\nThe Structure of the Government Must Furnish the Proper Checks
11	0.1066	paper_19.txt	FEDERALIST No. 19\r\n\r\nThe Same Subject Continued (The Insufficiency of the Present\r\

Index name: /Users/mit...book/src/applucene/temp.idx

Fig. 6.3: *Screenshot of Luke index browser, showing search results*

6.7.2 Index Searchers

Lucene supplies an `IndexSearcher` class that performs the actual search. Every index searcher wraps an index reader to get a handle on the indexed data. Once we have an index searcher, we can supply queries to it and enumerate results in order of their score.

There is really nothing to configure in an index searcher other than its reader, so we'll jump straight to the demo code.

6.7.3 Search Demo

We provide a simple implementation of Lucene search based on the index we created in the last section.

Code Walkthrough

The code is in the `main()` method of the demo class `LuceneSearch`. The method starts off by reading in command-line arguments.

```
public static void main(String[] args)
    throws ParseException, CorruptIndexException,
           IOException {

    File indexDir = new File(args[0]);
```

```
String query = args[1];
int maxHits = Integer.parseInt(args[2]);
```

We need the directory for the index, a string representing the query in Lucene's query language, and a specification of the maximum number of hits to return. The method is declared to throw a Lucene corrupt index exception if the index isn't well formed, a Lucene parse exception if the query isn't well formed, and a general Java I/O exception if there is a problem reading from or writing to the index directory.

After setting the command-line arguments, the next step is to create a Lucene directory, index reader, index searcher and query parser.

```
Directory fsDir = FSDirectory.open(indexDir);
IndexReader reader = IndexReader.open(fsDir);
IndexSearcher searcher = new IndexSearcher(reader);

String dField = "text";
Analyzer stdAn
    = new StandardAnalyzer(Version.LUCENE_36);
QueryParser parser
    = new QueryParser(Version.LUCENE_36,dField,stdAn);
```

It is important to use the same analyzer in the query parser as is used in the creation of the index. If they don't match, queries that should succeed will fail because the tokens won't match.[12] For instance, if we apply stemming in the indexing to reduce *codes* to *code*, then we better do the same thing for the query, because we won't find *codes* in the index, only its stemmed form *code*.

The last bit of code in the search demo uses the query parser to parse the query, then searches the index and reports the results.

```
Query q = parser.parse(query);

TopDocs hits = searcher.search(q,maxHits);
ScoreDoc[] scoreDocs = hits.scoreDocs;

for (int n = 0; n < scoreDocs.length; ++n) {
    ScoreDoc sd = scoreDocs[n];
    float score = sd.score;
    int docId = sd.doc;
    Document d = searcher.doc(docId);
    String fileName = d.get("file");
```

The `Query` object is created by using the parser to parse the text query. We then use the searcher instance to search given the query and an upper bound on the number of hits to return. This returns an instance of the Lucene class `TopDocs`, from package `search`, which encapsulates the results of a search (through references back into the index).

[12]The analyzers must produce the same tokenization. In the demo programs here, the analyzer used to create the index was a `StandardAnalyzer` wrapped in a `LimitTokenCountAnalyzer`. Since the `LimitTokenCountAnalyzer` doesn't change the underlying tokenization and we don't expect our queries to be very long, the query parser uses a `StandardAnalyzer`.

The TopDocs result provides access to an array of search results. Each result is an instance of the Lucene class ScoreDoc, also in the search package, which encapsulates a document reference with a floating point score.

The array of search results is sorted in decreasing order of score, with higher scores representing better matches. We then enumerate over the array, and for each ScoreDoc object, we pull its score out using the public member variable score. We then pull its document reference number (Lucene's internal identifier for the doc) out with the member variable doc. With the Lucene document identifier, we are able to retrieve the document from the searcher (which just delegates this operation to its index reader internally). Finally, with the document in hand, we retrieve its file name using the get() method on the document; we use get() here rather than getValues(), which returns an array, because we know there is only one file name for each document in the index. We could've also retrieved the text of the document, because we stored it in the index.

Running the Demo

The Ant target lucene-search invokes the demo with command-line arguments provided by the value of properties index.dir, query, and max.hits.

```
>      ant -Dindex.dir=temp.idx -Dquery="powers of the judiciary"
-Dmax.hits=15 lucene-search
```

```
Index Dir=/Users/mitzimorris/aliasi/lpbook/src/applucene/temp.idx
query=powers of the judiciary
max hits=15
Hits (rank,score,file name)
  0 0.29  paper_47.txt
  1 0.23  paper_48.txt
  2 0.19  paper_78.txt
...
 13 0.10  paper_81.txt
 14 0.09  paper_82.txt
```

We run this demo with the query string *powers of the judiciary*. As we saw in the previous section, the query parser stop-lists the words *of* and *the*, reducing this query to a boolean search for documents which contain the terms [text:powers] and/or [text:judiciary]. Lucene returns 15 results numbered 0 to 14. We see that paper 47 is the closest match to our query. This document contains 18 instances of the term *powers* and 24 instances of *judiciary*. This seems like a low score on a 0–1 scale for a document which matches all the tokens in the query; the reason is because the documents are long, so the percentage of tokens in the document matching the query tokens is relatively low.

The token *food* does not show up in any documents, so the query [text:food] returns no hits. If we enter the query string *powers of the judiciary food*, it returns exactly the same hits as the query string *powers of the judiciary*, in exactly the same order, but with lower scores. If we try the query string *judiciary +food*, we are insisting that the token *food* appear in any matching document. Because it doesn't appear in the corpus at all, the query string *judiciary +food* has zero hits. On the other hand, the must-not query *judiciary -food* has the same hits with the same scores as does the query for *judiciary*.

6.7.4 Ranking

For scoring documents against queries, Lucene uses the complex and highly configurable abstract base class `Similarity` in the package in `org.apache.lucene.search`. If nothing else is specified, as in our simple demos, the concrete subclass `DefaultSimilarity` will be used.

Similarity deals with scoring queries with `SHOULD`-occur terms. In the search language, all tokens are treated as `SHOULD` unless prefixed with the must-occur marking plus-sign (+) or must-not occur negation (-).

The basic idea is that the more instances of query terms in a document the better. Terms are not weighted equally. A term is weighted based on its inverse document frequency (IDF), so that terms that occur in fewer documents receive higher weights. Weights may also be boosted or lowered in the query syntax or programmatically with a query object.

All else being equal, shorter documents are preferred. The reason for this is that when two documents contain the same number of instances of query terms, the shorter document has a higher proportion of query terms and is thus likely to be a better match. This proportion may well be higher for a short document which contains only a few instances of the query term than it is for a very long document which contains many instances of the query term. This can be problematic when a document collection contains both very short and very long documents.

There is also a component of scoring based on the percentage of the query terms that appear in the document. All else being equal, we prefer documents that cover more terms in the query.

6.8 Deleting and Updating Documents

The the `IndexWriter` class supports methods to delete documents from an index based on a term or query. There is also a `deleteAll()` method to completely clear the index.

The the `IndexWriter` class supports methods to update a document or documents that contain a term. The update operation consists of first deleting the existing document(s) which contain a given term and then adding new one(s). This operation is atomic with respect to any index readers open on the index.

In many cases, the documents being indexes have unique identifiers. For example, in our file-based index, the file names are meant to be unique.[13] We can then create a term containing the application's document identifier field and delete by term. For instance, we could call `deleteDocuments(new`

[13]Unlike a database, which enforces unique ids by declaration, Lucene requires the programmer to treat fields as keys by convention.

Term("file","paper_12.txt")) to delete the document with file identifier paper_12.txt.

In earlier versions of Lucene, document deletion was handled by an IndexReader.[14] However, as of 3.6 these delete document(s) methods on IndexReader have been deprecated and in Lucene 4.0 all write support for this class will be removed.

6.8.1 Visibility of Deletes

When a delete method is called on a writer based on a term or document identifier, the documents are not immediately physically deleted from the index. Instead, their identifiers are buffered and they are treated as if virtually deleted.

This approach to document deletion was made for the same efficiency reasons and faces many of the same issues as concurrent sets of variables in Java's memory model. Even when deletes are called on one index, not every reader with a handle on that index will be able to see the delete, but any new reader opened on the index after the delete will see the deletion.

The storage space for a deleted document in the index directory is only reclaimed during a merge step.

6.8.2 Lucene Deletion Demo

We have implemented an example of deleting documents in the demo class LuceneDelete.

Code Walkthrough

The code is in the main() method of the demo class LuceneDelete. This method takes three command-line arguments, the index directory path, along with the field name and token used to construct the term to delete. We open the index directory and create a default analyzer just as we did in the demo class LuceneIndexing.

After setting the command-line arguments and creating the analyzer, we create an IndexWriterConfig object that will be passed in to the IndexWriter constructor.

```
IndexWriterConfig iwConf
    = new IndexWriterConfig(Version.LUCENE_36,ltcAn);
iwConf.setOpenMode(IndexWriterConfig.OpenMode.APPEND);

IndexWriter indexWriter
    = new IndexWriter(fsDir,iwConf);
```

This time we set the open mode to IndexWriterConfig.OpenMode.APPEND. Now we can attempt to delete a document from the index.

[14]Because the IndexWriter was originally designed just to append documents to an existing index, it didn't need to keep track of documents already in the index. But in order to delete a document, first it must be found. Since this couldn't be done with an IndexWriter, the deleteDocument methods were added to the IndexReader class.

```
int numDocsBefore = indexWriter.numDocs();

Term term = new Term(fieldName,token);
indexWriter.deleteDocuments(term);
boolean hasDeletedDocs = indexWriter.hasDeletions();
int numDocsAfterDeleteBeforeCommit = indexWriter.numDocs();

indexWriter.commit();
int numDocsAfter = indexWriter.numDocs();

indexWriter.close();
```

We then construct a term out of the field and token and pass it to the index writer's delete method. After the deletion, we query the writer to find out if deletions are pending. We check the number of docs before and after the call to the `commit()` method.

Running the Demo

The Ant target `lucene-delete` invokes the class, supplying the value of properties `index.dir`, `field.name`, and `token` as command-line arguments.

Make sure that before you run this demo, you've run the indexing demo exactly once. You can always delete the index directory and rebuild it if it gets messed up.

```
> ant -Dindex.dir=temp.idx -Dfield.name=file -Dtoken=paper_12.txt
lucene-delete

index.dir=/Users/mitzimorris/aliasi/lpbook/src/applucene/temp.idx
field.name=file
token=paper_12.txt
Num docs before delete=85
Has deleted docs=true
Num docs after delete before commit=85
Num docs after commit=84
```

After the demo is run, we can look at the index directory again.

```
> export BLOCKSIZE=1024; ls -1 -s temp.idx

total 1524
1144 _0.fdt
   4 _0.fdx
   4 _0.fnm
  84 _0.frq
   4 _0.nrm
 188 _0.prx
   4 _0.tii
  80 _0.tis
   4 _0_1.del
   4 segments.gen
   4 segments_2
```

There is now an extra file with suffix `.del` that holds information about which items have been deleted. These deletes will now be visible to index readers that open (or re-open) their indices.

6.9 Lucene and Databases

Lucene is like a database in its storage and indexing of data in a disk-like abstraction. The main difference between Lucene and a database is in the type of objects they store. Databases store multiple type-able tables consisting of small data objects in the rows. The Structured Query Language (SQL) is then used to retrieve results from a database, often calling on multiple tables to perform complex joins and filters.

Lucene, on the other hand, provides a configurable, but more homogeneous interface. It mainly operates through text search (with some lightweight numerical and date-based embellishments), and returns entire documents.[15]

6.9.1 Transactional Support

In enterprise settings, we are usually very concerned with maintaining the integrity of our data. In addition to backups, we want to ensure that our indexes don't get corrupted as they are being processed.

A transaction in the database sense, as encapsulated in the Java 2 Enterprise Edition (J2EE), is a way of grouping a sequence of complex operations, such as committing changes to an index, such that they all happen or none of them happen. That is, it's about making complex operations behave as if they were atomic in the concurrency sense.

Earlier versions of Lucene were not like databases in not having any kind of transactional support. More recently, Lucene introduced configurable commit operations for indexes. These commit operations are transactional in the sense that if they fail, they roll back any changes they were in the middle of. This allows standard database-type two-phase commits to be implemented directly with fine-grained control over preparing and rolling back.

[15]Nothing prevents us from treating paragraphs or even sentences in a "real" document as a document for Lucene's purposes.

Appendix A

Corpora

In this appendix, we list the corpora that we use for examples in the text.

A.1 Canterbury Corpus

The Canterbury corpus is a benchmark collection of mixed text and binary files for evaluating text compression.

Contents

The files contained in the archive are the following.

File	*Type*	*Description*	*Bytes*
`alice29.txt`	text	English text	152,089
`asyoulik.txt`	play	Shakespeare	125,179
`cp.html`	html	HTML source	24,603
`fields.c`	Csrc	C source	11,150
`grammar.lsp`	list	LISP source	3,721
`kennedy.xls`	Excl	Excel Spreadsheet	1,029,744
`lcet10.txt`	tech	Technical writing	426,754
`plrabn12.txt`	poem	Poetry	481,861
`ptt5`	fax	CCITT test set	513,216
`sum`	SPRC	SPARC Executable	38,240
`xargs.1`	man	GNU manual page	4,227

Authors

Ross Arnold and Tim Bell.

Licesning

The texts are all public domain.

Download

`http://corpus.canterbury.ac.nz/descriptions/`

Warning: you should create a directory in which to unpack the distribution because the tarball does not contain any directory structure, just a bunch of flat files.

A.2 20 Newsgroups

Contents

The corpus contains roughly 20,000 posts to 20 different newsgroups. These are

comp.graphics	rec.autos	sci.crypt
comp.os.ms-windows.misc	rec.motorcycles	sci.electronics
comp.sys.ibm.pc.hardware	rec.sport.baseball	sci.med
comp.sys.mac.hardware	rec.sport.hockey	sci.space
comp.windows.x		
misc.forsale	talk.politics.misc	talk.religion.misc
	talk.politics.guns	alt.atheism
	talk.politics.mideast	soc.religion.christian

Authors

It is currently being maintained by Jason Rennie, who speculates that Ken Lang may be the original curator.

Licensing

There are no licensing terms listed and the data may be downloaded directly.

Download

`http://people.csail.mit.edu/jrennie/20Newsgroups/`

A.3 WormBase MEDLINE Citations

The WormBase web site, `http://wormbase.org`, curates and distributes resources related to the model organism *Caenorhabditis elegans*, the nematode worm. Part of that distribution is a set of citation to published research on *C. elegans*. These literature distributions consist of MEDLINE citations in a simple line-oriented format.

Authors

Wormbase is a collaboration among many cites with many supporters. It is described in

Harris, T. W. et al. 2010. WormBase: A comprehensive resource for nematode research. *Nucleic Acids Research* **38**. `doi:10.1093/nar/gkp952`

Download

`ftp://ftp.wormbase.org/pub/wormbase/misc/literature/2007-12-01-wormbase-literature.endnote.gz` (15MB)

Appendix B

Further Reading

In order to fully appreciate contemporary approaches to natural language processing requires three fairly broad areas of expertise: linguistics, statistics, and algorithms. We don't pretend to introduce these areas in any depth in this book. Instead, we recommend the following textbooks, sorted by area, from among the many that are currently available.

B.1 Unicode

Given that we're dealing with text data, and that Java uses Unicode internally, it helps to be familiar with the specification. Or at least know where to find more information. The actual specification reference book is quite readable. The latest available in print is for version 5.0,

- Unicode Consortium, The. *The Unicode Standard 5.0.* Addison-Wesley.

  This is the final word on the unicode standard, and quite readable.

You can also find the latest version (currently 5.2) online at

```
http://unicode.org
```

This includes all of the code tables for the latest version, which are available in PDF files or through search, at

```
http://unicode.org/charts/
```

B.2 Java

B.2.1 Overviews of Java

For learning Java, we recommend the following three books pitched at beginners and intermediate Java programmers,

- Arnold, Ken, James Gosling and David Holmes. 2005. *The Java Programming Language*, 4th Edition. Prentice-Hall.

Gosling invented Java, and this is the "official" introduction to the language. It's also very good.

- Bloch, Joshua. 2008. *Effective Java*, 2nd Edition. Prentice-Hall.

  Bloch was one of the main architects of Java from version 1.1 forward. This book is about taking Java program design to the next level.

- Fowler, Martin, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley.

  Great book on making code better, both existing code and code yet to be created.

B.2.2 Books on Java Libraries and Packages

There is a substantial collection of widely used libraries for Java. Here are reference books for the Java libraries we use in this book.

- Loughran, Steve and Erik Hatcher. 2007 *Ant in Action.* Manning.

  This is technically the second edition of their 2002 book, Java Development with Ant. Both authors are committers for the Apache Ant project.

- Tahchiev, Peter, Felipe Leme, Vincent Massol, and Gary Gregory. 2010. *JUnit in Action*, Second Edition. Manning.

  A good overview for JUnit. Should be up-to-date with the latest attribute-driven versions, unlike either Kent Beck's book (he's the author of Ant). As much as we liked The Pragmatic Programmer, Hunt and Thomas's book on JUnit is not particularly useful.

- McCandless, Michael, Erik Hatcher and Otis Gospodnetić. 2010. *Lucene in Action*, Second Edition. Manning.

  A fairly definitive overview of the Apaceh Lucene search engine by three project committers.

- Hunter, Jason and William Crawford. 2001. *Java Servlet Programming*, Second Edition. O'Reilly.

  It only covers version 2.2, so it's a bit out of date relative to recent standards, but the basics haven't changed, and although there are more elementary introductions, this book is a very good introduction to what servlets can do. It also explains HTTP itself quite clearly. The first author is an Apache Tomcat developer and was involved in developing the original servlet standard.

- Goetz, Brian, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. 2006. *Java Concurrency in Practice.* Addison-Wesley.

  This fantastic book covers threading and concurrency patterns and their application, along with implementations in Java's `java.util.concurrent` *package. Doug Lea was the original author of* `util.concurrent` *before it was brought into Java itself, and his previous book, Concurrent Programming in Java, Second Edition (1999;*

Prentice Hall), is wroth reading for a deeper theoretical understanding of concurrency in Java.

- Hitchens, Ron. 2002. *Java NIO*. O'Reilly.

  A concise and readable introduction to the "new" I/O package, `java.nio` *and the concepts behind it, like buffers and multicast reads.*

- McLaughlin, Brett and Justin Edelson. 2006. *Java and XML*, Third Edition. O'Reilly.

  A good introduction to the set of tools available in Java for parsing and generating XML with SAX and DOM, specifying structure with DTDs and XML Schema, transforming XML with XSLT, as well as the general high-level libraries JAXP and JAXB.

B.2.3 General Java Books

There are also general-purpose books about Java that do not focus on particular classes or applications.

- Shirazi, Jack. 2003. *Java Performance Tuning*, 2nd Edition. O'Reilly

  Although already dated, it provides a good introduction to profiling and thinking about performance tuning. The author and friends also run the web site `javaperformancetuning.com`*, which is a monthly digest of Java performance tuning tips (and advertising). If you know C, start with Jon Bentley's classic, Programming Pearls (Addison-Wesley).*

B.2.4 Practice Makes Perfect

Really, the only way to learn how to code is to practice. Being a good coder means two things: being fast and efficient (assuming accuracy). By accurate, we mean writing code that does what it's supposed to do. By fast, we mean the coding goes quickly. By efficient, we mean the resulting code is efficient.

For practice, we highly recommend the algorithm section of the TopCoder contests. TopCoder provides a complete online Java environment, well thought out problems at varying specified levels of difficulty, unit tests that check your submissions, and examples of what others did to provide code-reading practice.

```
http://www.topcoder.com/tc
```

B.3 General Programming

We would also recommend a book about programming in general, especially for those who have not worked collaboratively with a good group of professional programmers.

- Hunt, Andrew and David Thomas. 1999. *The Pragmatic Programmer*. Addison-Wesley.

Read this book and follow its advice, which is just as relevant now as when it was written. The best book on how to program we know. A competitor to books like Steve McConnell's Code Complete (2004) and books on "agile" or "extreme" programming.

- Collins-Susman, Ben, Brian W. Fitzpatrick, and C. Michael Pilato. 2010. Collins-Sussman et al.'s *Version Control with Subversion*. Subversion 1.6. CreateSpace.

  This is the official guide to Subversion from the authors. It's also available online at

  `http://svnbook.red-bean.com/`

Appendix C

Licenses

This chapter includes the text of the licenses for all of the software used for demonstration purposes.

C.1 LingPipe License

C.1.1 Commercial Licensing

Customized commercial licenses for LingPipe or components of LingPipe are avaialble through LingPipe, Inc.

C.1.2 Royalty-Free License

Both LingPipe and the code in this book are distributed under the Alias-i Royalty Free License Version 1.

```
Alias-i ROYALTY FREE LICENSE VERSION 1

Copyright (c) 2003-2010 Alias-i, Inc All Rights Reserved

1.  This Alias-i Royalty Free License Version 1 ("License") governs the copying, modifying, and distributing of the computer
program or work containing a notice stating that it is subject to the terms of this License and any derivative works of that
computer program or work.  The computer program or work and any derivative works thereof are the "Software." Your copying,
modifying, or distributing of the Software constitutes acceptance of this License.  Although you are not required to accept
this License, since you have not signed it, nothing else grants you permission to copy, modify, or distribute the Software.  If
you wish to receive a license from Alias-i under different terms than those contained in this License, please contact Alias-i.
Otherwise, if you do not accept this License, any copying, modifying, or distributing of the Software is strictly prohibited by
law.

2.  You may copy or modify the Software or use any output of the Software (i) for internal non-production trial, testing and
evaluation of the Software, or (ii) in connection with any product or service you provide to third parties for free.  Copying or
modifying the Software includes the acts of "installing", "running", "using", "accessing" or "deploying" the Software as those
terms are understood in the software industry.  Therefore, those activities are only permitted under this License in the ways
that copying or modifying are permitted.

3.  You may distribute the Software, provided that you:  (i) distribute the Software only under the terms of this License, no
more, no less; (ii) include a copy of this License along with any such distribution; (iii) include the complete corresponding
machine-readable source code of the Software you are distributing; (iv) do not remove any copyright or other notices from the
Software; and, (v) cause any files of the Software that you modified to carry prominent notices stating that you changed the
Software and the date of any change so that recipients know that they are not receiving the original Software.

4.  Whether you distribute the Software or not, if you distribute any computer program that is not the Software, but that (a)
is distributed in connection with the Software or contains any part of the Software, (b) causes the Software to be copied or
modified (i.e., ran, used, or executed), such as through an API call, or (c) uses any output of the Software, then you must
distribute that other computer program under a license defined as a Free Software License by the Free Software Foundation or an
Approved Open Source License by the Open Source Initiative.

5.  You may not copy, modify, or distribute the Software except as expressly provided under this License, unless you receive a
different written license from Alias-i to do so.  Any attempt otherwise to copy, modify, or distribute the Software is without
```

Alias-i's permission, is void, and will automatically terminate your rights under this License. Your rights under this License may only be reinstated by a signed writing from Alias-i.

THE SOFTWARE IS PROVIDED "AS IS." TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, ALIAS-i DOES NOT MAKE, AND HEREBY EXPRESSLY DISCLAIMS, ANY WARRANTIES, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, CONCERNING THE SOFTWARE OR ANY SUBJECT MATTER OF THIS LICENSE. SPECIFICALLY, BUT WITHOUT LIMITING THE FOREGOING, LICENSOR MAKES NO EXPRESS OR IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS (FOR A PARTICULAR PURPOSE OR OTHERWISE), QUALITY, USEFULNESS, TITLE, OR NON-INFRINGEMENT. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL LICENSOR BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY DAMAGES OR IN RESPECT OF ANY CLAIM UNDER ANY TORT, CONTRACT, STRICT LIABILITY, NEGLIGENCE OR OTHER THEORY FOR ANY DIRECT, INDIRECT, INCIDENTAL, CONSEQUENTIAL, PUNITIVE, SPECIAL OR EXEMPLARY DAMAGES, EVEN IF IT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES, OR FOR ANY AMOUNTS IN EXCESS OF THE AMOUNT YOU PAID ALIAS-i FOR THIS LICENSE. YOU MUST PASS THIS ENTIRE LICENSE, INCLUDING SPECIFICALLY THIS DISCLAIMER AND LIMITATION OF LIABILITY, ON WHENEVER YOU DISTRIBUTE THE SOFTWARE.

C.2 Java Licenses

The Java executables comes with two licenses, one for the Java runtime environment (JRE), which is required to execute Java programs, and one for the Java development kit (JDK), required to compile programs. There is a third license for the Java source code.

C.2.1 Sun Binary License

Both the JRE and JDK are subject to Sun's Binary Code License Agreement.

Sun Microsystems, Inc
Binary Code License Agreement
for the JAVA SE RUNTIME ENVIRONMENT (JRE) VERSION 6 and JAVAFX RUNTIME VERSION 1

SUN MICROSYSTEMS, INC. ("SUN") IS WILLING TO LICENSE THE SOFTWARE IDENTIFIED BELOW TO YOU ONLY UPON THE CONDITION THAT YOU ACCEPT ALL OF THE TERMS CONTAINED IN THIS BINARY CODE LICENSE AGREEMENT AND SUPPLEMENTAL LICENSE TERMS (COLLECTIVELY "AGREEMENT"). PLEASE READ THE AGREEMENT CAREFULLY. BY USING THE SOFTWARE YOU ACKNOWLEDGE THAT YOU HAVE READ THE TERMS AND AGREE TO THEM. IF YOU ARE AGREEING TO THESE TERMS ON BEHALF OF A COMPANY OR OTHER LEGAL ENTITY, YOU REPRESENT THAT YOU HAVE THE LEGAL AUTHORITY TO BIND THE LEGAL ENTITY TO THESE TERMS. IF YOU DO NOT HAVE SUCH AUTHORITY, OR IF YOU DO NOT WISH TO BE BOUND BY THE TERMS, THEN YOU MUST NOT USE THE SOFTWARE ON THIS SITE OR ANY OTHER MEDIA ON WHICH THE SOFTWARE IS CONTAINED.

1. DEFINITIONS. "Software" means the identified above in binary form, any other machine readable materials (including, but not limited to, libraries, source files, header files, and data files), any updates or error corrections provided by Sun, and any user manuals, programming guides and other documentation provided to you by Sun under this Agreement. "General Purpose Desktop Computers and Servers" means computers, including desktop and laptop computers, or servers, used for general computing functions under end user control (such as but not specifically limited to email, general purpose Internet browsing, and office suite productivity tools). The use of Software in systems and solutions that provide dedicated functionality (other than as mentioned above) or designed for use in embedded or function-specific software applications, for example but not limited to: Software embedded in or bundled with industrial control systems, wireless mobile telephones, wireless handheld devices, netbooks, kiosks, TV/STB, Blu-ray Disc devices, telematics and network control switching equipment, printers and storage management systems, and other related systems are excluded from this definition and not licensed under this Agreement. "Programs" means (a) Java technology applets and applications intended to run on the Java Platform Standard Edition (Java SE) platform on Java-enabled General Purpose Desktop Computers and Servers, and (b) JavaFX technology applications intended to run on the JavaFX Runtime on JavaFX-enabled General Purpose Desktop Computers and Servers.

2. LICENSE TO USE. Subject to the terms and conditions of this Agreement, including, but not limited to the Java Technology Restrictions of the Supplemental License Terms, Sun grants you a non-exclusive, non-transferable, limited license without license fees to reproduce and use internally Software complete and unmodified for the sole purpose of running Programs. Additional licenses for developers and/or publishers are granted in the Supplemental License Terms.

3. RESTRICTIONS. Software is confidential and copyrighted. Title to Software and all associated intellectual property rights is retained by Sun and/or its licensors. Unless enforcement is prohibited by applicable law, you may not modify, decompile, or reverse engineer Software. You acknowledge that Licensed Software is not designed or intended for use in the design, construction, operation or maintenance of any nuclear facility. Sun Microsystems, Inc. disclaims any express or implied warranty of fitness for such uses. No right, title or interest in or to any trademark, service mark, logo or trade name of Sun or its licensors is granted under this Agreement. Additional restrictions for developers and/or publishers licenses are set forth in the Supplemental License Terms.

4. LIMITED WARRANTY. Sun warrants to you that for a period of ninety (90) days from the date of purchase, as evidenced by a copy of the receipt, the media on which Software is furnished (if any) will be free of defects in materials and workmanship under normal use. Except for the foregoing, Software is provided "AS IS". Your exclusive remedy and Sun's entire liability under this limited warranty will be at Sun's option to replace Software media or refund the fee paid for Software. Any implied warranties on the Software are limited to 90 days. Some states do not allow limitations on duration of an implied warranty, so the above may not apply to you. This limited warranty gives you specific legal rights. You may have others, which vary from state to state.

5. DISCLAIMER OF WARRANTY. UNLESS SPECIFIED IN THIS AGREEMENT, ALL EXPRESS OR IMPLIED CONDITIONS, REPRESENTATIONS AND WARRANTIES, INCLUDING ANY IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT ARE DISCLAIMED, EXCEPT TO THE EXTENT THAT THESE DISCLAIMERS ARE HELD TO BE LEGALLY INVALID.

6. LIMITATION OF LIABILITY. TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL SUN OR ITS LICENSORS BE LIABLE FOR ANY LOST REVENUE, PROFIT OR DATA, OR FOR SPECIAL, INDIRECT, CONSEQUENTIAL, INCIDENTAL OR PUNITIVE DAMAGES, HOWEVER CAUSED REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF OR RELATED TO THE USE OF OR INABILITY TO USE SOFTWARE, EVEN IF SUN HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. In no event will Sun's liability to you, whether in contract, tort (including negligence), or otherwise, exceed the amount paid by you for Software under this Agreement. The foregoing limitations will apply even if the above stated warranty fails of its essential purpose. Some states do not allow the exclusion of incidental or consequential damages, so some of the terms above may not be applicable to you.

JRE Supplemental Terms

In addition to the binary license, the JRE is additionally subject to the following supplemental license terms.

JDK Supplemental License Terms

The JDK is additionally subject to the following supplemental license terms.

SUPPLEMENTAL LICENSE TERMS

These Supplemental License Terms add to or modify the terms of the Binary Code License Agreement. Capitalized terms not defined in these Supplemental Terms shall have the same meanings ascribed to them in the Binary Code License Agreement . These Supplemental Terms shall supersede any inconsistent or conflicting terms in the Binary Code License Agreement, or in any license contained within the Software.

A. Software Internal Use and Development License Grant. Subject to the terms and conditions of this Agreement and restrictions and exceptions set forth in the Software "README" file incorporated herein by reference, including, but not limited to the Java Technology Restrictions of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license without fees to reproduce internally and use internally the Software complete and unmodified for the purpose of designing, developing, and testing your Programs.

B. License to Distribute Software. Subject to the terms and conditions of this Agreement and restrictions and exceptions set forth in the Software README file, including, but not limited to the Java Technology Restrictions of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license without fees to reproduce and distribute the Software, provided that (i) you distribute the Software complete and unmodified and only bundled as part of, and for the sole purpose of running, your Programs, (ii) the Programs add significant and primary functionality to the Software, (iii) you do not distribute additional software intended to replace any component(s) of the Software, (iv) you do not remove or alter any proprietary legends or notices contained in the Software, (v) you only distribute the Software subject to a license agreement that protects Sun's interests consistent with the terms contained in this Agreement, and (vi) you agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software.

C. License to Distribute Redistributables. Subject to the terms and conditions of this Agreement and restrictions and exceptions set forth in the Software README file, including but not limited to the Java Technology Restrictions of these Supplemental Terms, Sun grants you a non-exclusive, non-transferable, limited license without fees to reproduce and distribute those files specifically identified as redistributable in the Software "README" file ("Redistributables") provided that: (i) you distribute the Redistributables complete and unmodified, and only bundled as part of Programs, (ii) the Programs add significant and primary functionality to the Redistributables, (iii) you do not distribute additional software intended to supersede any component(s) of the Redistributables (unless otherwise specified in the applicable README file), (iv) you do not remove or alter any proprietary legends or notices contained in or on the Redistributables, (v) you only distribute the Redistributables pursuant to a license agreement that protects Sun's interests consistent with the terms contained in the Agreement, (vi) you agree to defend and indemnify Sun and its licensors from and against any damages, costs, liabilities, settlement amounts and/or expenses (including attorneys' fees) incurred in connection with any claim, lawsuit or action by any third party that arises or results from the use or distribution of any and all Programs and/or Software.

D. Java Technology Restrictions. You may not create, modify, or change the behavior of, or authorize your licensees to create, modify, or change the behavior of, classes, interfaces, or subpackages that are in any way identified as "java", "javax", "sun" or similar convention as specified by Sun in any naming convention designation.

E. Distribution by Publishers. This section pertains to your distribution of the Software with your printed book or magazine (as those terms are commonly used in the industry) relating to Java technology ("Publication"). Subject to and conditioned upon your compliance with the restrictions and obligations contained in the Agreement, in addition to the license granted in Paragraph 1 above, Sun hereby grants to you a non-exclusive, nontransferable limited right to reproduce complete and unmodified copies of the Software on electronic media (the "Media") for the sole purpose of inclusion and distribution with your Publication(s), subject to the following terms: (i) You may not distribute the Software on a stand-alone basis; it must be distributed with your Publication(s); (ii) You are responsible for downloading the Software from the applicable Sun web site; (iii) You must refer to the Software as JavaTM SE Development Kit 6; (iv) The Software must be reproduced in its entirety and without any modification whatsoever (including, without limitation, the Binary Code License and Supplemental License Terms accompanying the Software and proprietary rights notices contained in the Software); (v) The Media label shall include the following information: Copyright 2006, Sun Microsystems, Inc. All rights reserved. Use is subject to license terms. Sun, Sun Microsystems, the Sun logo, Solaris, Java, the Java Coffee Cup logo, J2SE, and all trademarks and logos based on Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries. This information must be placed on the Media label in such a manner as to only apply to the Sun Software; (vi) You must clearly identify the Software as Sun's product on the Media holder or Media label, and you may not state or imply that Sun is responsible for any third-party software contained on the Media; (vii) You may not include any third party software on the Media which is intended to be a replacement or substitute for the Software; (viii) You shall indemnify Sun for all damages arising from your failure to comply with the requirements of this Agreement. In addition, you shall defend, at your expense, any and all claims brought against Sun by third parties, and shall pay all damages awarded by a court of competent jurisdiction, or such settlement amount negotiated by you, arising out of or in connection with your use, reproduction or distribution of the Software and/or the Publication. Your obligation to provide indemnification under this section shall arise provided that Sun: (a) provides you prompt notice of the claim; (b) gives you sole control of the defense and settlement of the claim; (c) provides you, at your expense, with all available information, assistance and authority to defend; and (d) has not compromised or settled such claim without your prior written consent; and (ix) You shall provide Sun with a written notice for each Publication; such notice shall include the following information: (1) title of Publication, (2) author(s), (3) date of Publication, and (4) ISBN or ISSN numbers. Such notice shall be sent to Sun Microsystems, Inc., 4150 Network Circle, M/S USCA12-110, Santa Clara, California 95054, U.S.A , Attention: Contracts Administration.

F. Source Code. Software may contain source code that, unless expressly licensed for other purposes, is provided solely for reference purposes pursuant to the terms of this Agreement. Source code may not be redistributed unless expressly provided for in this Agreement.

G. Third Party Code. Additional copyright notices and license terms applicable to portions of the Software are set forth in the THIRDPARTYLICENSEREADME.txt file. In addition to any terms and conditions of any third party opensource/freeware license identified in the THIRDPARTYLICENSEREADME.txt file, the disclaimer of warranty and limitation of liability provisions in paragraphs 5 and 6 of the Binary Code License Agreement shall apply to all Software in this distribution.

H. Termination for Infringement. Either party may terminate this Agreement immediately should any Software become, or in either party's opinion be likely to become, the subject of a claim of infringement of any intellectual property right.

I. Installation and Auto-Update. The Software's installation and auto-update processes transmit a limited amount of data to Sun (or its service provider) about those specific processes to help Sun understand and optimize them. Sun does not associate the data with personally identifiable information. You can find more information about the data Sun collects at http://java.com/data/.

For inquiries please contact: Sun Microsystems, Inc., 4150 Network Circle, Santa Clara, California 95054, U.S.A.

C.2.2 Sun Community Source License Version 2.8

The license for the Java source code is

SUN COMMUNITY SOURCE LICENSE Version 2.8 (Rev. Date January 17, 2001)

RECITALS

Original Contributor has developed Specifications and Source Code implementations of certain Technology; and

Original Contributor desires to license the Technology to a large community to facilitate research, innovation andproduct development while maintaining compatibility of such products with the Technology as delivered by Original Contributor; and

Original Contributor desires to license certain Sun Trademarks for the purpose of branding products that are compatible with the relevant Technology delivered by Original Contributor; and

You desire to license the Technology and possibly certain Sun Trademarks from Original Contributor on the terms and conditions specified in this License.

In consideration for the mutual covenants contained herein, You and Original Contributor agree as follows:

AGREEMENT

1. Introduction. The Sun Community Source License and effective attachments ("License") may include five distinct licenses: Research Use, TCK, Internal Deployment Use, Commercial Use and Trademark License. The Research Use license is effective when You execute this License. The TCK and Internal Deployment Use licenses are effective when You execute this License, unless otherwise specified in the TCK and Internal Deployment Use attachments. The Commercial Use and Trademark licenses must be signed by You and Original Contributor in order to become effective. Once effective, these licenses and the associated requirements and responsibilities are cumulative. Capitalized terms used in this License are defined in the Glossary.

2. License Grants.

2.1. Original Contributor Grant. Subject to Your compliance with Sections 3, 8.10 and Attachment A of this License, Original Contributor grants to You a worldwide, royalty-free, non-exclusive license, to the extent of Original Contributor's Intellectual Property Rights covering the Original Code, Upgraded Code and Specifications, to do the following:

a) Research Use License: (i) use, reproduce and modify the Original Code, Upgraded Code and Specifications to create Modifications and Reformatted Specifications for Research Use by You, (ii) publish and display Original Code, Upgraded Code and Specifications with, or as part of Modifications, as permitted under Section 3.1 b) below, (iii) reproduce and distribute copies of Original Code and Upgraded Code to Licensees and students for Research Use by You, (iv) compile, reproduce and distribute Original Code and Upgraded Code in Executable form, and Reformatted Specifications to anyone for Research Use by You.

b) Other than the licenses expressly granted in this License, Original Contributor retains all right, title, and interest in Original Code and Upgraded Code and Specifications. 2.2. Your Grants.

a) To Other Licensees. You hereby grant to each Licensee a license to Your Error Corrections and Shared Modifications, of the same scope and extent as Original Contributor's licenses under Section 2.1 a) above relative to Research Use, Attachment C relative to Internal Deployment Use, and Attachment D relative to Commercial Use.

b) To Original Contributor. You hereby grant to Original Contributor a worldwide, royalty-free, non-exclusive, perpetual and irrevocable license, to the extent of Your Intellectual Property Rights covering Your Error Corrections, Shared Modifications and Reformatted Specifications, to use, reproduce, modify, display and distribute Your Error Corrections, Shared Modifications and Reformatted Specifications, in any form, including the right to sublicense such rights through multiple tiers of distribution.

c) Other than the licenses expressly granted in Sections 2.2 a) and b) above, and the restriction set forth in Section 3.1 d)(iv) below, You retain all right, title, and interest in Your Error Corrections, Shared Modifications and Reformatted Specifications.

2.3. Contributor Modifications. You may use, reproduce, modify, display and distribute Contributor Error Corrections, Shared Modifications and Reformatted Specifications, obtained by You under this License, to the same scope and extent as with Original Code, Upgraded Code and Specifications.

2.4. Subcontracting. You may deliver the Source Code of Covered Code to other Licensees having at least a Research Use license, for the sole purpose of furnishing development services to You in connection with Your rights granted in this License. All such Licensees must execute appropriate documents with respect to such work consistent with the terms of this License, and acknowledging their work-made-for-hire status or assigning exclusive right to the work product and associated Intellectual Property Rights to You.

3. Requirements and Responsibilities.

3.1. Research Use License. As a condition of exercising the rights granted under Section 2.1 a) above, You agree to comply with the following:

a) Your Contribution to the Community. All Error Corrections and Shared Modifications which You create or contribute to are automatically subject to the licenses granted under Section 2.2 above. You are encouraged to license all of Your other Modifications under Section 2.2 as Shared Modifications, but are not required to do so. You agree to notify Original Contributor of any errors in the Specification.

b) Source Code Availability. You agree to provide all Your Error Corrections to Original Contributor as soon as reasonably practicable and, in any event, prior to Internal Deployment Use or Commercial Use, if applicable. Original Contributor may, at its discretion, post Source Code for Your Error Corrections and Shared Modifications on the Community Webserver. You may also post Error Corrections and Shared Modifications on a web-server of Your choice; provided, that You must take reasonable precautions to ensure that only Licensees have access to such Error Corrections and Shared Modifications. Such precautions shall include, without limitation, a password protection scheme limited to Licensees and a click-on, download certification of Licensee status required of those attempting to download from the server. An example of an acceptable certification is attached as Attachment A-2.

c) Notices. All Error Corrections and Shared Modifications You create or contribute to must include a file documenting the additions and changes You made and the date of such additions and changes. You must also include the notice set forth in Attachment A-1 in the file header. If it is not possible to put the notice in a particular Source Code file due to its structure, then You must include the notice in a location (such as a relevant directory file), where a recipient would be most likely to look for such a notice.

d) Redistribution.

(i) Source. Covered Code may be distributed in Source Code form only to another Licensee (except for students as provided below). You may not offer or impose any terms on any Covered Code that alter the rights, requirements, or responsibilities of such Licensee. You may distribute Covered Code to students for use in connection with their course work and research projects undertaken at accredited educational institutions. Such students need not be Licensees, but must be given a copy of the notice set forth in Attachment A-3 and such notice must also be included in a file header or prominent location in the Source Code made available to such students.

(ii) Executable. You may distribute Executable version(s) of Covered Code to Licensees and other third parties only for the purpose of evaluation and comment in connection with Research Use by You and under a license of Your choice, but which limits use of such Executable version(s) of Covered Code only to that purpose.

(iii) Modified Class, Interface and Package Naming. In connection with Research Use by You only, You may use Original Contributor's class, interface and package names only to accurately reference or invoke the Source Code files You modify. Original Contributor grants to You a limited license to the extent necessary for such purposes.

(iv) Modifications. You expressly agree that any distribution, in whole or in part, of Modifications developed by You shall only be done pursuant to the term and conditions of this License.

e) Extensions.

(i) Covered Code. You may not include any Source Code of Community Code in any Extensions;

(ii) Publication. No later than the date on which You first distribute such Extension for Commercial Use, You must publish to the industry, on a non-confidential basis and free of all copyright restrictions with respect to reproduction and use, an accurate and current specification for any Extension. In addition, You must make available an appropriate test suite, pursuant to the same rights as the specification, sufficiently detailed to allow any third party reasonably skilled in the technology to produce implementations of the Extension compatible with the specification. Such test suites must be made available as soon as reasonably practicable but, in no event, later than ninety (90) days after Your first Commercial Use of the Extension. You must use reasonable efforts to promptly clarify and correct the specification and the test suite upon written request by Original Contributor.

(iii) Open. You agree to refrain from enforcing any Intellectual Property Rights You may have covering any interface(s) of Your Extension, which would prevent the implementation of such interface(s) by Original Contributor or any Licensee. This obligation does not prevent You from enforcing any Intellectual Property Right You have that would otherwise be infringed by an implementation of Your Extension.

(iv) Class, Interface and Package Naming. You may not add any packages, or any public or protected classes or interfaces with names that originate or might appear to originate from Original Contributor including, without limitation, package or class names which begin with "sun", "java", "javax", "jini", "net.jini", "com.sun" or their equivalents in any subsequent class, interface and/or package naming convention adopted by Original Contributor. It is specifically suggested that You name any new packages using the "Unique Package Naming Convention" as described in "The Java Language Specification" by James Gosling, Bill Joy, and Guy Steele, ISBN 0-201-63451-1, August 1996. Section 7.7 "Unique Package Names", on page 125 of this specification which states, in part:

"You form a unique package name by first having (or belonging to an organization that has) an Internet domain name, such as "sun.com". You then reverse the name, component by component, to obtain, in this example, "Com.sun", and use this as a prefix for Your package names, using a convention developed within Your organization to further administer package names."

3.2. Additional Requirements and Responsibilities. Any additional requirements and responsibilities relating to the Technology are listed in Attachment F (Additional Requirements and Responsibilities), if applicable, and are hereby incorporated into this Section 3.

4. Versions of the License.

4.1. License Versions. Original Contributor may publish revised versions of the License from time to time. Each version will be given a distinguishing version number.

4.2. Effect. Once a particular version of Covered Code has been provided under a version of the License, You may always continue to use such Covered Code under the terms of that version of the License. You may also choose to use such Covered Code under the terms of any subsequent version of the License. No one other than Original Contributor has the right to promulgate License versions.

5. Disclaimer of Warranty.

5.1. COVERED CODE IS PROVIDED UNDER THIS LICENSE "AS IS," WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, WARRANTIES THAT THE COVERED CODE IS FREE OF DEFECTS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. YOU AGREE TO BEAR THE ENTIRE RISK IN CONNECTION WITH YOUR USE AND DISTRIBUTION OF COVERED CODE UNDER THIS LICENSE. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE. NO USE OF ANY COVERED CODE IS AUTHORIZED HEREUNDER EXCEPT SUBJECT TO THIS DISCLAIMER.

5.2. You acknowledge that Original Code, Upgraded Code and Specifications are not designed or intended for use in (i) on-line control of aircraft, air traffic, aircraft navigation or aircraft communications; or (ii) in the design, construction, operation or maintenance of any nuclear facility. Original Contributor disclaims any express or implied warranty of fitness for such uses.

6. Termination.

6.1. By You. You may terminate this Research Use license at anytime by providing written notice to Original Contributor.

6.2. By Original Contributor. This License and the rights granted hereunder will terminate: (i) automatically if You fail to comply with the terms of this License and fail to cure such breach within 30 days of receipt of written notice of the breach; (ii) immediately in the event of circumstances specified in Sections 7.1 and 8.4; or (iii) at Original Contributor's discretion upon any action initiated in the first instance by You alleging that use or distribution by Original Contributor or any Licensee, of Original Code, Upgraded Code, Error Corrections or Shared Modifications contributed by You, or Specifications, infringe a patent owned or controlled by You.

6.3. Effect of Termination. Upon termination, You agree to discontinue use and return or destroy all copies of Covered Code in your possession. All sublicenses to the Covered Code which you have properly granted shall survive any termination of this License. Provisions which, by their nature, should remain in effect beyond the termination of this License shall survive including, without limitation, Sections 2.2, 3, 5, 7 and 8.

6.4. Each party waives and releases the other from any claim to compensation or indemnity for permitted or lawful termination of the business relationship established by this License.

7. Liability.

7.1. Infringement. Should any of the Original Code, Upgraded Code, TCK or Specifications ("Materials") become the subject of a claim of infringement, Original Contributor may, at its sole option, (i) attempt to procure the rights necessary for You to continue using the Materials, (ii) modify the Materials so that they are no longer infringing, or (iii) terminate Your right to use the Materials, immediately upon written notice, and refund to You the amount, if any, having then actually been paid by You to Original Contributor for the Original Code, Upgraded Code and TCK, depreciated on a straight line, five year basis.

7.2. LIMITATION OF LIABILITY. TO THE FULL EXTENT ALLOWED BY APPLICABLE LAW, ORIGINAL CONTRIBUTOR'S LIABILITY TO YOU FOR CLAIMS RELATING TO THIS LICENSE, WHETHER FOR BREACH OR IN TORT, SHALL BE LIMITED TO ONE HUNDRED PERCENT (100%) OF THE AMOUNT HAVING THEN ACTUALLY BEEN PAID BY YOU TO ORIGINAL CONTRIBUTOR FOR ALL COPIES LICENSED HEREUNDER OF THE PARTICULAR ITEMS GIVING RISE TO SUCH CLAIM, IF ANY. IN NO EVENT WILL YOU (RELATIVE TO YOUR SHARED MODIFICATIONS OR ERROR CORRECTIONS) OR ORIGINAL CONTRIBUTOR BE LIABLE FOR ANY INDIRECT, PUNITIVE, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES IN CONNECTION WITH OR ARISING OUT OF THIS LICENSE (INCLUDING, WITHOUT LIMITATION, LOSS OF PROFITS, USE, DATA, OR OTHER ECONOMIC ADVANTAGE), HOWEVER IT ARISES AND ON ANY THEORY OF LIABILITY, WHETHER IN AN ACTION FOR CONTRACT, STRICT LIABILITY OR TORT (INCLUDING NEGLIGENCE) OR OTHERWISE, WHETHER OR NOT YOU OR ORIGINAL CONTRIBUTOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGE AND NOTWITHSTANDING THE FAILURE OF ESSENTIAL PURPOSE OF ANY REMEDY.

8. Miscellaneous.

8.1. Trademark. You agree to comply with the then current Sun Trademark & Logo Usage Requirements accessible through the SCSL Webpage. Except as expressly provided in the License, You are granted no right, title or license to, or interest in, any Sun Trademarks. You agree not to (i) challenge Original Contributor's ownership or use of Sun Trademarks; (ii) attempt to register any Sun Trademarks, or any mark or logo substantially similar thereto; or (iii) incorporate any Sun Trademarks into your own trademarks, product names, service marks, company names, or domain names.

8.2. Integration. This License represents the complete agreement concerning the subject matter hereof.

8.3. Assignment. Original Contributor may assign this License, and its rights and obligations hereunder, in its sole discretion. You may assign the Research Use portions of this License to a third party upon prior written notice to Original Contributor (which may be provided via the Community Web-Server). You may not assign the Commercial Use license or TCK license, including by way of merger (regardless of whether You are the surviving entity) or acquisition, without Original Contributor's prior written consent.

8.4. Severability. If any provision of this License is held to be unenforceable, such provision shall be reformed only to the extent necessary to make it enforceable. Notwithstanding the foregoing, if You are prohibited by law from fully and specifically complying with Sections 2.2 or 3, this License will immediately terminate and You must immediately discontinue any use of Covered Code.

8.5. Governing Law. This License shall be governed by the laws of the United States and the State of California, as applied to contracts entered into and to be performed in California between California residents. The application of the United Nations Convention on Contracts for the International Sale of Goods is expressly excluded.

8.6. Dispute Resolution.

a) Any dispute arising out of or relating to this License shall be finally settled by arbitration as set out herein, except that either party may bring any action, in a court of competent jurisdiction (which jurisdiction shall be exclusive), with respect to any dispute relating to such party's Intellectual Property Rights or with respect to Your compliance with the TCK license. Arbitration shall be administered: (i) by the American Arbitration Association (AAA), (ii) in accordance with the rules of the United Nations Commission on International Trade Law (UNCITRAL) (the "Rules") in effect at the time of arbitration as modified herein; and (iii) the arbitrator will apply the substantive laws of California and United States. Judgment upon the award rendered by the arbitrator may be entered in any court having jurisdiction to enforce such award.

b) All arbitration proceedings shall be conducted in English by a single arbitrator selected in accordance with the Rules, who must be fluent in English and be either a retired judge or practicing attorney having at least ten (10) years litigation experience and be reasonably familiar with the technology matters relative to the dispute. Unless otherwise agreed, arbitration venue shall be in London, Tokyo, or San Francisco, whichever is closest to defendant's principal business office. The arbitrator may award monetary damages only and nothing shall preclude either party from seeking provisional or emergency relief from a court of competent jurisdiction. The arbitrator shall have no authority to award damages in excess of those permitted in this License and any such award in excess is void. All awards will be payable in U.S. dollars and may include, for the prevailing party (i) pre-judgment award interest, (ii) reasonable attorneys' fees incurred in connection with the arbitration, and (iii) reasonable costs and expenses incurred in enforcing the award. The arbitrator will order each party to produce identified documents and respond to no more than twenty-five single question interrogatories.

8.7. Construction. Any law or regulation which provides that the language of a contract shall be construed against the drafter shall not apply to this License.

8.8. U.S. Government End Users. The Covered Code is a "commercial item," as that term is defined in 48 C.F.R. 2.101 (Oct. 1995), consisting of "commercial computer software" and "commercial computer software documentation," as such terms are used in 48 C.F.R. 12.212 (Sept. 1995). Consistent with 48 C.F.R. 12.212 and 48 C.F.R. 227.7202-1 through 227.7202-4 (June 1995), all U.S. Government End Users acquire Covered Code with only those rights set forth herein. You agree to pass this notice to Your licensees.

8.9. Press Announcements. All press announcements relative to the execution of this License must be reviewed and approved by Original Contributor and You prior to release.

8.10. International Use.

a) Export/Import Laws. Covered Code is subject to U.S. export control laws and may be subject to export or import regulations in other countries. Each party agrees to comply strictly with all such laws and regulations and acknowledges their responsibility to obtain such licenses to export, re-export, or import as may be required. You agree to pass these obligations to Your licensees.

b) Intellectual Property Protection. Due to limited intellectual property protection and enforcement in certain countries, You agree not to redistribute the Original Code, Upgraded Code, TCK and Specifications to any country other than the list of restricted countries on the SCSL Webpage.

8.11. Language. This License is in the English language only, which language shall be controlling in all respects, and all versions of this License in any other language shall be for accommodation only and shall not be binding on the parties to this License. All communications and notices made or given pursuant to this License, and all documentation and support to be provided, unless otherwise noted, shall be in the English language.

PLEASE READ THE TERMS OF THIS LICENSE CAREFULLY. BY CLICKING ON THE "ACCEPT" BUTTON BELOW YOU ARE ACCEPTING AND AGREEING TO THE TERMS AND CONDITIONS OF THIS LICENSE WITH SUN MICROSYSTEMS, INC. IF YOU ARE AGREEING TO THIS LICENSE ON BEHALF OF A COMPANY, YOU REPRESENT THAT YOU ARE AUTHORIZED TO BIND THE COMPANY TO SUCH A LICENSE. WHETHER YOU ARE ACTING ON YOUR OWN BEHALF, OR REPRESENTING A COMPANY, YOU MUST BE OF MAJORITY AGE AND BE OTHERWISE COMPETENT TO ENTER INTO CONTRACTS. IF YOU DO NOT MEET THIS CRITERIA OR YOU DO NOT AGREE TO ANY OF THE TERMS AND CONDITIONS OF THIS LICENSE, CLICK ON THE REJECT BUTTON TO EXIT.

ACCEPT REJECT

GLOSSARY

1. "Commercial Use" means any use (excluding Internal Deployment Use) or distribution, directly or indirectly of Compliant Covered Code by You to any third party, alone or bundled with any other software or hardware, for direct or indirect commercial or strategic gain or advantage, subject to execution of Attachment D by You and Original Contributor.

2. "Community Code" means the Original Code, Upgraded Code, Error Corrections, Shared Modifications, or any combination thereof.

3. "Community Webserver(s)" means the webservers designated by Original Contributor for posting Error Corrections and Shared Modifications.

4. "Compliant Covered Code" means Covered Code that complies with the requirements of the TCK.

5. "Contributor" means each Licensee that creates or contributes to the creation of any Error Correction or Shared Modification.

6. "Covered Code" means the Original Code, Upgraded Code, Modifications, or any combination thereof.

7. "Error Correction" means any change made to Community Code which conforms to the Specification and corrects the adverse effect of a failure of Community Code to perform any function set forth in or required by the Specifications.

8. "Executable" means Covered Code that has been converted to a form other than Source Code.

9. "Extension(s)" means any additional classes or other programming code and/or interfaces developed by or for You which: (i) are designed for use with the Technology; (ii) constitute an API for a library of computing functions or services; and (iii) are disclosed to third party software developers for the purpose of developing software which invokes such additional classes or other programming code and/or interfaces. The foregoing shall not apply to software development by Your subcontractors to be exclusively used by You.

10. "Intellectual Property Rights" means worldwide statutory and common law rights associated solely with (i) patents and patent applications; (ii) works of authorship including copyrights, copyright applications, copyright registrations and "moral rights"; (iii) the protection of trade and industrial secrets and confidential information; and (iv) divisions, continuations, renewals, and re-issuances of the foregoing now existing or acquired in the future.

11. "Internal Deployment Use" means use of Compliant Covered Code (excluding Research Use) within Your business or organization only by Your employees and/or agents, subject to execution of Attachment C by You and Original Contributor, if required.

12. "Licensee" means any party that has entered into and has in effect a version of this License with Original Contributor.

13. "Modification(s)" means (i) any change to Covered Code; (ii) any new file or other representation of computer program statements that contains any portion of Covered Code; and/or (iii) any new Source Code implementing any portion of the Specifications.

14. "Original Code" means the initial Source Code for the Technology as described on the Technology Download Site.

15. "Original Contributor" means Sun Microsystems, Inc., its affiliates and its successors and assigns.

16. "Reformatted Specifications" means any revision to the Specifications which translates or reformats the Specifications (as for example in connection with Your documentation) but which does not alter, subset or superset the functional or operational aspects of the Specifications.

17. "Research Use" means use and distribution of Covered Code only for Your research, development, educational or personal and individual use, and expressly excludes Internal Deployment Use and Commercial Use.

18. "SCSL Webpage" means the Sun Community Source license webpage located at http://sun.com/software/communitysource, or such other url that Original Contributor may designate from time to time.

19. "Shared Modifications" means Modifications provided by You, at Your option, pursuant to Section 2.2, or received by You from a Contributor pursuant to Section 2.3.

20. "Source Code" means computer program statements written in any high-level, readable form suitable for modification and development.

21. "Specifications" means the specifications for the Technology and other documentation, as designated on the Technology Download Site, as may be revised by Original Contributor from time to time.

22. "Sun Trademarks" means Original Contributor's SUN, JAVA, and JINI trademarks and logos, whether now used or adopted in the future.

23. "Technology" means the technology described in Attachment B, and Upgrades.

24. "Technology Compatibility Kit" or "TCK" means the test programs, procedures and/or other requirements, designated by Original Contributor for use in verifying compliance of Covered Code with the Specifications, in conjunction with the Original Code and Upgraded Code. Original Contributor may, in its sole discretion and from time to time, revise a TCK to correct errors and/or omissions and in connection with Upgrades.

25. "Technology Download Site" means the site(s) designated by Original Contributor for access to the Original Code, Upgraded Code, TCK and Specifications.

26. "Upgrade(s)" means new versions of Technology designated exclusively by Original Contributor as an Upgrade and released by Original Contributor from time to time.

27. "Upgraded Code" means the Source Code for Upgrades, possibly including Modifications made by Contributors.

28. "You(r)" means an individual, or a legal entity acting by and through an individual or individuals, exercising rights either under this License or under a future version of this License issued pursuant to Section 4.1. For legal entities, "You(r)" includes any entity that by majority voting interest controls, is controlled by, or is under common control with You.

ATTACHMENT A REQUIRED NOTICES

ATTACHMENT A-1 REQUIRED IN ALL CASES

"The contents of this file, or the files included with this file, are subject to the current version of Sun Community Source License for [fill in name of applicable Technology] (the "License"); You may not use this file except in compliance with the License. You may obtain a copy of the License at http:// sun.com/software/communitysource. See the License for the rights, obligations and limitations governing use of the contents of the file.

The Original and Upgraded Code is [fill in name of applicable Technology]. The developer of the Original and Upgraded Code is Sun Microsystems, Inc. Sun Microsystems, Inc. owns the copyrights in the portions it created. All Rights Reserved.

Contributor(s): --

Associated Test Suite(s) Location: --"

ATTACHMENT A-2 SAMPLE LICENSEE CERTIFICATION

"By clicking the 'Agree' button below, You certify that You are a Licensee in good standing under the Sun Community Source License, [fill in name of applicable Technology] ("License") and that Your access, use and distribution of code and information You may obtain at this site is subject to the License."

ATTACHMENT A-3 REQUIRED STUDENT NOTIFICATION

"This software and related documentation has been obtained by your educational institution subject to the Sun Community Source License, [fill in name of applicable Technology]. You have been provided access to the software and related documentation for use only in connection with your course work and research activities as a matriculated student of your educational institution. Any other use is expressly prohibited.

THIS SOFTWARE AND RELATED DOCUMENTATION CONTAINS PROPRIETARY MATERIAL OF SUN MICROSYSTEMS, INC, WHICH ARE PROTECTED BY VARIOUS INTELLECTUAL PROPERTY RIGHTS.

You may not use this file except in compliance with the License. You may obtain a copy of the License on the web at http://sun.com/software/ communitysource."

ATTACHMENT B

Java (tm) Platform, Standard Edition, JDK 6 Source Technology

Description of "Technology"

Java (tm) Platform, Standard Edition, JDK 6 Source Technology as described on the Technology Download Site.

ATTACHMENT C INTERNAL DEPLOYMENT USE

This Attachment C is only effective for the Technology specified in Attachment B, upon execution of Attachment D (Commercial Use License) including the requirement to pay royalties. In the event of a conflict between the terms of this Attachment C and Attachment D, the terms of Attachment D shall govern.

1. Internal Deployment License Grant. Subject to Your compliance with Section 2 below, and Section 8.10 of the Research Use license; in addition to the Research Use license and the TCK license, Original Contributor grants to You a worldwide, non-exclusive license, to the extent of Original Contributor's Intellectual Property Rights covering the Original Code, Upgraded Code and Specifications, to do the following:

a) reproduce and distribute internally, Original Code and Upgraded Code as part of Compliant Covered Code, and Specifications, for Internal Deployment Use,

b) compile such Original Code and Upgraded Code, as part of Compliant Covered Code, and reproduce and distribute internally the same in Executable form for Internal Deployment Use, and

c) reproduce and distribute internally, Reformatted Specifications for use in connection with Internal Deployment Use.

2. Additional Requirements and Responsibilities. In addition to the requirements and responsibilities described under Section 3.1 of the Research Use license, and as a condition to exercising the rights granted under Section 3 above, You agree to the following additional requirements and responsibilities:

2.1. Compatibility. All Covered Code must be Compliant Covered Code prior to any Internal Deployment Use or Commercial Use, whether originating with You or acquired from a third party. Successful compatibility testing must be completed in accordance with the TCK License. If You make any further Modifications to any Covered Code previously determined to be Compliant Covered Code, you must ensure that it continues to be Compliant Covered Code.

ATTACHMENT D COMMERCIAL USE LICENSE

[Contact Sun Microsystems For Commercial Use Terms and Conditions]

ATTACHMENT E TECHNOLOGY COMPATIBILITY KIT

The following license is effective for the Java (tm) Platform, Standard Edition, JDK 6 Technology Compatibility Kit only upon execution of a separate support agreement between You and Original Contributor (subject to an annual fee) as described on the SCSL Webpage. The applicable Technology Compatibility Kit for the Technology specified in Attachment B may be accessed at the Technology Download Site only upon execution of the support agreement.

1. TCK License.

a) Subject to the restrictions set forth in Section 1.b below and Section 8.10 of the Research Use license, in addition to the Research Use license, Original Contributor grants to You a worldwide, non-exclusive, non-transferable license, to the extent of Original Contributor's Intellectual Property Rights in the TCK (without the right to sublicense), to use the TCK to develop and test Covered Code.

b) TCK Use Restrictions. You are not authorized to create derivative works of the TCK or use the TCK to test any implementation of the Specification that is not Covered Code. You may not publish your test results or make claims of comparative compatibility with respect to other implementations of the Specification. In consideration for the license grant in Section 1.a above you agree not to develop your own tests which are intended to validate conformation with the Specification.

2. Requirements for Determining Compliance.

2.1. Definitions.

a) "Added Value" means code which:

(i) has a principal purpose which is substantially different from that of the stand-alone Technology;

(ii) represents a significant functional and value enhancement to the Technology;

(iii) operates in conjunction with the Technology; and

(iv) is not marketed as a technology which replaces or substitutes for the Technology.

b) "Java Classes" means the specific class libraries associated with each Technology defined in Attachment B.

c) "Java Runtime Interpreter" means the program(s) which implement the Java virtual machine for the Technology as defined in the Specification.

d) "Platform Dependent Part" means those Original Code and Upgraded Code files of the Technology which are not in a share directory or subdirectory thereof.

e) "Shared Part" means those Original Code and Upgraded Code files of the Technology which are identified as "shared" (or words of similar meaning) or which are in any "share" directory or subdirectory thereof, except those files specifically designated by Original Contributor as modifiable.

f) "User's Guide" means the users guide for the TCK which Original Contributor makes available to You to provide direction in how to run the TCK and properly interpret the results, as may be revised by Original Contributor from time to time.

2.2. Development Restrictions. Compliant Covered Code:

a) must include Added Value;

b) must fully comply with the Specifications for the Technology specified in Attachment B;

c) must include the Shared Part, complete and unmodified;

d) may not modify the functional behavior of the Java Runtime Interpreter or the Java Classes;

e) may not modify, subset or superset the interfaces of the Java Runtime Interpreter or the Java Classes;

f) may not subset or superset the Java Classes;

C.3 Apache License 2.0

NekoHTML is licensed under version 2 of the Apache License.

C.4 Common Public License 1.0

JUnit is distributed under the Common Public License, version 1.0

b) Subject to the terms of this Agreement, each Contributor hereby grants Recipient a non-exclusive, worldwide, royalty-free patent license under Licensed Patents to make, use, sell, offer to sell, import and otherwise transfer the Contribution of such Contributor, if any, in source code and object code form. This patent license shall apply to the combination of the Contribution and the Program if, at the time the Contribution is added by the Contributor, such addition of the Contribution causes such combination to be covered by the Licensed Patents. The patent license shall not apply to any other combinations which include the Contribution. No hardware per se is licensed hereunder.

c) Recipient understands that although each Contributor grants the licenses to its Contributions set forth herein, no assurances are provided by any Contributor that the Program does not infringe the patent or other intellectual property rights of any other entity. Each Contributor disclaims any liability to Recipient for claims brought by any other entity based on infringement of intellectual property rights or otherwise. As a condition to exercising the rights and licenses granted hereunder, each Recipient hereby assumes sole responsibility to secure any other intellectual property rights needed, if any. For example, if a third party patent license is required to allow Recipient to distribute the Program, it is Recipient's responsibility to acquire that license before distributing the Program.

d) Each Contributor represents that to its knowledge it has sufficient copyright rights in its Contribution, if any, to grant the copyright license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under its own license agreement, provided that:

a) it complies with the terms and conditions of this Agreement; and

b) its license agreement:

i) effectively disclaims on behalf of all Contributors all warranties and conditions, express and implied, including warranties or conditions of title and non-infringement, and implied warranties or conditions of merchantability and fitness for a particular purpose;

ii) effectively excludes on behalf of all Contributors all liability for damages, including direct, indirect, special, incidental and consequential damages, such as lost profits;

iii) states that any provisions which differ from this Agreement are offered by that Contributor alone and not by any other party; and

iv) states that source code for the Program is available from such Contributor, and informs licensees how to obtain it in a reasonable manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

a) it must be made available under this Agreement; and

b) a copy of this Agreement must be included with each copy of the Program.

Contributors may not remove or alter any copyright notices contained within the Program.

Each Contributor must identify itself as the originator of its Contribution, if any, in a manner that reasonably allows subsequent Recipients to identify the originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with respect to end users, business partners and the like. While this license is intended to facilitate the commercial use of the Program, the Contributor who includes the Program in a commercial product offering should do so in a manner which does not create potential liability for other Contributors. Therefore, if a Contributor includes the Program in a commercial product offering, such Contributor ("Commercial Contributor") hereby agrees to defend and indemnify every other Contributor ("Indemnified Contributor") against any losses, damages and costs (collectively "Losses") arising from claims, lawsuits and other legal actions brought by a third party against the Indemnified Contributor to the extent caused by the acts or omissions of such Commercial Contributor in connection with its distribution of the Program in a commercial product offering. The obligations in this section do not apply to any claims or Losses relating to any actual or alleged intellectual property infringement. In order to qualify, an Indemnified Contributor must: a) promptly notify the Commercial Contributor in writing of such claim, and b) allow the Commercial Contributor to control, and cooperate with the Commercial Contributor in, the defense and any related settlement negotiations. The Indemnified Contributor may participate in any such claim at its own expense.

For example, a Contributor might include the Program in a commercial product offering, Product X. That Contributor is then a Commercial Contributor. If that Commercial Contributor then makes performance claims, or offers warranties related to Product X, those performance claims and warranties are such Commercial Contributor's responsibility alone. Under this section, the Commercial Contributor would have to defend claims against the other Contributors related to those performance claims and warranties, and if a court requires any other Contributor to pay any damages as a result, the Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE, NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each Recipient is solely responsible for determining the appropriateness of using and distributing the Program and assumes all risks associated with its exercise of rights under this Agreement, including but not limited to the risks and costs of program errors, compliance with applicable laws, damage to or loss of data, programs or equipment, and unavailability or interruption of operations.

6. DISCLAIMER OF LIABILITY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this Agreement, and without further action by the parties hereto, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against a Contributor with respect to a patent applicable to software (including a cross-claim or counterclaim in a lawsuit), then any patent licenses granted by that Contributor to such Recipient under this Agreement shall terminate as of the date such litigation is filed. In addition, if Recipient institutes patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Program itself (excluding combinations of the Program with other software or hardware) infringes such Recipient's patent(s), then such Recipient's rights granted under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to comply with any of the material terms or conditions

C.5 X License

ICU is distributed under the X License, also known as the MIT License.

C.6 Creative Commons Attribution-Sharealike 3.0 Unported License

Text drawn from the Wikipedia is licensed under the Creative Commons Attribution-Sharealike (CC-BY-SA) 3.0 Unported License, which we reproduce below. The creative commons website is `http://creativecommons.org/`.

C.6.1 Creative Commons Deed

Index