INSTANT

Short | Fast | Focused

Web Scraping with Java

Build simple scrapers or vast armies of Java-based bots to untangle and capture the Web

Ryan Mitchell

[PACKT] PUBLISHING

Instant Web Scraping with Java

Build simple scrapers or vast armies of Java-based bots to untangle and capture the Web

Ryan Mitchell

[PACKT]
PUBLISHING

BIRMINGHAM - MUMBAI

Instant Web Scraping with Java

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: August 2013

Production Reference: 1230813

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-84969-688-3

www.packtpub.com

Credits

Author
Ryan Mitchell

Reviewers
Benjamin Hill
Sumant Kumar Raja

Acquisition Editor
Mary Nadar

Commissioning Editors
Sharvari Tawde
Ameya Sawant

Technical Editor
Akashdeep Kundu

Project Coordinator
Joel Goveya

Proofreader
Jonathan Todd

Graphics
Abhinash Sahu

Production Coordinator
Conidon Miranda

Cover Work
Conidon Miranda

Cover Image
Sheetal Aute

About the Author

Ryan Mitchell has ten years of programming experience, including Java, C, Perl, PHP, and Python. In addition to "traditional" programming, she specializes in web technologies, with three years of Drupal development experience, and is Sitecore developer certified.

She graduated from Olin College of Engineering and is currently studying at the Harvard University Extension School for a Masters in Software Engineering.

In addition to academic life, she currently works at Velir Studios as a Web Systems Analyst, and has also worked as a developer for Harvard University and Abine Inc.

About the Reviewers

Benjamin Hill is a product manager working in online and mobile applications. His background includes everything from a bootstrapped crowdsourcing startup in 2006 to companies as big as Comcast and eBay. During that time he has learned a variety of techniques to automate websites that don't yet provide APIs.

Sumant Kumar Raja is an entrepreneur, integration architect, and independent consultant with seven years of experience working for global clients. He has worked on a wide range of complex technical environments including J2EE and SAP Process Integration. He is a Certified ScrumMaster.

He is director of Denarit Consulting Ltd. UK. In past, he has worked in various roles with Accenture for clients in the UK and the U.S.

> I am thankful to my family for their unconditional support.
>
> I would like to extend my thanks to Packt Publishing for giving me the opportunity to review this book.

Table of Contents

Preface

Welcome to *Instant Web Scraping with Java*! Although web scraping may seem like a fairly specific topic, there's more to it than simply turning URLs into HTML. What happens when you find that a page has a redirect, the server has placed a rate limiter on your IP address, or the data you want is behind a wall of Ajax or a form? How do you solve problems in a robust way? What do you do with the data after you get it?

This book assumes that you have some basic foundation in programming, and probably know your way around the command line a little. You can troubleshoot and improvise, and aren't afraid to play around with the code. Although we'll hold your hand and walk you through the basic examples, you're not going to get very far unless you combine different techniques learned in the book in order to fit your needs. In the *There's more...* section of each recipe, we'll give you some pointers on where to go from here, but there's a lot of ground to cover and not many pages to do it in.

If you're new to Java, I would recommend starting by carefully reading the recipes on *Setting up your Java Environment (Simple)* and *Writing and executing HelloWorld.java (Simple)*. More advanced programmers may be fine starting with the sections on RMI and threading. Whenever a later recipe depends on code used in a previous recipe, it will be mentioned in the *Getting ready* section of the recipe.

What this book covers

Setting up your Java Environment (Simple) explains that Java is not a lightweight language, and it requires some similarly heavy-duty installations. We'll guide you through setting up the Java Development Kit, as well as an Integrated Development Environment.

Writing and executing HelloWorld.java (Simple) shows you the basics of writing and executing basic Java programs, and introduces you to the architecture of the language. If you've never written a line of Java in your life, this recipe will be pretty helpful.

Writing a simple scraper (Simple) introduces you to some basic scraping functions and outputs some text grabbed from the Internet. If you only write one scraper this year, write this one.

Writing a more complicated scraper (Intermediate) teaches you that you have to crawl before you can walk. Let's write a scraper that traverses a series of links and runs indefinitely, reporting data back as it goes.

Handling errors (Simple) explains that the web is dark and full of terrors, but that won't stop our scrapers. Learn how to handle missing code, broken links, and other unexpected situations gracefully by throwing and catching errors.

Writing robust, scalable code (Advanced) teaches you how to create multiple packages, classes, and methods that communicate with one another, scrape multiple sites at once, and document your code so others can follow along. If you wanted to write one-page scripts you would have used Perl.

Persisting data (Advanced) explains how to connect to remote databases and store results from the web for later retrieval. Nothing lasts forever, but we can try.

Writing tests (Intermediate) shows you the tips and tricks of the JUnit testing framework. JUnit is especially important in web scraping, and many Java web scraping libraries are based off of JUnit. It will be used in several recipes in the book. It's important whether you want to check your own website for unexpected problems, or poke around someone else's.

Going undercover (Intermediate) explains how to masquerade as a web browser in order to prevent being blocked by sites. Although nearly every piece of data on the web is accessible to scrapers, some is a little harder to get to than others.

Submitting a basic form (Advances) teaches you to take off the gloves and actually interact with the sites by filling in fields, clicking buttons, and submitting forms. So far we've been looking but not touching when we crawl websites.

Scraping Ajax pages (Advanced) explains that the dynamic web brings some difficult challenges for web scrapers. Learn how to execute JavaScript as an actual web browser would, and access data that might be hidden by a layer of Ajax.

Faster scraping through threading (Intermediate) understands that you've moved beyond running scripts on your laptop and want to take full advantage of a dedicated server or larger machine, and explains how to create multithreaded scrapers that return data as fast as possible, without being hung up waiting for page loads.

Faster scraping with RMI (Advanced) shows you that using multiple servers for large scraping operations isn't necessary just for the increase in speed (although that is nice!) but for the increased robustness. Blocking one IP address is easy, but blocking all of them is hard. Learn how to communicate between servers using Java's built-in Remote Method Invocation.

What you need for this book

To work through most recipes in this book, all you need is an Internet-connected computer, running Windows, Mac OS X, or Linux. We'll walk you through installing all software you will need to complete the recipes. Access to a MySQL database, and/or access to a remote server might also come in handy, although they are not necessary.

Who this book is for

This book is aimed toward programmers who might be, but are not necessarily, Java developers. Although we'll mention a few of the basics of Java, and code samples will be progressively more advanced throughout the book, this book is not intended to be used as a Java instructional manual.

In addition to a programming background, we'll assume that you have a basic grasp of web technologies including HTML, CSS, XML, JavaScript, Ajax, SQL, and some understanding of how servers communicate across the web.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "The location doesn't matter, but keep in mind that you'll be typing the location into the command line quite a bit—something short like `/Users/rmitchell/workspace` is usually convenient."

A block of code is set as follows:

```
public class Hello {
  public static void main(String[] args) {
      System.out.println("hello, world!");
  }
```

Any command-line input or output is written as follows:

```
$start rmiregistry
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "You can keep an eye on the project, learn about changes with each release, and perhaps even get involved in this open source effort at the **Project Information** page:"

Warnings or important notes appear in a box like this.

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to `feedback@packtpub.com`, and mention the book title via the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on `www.packtpub.com/authors`.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting `http://www.packtpub.com/submit-errata`, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from `http://www.packtpub.com/support`.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt Publishing, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at `copyright@packtpub.com` with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at `questions@packtpub.com` if you are having a problem with any aspect of the book, and we will do our best to address it.

Instant Web Scraping with Java

Welcome to *Instant Web Scraping with Java*! Web scraping is an automated process that involves some amount of data parsing in order to obtain only the information that you need. Although using an API (commonly by sending a GET request to a host to retrieve JSON data) might technically be considered web scraping (after all, you are retrieving and parsing data), it is generally considered to be applicable only to sites that are designed to be viewed by humans through a web browser, rather than by an automated system. By retrieving the complete HTML from a website, parsing it into an object using any number of available libraries, and isolating and processing the desired data, you are freed from the limitations and availabilities of APIs already developed to retrieve the data.

> *As web scrapers like to say: "Every website has an API. Sometimes it's just poorly documented and difficult to use."*

To say that web scraping is a useful skill is an understatement. Whether you're satisfying a curiosity by writing a quick script in an afternoon or building the next Google, the ability to grab any online data, in any amount, in any format, while choosing how you want to store it and retrieve it, is a vital part of any good programmer's toolkit.

By virtue of reading this book, I assume that you already have some idea of what web scraping entails, and perhaps have specific motivations for using Java to do it. Regardless, it is important to cover some basic definitions and principles.

Very rarely does one hear about web scraping in Java—a language often thought to be solely the domain of enterprise software and interactive web apps of the 90's and early 2000's.

However, there are many reasons why Java is an often underrated language for web scraping:

- Java's excellent exception-handling lets you compile code that elegantly handles the often-messy Web
- Reusable data structures allow you to write once and use everywhere with ease and safety
- Java's concurrency libraries allow you to write code that can process other data while waiting for servers to return information (the slowest part of any scraper)
- The Web is big and slow, but the Java RMI allows you to write code across a distributed network of machines, in order to collect and process data quickly
- There are a variety of standard libraries for getting data from servers, as well as third-party libraries for parsing this data, and even executing JavaScript (which is needed for scraping some websites)

In this book, we will explore these, and other benefits of Java in web scraping, and build several scrapers ourselves. Although it is possible, and recommended, to skip to the sections you already have a good grasp of, keep in mind that some sections build up the code and concepts of other sections. When this happens, it will be noted in the beginning of the section.

How is this legal?

Web scraping has always had a "gray hat" reputation. While websites are generally meant to be viewed by actual humans sitting at a computer, web scrapers find ways to subvert that. While APIs are designed to accept obviously computer-generated requests, web scrapers must find ways to imitate humans, often by modifying headers, forging POST requests and other methods.

Web scraping often requires a great deal of problem solving and ingenuity to figure out how to get the data you want. There are often few roadmaps or tried-and-true procedures to follow, and you must carefully tailor the code to each website—often riding between the lines of what is intended and what is possible. Although this sort of hacking can be fun and challenging, you have to be careful to follow the rules.

Like many technology fields, the legal precedent for web scraping is scant. A good rule of thumb to keep yourself out of trouble is to always follow the terms of use and copyright documents on websites that you scrape (if any).

There are some cases in which the act of web crawling is itself in murky legal territory, regardless of how the data is used. Crawling is often prohibited in the terms of service of websites where the aggregated data is valuable (for example, a site that contains a directory of personal addresses in the United States), or where a commercial or rate-limited API is available. Twitter, for example, explicitly prohibits web scraping (at least of any actual tweet data) in its terms of service:

> *"crawling the Services is permissible if done in accordance with the provisions of the robots.txt file, however, scraping the Services without the prior consent of Twitter is expressly prohibited"*

Unless explicitly prohibited by the terms of service, there is no fundamental difference between accessing a website (and its information) via a browser, and accessing it via an automated script. The `robots.txt` file alone has not been shown to be legally binding, although in many cases the terms of service can be.

Setting up your Java Environment (Simple)

No more programming in Notepad. Although it's technically possible to write Java code in anything that can save a text file, it's also technically possible to recreate the Internet using carrier pigeons (See `http://www.ietf.org/rfc/rfc1149`). But we don't do that either.

Getting ready

The first step to writing in a new language is to make sure you have the language's compilers and/or interpreters installed. If you do not have the Java Development Kit (JDK) installed already, you can download it from Oracle's website from the following link:

`http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html`

Writing Java can often become tedious when dealing with Java's often finicky demands. Because Java is a strongly typed language, it is necessary to remember which datatype a particular variable is, in a particular section of code, and which operations can be done on it. A class that has been defined to accept a `float` value in one file cannot be called with an argument of a `double` type in another file, and a static class cannot be called in a non-static context later on. When navigating Java, it helps to have a tour guide on your team—that's where the IDE comes in.

A good Integrated Development Environment (IDE) is nearly indispensable to work with Java, especially with complex projects. An IDE allows you to catch compile time bugs and syntax errors on the fly, use advanced debugging features (no more `print` statements in every line), inspect objects, and even suggest functions and auto-complete code for you.

There are many excellent IDEs on the market. The most popular among them are Eclipse, NetBeans, and JCreator. However, you should use the IDE you feel most comfortable with if you have a preference. We will be working with Eclipse in this book, and any reference to tasks done in the IDE will assume that you are using Eclipse.

Eclipse Classic can be found for Windows, Mac OS X, and Linux at the following link:

`http://www.eclipse.org/downloads/packages/eclipse-classic-422/junosr2`

The download will likely take a few minutes to finish, so read through the rest of the recipe while it's working.

How to do it...

1. Assuming you don't have the JDK installed (if you have it, congratulations! Feel free to skip the steps involving downloading and installing the JDK), make sure that you have the files for both Eclipse and the JDK downloaded.
2. Install the JDK first (unless you are an expert and have a strong reason not to, opt to install all packages that it suggests).
3. Use the default folder for installation (for Windows the default folder is `C:\java\jre7`).

> For more information about system requirements, installation, and troubleshooting installation, Oracle's JDK 7 and JRE 7 Installation Guide is an excellent resource. You can find the guide from the following link:
>
> `http://docs.oracle.com/javase/7/docs/webnotes/install`

4. After successfully installing the JDK, extract the Eclipse downloaded file to any folder. You will find the Eclipse application inside the extracted folder. I suggest creating a shortcut to Eclipse somewhere convenient, such as in your `Programs` or `Applications` folder, or on your desktop.
5. The first time you run Eclipse, it will prompt you to create a workspace. This is a location where all of your Java files will be stored. The location doesn't matter, but keep in mind that you'll be typing the location into the command line quite a bit—something short like `/Users/rmitchell/workspace` is usually convenient.

How it works...

The JDK is a collection of useful tools for developing Java software. The most important of these is the **javac** (Java compiler, pronounced as "java-see"), which translates your written Java files into Java bytecode. You may be familiar with Java bytecode as the `.class` files that are created after your `.java` files are compiled.

> This Java bytecode is read by the **Java Virtual Machine** (**JVM**) and is translated into lower-level assembly language specific to each operating system. JVMs can also be written to run at a hardware level, resulting in faster-running code, although this is less common.

The mantra of Sun Microsystems, while developing Java, was:

> *"Write once, run anywhere."*

This was accomplished through the creation of JVMs, custom-written to precise specifications, in order to add a buffer between inconsistent operating systems and processors, and the Java bytecode that was being run on them. Rather than writing software for each environment it would be running in, Java programmers could write it once, compile, and let the JVM take care of the rest.

Several other useful applications come bundled with the JDK that we will use. Among these are the Java Archive (JAR), which is based on the `.zip` format and used to bundle collections of executable Java files and any associated resources needed. Another application of this type is `Javadoc`, a documentation generator that aggregates source code comments and other information to create HTML-based documents for your code.

There's more...

We've only mentioned a few of the dozens of technologies that combine to form the Java platform. However, you don't really need to know about much other than a handful of interface-level tools and APIs in order to write some serious Java code. Having a better understanding of how everything fits together can be useful for improving your control over the language and understanding the documentation when the going gets tough. An overview of the Java platform and complete documentation for the language can be found on Oracle's website, at `http://docs.oracle.com/javase/7/docs/`.

Writing and executing HelloWorld.java (Simple)

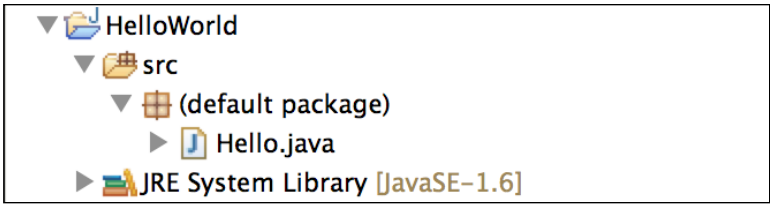
The classic "Hello, World" program was first introduced in a 1974 Bell Labs paper on C by Brian Kernighan. As C is the spiritual ancestor of Java, it seems appropriate to include it here. In this recipe, we will cover the basic syntax of a Java program, and ensure that your Java environment is working correctly.

Getting ready

You will need to have completed the previous recipe, *Setting up your Java Environment (Simple)*, to continue through this recipe. Although the instructions will be approximately the same using other IDEs, we recommend that you use Eclipse in order to follow along with the recipes in this book.

How to do it...

1. Open Eclipse.
2. In the main menu, navigate to **File** | **New** | **Project...**.
3. Select the top option, **Java Project,** and click on the **Next** button.
4. Enter `HelloWorld` as the name of the new project, keep the **Use default location** checkbox selected, and then click on the **Finish** button.
5. In the main menu, navigate to **File** | **New** | **Class**.
6. Enter `Hello` as the **Name** for your new class. Under the **Which method stubs would you like to create?** checkbox heading, select **public static void main(String[] args)** and then click on the **Finish** button.
7. In Eclipse's Package Explorer, on your left (if you do not see the Package Explorer, enable it by selecting from the main menu: **Window** | **Show View** | **Package Explorer**) you should see an architecture of your new Java package that looks like the following screenshot:

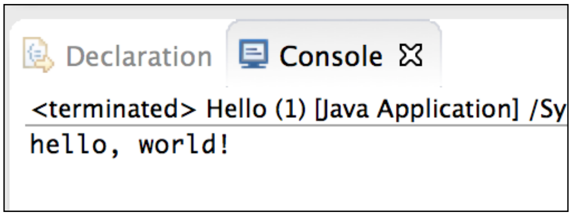


8. Select the `Hello.java` class that you've created, and you'll find that most of the code has been written for you automatically by Eclipse (see, I told you using an IDE is fantastic!). To complete your "Hello, World" program, add the code line `System.out.println("Hello, World");` to your main method so that the code looks like the following:

```
public class Hello {
  public static void main(String[] args) {
      System.out.println("hello, world!");
  }
}
```

9. Save the file, and run it. There are several ways to execute code in Eclipse:
   - By clicking on the round green run icon in Eclipse's toolbar below the main menu
   - By selecting **Run** | **Run** in the main menu
   - Or by pressing *Ctrl* + *F11* (*shift* + *command* + *F11* on a Mac)

10. In the **Console** tab below the code window (if you do not see the **Console** tab, go to **Window** | **Show View** | **Console** in the main menu), you should see the following text:



```
<terminated> Hello (1) [Java Application] /Sy
hello, world!
```

How it works...

One of the common complaints about Java is that it requires at least five lines of code to create a basic "Hello, World" program. But if all you wanted to do was print out **Hello, World** to the console, you'd be using a different language. With the program we've created, we've actually created several things, which are:

- A new project, `HelloWorld` project provides a way for Java programmers to keep all components of a piece of software together. For example, packages, which are generally bundled up into `.jar` files, for distribution.
- A new class, `Hello`, that exists in the project `HelloWorld`. Classes are similar to objects in other languages, and contain multiple methods (functions) that define the behavior of that object.

- A main method within the class `Hello`. This main method is always the starting point at the time of calling a Java class, and is declared to be of type `public static` (which make it publicly available outside its own class, and static means that it is callable) with a return type of `void`, and an argument type of an array of `Strings`.
- Some code inside this main class that handles the *work* of the program (in this case, it simply prints **Hello, World!**, but it gets better from here—I promise!).

There's more...

Especially if you have no exposure, or are rusty with Java, now is the time to play around with it. Although we will be going over most of the complex syntax used in the exercises in the book, depending on your level of Java experience, I recommend that you take a look at Oracle's Language Syntax guide at `https://wikis.oracle.com/display/code/Language+Syntax` and get comfortable with the basic operations, string manipulation, arrays, and control flow syntax. From here on, we'll assume that you have some basic understanding of Java syntax—at least enough to follow along with the basic written code.

Writing a simple scraper (Simple)

Wikipedia is not just a helpful resource for researching or looking up information but also a very interesting website to scrape. They make no efforts to prevent scrapers from accessing the site, and, with a very well-marked-up HTML, they make it very easy to find the information you're looking for. In this project, we will scrape an article from Wikipedia and retrieve the first few lines of text from the body of the article.

Getting ready

It is recommended that you complete the first two recipes in this book, or at least have some working knowledge of Java, and the ability to create and execute Java programs at this point.

As an example, we will use the article from the following Wikipedia link:

`http://en.wikipedia.org/wiki/Java`

Note that this article is about the Indonesian island nation Java, not the programming language. Regardless, it seems appropriate to use it as a test subject.

We will be using the jsoup library to parse HTML data from websites and convert it into a manipulatable object, with traversable, indexed values (much like an array). In this exercise, we will show you how to download, install, and use Java libraries. In addition, we'll also be covering some of the basics of the jsoup library in particular.

How to do it...

Now that we're starting to get into writing scrapers, let's create a new project to keep them all bundled together. Carry out the following steps for this task:

1. As shown in the previous recipe, open Eclipse and create a new Java project called `Scraper`.
2. Although we didn't create a Java package in the previous recipe, packages are still considered to be handy for bundling collections of classes together within a single project (projects contain multiple packages, and packages contain multiple classes). You can create a new package by highlighting the `Scraper` project in Eclipse and going to **File** | **New** | **Package**.
3. By convention, in order to prevent programmers from creating packages with the same name (and causing namespace problems), packages are named starting with the reverse of your domain name (for example, `com.mydomain.mypackagename`). For the rest of the book, we will begin all our packages with `com.packtpub.JavaScraping` appended with the package name. Let's create a new package called `com.packtpub.JavaScraping.SimpleScraper`.
4. Create a new class, `WikiScraper`, inside the `src` folder of the package.
5. Download the jsoup core library, the first link, from the following URL:

   `http://jsoup.org/download`
6. Place the `.jar` file you downloaded into the `lib` folder of the package you just created.
7. In Eclipse, right-click in the **Package Explorer** window and select **Refresh**. This will allow Eclipse to update the Package Explorer to the current state of the `workspace` folder. Eclipse should show your `jsoup-1.7.2.jar` file (this file may have a different name depending on the version you're using) in the Package Explorer window.
8. Right-click on the jsoup JAR file and select **Build Path** | **Add to Build Path**.
9. In your `WikiScraper` class file, write the following code:

```
package com.packtpub.JavaScraping.SimpleScraper;

import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import java.net.*;
import java.io.*;
```

```
public class WikiScraper {
  public static void main(String[] args) {
    scrapeTopic("/wiki/Python");
  }

  public static void scrapeTopic(String url){
    String html = getUrl("http://www.wikipedia.org/"+url);
    Document doc = Jsoup.parse(html);
    String contentText = doc.select("#mw-content-text >
      p").first().text();
    System.out.println(contentText);
  }

  public static String getUrl(String url){
    URL urlObj = null;
    try{
      urlObj = new URL(url);
    }
    catch(MalformedURLException e){
      System.out.println("The url was malformed!");
      return "";
    }
    URLConnection urlCon = null;
    BufferedReader in = null;
    String outputText = "";
    try{
      urlCon = urlObj.openConnection();
      in = new BufferedReader(new
        InputStreamReader(urlCon.getInputStream()));
      String line = "";
      while((line = in.readLine()) != null){
        outputText += line;
      }
      in.close();
    }catch(IOException e){
      System.out.println("There was an error connecting to
        the URL");
      return "";
    }
    return outputText;
  }
}
```

Assuming you're connected to the internet, this should compile and run with no errors, and print the first paragraph of text from the article.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at `http://www.packtpub.com`. If you purchased this book elsewhere, you can visit `http://www.packtpub.com/support` and register to have the files e-mailed directly to you.

How it works...

Unlike our `HelloWorld` example, there are a number of libraries needed to make this code work. We can incorporate all of these using the `import` statements before the class declaration. There are a number of jsoup objects needed, along with two Java libraries, `java.io` and `java.net`, which are needed for creating the connection and retrieving information from the Web.

As always, our program starts out in the main method of the class. This method calls the `scrapeTopic` method, which will eventually print the data that we are looking for (the first paragraph of text in the Wikipedia article) to the screen. `scrapeTopic` requires another method, `getURL`, in order to do this.

`getUrl` is a function that we will be using throughout the book. It takes in an arbitrary URL and returns the raw source code as a string. Understanding the details of this method isn't important right now—we'll dive into more depth about this in other sections of the book. Essentially, it creates a Java URL object from the URL string, and calls the `openConnection` method on that URL object. The `openConnection` method returns a `URLConnection` object, which can be used to create a `BufferedReader` object.

`BufferedReader` objects are designed to read from, potentially very long, streams of text, stopping at a certain size limit, or, very commonly, reading streams one line at a time. Depending on the potential size of the pages you might be reading (or if you're reading from an unknown source), it might be important to set a buffer size here. To simplify this exercise, however, we will continue to read as long as Java is able to.

The `while` loop here retrieves the text from the `BufferedReader` object one line at a time and adds it to `outputText`, which is then returned.

After the `getURL` method has returned the HTML string to `scrapeTopic`, `jsoup` is used. `jsoup` is a Java library that turns HTML strings (such as the string returned by our scraper) into more accessible objects. There are many ways to access and manipulate these objects, but the function you'll likely find yourself using most often is the `select` function. The jsoup `select` function returns a jsoup object (or an array of objects, if more than one element matches the argument to the `select` function), which can be further manipulated, or turned into text, and printed.

The crux of our script can be found in this line:

```
String contentText = doc.select("#mw-content-text >
  p").first().text();
```

This finds all the elements that match `#mw-content-text > p` (that is, all `p` elements that are the children of the elements with the CSS ID `mw-content-text`), selects the first element of this set, and turns the resulting object into plain text (stripping out all tags, such as `<a>` tags or other formatting that might be in the text).

The program ends by printing this line out to the console.

There's more...

Jsoup is a powerful library that we will be working with in many applications throughout this book. For uses that are not covered in the book, I encourage you to read the complete documentation at `http://jsoup.org/apidocs/`.

What if you find yourself working on a project where jsoup's capabilities aren't quite meeting your needs? There are literally dozens of Java-based HTML parsers on the market.

Writing more complicated scraper (Intermediate)

This recipe isn't a huge technical leap over the last one, but it will introduce you to automated scraping (hitting the go button and letting your scraper traverse the Web by itself) and scraping multiple pages.

Getting ready

Before beginning this recipe, you'll need a copy of the project we created in the previous recipe.

How to do it...

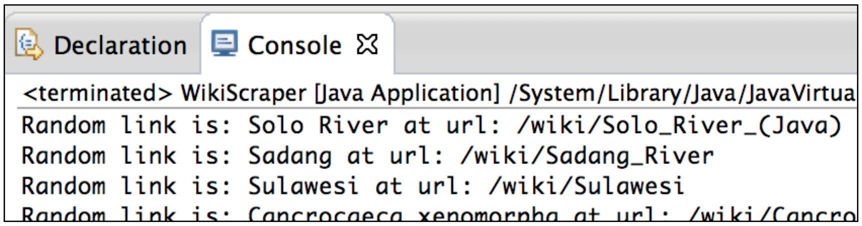
1. Although our code is still fairly simple, it's a good idea to think about organizing it into packages within our project. Java packages are good for bundling up sets of functionalities within a project, and reducing `namespace` problems when you have lots of code lying around.
2. To create a new package within the `Scraper` project, go to **File** | **New** | **Package** in the Eclipse main menu. Let's call it `com.packtpub.JavaScraping.ComplicatedScraper`.

3. Move your `WikiScraper.java` file to the newly created class. You'll see that Eclipse has helpfully added the `Wikipedia` package at the top of it. This tells the compiler that it is a part of the `Wikipedia` package.
4. The only modifications made to the code from the previous recipe are the inclusion of a new library, `java.util.Random`, the addition of a new global variable, Random generator, and this additional code in the `scrapeTopic` method:

```
package com.packtpub.JavaScraping.ComplicatedScraper;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.select.Elements;
import java.net.*;
import java.io.*;
import java.util.Random;

public class WikiScraper {
  private static Random generator;
  public static void main(String[] args) {
    generator = new Random(31415926);
    scrapeTopic("/wiki/Java");
  }
  public static void scrapeTopic(String url){
    String html = getUrl("http://www.wikipedia.org/"+url);
    Document doc = Jsoup.parse(html);
    Elements links = doc.select("#mw-content-text
      [href~=^/wiki/((?!:).)*$]");

    if(links.size() == 0){
      System.out.println("No links found at "+url+". Going
        back to Java!");
      scrapeTopic("wiki/Java");
    }
    int r = generator.nextInt(links.size());
    System.out.println("Random link is:
      "+links.get(r).text()+" at url:
      "+links.get(r).attr("href"));
    scrapeTopic(links.get(r).attr("href"));
  }

  public static String getUrl(String url){
    URL urlObj = null;
    try{
      urlObj = new URL(url);
    }catch(MalformedURLException e){
      System.out.println("The url was malformed!");
      return "";
    }
    URLConnection urlCon = null;
    BufferedReader in = null;
    String outputText = "";
    try{
      urlCon = urlObj.openConnection();
      in = new BufferedReader(new
        InputStreamReader(urlCon.getInputStream()));
      String line = "";
      while((line = in.readLine()) != null){
        outputText += line;
      }
      in.close();
    }catch(IOException e){
      System.out.println("There was an error connecting to
        the URL");
      return "";
    }
    return outputText;
  }
}
```

Assuming you are connected to the Internet, this code should run indefinitely, traversing random links through Wikipedia and printing them to the console as it goes. If it encounters a problem (no links found on the page, whether that is because there are no links or because the code encountered a format it wasn't expecting and couldn't find them), it will go back to the original Java article and continue on a different path.

You should see something like this:



```
<terminated> WikiScraper [Java Application] /System/Library/Java/JavaVirtua
Random link is: Solo River at url: /wiki/Solo_River_(Java)
Random link is: Sadang at url: /wiki/Sadang_River
Random link is: Sulawesi at url: /wiki/Sulawesi
Random link is: Cancrocaeca xenomorpha at url: /wiki/Cancro
```

]

How it works...

By declaring a random number-generator object inside the class declaration and before the main method, and then defining that number-generator object in the main method, we can create a global object that is able to be used across all the methods in the object (although, because it has been declared to be a `private` variable, and not a `public` one, the object cannot be accessed from outside of the class). This is a common concept in object-oriented programming, and not unique to Java.

As before, `scrapeTopic` gets the HTML string for the URL passed to it, and creates a `jsoup` object from that string.

It is worth taking a closer look at this line:

```
Elements links = doc.select("#mw-content-text
  [href~=^/wiki/((?!:).)*$]");
```

This creates a jsoup `Elements` object that contains a collection of jsoup `Element` (note the singular) objects. As before, we will use the `select()` function to retrieve the elements that we want, but the argument is different:

```
"#mw-content-text [href~=^/wiki/((?!:).)*$]"
```

As in the first example, we are selecting the element with the CSS ID `mw-content-text` (the main body of the Wikipedia article), but rather than selecting the first paragraph (as in the previous example), we are looking for all article links inside the main body.

The jsoup `select` expression `[attr ~= expression]` means select all the items containing the attribute values for `attr` that match the given **regular expression**. (jsoup is powerful stuff.)

In this case, we are selecting all the elements (presumably links) where the `href` attribute matches the regular expression `^/wiki/((?!:).)*$`.

Regular expressions are outside the scope of this book, but in this case, it matches all the links that begin with `/wiki/` (all article pages begin with this) and excludes any links that contain a colon (`:`) following that. Wikipedia links containing a colon are mostly special pages, such as user pages, or Wikipedia guideline pages, and not the article pages that we're looking for.

There's more...

For more information about jsoup selectors and the power and flexibility they offer, see the selector documentation at the following link:

`http://jsoup.org/apidocs/org/jsoup/select/Selector.html`

One of the best quick reference guides to regular expressions I've seen can be found at `http://web.mit.edu/hackl/www/lab/turkshop/slides/regex-cheatsheet.pdf`, and a more in-depth explanation of what they are and how they work can be found in many places on the Web.

Handling errors (Simple)

Errors are something that you will encounter often in web scraping. The web is messy, and you can never be certain whether an element exists, or if a page returns the data you want, or even that a site's server is up and running. Being able to catch, throw, and handle these exceptions (errors) is essential to web scraping.

Getting ready

Take a look at the `getUrl` method in the previous scrapers. You'll notice the following lines:

```
try{
...
}catch(IOException e){
    System.out.println("There was an error connecting to the URL");
    return "";
}
```

This code snippet indicates that the statement in the `try` brackets is capable of throwing an `IOException` if things go wrong, and defines how to gracefully handle this exception if it occurs, inside the `catch` brackets.

In this recipe, we will look at, not just how to handle exceptions, but also how to create and throw our own exceptions, in more detail.

How to do it...

Although there are many built-in general exceptions Java has that can be thrown, it's best to be specific when it's possible to anticipate a problem. For this reason, we will create our own `error` classes for our software to throw.

1. Create a new package in the `Scraper` project called `com.packtpub.JavaScraping.HandlingErrors`.
2. Create the class `LinkNotFoundException` inside the new package. The code will look like this:

```
package com.packtpub.JavaScraping.HandlingErrors;
public class LinkNotFoundException extends Exception{
   public LinkNotFoundException(String msg) {
   super(msg);
   }
}
```

Modify `WikiScraper.java` to match the following (where methods are the same as they appear in previous sections, the body of the method is replaced by "..."):

```
package com.packtpub.JavaScraping.HandlingErrors;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.select.Elements;
import java.net.*;
import java.io.*;
import java.util.Random;

public class WikiScraper {
  private static Random generator;
  public static void main(String[] args) {
    generator = new Random(31415926);
    for(int i = 0; i < 10; i++){
      try{
        scrapeTopic("/wiki/Java");
      }
      catch(LinkNotFoundException e){
        System.out.println("Trying again!");
      }
    }
  }

  public static void scrapeTopic(String url) throws
    LinkNotFoundException{
    String html = getUrl("http://www.wikipedia.org/"+url);
    Document doc = Jsoup.parse(html);
    Elements links = doc.select("#mw-content-text
      [href~=^/wiki/((?!:).)*$]");
    if(links.size() == 0){
      throw new LinkNotFoundException("No links on page, or
        page malformed");
    }
    int r = generator.nextInt(links.size());
    System.out.println("Random link is:
      "+links.get(r).text()+" at url:
      "+links.get(r).attr("href"));
    scrapeTopic(links.get(r).attr("href"));
  }

  public static String getUrl(String url){
    ...
  }
}
```

How it works...

This is the first example we've seen of extending classes in Java. If you've done object-oriented programming before, you're likely to be familiar with the concept of extending classes. When we extend a class, the extended class contains all the attributes and methods of the parent class. The classic zoological example of class extension goes something like this:

```
Public class Animal{...}
Public class FourLeggedMammal extends Animal(){...}
Public class Dog extends FourLeggedMammal{...}
```

In this example, the `Dog` class contains all the attributes/methods/behaviors of `FourLeggedMammal`, while perhaps adding some additional methods of their own, specific to `Dog` (for example, `public String void bark()`).

If you were to take a look at the Java core, you'd see many exception classes similar to the one we've just created. All exceptions extend the Java Exception class, which provides methods for getting the stack trace for the error, printing error messages, and other useful things.

> Be aware that throwing an error requires a large amount of overhead, and should not be used in place of a normal return value. Errors should indicate that something has gone seriously wrong, and things need to be handled or reset appropriately.

There's more...

Oracle has an excellent Java tutorial series at `http://docs.oracle.com/javase/tutorial/index.html` that provides not just an introductory series of lessons to Java development but also resources on many advanced topics.

Although the throwing, catching, and handling of exceptions might seem like a fairly straightforward exercise as it was presented in this recipe, exception handling is still far from a simple subject. Oracle's lesson on exception handling, which can be found at `http://docs.oracle.com/javase/tutorial/essential/exceptions/`, is a good starting point for learning more advanced techniques.

Writing robust, scalable code (Advanced)

You didn't start learning Java so that you could write one-page scripts, and our ability to crawl the Web is going to be limited very quickly if that's all we're using it for. In this recipe, we will write a multi class scraper that scrapes across multiple websites.

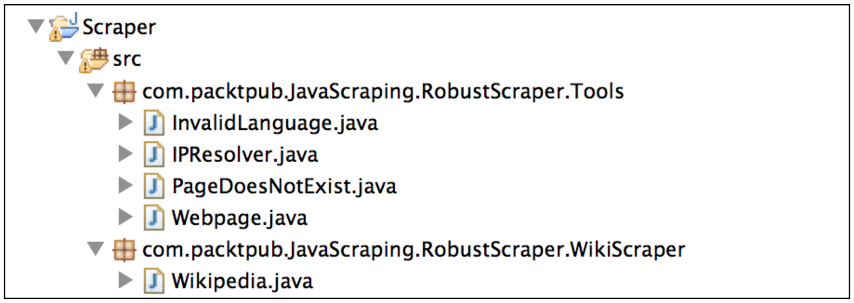
Getting ready

We'll build on our work in the earlier recipe *Writing a simple scraper (Simple)*, but will create two new packages that are able to import, create instances of, and communicate between each other.

How to do it...

1. Create a new package within the `Scraper` project named `com.packtpub.JavaScraping.RobustScraper.WikiScraper`, and create an additional project called `com.packtpub.JavaScraping.RobustScraper.Tools`.
2. Create a class in the `WikiScraper` package called `Wikipedia.java`.
3. Create four classes in `Tools` called `InvalidLanguage.java`, `IPResolver.java`, `PageDoesNotExist.java`, and `Webpage.java`.

   Your file structure in Eclipse's Package Explorer should look like this:



```
▼ Scraper
  ▼ src
    ▼ com.packtpub.JavaScraping.RobustScraper.Tools
      ▶ InvalidLanguage.java
      ▶ IPResolver.java
      ▶ PageDoesNotExist.java
      ▶ Webpage.java
    ▼ com.packtpub.JavaScraping.RobustScraper.WikiScraper
      ▶ Wikipedia.java
```

4. The code for each of the classes in the `Tools` package is as follows:

   The code for the `Wikipedia` class:

```
package com.packtpub.JavaScraping.RobustScraper.WikiScraper;
import java.io.IOException;
import java.net.MalformedURLException;
import org.jsoup.nodes.Document;
import org.jsoup.nodes.Element;
import org.jsoup.select.Elements;

import com.packtpub.JavaScraping.RobustScraper.Tools.IPResolver;
import com.packtpub.JavaScraping.RobustScraper.Tools.Webpage;

public class Wikipedia {

  /**
   * Crawls both English and German Wikipedia. Finds 10 random
German Wikipedias by following redirects
   * through the "special:random" page, and returns the country of
origin for the first IP address
   * (if any) that it finds on the page. After this, does the same
for English Wikipedia edits.
   * @param args
   */
  //Crawls both English and German Wikipedias. Returns physical
address, and datetime edits were made
  public static void main(String[] args) {
    for(int i = 0; i < 10; i++){
      String randomPage = getRandomHist("de");
      System.out.println("This page has been edited in
        "+getEditCountry(randomPage));
    }
    for(int j = 0; j < 10; j++){
      String randomPage = getRandomHist("en");
      System.out.println("This page has been edited in
        "+getEditCountry(randomPage));
    }
  }

  private static String getRandomHist(String language){
    Webpage randomPage = new
      Webpage("http://"+language+".wikipedia.org/wiki/
      Special:Ran      dom");
    String randomUrl = "";
    try {
        if(language == "en"){
          randomUrl = randomPage.getRedirect();
        }else{
          randomUrl = randomPage.getRedirect(true);
        }
      } catch (MalformedURLException e) {
        e.printStackTrace();
      } catch (IOException e) {
        e.printStackTrace();
      }

    if(randomUrl == ""){
      return "";
    }
    String randHist =
      randomUrl.replace("http://"+language+
      ".wikipedia.org/wiki/", "");
    randHist = "http://en.wikipedia.org/w/index.php?title="
      +randHist+"&action=history";
    return randHist;
  }

  private static String getEditCountry(String histPageUrl){
    Webpage histPage = new Webpage(histPageUrl);
    Document histDoc = histPage.getJsoup();
    Elements rows = histDoc.select("li");
    for(Element row : rows){
      System.out.println(row.select(".mw-
        userlink").text());
      if(row.select(".mw-userlink").text().matches("^(?:[0-
        9]{1,3}.){3}[0-9]{1,3}$")){
        //"Username" is an IP Address
        IPResolver IPAddress = new
          IPResolver(row.select(".mw-userlink").text());
        return IPAddress.getCountry();
      }
    }
    return "No IP Addresses Found in Edit History";
  }
}
```

Here is the code for our new Exception class, InvalidLanguage:

```
/* InvalidLanguage.java is the exception method that is thrown
when the language given to the Wikipedia class does not exist:
*/
package com.packtpub.JavaScraping.RobustScraper.Tools;
/**
 * Is thrown if a two letter language code (e.g. "en," "fr," "de")
does not actually exist
 */
public class InvalidLanguage extends Exception{

  private static final long serialVersionUID = 1L;

  public InvalidLanguage(String msg)
  {
    System.out.println(msg);
    }
}
```

The code for `IPResolver`:

```
package com.packtpub.JavaScraping.RobustScraper.Tools;

/**
 * IPResolver uses the free service provided by freegeoip.net to
resolve IP addresses into a geographic location
 * @author rmitchell
 *
 */
public class IPResolver {

  /**
   * @param ipAddress the IPv4 address to be resolved to location
   * @param ipInfo plaintext location or other information that is
returned
   */
  String ipAddress;
  String ipInfo;
  public IPResolver(String declaredAddress){
    ipAddress = declaredAddress;
    Webpage ipInfoObj = new
      Webpage("http://freegeoip.net/xml/"+ipAddress);
    ipInfo = ipInfoObj.getString();
  }
  /**
   * Simple method, returns anything between the \<CountryName\>
tags (presumably the country of origin)
   * @return String containing the name of the ipAddress country
of origin
   */
  public String getCountry(){
    int startCtry = ipInfo.indexOf("<CountryName>")+13;
    int endCtry = ipInfo.indexOf("</CountryName>");
    return ipInfo.substring(startCtry, endCtry);
  }
}
```

The code for `PageDoesNotExist`:

```
package com.packtpub.JavaScraping.RobustScraper.Tools;

/**
 * Error is thrown if the webpage does not exist
 */
public class PageDoesNotExist extends Exception{

  public PageDoesNotExist(String msg) {
      System.out.println(msg);
  }
}
```

5. The `Webpage` class is the longest piece of code that we will encounter in this book, although much of this code is taken from the recipe named *Writing a simple scraper (Simple)*. Where methods are self-explanatory, or have been used in the past, we replace the body of the code with "..." The full working code can be downloaded from the Packt Publishing website.

```
package com.packtpub.JavaScraping.RobustScraper.Tools;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import java.net.*;
import java.io.*;

public class Webpage {

  /**
   * @param url the url of the target page
   * @param urlObj Java url object used by several methods,
    global variable here for convenience
   */
```

```
public String url;
public URL urlObj;

/**
 * Sets urlObj, throws error if the URL is invalid
 * @param declaredUrl
 */
public Webpage(String declaredUrl){
  url = declaredUrl;
  urlObj = null;
  try{
    urlObj = new URL(url);
  }catch(MalformedURLException e){
    System.out.println("The url was malformed!");
  }
}

/**
 * Used to overload the getRedirect(Boolean
   secondRedirect) method if no Boolean is provided.
 * By default, this will perform only one redirect and
   return the result.
 * @return Returns final redirected URL, or an empty
   string if none was found.
 * @throws MalformedURLException
 * @throws IOException
 */
public String getRedirect() throws MalformedURLException,
  IOException{
  try{
    return getRedirect(false);
  }catch(MalformedURLException e){
    System.out.println("The url was malformed!");
  }catch(IOException e){
    System.out.println("There was an error connecting to
      the website");
  }
  return "";
}

/**
 * Used to get the URL that a page redirects to, or,
   optionally, the second redirect that a page goes to.
 * @param secondRedirect "True" if the redirect should be
   followed twice
 * @return URL that the page redirects to
 * @throws MalformedURLException
 * @throws IOException
 */
public String getRedirect(Boolean secondRedirect) throws
  MalformedURLException, IOException{
  HttpURLConnection URLCon =
    (HttpURLConnection)urlObj.openConnection();
  URLCon.setInstanceFollowRedirects(false);
  URLCon.connect();
  String header = URLCon.getHeaderField( "Location" );
  System.out.println("First header is:
    "+URLCon.getHeaderField(7));
  System.out.println("Redirect is: "+header);

  //If a second redirect is required
  if(secondRedirect){
    try{
      URL newURL = new URL(header);
      HttpURLConnection newCon =
        (HttpURLConnection)newURL.openConnection();
      newCon.setInstanceFollowRedirects(false);
      newCon.connect();
      String finalHeader = newCon.getHeaderField(
        "Location" );
      return finalHeader;
    }
    catch(MalformedURLException e){
      System.out.println("The url was malformed!");
    }
    return "";
  }
  else{
    return header;
  }
}

/**
 * Gets the text version of the webpage
 * @return
 */
public String getString(){
  ...
}
```

```
    /**
     * @return Jsoup of urlObj page
     */
    public Document getJsoup(){
      ...
    }

  }
```

How it works...

You may notice the different comment types that are used throughout the code. A comment type you may be less familiar with is the Javadoc comment:

```
/**
 * Used to get the URL that a page redirects to, or, optionally,
   the second redirect that a page goes to.
 * @param secondRedirect "True" if the redirect should be
   followed twice
 * @return URL that the page redirects to
 * @throws MalformedURLException
 * @throws IOException
 */
```

These type of comments are placed before methods and class definitions. Javadoc provides a convenient way to create documentation, and documents your code at once. You can see the documentation that is automatically created if you go to **Project** | **Generate Javadoc** in Eclipse.

In order to save space, and because we will be explaining the code later, we will not be including Javadoc comments within the code printed in this book. However, the complete code along with `.javadoc` files can be downloaded at the Packt Publishing website.

Sometimes the easy way to code something is not always the best way. Classes should always be created in the most reusable and flexible way possible. We could have written a more specific method that goes directly to the following German "random" Wikipedia page, `http://de.wikipedia.org/wiki/Spezial:Zuf%C3%A4llige_Seite`. However, this would mean that we would have to hardcode all the random article URLs for each language, or know it off the top of our heads when we are writing code that uses the **scraping tools**. With the scraping tools as they are written, Wikipedia lets us know what the correct URL is (by redirecting from `http://de.wikipedia.org/wiki/Special:Random`, to `http://de.wikipedia.org/wiki/Spezial:Zuf%C3%A4llige_Seite`), and we are free to enter any language code we want (which means that we could use `http://ja.wikipedia.org/wiki/Special:Random` for a random Japanese article).

Breaking out tools into separate packages allows us to reuse them for other purposes. We can create multiple scraper packages that scrape many different websites using the same package of tools without sacrificing functionality.

There's more...

What are the other ways in which you could expand this code? An obvious way of expanding this code is to make the `IPResolver` object more versatile, allowing you to look up cities, latitudes, and longitudes, and other information.

Because we did not require extensive formatted information, it would have been needless overhead at this point to import a JSON library, or an XML parsing library in order to interpret the results from the `IPResolver`. However, it is worth having one or two of these libraries in hand.

You can get a JSON parsing library from the following link:

`https://github.com/douglascrockford/JSON-java`

Just like what we did with the jsoup library, place the `.jar` file in your source folder and use Eclipse to add it to the build path.

Persisting data (Advanced)

While watching data stream down a console is fun, it is nearly useless unless you're storing the data for useful applications. Being able to store, retrieve, manipulate, and display data is an inescapable component of web scraping. In this section, we will show you how to store data in a MySQL database.

Getting ready

Although PostgreSQL, MongoDB, and a multitude of other database types are increasingly used, MySQL is still ubiquitous in the industry, and, partly because of its ownership by Oracle, it has fantastic support in Java. We will be using MySQL throughout this book, but feel free to substitute your own database syntax wherever appropriate.

Java's JDBC (Java Database Connectivity) API provides a way for Java to connect to, query, and update a database.

For this section, you will need to have access to a server with MySQL installed, or download software such as XAMPP or MAMP to run a MySQL database from your local machine. Although setting up a MySQL server is outside the scope of this book, you can find the downloadable file of MAMP for Mac OS X at `http://www.mamp.info/en/index.html`, or XAMPP for Windows at `http://download.cnet.com/XAMPP/3000-10248_4-10703782.html`, and follow the instructions provided for the installation and setup.

We will be modifying the code in the earlier section named *Writing more complicated scraper (Intermediate)*. So it is recommended that you skim that section and be comfortable with the code contained in it.

Get the MySQL Connector JAR file from the following link:

`http://dev.mysql.com/downloads/connector/j/`

There is a fairly annoying Oracle signup process involved to get the `.jar` file downloaded. However, downloading from Oracle ensures that the JAR file is up-to-date and secure. Change the extension of the `.jar` file to `.zip` and extract it, and add the binary JAR file (the file looks something like `mysql-connector-java-5.1.25-bin.jar`) to your build path in Eclipse. These are the drivers for JDBC and essential for connecting to a database in Java.

How to do it...

1. Make sure to add the JDBC drivers for MySQL to your library, and add them to the build path for your project.
2. Create a new package called `com.packtpub.JavaScraping.PersistingData`, and copy the `Wikipedia` class from the package created in the previous recipe, `com.packtpub.JavaScraping.ComplicatedScraper`.
3. In the `Wikipedia` class, add the following line below your list of imports:

```
import java.sql.*;
```

4. Add the line to declare a global `Connection` object `dbConn`, immediately before the similar line that declares the global `Random` generator:

```
public class WikiScraper {
  private static Random generator;
  private static Connection dbConn;
```

5. Replace the main method with the following code, where `<username>`, `<password>`, `<host>`, and `<database>` should be replaced with your MySQL username, password, the host your database is running on, and the name of your database, respectively. If you are running MySQL on a port other than 3306, change that as well:

```
public static void main(String[] args) {
  generator = new Random(31415926);
  String dbURL = "jdbc:mysql://<host>:3306/<database>";
  Properties connectionProps = new Properties();
  connectionProps.put("user", "<username>");
  connectionProps.put("password", "<password>");
  dbConn = null;
  try {
    dbConn = DriverManager.getConnection(dbURL,
      connectionProps);
  }
  catch (SQLException e) {
    System.out.println("There was a problem connecting to
      the database");
    e.printStackTrace();
  }

  PreparedStatement useStmt;
  try {
    useStmt = dbConn.prepareStatement("USE java");
    useStmt.executeUpdate();
  } catch (SQLException e) {
    e.printStackTrace();
  }

  scrapeTopic("/wiki/Java");
}
```

6. Create a new method, `writeToDB`:

```
    private static void writeToDB(String title, String url){
      PreparedStatement useStmt;
      try {
        useStmt = dbConn.prepareStatement("INSERT INTO
          wikipedia (title, url) VALUES (?,?)");
        useStmt.setString(1, title);
        useStmt.setString(2, url);
        useStmt.executeUpdate();
      } catch (SQLException e) {
        e.printStackTrace();
      }
    }
```

7. Call `writeToDB` in your `scrapeTopic` method, immediately before `scrapeTopic` calls itself recursively:

```
    System.out.println("Random link is: "+links.get(r).text()+"
      at url: "+links.get(r).attr("href"));
    writeToDB(links.get(r).text(), links.get(r).attr("href"));
    scrapeTopic(links.get(r).attr("href"));
```

How it works...

The WikiScraper class defines a global database connection so that the connection can be set in the main method and called repeatedly in the writeToDB method. If we didn't do this, a new connection would have to be created and set up every time we wanted to write to the database.

The getConnection() method takes in a JDBC URL as well as a username and a password (stored as a Properties object) in order to connect to the database defined by the URL. The format for JDBC URLs are:

```
jdbc:mysql://<host>:<port>/<database>
```

You never know what you might be picking up from the Web. Always, always, always sanitize your inputs. Java has a nifty way of doing this using the prepareStatement method—use a ? character where your unsanitary variable should be inserted, and add variables using the setString method. Indexes for added variables start at 1.

Writing tests (Intermediate)

As a programmer, you may be familiar with **unit testing**. Unit testing provides a set of written requirements, or challenges, that programs must satisfy in order to pass. A unit test might be something like, "Connect to a server and fetch a web page. Check to make sure that the page has an HTML title. If it does not, the test fails."

In this recipe, we will be writing simple unit tests, and building off of that foundation for the next several recipes, using a library called HtmlUnit. Even if you do not plan on using unit tests, it might be a good idea to at least skim this recipe to prepare yourself.

Getting ready

A good unit test tests only one thing at a time, and builds on the groundwork laid by other tests. If one test is passed indicating that a failure could have stemmed from any one of a dozen causes, it's just not a very well-written test.

Although testing may seem needlessly rigorous, it is extremely valuable for web scraping for several reasons:

- It allows you to check on websites themselves to make sure that they have not changed, and that your scrapers will still be functional against them

- It allows you to check on the scraper code itself, to make sure that it is still working, and that any additional scrapers or scraping tools you write will play nicely with the others
- Some useful scraping tools, such as HtmlUnit (which we will cover in the next recipe), require that the code is written in the form of a unit test

Java has a third-party unit-testing library, `JUnit`, which is almost ubiquitous in the industry. The libraries should already be included at the time of Java installation, although you can download the latest version, or read more about it at the following link:

`https://github.com/junit-team/junit/wiki/Download-and-Install`

How to do it...

1. Create a new package called `tests` in your `Scraper` project.
2. Create a `JUnit` class in this project by right-clicking on the package in Eclipse's Package Explorer and selecting **New | JUnit Test Case**.
3. Call this new class `ScraperTest`, and write the following code in it:

```
package com.packtpub.JavaScraping.Test;

import junit.framework.TestCase;
import org.junit.Test;

public class ScraperTest extends TestCase{
  public static void main(String args[]) {
    org.junit.runner.JUnitCore.main("tests.ScraperTest");
  }

  @Test
  public void test() {
    WikiScraper scraper = new WikiScraper();
    String scrapedText = WikiScraper.scrapeTopic();
    String openingText = "A python is a constricting snake
      belonging to the Python (genus), or, more generally,
      any snake in the family Pythonidae (containing the
      Python genus).";
    assertEquals("Nothing has changed!", scrapedText,
      openingText);
  }
}
```

The `WikiScraper` class contains:

```
package com.packtpub.JavaScraping.Test;

import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import java.net.*;
import java.io.*;

public class WikiScraper {


  public WikiScraper(){
    System.out.println("Initialized");
  }

  public static String scrapeTopic(){
    String url = "/wiki/Python";
    String html = getUrl("http://www.wikipedia.org/"+url);
    Document doc = Jsoup.parse(html);
    String contentText = doc.select("#mw-content-text >
      p").first().text();
    return contentText;
  }

  public static String getUrl(String url){
    ...
  }
}
```

How it works...

Any word or line preceded by an `@` sign is known as an **annotation** in Java. Annotations have no real meaning to the Java compiler by themselves, but they affect how third-party libraries (such as the `JUnit` testing framework) might interpret the code that follows them.

The `@Before`, `@Test`, and `@After` annotations tell `JUnit` that the methods that follow have special purposes, and should be interpreted accordingly. `@Before` methods provide the setup for any tests. If you have variables that need to be initialized, connections that need to be started, or data that needs to be populated, now is the time to do it.

The `@Test` annotation must contain an assertion specific to `JUnit` at some point, and cannot give a return value.

The `@After` annotation does any teardown that is needed after your tests. If you need to close connections, print some `All done` message, or do any other closing actions, do it here.

There's more...

You can find a full list of JUnit assert methods in the JUnit documentation at http://junit.sourceforge.net/javadoc/org/junit/Assert.html.

Pay close attention to the argument types that the assert methods require. Some of the methods are overridden (two methods with the same name, but with different argument types, such as assertNull and assertNull(<object>)—Java will either call one method or the other depending on which arguments you give it).

Although it can often be overwhelming to dive into the formal documentation for a new framework (as opposed to searching Google for choice of keywords and hoping you're lucky), JUnit has some of the better documentation complete with examples and plain text explanations for each feature. Pay special attention to the list of annotations, which may come in handy for ordering the execution of your JUnit methods. Check it out in the following link:

http://junit.sourceforge.net/javadoc/org/junit/package-summary.html

Going undercover (Intermediate)

Not every website is as welcoming to scraping as Wikipedia. There are many sites that will check to see if you're actually a web browser (or if you say that you are, at least) before sending you the site data. In this recipe, we will learn how to subvert this check (while making sure to comply with the Terms of Service) in order to get the desired data from a website.

Getting ready

Web servers can check which browser you are using by checking the HTTP header information you are sending with every request you make for a web page.

HTTP header information looks like this:

```
Host: www.google.com
Connection: keep-alive
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_2)
AppleWebKit/537.31 (KHTML, like Gecko) Chrome/26.0.1410.65
Safari/537.31
Accept: */*
Referer: http://www.google.com
Accept-Encoding: gzip,deflate,sdch
Accept-Language: en-US,en;q=0.8
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.3
```

Although many items in the header can reveal your true identity as a web scraper (not setting the language encoding, for example, may tip off some of the better-protected sites), the `User-Agent` is the one that is, for obvious reasons, most likely to give you away. Fortunately, we can control the HTTP header information that we send to a site, essentially disguising our program as a more innocuous visit from a web browser.

In this section, we will collect data from `http://www.freeproxylists.net/` (many other "free-proxy IP sites" will work, and have similar securities if this one is broken or missing in the future). Notice that they do not have a Terms of Service and even their `robots.txt` file does not prohibit us from scraping their site. While their legal security may be lacking, their actual security is certainly present. If we do not modify our web scraper's headers when trying to scrape the site, the site presents us with a CAPTCHA challenge, which our script, of course, cannot pass. If it does not suspect us, it will present the actual site.

How to do it...

Although Java's built-in methods to handle URL connections are good, HtmlUnit provides greater flexibility and power that we need in order to modify our HTTP headers. Because HtmlUnit was designed as a framework for website testing, based on `JUnit`, you will need to have `JUnit` installed and may want to read over the previous recipe. You will also need to download HtmlUnit and jsoup from the following link (used in previous recipes) and add the JAR files to your build path:

`http://sourceforge.net/projects/HtmlUnit/files/HtmlUnit/`

1. Create a new package, `com.packtpub.JavaScraping.Undercover`.
2. Create a class, `FreeProxyLists`, with the following code:

```
package com.packtpub.JavaScraping.Undercover;

import java.io.IOException;
import java.net.MalformedURLException;

import junit.framework.TestCase;

import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.junit.Test;

import com.gargoylesoftware.HtmlUnit.BrowserVersion;
import com.gargoylesoftware.HtmlUnit.
  FailingHttpStatusCodeException;
import com.gargoylesoftware.HtmlUnit.WebClient;
import com.gargoylesoftware.HtmlUnit.html.HtmlPage;
```

```
public class FreeProxyLists extends TestCase{
  public static void main(String args[]) {
    org.junit.runner.JUnitCore.main("tests.
      FreeProxyLists");
  }

  @Test
  public void test() throws FailingHttpStatusCodeException,
    MalformedURLException, IOException {
      WebClient webClient = new
        WebClient(BrowserVersion.FIREFOX_10);
      webClient.setThrowExceptionOnScriptError(false);
      HtmlPage page =
      webClient.getPage("http://www.freeproxylists.net/");
      Document jPage = Jsoup.parse(page.asXml());
      String ipAddress =
        jPage.getElementsByClass("DataGrid").get(0).
        getElementsByTag("tbody").get(0).
        getElementsByClass("odd").get(0).getElementsByTag
        ("td").get(0).getElementsByTag("a").get(0).text();
      System.out.println(ipAddress);
      webClient.closeAllWindows();
      assertTrue(ipAddress.matches("(?:[0-9]{1,3}.){3}[0-
        9]{1,3}"));
  }
}
```

How it works...

As you can see, forging your browser's header becomes as trivial as passing an argument to `WebClient` in HtmlUnit:

```
WebClient webClient = new WebClient(BrowserVersion.FIREFOX_10);
```

> The `WebClient` function defaults to Internet Explorer 8 if no browser is specified. This can be changed by using `BrowserVersion.setDefault()`.

While HtmlUnit is a fantastic tool for getting data, it does lack some flexibility in processing the data. If all you need is the content of a web page with a particular CSS ID, you're fine using something like this:

```
String content = page.getElementById("particularID").asText();
```

However, HtmlUnit does lack some tools, most notably a function to get elements by class name. For this reason, we convert the `HtmlUnit DomElement` to a jsoup document in order to process it. While this is very convenient, be careful when combining these two libraries—there are some namespace issues to watch out for (for example, `com.gargoylesoftware.HtmlUnit.javascript.host.Element` and `org.jsoup.nodes.Element`), but careful use of the `import` statements and your IDE's warnings go a long way to solve this.

In order to get the first IP address in the list, from the jsoup document, we use the intimidating-looking:

```
jPage.getElementsByClass("DataGrid").get(0).getElementsByTag("tbod
y").get(0).getElementsByClass("odd").get(0).getElementsByTag("td")
.get(0).getElementsByTag("a").get(0).text();
```

It's common to write lines like this when extracting data from pages that don't want to be scraped. When scraping pages that lack sufficient class and ID attributes to pinpoint elements more directly, you shouldn't be afraid of getting dirty—write inelegant code to deal with inelegant code.

In the last line of the test method, we encounter regular expressions again, to make sure that the string we've retrieved from the page really is a valid IP address:

```
assertTrue(ipAddress.matches("(?:[0-9]{1,3}.){3}[0-9]{1,3}"));
```

There's more...

Occasionally, you may need to get more specific in your testing or scraping than just modifying the type of browser (that is, changing the language preferences to scrape a website's French version). HtmlUnit provides an easy way to do that through setting browser property values, described here:

`http://HtmlUnit.sourceforge.net/apidocs/com/gargoylesoftware/HtmlUnit/BrowserVersion.html`

Submitting a basic form (Advanced)

A web scraper wouldn't be much good if it got stymied the first time it encountered a login form. In this recipe, we will learn how to submit forms to get information on the other side of the form, log in to a website, or simply send information to a web server in an automated way.

Getting ready

You probably have at least some familiarity with the two basic methods of web form submission—GET and POST. Submitting forms with GET is easy enough. Simply use your scraper to connect to a page at the following link:

```
http://example.com/thanksforsubmitting.html?field1=foo&field2=bar
```

You can even obtain Google results (after setting your headers appropriately, of course) by scraping the page at:

```
http://www.google.com/search?q=java
```

However, submitting forms by POST isn't quite as intuitive. There are two ways to submit post data automatically:

- The first is to send your data as a POST request directly
- The second way is to fill out the form with a **headless browser**, like the one HtmlUnit uses, and click on the submit button. This lets the page send its own request

To complete this recipe, you will need to download JUnit and HtmlUnit and add both of them to your build path. You will also need to create or own a Wikipedia account (you can create an account here:

```
http://en.wikipedia.org/w/index.php?title=Special:UserLogin&type=sign
up
```

Or you can click on the **Create account** link at the top-right corner of the Wikipedia home page.

How to do it...

1. Create a new package, `com.packtpub.JavaScraping.SubmittingForm`, in your `Scraper` project.
2. Create a new `JUnit` test class called `WikiLogin`:

   ```
   package com.packtpub.JavaScraping.SubmittingForm;

   import java.io.IOException;
   import java.net.MalformedURLException;
   import java.util.Iterator;
   import java.util.Set;

   import junit.framework.TestCase;

   import org.junit.Test;

   import com.gargoylesoftware.HtmlUnit.BrowserVersion;
   import com.gargoylesoftware.HtmlUnit.
   FailingHttpStatusCodeException;
   import com.gargoylesoftware.HtmlUnit.WebClient;
   import com.gargoylesoftware.HtmlUnit.html.*;
   ```

```
public class WikiLogin extends TestCase{
public static void main(String args[]) {
org.junit.runner.JUnitCore.main("SubmitForm.WikiLogin");
}

@Test
public void test() throws FailingHttpStatusCodeException,
MalformedURLException, IOException {

WebClient webClient = new
WebClient(BrowserVersion.FIREFOX_10);

  HtmlPage page1 =
    webClient.getPage("http://en.wikipedia.org/w/index.php?
    title=Special:UserLogin");
  HtmlForm form = page1.getForms().get(0);
  HtmlSubmitInput button =
    form.getInputByName("wpLoginAttempt");

  HtmlTextInput userField = form.getInputByName("wpName");
  HtmlPasswordInput passField =
    form.getInputByName("wpPassword");

    userField.setValueAttribute("<username>");
    passField.setValueAttribute("<password>");

    HtmlPage page2 = button.click();

    String headerText =
      page2.getHtmlElementById("firstHeading").
      getElementsByTagName("span").get(0).asText();
    webClient.closeAllWindows();
    assertEquals(headerText, "Login successful");
  }
}
```

How it works...

Although Wikipedia welcomes the use of scrapers, we are using the Firefox browser from the previous recipe, just to get in the habit of modifying our browser headers whenever we build scrapers (it can save a lot of headache down the line).

The scraper first grabs the page at `http://en.wikipedia.org/w/index.php?title=Special:UserLogin`, and gets the first form on the site (`HtmlForm form = page1.getForms().get(0);`), which happens to be the login form.

> Although Wikipedia is a very stable site, and hasn't significantly changed the home page in years, it is important to remember that websites are subject to change, and what was the first form on the site may not be the first form on the site next year, or next month. This is an example of something that could be unit tested.

The username and password fields, as well as the submit button, are selected from the form. The username and password fields have their text set to the username and password for the account, and the button is clicked with HtmlUnit's headless browser functionality:

```
HtmlPage page2 = button.click();
```

The `click` function returns a page, whether or not the page changes when the button is clicked (if it doesn't, the function returns the same page you were already working with). In this case, the second page should now be a logged in Wikipedia userpage.

The first thing that greets Wikipedia users when they log in is the header tag:

```
<h1 id="firstHeading" class="firstHeading" lang="en"><span
  dir="auto">Login successful</span></h1>
```

We check for the presence of this by selecting the element with the ID `firstHeading`, locating its inner `span` tag, and asserting so that the text says **Login successful** as part of the `JUnit` test.

There's more...

If you can access the page, and isolate the element on the page you want to interact with, there is a little limit to the type of interaction HtmlUnit is capable of doing. A comprehensive list of interactions for `HtmlElement` objects can be found in HtmlUnit's documentation pages. See the following link:

`http://HtmlUnit.sourceforge.net/apidocs/com/gargoylesoftware/HtmlUnit/html/HtmlElement.html`

Also be sure to explore the subclasses of `HtmlElement`, such as `HtmlTextInput`, `HtmlSubmitInput`, and `HtmlForm`, which we've used in this recipe. The additional methods they add may come in useful. Although it is possible to treat everything as an `HtmlElement`, you will be limiting yourself only to the interactions that can be done on all elements, rather than something more specific, such as writing text in a text area.

Although HtmlUnit provides ways to access form elements by name and value, that's not very useful if the forms you're trying to log into, eschew these nice naming conventions. The `getByXPath` method can be of great help in this instance. It allows you to access any HtmlUnit form element that you can define. A guide to formatting `XPath` strings that `getByXPath` requires can be found at `http://www.w3schools.com/xpath/xpath_syntax.asp`.

Scraping Ajax Pages (Advanced)

One of the fundamental (and frustrating) differences between a web scraper and a web browser is the ability to execute client-side code, such as JavaScript. This isn't usually a problem, but JavaScript can sometimes be used to encode or hide data that makes it difficult to get at for a web scraper. You'll never take your browser's JavaScript execution capabilities for granted again, as we try to recreate the same functionality in Java.

Getting ready

It is sometimes possible to get around needing to execute JavaScript on a page by scraping the JavaScript itself (sometimes obfuscated and annoying, sometimes pre-packaged in convenient Google Maps address arrays—JavaScript parsing is always an adventure). But because of Ajax, it is not always possible to take this route.

What makes this problem so difficult? After an initial page load, a JavaScript script will execute and send an additional request back to the server for more information. The server will then send the information back that you wanted to the page without doing an additional page reload. Here is an example of one of these pages:

`http://www.peoplefinders.com/`

Observe that the loading screen, after a search is performed, does not take you to an additional page at all—the URL remains the same, only the data in the page changes. There are many similar "people search" sites that use this sort of protection.

The easiest way to get this information is to execute the JavaScript on the page using a headless browser, like HtmlUnit (discussed in the previous recipe).

We will use HtmlUnit to execute JavaScript that will allow us to retrieve the data on these Ajax-protected pages.

How to do it...

1. Create a new package, `com.packtpub.JavaScraping.Ajax`.
2. Add a new `JUnit` test class, with the following code:

```
package com.packtpub.JavaScraping.Ajax;

import java.io.IOException;
import java.net.MalformedURLException;

import junit.framework.TestCase;
import org.junit.Test;

import com.gargoylesoftware.HtmlUnit.BrowserVersion;
import com.gargoylesoftware.HtmlUnit.ElementNotFoundException;
import com.gargoylesoftware.HtmlUnit.
FailingHttpStatusCodeException;
import com.gargoylesoftware.HtmlUnit.WebClient;
import com.gargoylesoftware.HtmlUnit.html.*;

public class Intelius extends TestCase{

  @Test
  public void test() throws FailingHttpStatusCodeException,
    MalformedURLException, IOException {

  WebClient webClient = new
    WebClient(BrowserVersion.FIREFOX_10);
  HtmlPage page = null;

  webClient.setThrowExceptionOnFailingStatusCode(false);
  webClient.setThrowExceptionOnScriptError(false);
  try {
    page =
      webClient.getPage("http://www.intelius.com/results.
      php?ReportType=1&formname=name&qf=Ryan&qmi=E&qn=
      Mitchell&qcs=Needham%2C+MA&focusfirst=1");
    }
  catch (FailingHttpStatusCodeException e) {
    e.printStackTrace();
  }
  catch (MalformedURLException e) {
    e.printStackTrace();
  }
  catch (IOException e) {
    e.printStackTrace();
  }
    DomElement name = null;
     for (int i = 0; i < 20; i++) {
        Boolean mblueExists = true;
        try{
          name = page.getElementById("name");
        }
        catch(ElementNotFoundException e){
          mblueExists = false;
        }
```

```
                if (mblueExists) {
                    break;
              }
                synchronized (page) {
                  try {
                    page.wait(500);
                  }
                  catch (InterruptedException e) {
                    e.printStackTrace();
                  }
                }
              }
            webClient.closeAllWindows();
        try{
          assertEquals("Ryan Elizabeth Mitchell",
            name.getElementsByTagName("em").get(0).asText());
        }
        catch(NullPointerException e){
          fail("name not found");
        }
      }
    }
```

How it works...

As with many libraries similar to HtmlUnit, JavaScript needs to be explicitly executed, or handled somehow. HtmlUnit is nice in that it handles things by default. In fact, it handles them so well that it often trips up on errors (at the time of this writing, a relatively inconsequential 404 not found error for a library that Intelius's JavaScript requires) and will throw exceptions for JavaScript you didn't even realize you were executing.

By setting the following flags we can avoid these pitfalls:

```
webClient.setThrowExceptionOnFailingStatusCode(false);
webClient.setThrowExceptionOnScriptError(false);
```

Although you will still see warnings in Eclipse, the exceptions will be ignored and the page will continue to be executed regardless of whether the JavaScript is successful or not.

We'll learn more about threading in the next recipe, but consider this to be your unofficial introduction. HtmlUnit often creates new threads (similar to a process—it is an independent execution of a program that can be performed while other threads/processes are working on other parts of the same program) when efficient to do so, like when it is loading and processing pages. The synchronized function suspends the execution of all threads; in this case, waiting 500 milliseconds (half a second) before checking to see if the Ajax on the page has finished loading:

```
synchronized (page) {
  try {
    page.wait(500);
  }
  catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

We can check to see when the page is done loading by checking for the existence of the tag with the ID `mblue`, which only appears on the loaded version of the page, and nowhere in the loading screen. Conveniently, this tag also wraps around the name of the subject we are searching for (in this case, "Ryan Elizabeth Mitchell") and lets us compare the name we find in the page to the target name in order to pass the `JUnit` test.

There's more...

Although we've done a lot with HtmlUnit in the last few recipes, the library is capable of many more functions than we could possibly cover in this section. The project's `SourceForge` page, `http://HtmlUnit.sourceforge.net/index.html`, is an invaluable resource in getting documentation, answering questions, and getting code snippets for common functions (see **Using a proxy server** on the Introduction page of this link, `http://HtmlUnit.sourceforge.net/gettingStarted.html`, for another cool technique).

HtmlUnit has been modified and developed for more than 10 years now, and some of the methods and best practices with the library have been deprecated or are no longer usable. On the other hand, this also means that new functionality is being constantly added (such as, hopefully soon, the ability to select elements by class). You can keep an eye on the project, learn about changes with each release, and perhaps even get involved in this open source effort at the **Project Information** page:

`http://HtmlUnit.sourceforge.net/project-info.html`

Faster scraping through threading (Intermediate)

Although processors were originally developed with only one core, modern multicore processors (processors with multiple separate and independent CPUs embedded on a single chip) run most efficiently when they are executing multiple threads (separate, independent sets of instructions) at once.

If you have a powerful quad-core processor completely dedicated to executing your program, but your program is not using threading, you are essentially limited to using 25 percent of your processor's capabilities. In this recipe, we will learn how to take advantage of your machine (especially important when running on a dedicated server) and speed up your scraping.

Getting ready

Writing software that uses threading can be extremely important in web scraping for several reasons:

- You're often working on machines that are completely dedicated to scraping and you need to be able to take advantage of multicore processors appropriately
- It often takes a long time for web pages to respond, and having multiple threads running at once can allow the processor to help optimize efficiency during these "down" times
- You need all your scrapers to share resources easily; global variables, for example, are shared between all threads

Because we will be using the code from persisting data, it would be helpful to review the recipe *Persisting data (Advanced)*, have your MySQL database ready, and download the JDBC drivers from `http://dev.mysql.com/downloads/connector/j/`. Alternatively, you can relatively easily replace the code that actually stores the results in a database and do something else with it, such as simply printing it on the screen.

How to do it...

Up until this point, every class we've written has either begun execution in the main method (if called statically) or in the method that we explicitly call (for example, `myWebsite.getString()`). However, threads, which extend the core Java `Thread` class, have a `run` method that is started when `myThreadClass.start()` is called.

1. We will be basing our thread code off of the code created in the *Persisting data (Advanced)* recipe. Review this recipe if you haven't completed it already.

2. Methods that remain unchanged are marked as `methodName(){...}`. You should create a new package, `com.packtpub.JavaScraping.Threading`, that contains two methods: `ThreadStarter` and `WikiCrawler`.

   The `WikiCrawler` method:

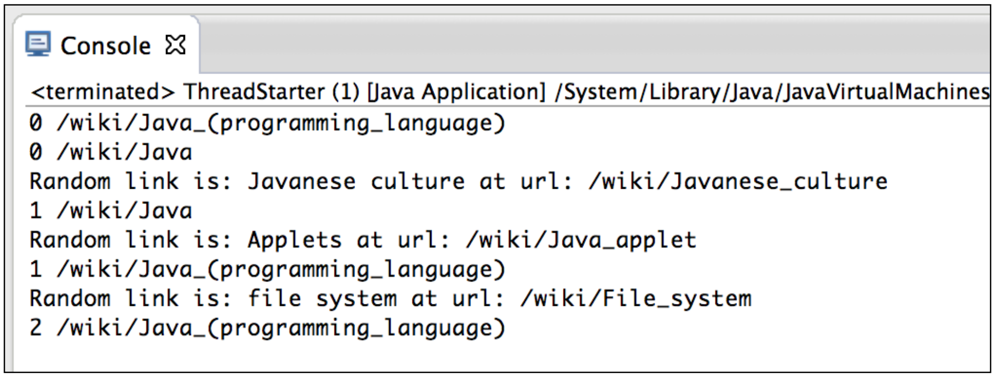
```
package com.packtpub.JavaScraping.Threading;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.net.MalformedURLException;
import java.net.URL;
import java.net.URLConnection;
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.PreparedStatement;
import java.sql.SQLException;
import java.util.Properties;
import java.util.Random;

import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.select.Elements;

class WikiCrawler extends Thread {

  private static Random generator;
  private static Connection dbConn;

    public WikiCrawler(String str) {
      super(str);
    generator = new Random(27182845);

    String dbURL = "jdbc:mysql://localhost:3306/java";
    Properties connectionProps = new Properties();
    connectionProps.put("user", "<username>");
    connectionProps.put("password", "<password>");
    dbConn = null;
    try {
      dbConn = DriverManager.getConnection(dbURL,
        connectionProps);
    } catch (SQLException e) {
      System.out.println("There was a problem connecting to
        the database");
      e.printStackTrace();
    }

    PreparedStatement useStmt;
    try {
      useStmt = dbConn.prepareStatement("USE java");
      useStmt.executeUpdate();
    } catch (SQLException e) {
      e.printStackTrace();
    }
    }

    public void run() {
      String newUrl = getName();
        for (int i = 0; i < 10; i++) {
            System.out.println(i + " " + getName());
            newUrl = scrapeTopic(newUrl);
            if(newUrl == ""){
              newUrl = getName();
            }
            try {
                sleep((int)(Math.random() * 1000));
            } catch (InterruptedException e) {}
        }
        System.out.println("DONE! " + getName());
    }


  public static String scrapeTopic(String url){
    ...
  }

  private static void writeToDB(String title, String url){
    ...
  }

  public static String getUrl(String url){
...
  }

}
```

3. Replace `<username>` and `<password>` in `WikiCrawler` with the username and password for your MySQL database.

4. The following class, `ThreadStarter`, creates two `WikiCrawler` threads, each with a different Wikipedia URL, and starts them:

```
package com.packtpub.JavaScraping.Threading;

class ThreadStarter {
    public static void main (String[] args) {
        WikiCrawler javaIsland = new
          WikiCrawler("/wiki/Java");
        WikiCrawler javaLanguage = new
          WikiCrawler("/wiki/Java_(programming_language)");
        javaLanguage.start();
        javaIsland.start();
    }
}
```

The output would look something like this:



```
<terminated> ThreadStarter (1) [Java Application] /System/Library/Java/JavaVirtualMachines
0 /wiki/Java_(programming_language)
0 /wiki/Java
Random link is: Javanese culture at url: /wiki/Javanese_culture
1 /wiki/Java
Random link is: Applets at url: /wiki/Java_applet
1 /wiki/Java_(programming_language)
Random link is: file system at url: /wiki/File_system
2 /wiki/Java_(programming_language)
```

How it Works...

As mentioned earlier, `WikiCrawler.run()` acts as a jump-off point in the thread execution, in much the same way that `main` does in a normal class. Although `WikiCrawler.scrapeTopic()` has been slightly modified from the code presented in the *Persisting data (Advanced)* recipe, it serves much the same purpose in that it retrieves from the web page for a given Wikipedia article, gets a new random number based on the number of links to other articles found, and writes a new line to a table in the database. In this case, however, it returns the next URL as a string for the main loop in the `run` method to use on its next pass.

It is tempting to create global variables in the Thread class for things like the starting URL (for example, /wiki/Java) that we feed it, but keep in mind that threads share memory. If this global variable were created, each thread would go off of the URL that was given to the last thread (in this case, /wiki/Java_Road). We use thread names to differentiate the threads. In this case, it is convenient to use the names as the URL that they are assigned to as a starting point for the web crawling.

The implementation of the ThreadStarter class is relatively straightforward. This class, as a normal, non-thread Java class, starts execution in the main method. It initializes four new WikiScraper threads and starts them in one line each.

There's more...

This was a very simple, two-class example designed to demonstrate how to create and run threads. Although the threads were all of one type in this example (the WikiScraper class), it is certainly possible to have threads that are designed only to wait for pages to load and return the page to a main class, threads to analyze the data in the pages, and threads to store the data in a database. Because threads share memory, it is easy to pass information between them.

Although it is impossible to be certain that a thread has executed (and therefore, if the data you want is present yet), you can certainly create a loop that continuously checks to see if the data has been added to an array yet, and, if so, does something with the data before continuing the loop and waiting for the next piece of data to be added.

Threading gets very complicated very fast, although it has immense power and can vastly improve the performance of your programs. For more information about different methods of using threads to improve runtime, the following links are a good start:

http://docs.oracle.com/javase/tutorial/essential/concurrency/

http://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html

Faster scraping with RMI (Advanced)

Now that we've learned how to speed up our code by using multiple threads on the same server, we will learn how to speed up our code by using multiple servers. **Distributed computing** is a powerful and tricky tool to master, but the basics are straightforward. If you are interested in RMI, I highly recommend that you pay close attention to the *There's more...* section at the end of this recipe.

Getting ready

Using multiple servers to scrape the Web can be useful for several reasons. For one, even with threading, the limitations of one machine can be a problem for large-scale applications. For another, while visiting a website repeatedly from a single IP address might be a big red flag, visiting a website repeatedly from a multitude of helper cloud machines with separate addresses, all sending data to a parent database, might not be so noticeable.

Java's **Remote Method Invocation** (**RMI**) handles the communication between two distant JVMs by sending serialized objects, handling calls to those objects, and sending return values from those calls. Object serialization, converting objects into streams of bytes for portability, makes it possible for a local class to instantiate and interact with a remote object as if it were local.

The best way to really understand how this works, and how it can be immensely useful in web scraping, is to take a look at a simple example. Our program architecture will consist of a **Client** (our home server that will be collecting scraping data but not doing any scraping itself), a **Server** (remote machine, responsible for instantiating and registering scrapers with the RMI), and a scraper, `ScraperImp` (this is what does the actual scraping, and is what is serialized and sent back to the Client). Because all RMI-serialized objects must have an associated interface that they implement, we also have a `ScraperInterface`, for a total of four classes.

How to do it...

1. Create a new package, `com.packtpub.JavaScraping.RMI`, that contains four classes: `Client`, `ScraperImp`, `ScraperInterface`, and `Server`. Although it is possible to actually run this using a remote server, if you have access to one, you can also run the classes in different processes on the same local machine. We will provide code and instructions for doing both, but it is recommended to test the code locally first.
2. The following three classes are run from the remote machine that is doing the actual scraping.

   The Remote Interface (this should also be included on the Client machine so that it has the object definition):

   ```
   package com.packtpub.JavaScraping.RMI;

   import java.rmi.*;

   public interface ScraperInterface extends  Remote {
     public String getMessage() throws RemoteException;
   }
   ```

3. The implementation of the Remote Interface (this contains our scraping code, and gathers the data to return):

```
package com.packtpub.JavaScraping.RMI;

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.net.*;
import java.io.*;

import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;

public class ScraperImp extends UnicastRemoteObject implements
ScraperInterface {
  public ScraperImp() throws RemoteException {}
  public static void main() {
    System.out.println("Hello, World!");
  }

  public String getMessage() throws RemoteException {
    return(scrapeTopic("wiki/Java"));
  }

  public static String scrapeTopic(String url){
    String html = getUrl("http://www.wikipedia.org/"+url);
    Document doc = Jsoup.parse(html);
    String contentText = doc.select("#mw-content-text >
      p").first().text();
    return contentText;
  }

  public static String getUrl(String url){
    URL urlObj = null;
    try{
      urlObj = new URL(url);
    }catch(MalformedURLException e){
      System.out.println("The url was malformed!");
      return "";
    }
    URLConnection urlCon = null;
    BufferedReader in = null;
    String outputText = "";
    try{
      urlCon = urlObj.openConnection();
      in = new BufferedReader(new
        InputStreamReader(urlCon.getInputStream()));
      String line = "";
      while((line = in.readLine()) != null){
        outputText += line;
      }
      in.close();
    }catch(IOException e){
      System.out.println("There was an error connecting to
        the URL");
      return "";
    }
    return outputText;
  }

}
```

4. The `Server` class creates Remote Implementations, and registers them with the Java RMI. This allows the client to access them across a network.

```
package com.packtpub.JavaScraping.RMI;
public class Server {
  public static void main(String[] args) {
    try {
      ScraperImp localObject = new ScraperImp();
      Naming.rebind("rmi://localhost:1099/Scraper", localObject);
      System.out.println("Successfully bound Remote
Implementation!");
    } catch(RemoteException re) {
      System.out.println("RemoteException: " + re);
    } catch(MalformedURLException mfe) {
      System.out.println("MalformedURLException: " + mfe);
    }
  }
}
```

5. The `Client` class is run on your local machine, and connects to the registered remote interface:

```
package com.packtpub.JavaScraping.RMI;

import java.rmi.*;
import java.net.*;
import java.io.*;
```

```
public class Client {
  public static void main(String[] args) {
    try {
      String host =
        (args.length > 0) ? args[0] : "localhost";
      ScraperInterface remObject = (ScraperInterface)Naming.
lookup("//127.0.0.1:1099/Scraper");
      System.out.println(remObject.getMessage());
    } catch(RemoteException re) {
      System.out.println("RemoteException: " + re);
    } catch(NotBoundException nbe) {
      System.out.println("NotBoundException: " + nbe);
    } catch(MalformedURLException mfe) {
      System.out.println("MalformedURLException: " + mfe);
    }
  }
}
```

6. Although we will be building and compiling this in Eclipse, we will be running it from the command line. It is necessary to run the `rmiregistry` manually, and load the classes in a particular order, either on different physical servers or in different processes on your local machine, for everything to communicate with one another.
7. Because this cannot be done in Eclipse, you will need to open three shells—two on your client machine (the machine receiving the data from the remote scrapers), and three on each remote machine (in this example, we are using just one remote machine, but it can be easily extended).
8. To run this locally, using separate processes, open three terminal windows. Navigate to the `bin` folder of your `Scraper` project in each terminal. In the first terminal, run the command:

   ```
   $rmiregistry
   ```

   Or, in Windows:

   ```
   $start rmiregistry
   ```

9. This will start the RMI registry in preparation for the Server to register the `Scraper` implementation. In the second terminal window, navigate to your `bin` folder inside the project and start the server using the command:

   ```
   $java com.packtpub.JavaScraping.RMI.Server
   ```

10. Finally, start the Client using the following command:

    ```
    $java -cp .\:../lib/jsoup-1.7.2.jar com.packtpub.JavaScraping.RMI.
    Client
    ```

    And in Windows:

    ```
    $java -classpath .;../lib/jsoup-1.7.2.jar com.packtpub.
    JavaScraping.RMI.Client
    ```

11. To do this on two separate physical servers, you first need to replace the `localhost` address with your remote server's IP address, start up the RMI registry then run `WikiRMI.Server` on the remote machine, and run `WikiRMI.Client` on your local machine.

How it works...

When the Server is started, it immediately instantiates a `ScraperImp` object containing methods for getting information from `http://www.wikipedia.org/wiki/Java` (the particular URL could also be passed as an argument, either on instantiation by the Server, or by the Client later, when `ScraperImp.getMessage()` is called, but hard coding it simplifies things in this example).

After the `ScraperImp` object is created, it binds it to the RMI on the localhost, port 1099, under the lookup name `Scraper`, using the following line:

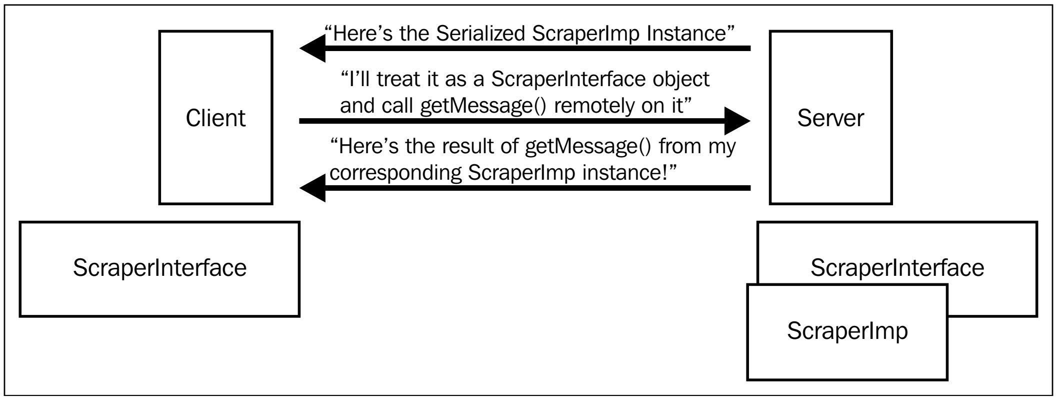
```
Naming.rebind("rmi://localhost:1099/Scraper", localObject);
```

The `rebind` method is used rather than `bind`, in order to remove any objects that may already be bound under `Scraper` to begin with, and avoid an exception if a conflict occurs. It is not necessary to use rebind, rather than bind, but it may be preferable in some situations.

When the Client is started, it uses a lookup to find the `Scraper`:

```
Naming.lookup("//127.0.0.1:1099/Scraper");
```

After that, the entire exchange looks something like this:


Client
Server
"Here's the Serialized ScraperImp Instance"
"I'll treat it as a ScraperInterface object and call getMessage() remotely on it"
"Here's the result of getMessage() from my corresponding ScraperImp instance!"
ScraperInterface
ScraperInterface
ScraperImp

It is important to remember that the Client is dealing with a copy of the Server's remote object, made by the Client through the serialized data it received from the Server. Any changes made to the Client's object by the client will not necessarily be reflected in the server, unless an explicit method call is made.

There's more...

As you may have gathered, the Java RMI is an extremely complex (but extremely useful) mechanism that this simple example does not do justice. Although it can be difficult to understand at first glance, writing code that spans across distributed networks, and the ability to interact with remote objects as if they were local, is extremely useful when trying to coordinate multiple web scrapers.

Oracle's tutorial section does have a decent introduction to RMI. To know more, go to the following link:

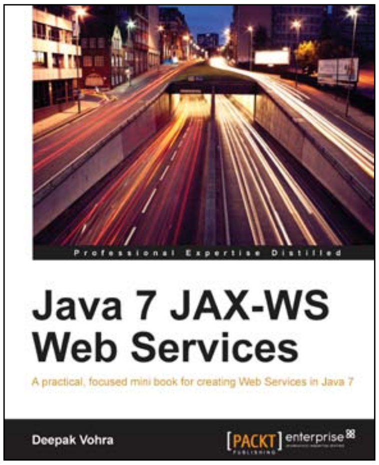
`http://docs.oracle.com/javase/tutorial/rmi/`

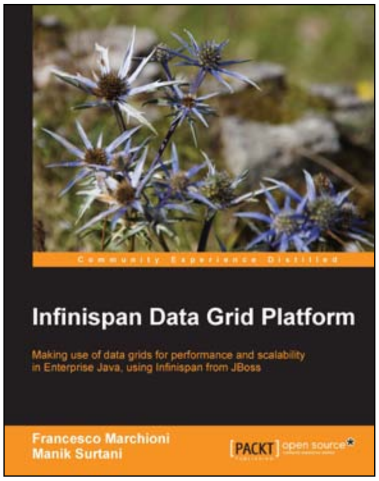
If you are looking for a slightly more complex material, they also have an excellent introduction to Distributed Computing with RMI; this can be found at the following link:

`http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html`

Not only is the RMI mechanism itself difficult to master but also a great deal of thought must be put into the design patterns used when architecting systems that implement it, particularly if these systems grow beyond just a remote server or two. Although it might be tempting to mindlessly copy a suitable-looking design pattern from a book, it's important to put thought into the most efficient way to architect your system, based on the job it needs to perform.

INSTANT
Short | Fast | Focused
PHP Web Scraping
Get up and running with the basic techniques of web scraping using PHP
Jacob Ward
PACKT


Professional Expertise Distilled
Java 7 JAX-WS Web Services
A practical, focused mini book for creating Web Services in Java 7
Deepak Vohra
PACKT enterprise



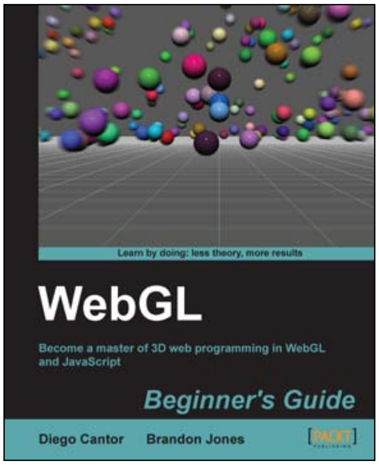
Community Experience Distilled
Infinispan Data Grid Platform
Making use of data grids for performance and scalability in Enterprise Java, using Infinispan from JBoss
Francesco Marchioni
Manik Surtani
PACKT open source

Infinispan Data Grid Platform

ISBN: 978-1-849518-22-2 Paperback: 150 pages

Making use of data grids for performance and scalability in Enterprise Java, using Infinispan from JBoss

1. Configure and develop applications using the Infinispan Data grid platform
2. Follow a simple ticket booking example to easily learn the features of Infinispan in practice
3. Draw on the experience of Manik Surtani, the leader, architect and founder of this popular open source project



Learn by doing: less theory, more results
WebGL
Become a master of 3D web programming in WebGL and JavaScript
Beginner's Guide
Diego Cantor
Brandon Jones
PACKT

WebGL Beginner's Guide

ISBN: 978-1-849691-72-7 Paperback: 376 pages

Become a master of 3D web programmingin WebGL and JavaScript

1. Dive headfirst into 3D web application development using WebGL and JavaScript.
2. Each chapter is loaded with code examples and exercises that allow the reader to quickly learn the various concepts associated with 3D web development
3. A practical beginner's guide with a fast paced but friendly and engaging approach towards 3D web development

Please check **www.PacktPub.com** for information on our titles