Pro Jakarta Velocity: From Professional to Expert

Rob Harrop

ISBN: 1-59059-410-X

Available September 2004

Apress™

Chapter 1: Introducing Velocity*

Chapter 2: Getting Started with Velocity*

Chapter 3: Using the Velocity Template Language

Chapter 4: Introducing Patters and Best Practices

Chapter 5: Creating Stand-Alone Applications with Velocity*

Chapter 6: Creating Web Applications with Velocity

Chapter 7: Using Velocity with Anakia

Chapter 8: Using Additional Velocity Tools

Chapter 9: Going Inside Velocity

Chapter 10: Extending Velocity

Appendix: Velocity Reference

*These chapters are included in the reader

Please note that the chapters included here are in their "beta" form, and are subject to modification and correction before the final book ships.

About the Author

Rob Harrop is Lead Software Architect of UK-based development house, Cake Solutions Limited. At Cake, Rob leads a team of six developers working on enterprise solutions for a variety of clients including the Department of Trade and Industry, the Metropolitan Police and NUS Services Limited. Rob, and Cake, specialises in both .NET and J2EE-based development, with Rob having been involved with .NET since the alpha stages.

Rob is the author of Pro Jakarta Velocity (Apress, Not Yet Published) as well as co-author of Pro Jakarta Struts (Apress, 2004), Pro Visual Studio .NET (Apress, Not Yet Published) and Oracle Application Server 10g: J2EE Deployment and Administration (Apress, Not Yet Published).

In his limited spare time, Rob enjoys playing about with different technologies, his current favourites being Groovy and AOP. Rob is a committer on the open source Spring project (www.springframework.org), a Java and .NET application framework built around the principle of Dependency Injection. When not sat in front of the computer Rob usually has his head buried in a book and prefers the fantasy parodies of Terry Pratchett's Discworld.

About the Technical Reviewer

Jan Machacek is lead programmer of UK-based software company Cake Solutions Limited(http://www.cakesolutions.net), where he has helped design and implement enterprise-level applications for a variety of UK and US based clients. In his spare time he enjoys discovering new software and hardware technologies. Apart from Java, Jan is interested in the .NET framework and non-procedural and AI programming.As a proper computer geek, Jan loves the Star Wars and The Lord of the Rings. Jan lives in Manchester, UK, and can be reached at jan@cakesolutions.net.

Introduction

As Java programmers we should consider ourselves lucky. We have a vast range open source tools available to us. For almost any problem you can conceive there are numerous open source tools that can be used to solve that problem. This is especially evident in the glut of MVC frameworks available for building Java-based web applications. It was while I was working on a book for one of these frameworks, Struts, that I realised, that although the ability for Struts to use different technologies for the View layer of the MVC triad was much lauded, these View technologies suffered from a lack of exposure and documentation. During the course of writing the Struts book I worked on a single chapter to demonstrate Velocity integration but it quickly became apparent that much more could, and should, be written about Velocity.

I am a great fan of Velocity mainly due to its simplicity and fantastic performance. Wherever possible I prefer to use Velocity in preference to JSP, as the View technology in my web applications. More than that, I find Velocity is the ideal tool for a myriad of tasks that go well beyond the web and well beyond the capabilities of JSP.

What shocks me most is that Velocity does not have the same level of exposure and widespread acceptance as Struts. In my eyes there is a much clearer benefit, namely in performance, to using Velocity rather than JSP that Velocity should be the natural choice for developers in many cases. Add to this the ability to use Velocity in applications that run outside of the web context, and you have a powerful tool ready to exploit. I was surprised during a conversation with several developers at a conference how many were unaware of Velocity and how many were aware of Velocity but had never used it. For this reason I felt the time was right for a book on Velocity to cover all aspects of the technology and to act as an instructive guide to correct use of all things Velocity-related.

My overriding aim with this book is to help other developers enjoy the same success with Velocity that my team and I enjoy. I believe that I have achieved this aim and I hope that in this book you will find all the information necessary to make use of Velocity in your projects.

CHAPTER 1

Introducing Velocity

I'M GUESSING that since you've picked up this book, you already have more than a passing familiarity with Java. You may have noticed over the past few years that text-based output formats have been at the forefront of the Java world. Certainly XML is perhaps the most talked about output format, and HTML is definitely the most widely used. Whilst these formats are fantastic in terms of ease of creation when dealing with simple, static content, creating complex output from dynamic data can prove to be much trickier, and maintaining that content is even harder. Some of you may be thinking that the problem is solved, in part, by JSP.

Sure, JSP makes creating text-based output to send to Web clients easier, but it's by no means the simplest solution, and it doesn't solve the problem of creating text-based output for use in other areas, such as e-mail messaging or desktop applications. The common solution in those areas is to assemble the text content manually, intermixing static content with dynamic data in a mish-mash of hideous Java code. Creating code in this way is error-prone and difficult to maintain, especially if it has to be maintained by someone other than the original developer. This is where Velocity comes in. Simply put, Velocity takes the pain out of creating complex text output and makes creating simple text output even simpler. Velocity isn't perfect, however; it lacks in one important area—documentation. Whilst the documentation is not particularly bad for an open-source project, it leaves a lot of things uncovered and doesn't cover other features in enough detail. That's where this book can help. During the course of the book, I'll take you through all aspects of Velocity, from downloading and getting started to dissecting the code that goes into making Velocity work.

What Is Velocity?

The Jakarta Velocity project is a Java implementation of a template engine. The term *template engine* is quite broad, but in the context of Velocity and the like, it means a piece of software that takes as input a template or templates and couples this with variable data to produce some form of output. In the case of Velocity, your template is simply plain text containing Velocity Template Language (VTL) directives, coupled with static content. The VTL directives tell Velocity how to

combine the static content in the template file with the variable data from your Java application to provide the desired output. The variable data can come from anywhere within your Java applications; as you'll see in Chapter 2, the mechanism for transferring data between your application and Velocity is flexible, allowing you to work with all kinds of Java `Objects`, including `Collections` and `Arrays`. Output from Velocity is always text-based, but the format of the text you generate isn't constrained. This means you can use Velocity to create HTML or XML output as well as plain-text output.

Although what Velocity does is simple and using it is easy, the Velocity engine is sophisticated and offers a quick, robust parser, template caching, and pluggable introspection. Many other tools have been built on top of Velocity, and some of these, such as Anakia, which is covered in Chapter 8, come in the standard Velocity distribution. For the most part, these tools make it simpler to use Velocity in a specific context. Chapters 7–8 cover a choice selection of these add-on tools.

Introducing Java Template Engines

Of course, Velocity isn't the only template engine available for you to use. In keeping with most open-source projects, you have plenty of alternatives to Velocity. Since this book focuses on Velocity, I won't cover these other tools in too much detail; I'll leave the exploration of these topics to you, but the following list describes them briefly:

WebMacro: The WebMacro template engine is kind of the precursor to Velocity. In fact, the reason Velocity was originally created was because of the then-restrictive licensing of WebMacro. Velocity and WebMacro share similar syntax, making it quite simple to move templates between the two systems. You can find more details at `http://www.webmacro.org`.

StringTemplate: This is one system to look at if you require inheritance in your templates. StringTemplate was created by Terrence Parr of ANTLR fame. You can find more details on StringTemplate at `http://www.antlr.org/stringtemplate/index.html`.

Jamon: Jamon is quite a complex and involved template engine. It differs greatly from Velocity in that it compiles your template files into Java classes, which you can then use within your application just like normal classes. Jamon resides at `http://www.jamon.org`.

JBYTE: The JavaBY Template Engine (JBYTE) is a simplistic, lightweight template engine that can generate any kind of text-based output. You can find more information on JBYTE at `http://javaby.sourceforge.net/`.

Enhydra XMLC: Like Jamon, XMLC creates compiled templates that can then be accessed directly from your code. XMLC is quite tightly coupled to produce output for Web applications, so it may not be ideal if you want to produce a desktop application. You can find the XMLC Web site at `http://xmlc.objectweb.org/`.

So, Why Use Velocity?

With so many choices, you may be wondering why you should use Velocity and not one of the other solutions. Well, obviously the choice is yours; however, I recommend Velocity for the following four main reasons:

Simplicity: A template engine should be lightweight and unobtrusive. In other words, you don't want a template engine that reduces your application to a crawl, and you certainly don't want to have your application constrained by a restrictive API. In addition, you don't want a template engine that's bogged down by unnecessary features that deter from the main aim of generating content. Velocity satisfies the need for a simple, capable template engine. The Velocity API places no specific requirements on the architecture of your application, allowing you total control, and Velocity does nothing other than templating.

Integration: As part of the Apache Jakarta project, Velocity is in the company of many other excellent projects. Wherever possible, you'll find that Velocity is well integrated with other Jakarta projects such as Struts, Torque, and Tapestry. Chapter 6 covers integration with Struts in more detail.

Proven success: As you'll see in the "Seeing Velocity in Action" section, Velocity is already in use in many different areas, including some commercial desktop applications. Many developers, myself included, have successfully used Velocity in a wide range of applications. Velocity has been through many revisions and is now a highly robust, mature technology.

Supporting documentation: None of the other projects has such a fantastic book to accompany them! On a serious note, Velocity has at least one other book written about it, but the other engines have none. Velocity is established enough as a technology to warrant books being written about it, which means you have access to useful documentation.

The Benefits of Templating with Velocity

Depending on the type of applications you create, using a template engine has many benefits. For applications where you have to produce the output manually, such as a desktop application or an e-mail newsletter service, the following core benefits make using Velocity a sensible decision:

Code separate from content: By using a template engine, you can factor all the static content into template files so that it isn't intermixed with your Java code. By using this approach, you'll find that obtaining correct output is far easier and also that you can make most modifications without having to touch your Java code.

Simple syntax: Nondevelopers can understand the simple syntax employed by Velocity. In the case of the e-mail newsletter service, this means that the marketing people could change the content of the newsletter without having to get the development team involved.

Performance: For the most part, generating output using Velocity will be quicker than manually constructing the text. Also bear in mind that Velocity is in constant development, so the performance will continue to improve.

In the case of Web-based applications, you get all the same benefits, but you get the most important benefit of all: complete separation of the logic of your application from the presentation. With JSP, you can quite easily perform a bunch of business logic using JSP scriptlets, but with Velocity, the language constructs are limited to allowing you to perform the simple decisions and looping necessary to produce your output—you can't, for instance, modify a database from a Velocity template.

CAUTION *What I've said here is only partially true. Velocity itself doesn't provide support for any kind of business logic or data access. However, as you'll come to see, you can make any object accessible to a template, and you can access any of the methods on that object from within that template. So, if you really go out of your way to do so, you can perform data access from a template; it's just not as simple as with JSP scriptlets where you can directly instantiate an instance of any object.*

Understanding the Uses of Velocity

I've included this section in the hope of debunking a common misconception about Velocity. Velocity isn't just a tool for building Web applications. Velocity is

an excellent tool for this purpose, but that's just a small part of what it's all about. If you were expecting to see me jump straight into MVC-based Web applications and how Velocity can help you to build this kind of application, then don't worry—that's all covered; it's just later in the book than you may have expected.

NOTE *If you aren't familiar with the term* MVC*, don't worry too much; it's explained in much more detail in later chapters.*

The focus of this book isn't on building Web applications but on improving all kinds of applications using Velocity. Web applications are just a one part of this. Velocity has many uses, and during the course of this book I'll explore the most common types of application including Web applications, e-mail applications, console applications, and desktop GUI applications.

Seeing Velocity in Detail

Now that you're familiar with the concept of what Velocity and template engines are in general, the following provides a more detailed look at the features of Velocity:

Flexible template mechanism: Whilst the easiest method of generating output is to use templates that are stored as files, Velocity is just as comfortable working with templates that are dynamically generated or that are loaded from a database. If you find that you can't load templates from your desired source, then you can easily extend Velocity to provide this functionality, as detailed in Chapter 10.

Servlet integration: As part of the standard distribution, Velocity includes the `VelocityServlet`, which eases the use of Velocity in a servlet environment. You need to create your own servlet derived from this class to process the Velocity content.

Simple syntax: The template language used by Velocity, VTL, is simple enough to be understood by nonprogrammers, making Velocity an ideal solution for use in Web development where designers need to modify page layout.

Texen: Texen is a text-generation tool, built on top of Velocity and integrated with Ant. You can use Texen to generate any kind of text-based output, and it's used to great effect in the Jakarta Torque project for generating SQL scripts. Texen is included as part of the standard Velocity download and is covered in much more detail in Chapter 8.

Anakia: Using Anakia, you can transform XML documents into any other text-based format. Anakia uses Velocity and JDOM internally and, like Texen, is integrated with Ant. Also like Texen, you'll find Anakia in the standard Velocity distribution. I cover Anakia in more detail in Chapter 7.

VelocityTools: This is a subproject of the main Velocity project and as such has to be downloaded separately. The VelocityTools project is a collection of plug-ins for Velocity that overcomes some of the shortcomings of the standard engine. Specifically, the VelocityTools project makes it much easier to get desirable output from your templates by providing a range of math and formatting functions. You'll also find that building Web applications with Velocity is much easier using VelocityTools because of the extra support for servlets and for Struts. VelocityTools is covered in more detail throughout the book.

Seeing Velocity in Action

As I mentioned earlier, Velocity is in use in a wide range of applications, both on the Web and on the desktop. It's worthwhile to take a few minutes to check out a couple of these projects to see how other people are putting Velocity to use—who knows, you may be inspired.

Velocity on the Web

The most common use for Velocity is in Web applications, and because of this, many Web-based tools run on Velocity. I've included two I think are quite cool. Download them and give them a go, and then you can dissect the code to see how Velocity is employed.

Scarab

Scarab is an open-source issue and project artifact–tracking application, which has been built using Velocity and Apache Turbine. You should download this application anyway—it's really quite handy for use in your projects, and it has some good Velocity code to explore. You can obtain the latest version of Scarab from `http://scarab.tigris.org/`.

Roller Weblogger

Weblogging is the current craze, and a whole host of open-source Weblogging tools have popped up, built not just in Java but also in PHP, Python, and Perl.

Roller is one of the better Webloggers and is a great example of how to use Struts and Velocity to put together a robust Web application. Velocity is used as an alternative to JSP to allow end users to customize their own pages on a Roller-based Weblog. The Roller project is hosted at `http://www.rollerweblogger.org/`. If you're going to download this application, you should definitely refer to the article at `http://www.onjava.com/pub/a/onjava/2002/04/17/wblogosj2ee.html`, which includes one of the developers of Roller talking about the development of Roller using various open-source technologies including Velocity.

Velocity on the Desktop

As I mentioned before (and no doubt will mention again), Velocity is much more than just a tool for building Web applications. Many quality desktop applications use Velocity to produce their text-based output.

IntelliJ IDEA

A very well-known project, IDEA is one of the best Java IDEs on the market. IDEA uses Velocity for code generation, reducing the amount of repetitive code that you as a developer need to write. If you want to see this generation process in action, you can download a 30-day trial of IDEA from the IntelliJ Web site at `http://www.intellij.com/`. Unfortunately, the source code for IntelliJ isn't available, so you can't see how it was done, but by the end of the book you'll be able to reproduce the behavior anyway.

Gentleware Poseidon

Gentleware Poseidon is one of my favorite UML tools, which will transform my UML model directly into Java code. Can you guess which tool they use to do this? That's right, Velocity! You can try out the Java code generation capability by downloading the free community edition at `http://www.gentleware.com/products/descriptions/ce.php4`.

Generic Velocity Tools

Of course, Velocity isn't limited to use in a desktop or Web application. Many of the best applications are cool little utilities or plug-ins that make life that little bit easier. Here again, Velocity makes creating text-based output simple and fast.

Middlegen

Middlegen is a nifty little tool for anyone who has to write data access code. Using Middlegen, you can automatically generate the code required to access a database using EJB 2.0, JDO, Hibernate, or JSP/Struts. Velocity is used as a means to generate the code necessary to access the database. Middlegen is all open source, so you can delve into the code to see how Velocity is used. The code is available for download along with a binary at `http://boss.bekk.no/boss/middlegen/index.html`.

Luxor

Luxor is a Java XML User Interface Language (XUL) toolkit. Using Luxor, you can build cross-platform rich-client applications without having to get your hands dirty with Swing. As part of the toolkit, Luxor includes a templating engine, which is, of course, Velocity. Luxor is an interesting project even without the Velocity connection, so you should download the code and have look around—you'll find it useful for a variety of reasons. Just visit `http://luxor-xul.sourceforge.net/`.

NOTE *I found all these projects when looking for Velocity samples on the Velocity Web site. You can find a much more comprehensive list of projects on the Powered by Velocity page at* `http://jakarta.apache.org/velocity/powered.html`.

Summary

By now you should be more than familiar with the concept of a template engine and also with the specific uses of a template engine. I've given you a whirlwind rundown on the basics of Velocity's feature set. No doubt you can see from the simple descriptions that Velocity is quite a simple, lightweight technology, but you may not yet appreciate the flexibility it offers. In next chapter, I'll take you through the basics of getting Velocity installed, writing some templates, and creating some output. After that, I'll move on and give you a fuller look at VTL and some advanced features of the Velocity engine, and then it's onto the good stuff when I start to show you how to put Velocity to good use. If you're already familiar with how to install Velocity and VTL, you can skip the next two chapters and move straight onto the specific discussions of how to use Velocity. For those of you who are completely unfamiliar with Velocity, I recommend you look at some of the applications mentioned previously to get some idea of what Velocity can do and how it works and then move onto the next chapter.

CHAPTER 2

Getting Started with Velocity

Now that you're familiar with what Velocity is all about, it's time to get on with the show. In this chapter, I'll discuss how to obtain both Velocity and VelocityTools and how to configure the tools within your environment. Once you have Velocity up and running, I'll show some simple examples to demonstrate how to use Velocity.

Installing Velocity

Before you can start with any of the examples, you'll need to install Velocity on your machine. You can download the latest version of Velocity from the Web site at `http://jakarta.apache.org/velocity`. The download comes in two versions: the precompiled binary version and the source code version. I prefer to get the source code version and build from there, as it means that I have the source code if I need to do any debugging. You'll need the source code for some of the discussions in later chapters, but if you choose to download the binary version instead, you can skip to the "Creating Hello World" section. The version of Velocity I've used in this book is 1.4.

Building from Source Code

The first thing to do is obtain the correct source package for your platform. Since Velocity is 100 percent Java, there are no actual differences in the contents of the package; they're just compressed differently to suit both Windows and Unix users. Once you've downloaded the source archive, extract it to somewhere on your machine. I recommend you don't include any spaces in the path, since some machines may experience problems.

Before you proceed, make sure that the JAVA_HOME environment variable on your machine is pointing to the directory containing your Java SDK.

Installing Ant

The next step is to install Apache Ant. If you already have Apache Ant on your machine, make sure it's at least version 1.3; otherwise, you'll need to download

a later version. You can obtain the current version of Ant from `http://ant.apache.org`; for this example, I used version 1.5.4. For this part, I recommend you download the binary package unless you're familiar with Ant, since building Ant from the source is a little more complicated than building Velocity.

Once you've downloaded Ant, you need to extract the archive to a directory on your machine, again avoiding spaces in the path. Next, you need to set the ANT_HOME environment variable on your machine to point to the directory to which you extracted the Ant files. You should also add the Ant bin directory to your PATH environment variable so that you can run Ant easily from other directories. If you're unsure on how to set environment variables, you can find a sample in the Ant user manual at `http://ant.apache.org/manual/index.html`.

Verifying Ant

Once you've done that, you can verify that Ant is installed by typing **ant -help** at the command prompt. If Ant is installed correctly, you'll receive a message from Ant informing you of the command-line options available. If you don't get this message, then you should verify that you've completed the previous steps correctly. Failing that, check the Ant Web site for troubleshooting tips.

Building the Velocity Source Code

Once Ant is installed, building Velocity is a simple job. At the command prompt, switch to the directory where you extracted the Velocity source package, and from there switch into the build directory. Now simply type the following command:

```
ant
```

You should see a mass of Ant build messages scroll by until you get the `BUILD SUCCESSFUL` message. If you don't see any messages, make sure the `ant` command is accessible from your path. If you get a `BUILD FAILED` message, check that you're using an Ant version greater than 1.3. This will build the standard Velocity JAR file without any of the dependencies; you can build a JAR file with the dependencies included using the command `ant jar-dep`.

Once the build is complete, you'll find the Velocity JAR file in the Velocity home directory. You'll also find that all the JavaDoc documentation has been generated for the code as well, which you'll no doubt find useful as you use Velocity.

Building with J2EE Support

Velocity includes one class, `DataSourceResourceLoader`, which provides additional support in a J2EE environment. Chapter 10 covers using `DataSourceResourceLoader` in more detail. However, if you want to take advantage of this feature, you must

build Velocity with this class included. To do this, you need to copy the JAR file from your J2EE application server that contains the `javax.sql.DataSource` class into the `build/lib` directory inside your Velocity directory. Then you can build with Velocity with the `DataSourceResourceLocator` class using either `ant jar-J2EE` or `ant jar-J2EE-dep` to build without dependencies and with dependencies, respectively.

Testing the Velocity Build

Before moving on, it'd be wise to check that Velocity did indeed build correctly and that the build is functioning as the developers intended. Fortunately, the developers took the time to package the JUnit unit tests with the Velocity download. To run these tests, simply run the following command from the Velocity build directory:

```
ant test
```

Ant will now run your Velocity through a series of tests, which should result in a nice `BUILD SUCCESSFUL` message. If you get `BUILD FAILED`, this means one of your tests failed. You should try building the source again and running the tests again. If the problem still exists, try to build and test against the latest nightly build of Velocity available from the Web site. If still you have problems, it's most likely a configuration issue on your machine. Check the Velocity Web site for assistance.

Which JAR File?

As I mentioned previously, you can build two types of JAR files containing Velocity. One of the JAR files contains just the Velocity classes and nothing else; if you want to use Velocity, you'll have to add all the appropriate dependencies to your classpath as appropriate. The second JAR file, which has the word *dep* in the filename, contains all the dependencies that are needed by Velocity, so you can avoid having to find and download the correct versions yourself. For the most part, it's easier to use the JAR file containing all the dependencies, but if you want to use the version without the dependencies, then you'll need to download Jakarta Commons Collections and Jakarta ORO from `http://jakarta.apache.org`. You'll also need to have a logging tool; the "Configuring Velocity Logging" section covers this in more detail.

Building VelocityTools from Source Code

For many of the examples in this book that you'll build with Velocity, and indeed for most if not all of the applications, you'll need VelocityTools. As I mentioned

previously, the VelocityTools project overcomes some of the shortcomings in the Velocity runtime and makes your life that little bit easier. Rather than address all the features of VelocityTools here, I'll cover them as they become required by the examples.

Building VelocityTools isn't much different from building Velocity, given that it uses Ant to control the build. The main differences lie in the way the project is structured. To start with, you need to download the source code from the VelocityTools site at `http://jakarta.apache.org/velocity/tools/index.html`. The examples in this book are written from VelocityTools version 1.2.

Once you've downloaded the source code, extract it from the archive and put the resulting folder in an accessible location on your machine. From the command line, change to the directory where you placed the VelocityTools files. Unlike Velocity, there's no build directory, and the Ant build file resides in the main VelocityTools directory. To build the VelocityTools JAR files, simply run this command:

```
ant
```

Note that you don't need to specify a build target, since the default target will build the appropriate JAR files. By default, the build produces two JAR files: one containing all the tools and another containing everything but the Struts integration tools. You can also build a smaller JAR containing none of the Struts or servlet tools. To do this, simply run the following command from the command line:

```
ant jar.generic
```

Now you're ready to get started with Velocity.

About the Examples

Before I start with the examples, I'll add this a quick note about compiling and running the examples. As you're aware, compiling even a simple application with javac can be a nightmare, with many issues related to classpaths and JAR locations. I wrote all the examples using Eclipse, but I used Ant to build them all because it's so simple. If you're experiencing problems building any of the examples, you can download all the code and the Ant build script from the Downloads section of the Apress Web site (`http://www.apress.com`).

When you're running the examples, you need to make sure you have all the correct libraries on your classpath. If you're having problems running the examples, then it's most likely because of issues with the classpath.

Creating Hello World

It would be almost sacrilegious if I neglected the traditional Hello World example with Velocity. Getting Velocity to say "hello" to the world is quite simple. First, you need a template. A template is just a plain-text file containing VTL markup and static content. In this case, you have no dynamic content, so you just want to include static content, like so:

```
Hello World!
```

Save this file to a location that will be accessible from your code. I suggest you follow a directory structure similar to the sample code so the examples are easier to follow. With the template in place, you can move onto the Java code (see Listing 2-1).

Listing 2-1. The Hello World App

```
package com.apress.pjv.ch2;

import java.io.StringWriter;
import java.io.Writer;

import org.apache.velocity.Template;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

public class HelloWorld {

    public static void main(String[] args) throws Exception {
        // initialize Velocity
        Velocity.init();

        // get the template
        Template template =
            Velocity.getTemplate("src/templates/ch2/HelloWorld.vm");

        // create a context for Velocity
        VelocityContext context = new VelocityContext();

        // create the output
        Writer writer = new StringWriter();
        template.merge(context, writer);

        // write out
        System.out.println(writer.toString());
    }
}
```

Once you've completed the code, compile it and then run it. You'll see the following familiar message:

```
Hello World!
```

Now, you're probably thinking that Listing 2-1 contained an awful lot of code just to write "Hello World" to the console, and you'd be right. However, this is just a simplified example; as you move onto some more examples, you'll see how this code doesn't change much, but the amount of output can greatly increase.

Dissecting Hello World

So now that you've seen Velocity in action, you'll look at what you just wrote. First, you created the template. The Hello World template was basic and contained absolutely no VTL content whatsoever; it was simply the content you wanted to display on the screen. Next came the Java class, which was a little more complicated; you'll now look at it bit by bit:

```
// initialize Velocity
        Velocity.init();
```

The first line of code initializes the Velocity runtime with the default set of parameters. The `init()` method can be passed as either a `Properties` argument containing configuration properties for the Velocity runtime or a `String` argument that indicates the path to a file containing the configuration properties. The "Configuring the Velocity Runtime" section discusses the available configuration properties in more detail.

The next step is to obtain a reference to the actual template. Velocity has the following `Template` class to refer to templates:

```
// get the template
        Template template =
            Velocity.getTemplate("src/templates/ch2/HelloWorld.vm");
```

In this case, the template is loaded from the file system using `Velocity.getTemplate()`. With the template now loaded, the code now creates the following instance of the `VelocityContext` class:

```
// create a context for Velocity
VelocityContext context = new VelocityContext();
```

The `VelocityContext` class is pivotal to making your Velocity-based applications do anything useful. Using `VelocityContext`, you can pass data from your

Java program into the Velocity runtime when rendering output from your templates. You can see this in action in the next example:

```
// create the output
         Writer writer = new StringWriter();
         template.merge(context, writer);

         // write out
         System.out.println(writer.toString());
```

In the last step of this example, the code creates the output and writes it to the console. The `mergeTemplate` method will merge the data in the `VelocityContext` object, with the static content and VTL markup in the template, and write the result to the `Writer` instance provided by the code. In this case, the template contains no VTL markup, and the `VelocityContext` contains no data, so the output is simply the static content from the template file.

Improving Hello World

In the previous example, the code really did nothing spectacular. In other words, there was nothing dynamic about the output at all; all the code in the `main()` method could have been replaced with a single line, and the output wouldn't have been different.

The real power of Velocity comes when combining the static content in a template with some VTL constructs and some dynamic data from your Java application. Quite surprisingly, it takes hardly any extra code to give the Hello World application some dynamic behavior. First, you need to modify the template, adding some VTL markup, like so:

```
Hello $who!
```

Don't worry too much about the syntax just yet; Chapter 3 explains VTL in full, so the behavior of this particular piece of code will become apparent soon. Although I've created a separate class, the changes to the code are minimal (see Listing 2-2).

Listing 2-2. The Improved Hello World App

```
package com.apress.pjv.ch2;

import java.io.StringWriter;
import java.io.Writer;
```

```
import org.apache.velocity.Template;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

public class HelloWorldImproved {

    public static void main(String[] args) throws Exception{
        //initialize Velocity
        Velocity.init();

        // get the template
        Template template =
            Velocity.getTemplate("src/templates/ch2/HelloWorldImproved.vm");

        // create a context for Velocity
        VelocityContext context = new VelocityContext();

        // set the who variable
        context.put("who", "Gandalf");

        // create the output
        Writer writer = new StringWriter();
        template.merge(context, writer);

        // write out
        System.out.println(writer.toString());
    }
}
```

Notice that the only differences are the name of the template file and the addition of this extra line:

```
// set the who variable
        context.put("who", "Gandalf");
```

The `VelocityContext` class allows you to pass data from your application into the Velocity runtime. `VelocityContext` works on a name/value basis, with the name you provide when adding data to the `VelocityContext` class being used to refer to that data from within your template. If you recall the change I made to the template, I replaced the word *World* with the VTL variable `$who`. Then in the code I set the value of the `$who` variable to Gandalf. When you run the example now, you'll get the following output:

```
Hello Gandalf!
```

Try changing the value of the `$who` variable from within your Java code, and verify that the output changes appropriately.

Using Context Chaining

With Velocity it's possible for multiple contexts to be merged and passed to the runtime as a single context. This can help when many components are interacting to produce the context; rather than pass a single context from component to component, which will have performance implications if the context is particularly large and it's being passed across the network and back, each component can create its own context, and then the main application can chain the contexts together. To see this in action, you'll now build a simple example. First, create a new template with the following code:

```
This is my first name $firstName
This is my last name $lastName
```

Second, create a Java class to create some output from this template. Listing 2-3 shows the code in full; the lines of interest are explained afterward.

Listing 2-3. Context Chaining

```
package com.apress.pjv.ch2;

import java.io.StringWriter;
import java.io.Writer;

import org.apache.velocity.Template;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

public class ContextChaining {

    public static void main(String[] args) throws Exception {
        // initialize Velocity
        Velocity.init();

        // get the template
        Template template =
            Velocity.getTemplate("src/templates/ch2/ContextChaining.vm");

        // create two separate contexts
        VelocityContext context1 = new VelocityContext();
        VelocityContext context2 = new VelocityContext(context1);

        // set the firstName variable
        context1.put("firstName", "Rob");
        context2.put("lastName", "Harrop");
```

```
        // create the output
        Writer writer = new StringWriter();
        template.merge(context2, writer);

        // write out
        System.out.println(writer.toString());
    }
}
```

Notice that when you're creating the `context2` object, you pass the `context1` object as an argument as the constructor, like so:

```
// create two separate contexts
VelocityContext context1 = new VelocityContext();
VelocityContext context2 = new VelocityContext(context1);
```

This will place the data in `context1` inside `context2`. Note, however, that the data isn't simply copied from one context to the other; the second context, `context2`, maintains a reference to `context1` and will check this context when trying to resolve references within a template. For this reason, you're able to add data to `context1` after the constructor on `context2` has finished executing and still have the data accessible via `context1`. Running this code will yield the following output:

```
This is my first name Rob
This is my last name Harrop
```

As you can see, both variables are resolved correctly, and the correct data is rendered, even though only the `context2` object was passed to the Velocity runtime directly. When trying to resolve the `$firstName` reference, the `context2` object has delegated to the chained `context1`, once it determined that it didn't contain the data. For the most part, you won't use this feature in your day-to-day programming—you'll simply create a context, populate it, and then generate content from it. However, it can be useful if you're building complex applications with many different components because it helps decouple the components from each other. Context chaining also becomes useful when you build framework services for your own application—perhaps you want guarantee that a certain piece of data always exists in the context—and then use context chaining to allow your application to provide its own context and have the framework chain the extra data on top.

Configuring the Velocity Runtime

The Velocity runtime is highly configurable and has a large selection of configuration properties that you can use to customize the behavior of the runtime.

Understanding how to configure the Velocity runtime correctly will help when building more complex applications.

Setting Configuration Parameters

You have two ways to set the Velocity configuration parameters when first initializing the Velocity runtime. The first way is simply to create a `Properties` instance from within your code and then pass this instance to the `Velocity.init()` method (see Listing 2-4).

Listing 2-4. Configuration Using the `Properties` *Class*

```
public class HelloWorldProperties {

    public static void main(String[] args) throws Exception {
        // set some properties
        Properties props = new Properties();
        props.put("input.encoding", "utf-8");

        // initialize Velocity
        Velocity.init(props);

        // get the template
        Template template =
            Velocity.getTemplate("src/templates/ch2/HelloWorld.vm");

        // create a context for Velocity
        VelocityContext context = new VelocityContext();

        // create the output
        Writer writer = new StringWriter();
        template.merge(context, writer);

        // write out
        System.out.println(writer.toString());
    }
}
```

The main drawback of this method is obvious; any changes to the configuration require a change to your Java code, which in a large application means a recompile and no doubt a bunch of testing to make sure something didn't go wrong in the process. A better solution to this is to use an external properties file and pass the name of the properties file to the Velocity runtime when you call `init()`. I'm sure most of you are familiar with the format of Java property files, but just in case, you can achieve the same behavior as the previous example with this property file:

```
# set the input encoding
input.encoding=utf-8
```

Then, to use this property file to configure the Velocity runtime, you just pass the filename to the `Velocity.init()` method (see Listing 2-5).

Listing 2-5. Configuration Using a Properties File

```
public class HelloWorldExternalProperties {

    public static void main(String[] args) throws Exception{
        // initialize Velocity
        Velocity.init("src/velocity.properties");

        // get the template
        Template template =
            Velocity.getTemplate("src/templates/ch2/HelloWorld.vm");

        // create a context for Velocity
        VelocityContext context = new VelocityContext();

        // create the output
        Writer writer = new StringWriter();
        template.merge(context, writer);

        // write out
        System.out.println(writer.toString());
    }
}
```

Unless you plan to have some kind of dynamic configuration for the Velocity runtime that can be modified through some user interface in your application, I recommend you use an external properties file for configuration. That way, you can make any changes to the configuration quickly and without touching your application code.

Introducing General Configuration Parameters

You can use some general configuration parameters to configure the behavior of the Velocity runtime (see Table 2-1).

Table 2-1. General Velocity Configuration Parameters

Parameter Name	Description
`input.encoding`	This allows you to specify the encoding of your input data. By default, this is set to ISO-8859-1. Change this only if you have templates that aren't ISO-8859-1 encoded.
`output.encoding`	This property is used by the VelocityServlet to set the response encoding, and by Anakia when writing output files. The default is ISO-8859-1.
`parser.pool.size`	The Velocity runtime maintains a pool of parsers that parse your templates. If you have enough memory available on your machine and your application is parsing a lot of templates concurrently, you should consider increasing the size of the pool. By default, the pool size is set to 20.
`runtime.interpolate.literals`	The value of this property determines whether the Velocity runtime will interpolate string literals that are contained within a template. Chapter 3 includes a more detailed discussion of the effect of this parameter.
`runtime.introspector.uberspect`	Velocity heavily uses introspection, mainly to provide a simple, JavaBeans-based syntax for accessing properties on your Java objects. The actual introspection is pluggable, so you can replace the default implementation with one of your own. Bear in mind that Velocity already has an excellent implementation that has been improved even more in version 1.4. You should use this only in special cases. The value of this property should be set to the fully qualified class name of the introspector; the default is `org.apache.velocity.util.introspection.UberspectImpl`.

Configuring Velocity Logging

As you start to build more complex applications using Velocity, you're bound to come across the need to debug your templates. Often you'll find that the output isn't as you expected or that output isn't generated at all and you get an exception. The logging mechanism is useful when you're experiencing problems with Velocity locating templates, which becomes even more complicated when you start to use multiple resource loaders. Thankfully, Velocity has quite a good logging system, which outputs some detailed information about the parsing and execution of your template. The information in the logs is often sufficient for you to find out what's wrong with your templates and fix it.

If you've been running the examples up to this point, you may have noticed the log. By default, Velocity will look for the Avalon LogKit on your classpath and use this if it's found. If Avalon LogKit isn't found, then Velocity will look for log4j and use that. You'll find that Avalon LogKit is included in the dependencies JAR file that's created when you build Velocity, so if, like me, you've used this dependency JAR to build the examples, your logs will have been written using Avalon LogKit. The default output path for the log is from the root of your execution path to a file named `velocity.log`.

Introducing Velocity LogSystem

At the core of the Velocity logging system is the `LogSystem` interface. All log messages from the Velocity runtime are written to an object that implements this interface. The standard distribution of Velocity contains implementations of `LogSystem` for Avalon LogKit and Apache log4j, along with an implementation to ignore all log messages. As well as these implementations, you'll also find an implementation of `LogSystem` that will write log messages using the Jakarta Common Logging system included in the VelocityTools project. If you don't see your favorite logging tool listed here, don't worry; it's a trivial job to create a custom implementation of `LogSystem` and have Velocity use that as its logger. Chapter 10 covers this topic in more detail.

Configuring Velocity to Use log4j

As I mentioned earlier, the Velocity runtime will, by default, use Avalon LogKit as the logging implementation when writing any log messages. If Avalon LogKit isn't found on the classpath, Velocity will then look for log4j and use that, if found. This is all well and good when you're using the simple Velocity JAR file without all the dependencies, as you can choose to have just log4j on classpath. However, when using the JAR file containing the dependencies, Avalon LogKit is included in that JAR so Velocity will always pick it up before log4j. Fortunately, you can tell Velocity to ignore that Avalon LogKit is available and use the log4j logger by setting the `runtime.log.logsystem.class` configuration parameter to the class name of the log4j `LogSystem` implementation (in this case, `org.apache.velocity.runtime.log.SimpleLog4JLogSystem`). You can set this parameter either by passing it to `Velocity.init()` method as described earlier or by placing it in an external properties file as described in the previous example.

NOTE *Another implementation of* `LogSystem` *for log4j is included with the Velocity distribution:* `Log4JLogSystem`*. This class has been deprecated since version 1.3 of Velocity and shouldn't be used. You should also be aware that the current log4j support in Velocity uses the older concept of a* `Category`*, which has been replaced in log4j by the* `Logger` *class. The* `Category` *class still works in log4j, and version 1.5 of Velocity uses the log4j* `LogSystem` *implementation with* `Logger` *instead of* `Category`*.*

So, you'll now try it out. Using the previous example, when you first run it without any special configuration parameters, the first line of the log file reads as follows:

```
Fri Feb 13 10:43:10 GMT 2004  [debug] AvalonLogSystem initialized using logfile
'velocity.log'
```

As you can see, Velocity is clearly using Avalon LogKit as the logging implementation. Now to switch to log4j, you simply add the following line of code to the configuration file and run this:

```
runtime.log.logsystem.class=org.apache.velocity.runtime.log.SimpleLog4JLogSystem
```

Now when you look into the log file, you'll see that the first line of log messages written is this:

```
2004-02-13 10:45:21,125 - SimpleLog4JLogSystem initialized using logfile ➥
'velocity.log'
```

Now you can see that log4j log system is being used to write the log messages. By default the `SimpleLog4JLogSystem` will use its own class name as the name of the log4j `Category`. However, you can change this by setting the `runtime.log.logsystem.log4j.category` property to the name of the category you want to use. Once you're using log4j to log the Velocity messages, you can configure the output of the messages just as you would with any log4j application. For instance, to output all the Velocity messages to the console, create a simple `log4j.properties` file with the following code and place it in the root of your classpath:

```
log4j.rootLogger=DEBUG, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

# Pattern to output the caller's filename and line number.
log4j.appender.stdout.layout.ConversionPattern=%p [%t] [%c] %C{1}.%M(%L) | %m%n
```

Be aware that it seems that the configuration file will be picked up by log4j and Velocity only if you set your own `Category` name as described. I found that when I had this configuration file in the classpath and I passed Velocity a specific `Category` name to use, the log message went to the console as desired. However, when I didn't pass Velocity the `Category` name, no log messages were written to the console. For more information on log4j configuration, read *Logging in Java with the JDK 1.4 Logging API and Apache log4j* (Apress, 2003).

Using Commons Logging with Velocity

As part of the VelocityTools project, a `LogSystem` implementation will allow you to write all log output via the Commons Logging tool. If you intend to use log4j as the actual logging implementation, I advise that you use the `SimpleLog4JLogSystem` and resist the temptation to pass log messages from Velocity to Commons Logging and then onto log4j. However, if you want to use a logging provider that has no LogSystem implementation and you don't fancy writing your own, then Commons Logging may well have an adapter. A good example of this would be if you wanted to use Java 1.4 Logging as the mechanism for writing your log messages.

For this example, I used the code from the `HelloWorldExternalProperties` example. No modifications to the Java code were needed; only the configuration file was changed. To use Commons Logging, first you need to download it from the Jakarta Commons Web site at `http://jakarta.apache.org/commons`. Once you've downloaded it, include the JAR file in the classpath of your application. You'll also need to include the VelocityTools JAR file, as this contains the `LogSystem` implementation for Commons Logging. Once that's done, you need to set the `runtime.log.logsystem.class` property to `org.apache.velocity.tools.generic.log.CommonsLogLogSystem`. If you still have log4j in your classpath and configured correctly, then you can run the example now, but you'll still be using log4j as the underlying logging provider. To switch to the JDK 1.4 Logging API as the logging mechanism, simply remove log4j from the classpath and, of course, make sure you're running on a 1.4 JVM. Now run the example. You'll see that the Velocity log file shows that `CommonsLogLogSystem` is being used to write the log messages, and you should also see that the log messages are written to the console. This is the default JDK 1.4 logging setup provided to Velocity via Commons Logging. Phew! So many different adapters—it can get quite complicated. For the most part, I recommend you stick with log4j as your logger and that you use `SimpleLog4JLogSystem` to interface it with Velocity. Log4j is a powerful logging tool, and using `SimpleLog4JLogSystem` provides the simplest and best-performing mechanism to link it with Velocity. For most applications, I recommend you use Commons Logging at code level but use log4j as the actual logging tool. This way, you can easily replace log4j with another logging tool for both your application and Velocity log messages without having to rework your code. For a more

in-depth discussion of using Commons Logging, you should check out the Jakarta Web site and also read my discussion of Commons Logging in *Pro Jakarta Struts*, Second Edition (Apress, 2004).

Additional Logging Configuration

Besides just configuring which tool is going to perform the actual logging, you can use a few additional configuration parameters to customize the logging behavior (see Table 2-2).

Table 2-2. Additional Logging Configuration Parameters

Parameter	Description
`runtime.log`	Using this parameter, you can specify the name of the log file created by Velocity. By default, this parameter is set to `velocity.log`, and the file is created in the root of the execution path.
`runtime.log.logsystem`	You can use this parameter only from code; you can't use it from the configuration file. Using this parameter, you can pass an instance of a class that implements `LogSystem` to Velocity and have it use that class to write the log messages. This can be quite useful if you want to be able to build the logging configuration dynamically in your code.
`runtime.log.xxx.stacktrace`	Using this parameter, you can specify whether Velocity should output stack traces in the log files for the error, warn, and info message levels. By default, this option is set to `false` for each level. To turn stack traces on, set this parameter to `true`, replacing `xxx` with the level name.
`runtime.log.invalid.references`	By default, Velocity will write a log message each time it finds a reference in one of your templates that isn't valid. This is one of my main uses of the log file when I'm debugging my applications, but it's useful to turn this off in a production. Set this option to `false` to disable invalid reference logging.

Understanding Resource Loaders

In each of the previous examples, the templates have been loaded from a file, and you'd be forgiven for thinking this is the only way to load a template for Velocity. This is, however, not the case. Velocity has a flexible resource management system

that allows for resources to be retrieved from a wide variety of sources. Templates are the most common resource loaded by Velocity, but as you'll see when you look at the `#include` directive in Chapter 3, they aren't the only resources loaded by Velocity.

Central to the resource management system in Velocity are *resource loaders*. Resource loaders are responsible for retrieving the resource from a specific location and passing the contents to the Velocity runtime. Consider the examples you've seen. All of them include a call to `Velocity.getTemplate()`, which loads a template resource into the Velocity runtime. At first glance, you'd think that the `String` argument the `getTemplate()` method takes is actually the path to the template, but in fact it's just an identifier that's used by the resource loader to retrieve a particular resource. In some cases, this resource can be a filename; in others, it may just be a designated identifier for your templates. The Velocity distribution comes complete with four resource loaders for loading resources: from the file system directly, from a JAR file, from the classpath, and from a J2EE `DataSource`. In the next section, I'll demonstrate how to use the first three and also show configuration options available for the fourth.

Using Multiple Resource Loaders

For this section, I've built an example to test three of the four supplied resource loaders. Rather than talk you through entering the code bit by bit, I'll include it all here and then show the relevant bits for each resource loader.

The Java code for this example should seem quite familiar; I've just refactored it a little to make it simpler (see Listing 2-6).

Listing 2-6. Using Resource Loaders

```
package com.apress.pjv.ch2;

import java.io.StringWriter;
import java.io.Writer;

import org.apache.velocity.Template;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.Velocity;

public class ResourceLoaders {

    private static final String FILE_RESOURCE_LOADER_TEMPLATE =
        "ResourceLoader1.vm";
    private static final String JAR_RESOURCE_LOADER_TEMPLATE =
        "ResourceLoader2.vm";
    private static final String CLASSPATH_RESOURCE_LOADER_TEMPLATE =
        "ResourceLoader3.vm";
```

```
    public static void main(String[] args) throws Exception {
        processTemplate(FILE_RESOURCE_LOADER_TEMPLATE);
        processTemplate(JAR_RESOURCE_LOADER_TEMPLATE);
        processTemplate(CLASSPATH_RESOURCE_LOADER_TEMPLATE);
    }

    private static void processTemplate(String templateName) throws Exception {
        // initialize Velocity
        Velocity.init("src/velocity.properties");

        // get the template
        Template template = Velocity.getTemplate(templateName);

        // create a context for Velocity
        VelocityContext context = new VelocityContext();

        // create the output
        Writer writer = new StringWriter();
        template.merge(context, writer);

        // write out
        System.out.println(writer.toString());
    }

}
```

I've created three templates: `ResourceLoader1.vm`, `ResourceLoader2.vm`, and `ResouceLoader3.vm`, each with a single line of static text. I've placed the first template, `ResourceLoader1.vm`, in the same directory as all the templates from the previous examples. I've packaged the second template, `ResourceLoader2.vm`, inside a JAR file and then placed it in the same directory as `ResourceLoader1.vm`. I've stored the third template, `ResourceLoader3.vm`, in the root of the classpath. The placement of these files is important; if you place all the templates in the same directory as `ResourceLoader1.vm`, they will be picked up by the same resource loader, and you won't be able to see the others in action.

Testing FileResourceLoader

The first resource loader to test is the file resource loader, which is implemented by the `FileResourceLoader` class in the `org.apache.velocity.runtime.resource.loader` package. By default, Velocity will use an instance of `FileResourceLoader` configured to look in the current directory when searching for resources. So, in

all the previous examples, the FileResourceLoader has been used to load the templates based on the path specified in the call to Velocity.getTemplate(). If you look closely at the current example, you'll notice that I've specified just the filename for ResourceLoader1.vm, not the whole path. If you ran the example without any further configuration, Velocity would be unable to find this template because it doesn't reside in the current directory. However, you can configure a different resource loader than the default and have Velocity look for my resources there. To do this, first you need to tell Velocity the name of my resource loader and then, using this name, configure the options for it. You can do this with a few lines in the velocity.properties file:

```
resource.loader=file

# File Resource Loader
file.resource.loader.class= ➥
org.apache.velocity.runtime.resource.loader.FileResourceLoader
file.resource.loader.path=./src/templates/ch2
file.resource.loader.cache=true
file.resource.loader.modificationCheckInterval=2
```

Note that you can also pass these options to Velocity inside an instance of the Properties class. The resource.loader option specifies the name of the resource loader that Velocity should find. By default, Velocity will look for a resource loader named file, but I prefer to define the name explicitly—if only because it makes my intentions clearer. To configure a particular resource loader, you need to include the configuration options prefixed by the name given to the resource.loader option. As you can see from the previous code, the four configuration parameters after the resource.loader parameter are prefixed with the name file. This means that these parameters are being set for just the resource loader named file. This is useful when you have more than one resource loader working in your application.

If you run the example now, you'll see output for the first template, but you should get an error for the second template, as Velocity won't be able to find the appropriate resource until you configure the JAR resource loader. It's useful to check the Velocity log file, as it will tell what resource loaders it's using and which one was used when a particular resource was loaded.

Table 2-3 describes the configuration options for the FileResourceLoader.

Table 2-3. FileResourceLoader *Configuration Parameters*

Parameter	Description
<name>.resource.loader.class	This specifies the name of the implementation class of the resource loader. For the FileResourceLoader, this will be org.apache.velocity.runtime.resource.loader.FileResourceLoader. This parameter is always required.

Table 2-3. `FileResourceLoader` *Configuration Parameters (continued)*

Parameter	Description
`<name>.resource.loader.path`	Use this parameter to specify the path in which the `FileResourceLoader` should look for resources. You can specify more than one path, each one separated by a comma.
`<name>.resource.loader.cache`	By default, this parameter is set to `false`, which means each time you want a resource from this resource loader, it will be reloaded from the filesystem. This is quite useful when debugging, since you're likely to change your templates quite a bit. However, in a production environment, you can really improve performance by setting this parameter to `true`, in which case the resource loader will place resources in a cache once they've been loaded and then retrieve them from the cache for subsequent requests.
`<name>.resource.loader .modificationCheckInterval`	This parameter is used only if you turn on caching. Use this parameter to specify the interval, in seconds, between checks to see if a cache resource has been modified. If you're in a production environment and want the performance benefit of caching but don't want the hassle of having to restart your application every time you change a resource, then set this parameter to the highest interval you can manage.

Testing JarResourceLoader

If you ran the example after the previous step, you'll no doubt have received an error message telling you that Velocity couldn't find `ResourceLoader2.vm`. This is no wonder, really, when you consider that this resource is actually packaged inside a JAR file. For this you need to use the `JarResourceLoader`, which will load resources that are packaged within a JAR file.

Configuring the `JarResourceLoader` isn't much different from configuring the `FileResourceLoader`. First, you have to add another resource loader name to the `resource.loader` parameter; second, you declare the configuration parameters for this resource loader, like so:

```
resource.loader=  file, jar

# File Resource Loader
...

# JAR Resource Loader
jar.resource.loader.class = ➥
org.apache.velocity.runtime.resource.loader.JarResourceLoader
jar.resource.loader.path = jar:file:/tmp/ResourceLoader2.jar
```

Notice that I've specified only two parameters for this resource loader; Table 2-4 describes the full list of parameters recognized by the `JarResourceLoader`. Run the example again, and this time you should get output from both the first and second templates, with only the third throwing up an error.

Table 2-4. `JarResourceLoader` *Configuration Parameters*

Parameter	Description
`<name>.resource.loader.class`	This parameter has the same meaning as for the `FileResourceLoader` and is always required. For the `JarResourceLoader`, this will be `org.apache.velocity.runtime.resource.loader.JarResourceLoader`.
`<name>.resource.loader.path`	Use this parameter to specify the paths to the JAR files that the `JarResourceLoader` will look in for resources. As with `FileResourceLoader`, you can specify more than one path using commas to separate each path. The syntax of the paths should correspond with that defined by the `java.net.JarURLConnection` class. See the JavaDoc for that class for more details.
`<name>.resource.loader.cache`	Although you can turn on caching for the `JarResourceLoader`, support for caching in this loader isn't yet fully implemented. The `JarResourceLoader` will always inform the runtime that a particular resource has changed, regardless of whether this is the case. You should note, however, that the resource loader will have the opportunity only to notify the runtime that the resource has been modified after the amount of seconds set in `modificationCheckInterval` has elapsed. For instance, consider an application where you've set `modificationCheckInterval` to 20 seconds. You run the application and the template is processed once, being loaded from the JAR file. Then, 19 seconds later the template is processed again, being retrieved from the cache since the `modificationCheckInterval` hasn't yet elapsed. Then the template is processed again in 30 seconds. This time, the runtime will check to see if the resource has been modified since the amount of time specified by `modificationCheckInterval` has elapsed. Now, regardless of whether the resource has been modified, it will be reloaded.
`<name>.resource.loader.modificationCheckInterval`	As with `FileResourceLoader`, this is the amount of time, in seconds, that the runtime will wait before checking to see if the resource has changed. See the previous discussion of the cache parameter for more details.

Testing ClasspathResourceLoader

All that's left is to configure Velocity to find the third template. As you may have noticed, I got you to place this template in the root of the classpath. The reason for this is so that I can highlight the use of the `ClasspathResourceLoader`. This resource loader is the easiest to configure and is perhaps the most useful in a Web-based environment, as you have no issues with paths; all the resources are simply included in the classpath. You should note that you don't have to place your resources loose at the root of your classpath; you can put them in a JAR file and place the JAR file in your classpath. This is useful in a servlet environment since you can place all your resources in a JAR file and pop that in the `/WEB-INF/lib` directory, and your resources are all available through the classpath.

I mentioned that the `ClasspathResourceLoader` was easy to configure, and it is. You don't have any paths to worry about, just the following class name:

```
resource.loader=  file, jar, class

# File Resource Loader
...

# JAR Resource Loader
...

# Classpath Resource Loader
class.resource.loader.class =
org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader
```

Now, run the example again, and you should get output from all three templates, with no errors at all. Table 2-5 describes the configuration parameters for this resource loader.

Table 2-5. `ClasspathResourceLoader` *Configuration Parameters*

Parameter	Description
`<name>.resource.loader.class`	This parameter has the same meaning as for the `FileResourceLoader` and is always required. For the `ClasspathResourceLoader`, this will be `org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader`.
`<name>.resource.loader.cache`	As with `FileResourceLoader`, you can enable caching for resources loaded by the `ClasspathResourceLoader` by setting this parameter to `true`. You should note that this resource loader doesn't support modification detection; it will always inform the runtime that the resource hasn't been changed, even if you change it. This can be quite infuriating during debugging, since you need to restart your application just to check for changes in your template.

Table 2-5. `ClasspathResourceLoader` *Configuration Parameters (continued)*

Parameter	Description
`<name>.resource.loader.modificationCheckInterval`	This parameter, although supported for this resource loader, is kind of pointless, since if you have caching enabled, the resource will never be reloaded despite any modifications you may make to it.

Configuring DataSourceResourceLoader

Before proceeding with this section, you should refer to the earlier "Building with J2EE Support" section, which discusses building Velocity with support for the `DataSourceResourceLoader` class.

Using `DataSourceResourceLoader`, you can store your Velocity resources in any data store that can be accessed via a `javax.sql.DataSource` instance. Using this resource loader is restricted to J2EE environment. I've purposely avoided going over the details of configuring a `DataSource` in your application server, since I think that if you want to use this feature, then you're already more than familiar with that process. I will, however, show you how to set up Velocity so that you can use the `DataSourceResourceLoader` in your J2EE applications.

The first step is to create a table in your data store that will store the Velocity resources. This table must include at least three columns: one to store the resource name, one to store the resource content, and one to act as a time stamp. You'll find a sample SQL script included in the JavaDoc for the `DataSourceResourceLoader` class.

Next, you need to configure the resource loader in Velocity. You do this just like any other resource loader; however, you have to specify many more parameters. This is a sample configuration for the `DataSourceResourceLoader`:

```
 ds.resource.loader.class
org.apache.velocity.runtime.resource.loader.DataSourceResourceLoader
 ds.resource.loader.resource.datasource = java:comp/env/jdbc/Velocity
 ds.resource.loader.resource.table = velocityResources
 ds.resource.loader.resource.keycolumn =  resourceKey
 ds.resource.loader.resource.templatecolumn = resourceDef
 ds.resource.loader.resource.timestampcolumn = resourceTimestamp
 ds.resource.loader.cache=true
 ds.resource.loader.modificationCheckInterval = 60
```

Table 2-6 gives a full description of each parameter.

Table 2-6. `DataSourceResourceLoader` *Configuration Parameters*

Parameter	Description
`<name>.resource.loader.class`	As with all the previous resource loaders, you should set this to the name of the resource loader class (in this case, `DataSourceResourceLoader`).
`<name>.resource.loader.resource.datasource`	This is the full JNDI name of the data source. You must configure the data source separately within your application server.
`<name>.resource.loader.resource.table`	This is the name of the table in the data store that contains your resources.
`<name>.resource.loader.resource.keycolumn`	This is the name of the column in your table that contains the resource name. This is the argument you pass to `Velocity.getTemplate()` when you want to load a template.
`<name>.resource.loader.resource.templatecolumn`	This is the name of the column that contains the actual resource contents.
`<name>.resource.loader.resource.timestampcolumn`	This is the name of the column containing a time stamp. This is used for caching purposes.
`<name>.resource.loader.cache`	This resource loader fully supports caching, using a time stamp stored alongside a resource in the data store. Set this parameter to `true` to enable caching.
`<name>.resource.loader.modificationCheckInterval`	This one should be familiar—it's the amount of time the Velocity runtime will wait before checking to see if a resource has been modified. Bear in mind that this will involve queries being run against the data store.

Configuring the Resource Manager

You can use three parameters to control the overall behavior of the Velocity resource manager (see Table 2-7).

Table 2-7. Resource Manager Configuration Parameters

Parameter	Description
`resource.manager.logwhenfound`	By default, this parameter is set to `true`, which causes the resource manager to write a log message the first time it finds a resource. You can set this to `false` in a production environment, but it's useful in a debugging environment.

Table 2-7. Resource Manager Configuration Parameters (continued)

Parameter	Description
`resource.manager.cache.class`	Using this parameter, you can specify the name of class that will be used to cache resources once they're loaded. Chapter 10 covers how to create a custom cache implementation. The default for this parameter is `org.apache.velocity.runtime.resource.ResourceCacheImpl`. This class uses a least recently used caching algorithm to remove old items from memory, keeping memory consumption at a sensible level.
`resource.manager.cache.size`	Use this parameter to set the default cache size. When using the default cache, this parameter is set to 89.

Choosing a Resource Loader

You have quite a few choices available for resource loaders; as you'll see in Chapter 10, you can even implement your own resource loaders. However, for the most part you should stick with either `FileResourceLoader` or `ClasspathResourceLoader` because that way you can easily modify the template contents; when you have your templates packaged away in a JAR file or stored in a database, editing the template isn't as easy as just modifying a file in the file system. This is especially true if the templates are used for output created by nonprogrammers. Non-programmers already have to learn the Velocity syntax; you don't really want to start having to show them how to create and extract JAR files or interact with your RDBMS.

The lack of caching on the `JarResourceLoader` makes it a bad choice for production environments, as you can get quite a performance boost from the effective use of caching. I imagine that caching will eventually be added to the `JarResourceLoader`, but my guess is that it will detect only when the entire JAR file has changed, which means that changing one template in the file will invalidate the cached versions of any other templates in the JAR file, whether or not they've changed.

In a Web environment, I almost certainly recommend you use the `ClasspathResourceLoader`; it'll remove the need to worry about paths and will ensure that your application will still function when deployed as a WAR file in containers such as JBoss, which run the application directly from the WAR without extracting the files. One thing to be aware of in `ClasspathResourceLoader` is that if you have caching enabled, once a template has been loaded and placed in the cache, it'll always be retrieved from the cache, irrespective of whether the resource has changed. This is different from `JarResourceLoader`, which will always tell the runtime that the resource has been modified; `ClasspathResourceLoader` always tells the runtime that the resource has *not* been modified.

Configuring VTL

Using configuration, it's possible to modify the way in which certain VTL constructs work. I've saved the discussion of these configuration options until Chapter 3 where they're discussed in the context of the corresponding VTL constructs.

Configuring Velocimacro Libraries

Velocimacros are an advanced feature of VTL that are discussed in full in Chapter 3. Essentially, Velocimacros allow you to build reusable snippets of VTL code that you can use multiple times throughout a template. Velocity allows Velocimacros to be grouped together and stored globally in a library—Chapter 4 covers this topic in more detail.

Using VelocityEngine

In the previous examples, all interaction with the Velocity runtime has been through the Velocity helper class. Using this class, you're accessing a single instance of the Velocity runtime, which in most cases is perfectly adequate. However, as the needs of your applications increase, you may find that you need to use two instances of the Velocity runtime that are configured differently, within the same JVM. Fortunately, this capability has been present in Velocity since version 1.2, in the `VelocityEngine` class.

Using the `VelocityEngine` class is very much like using the `Velocity` class, except you're dealing with methods on a specific instance rather than static methods on a class. You still initialize the runtime with a call to the `init()` method, still get an instance of a template using the `getTemplate()` method, and still generate the final output with a call to `mergeTemplate()`. So, what's the point? Well, as I mentioned previously, you may want to have multiple instances of the runtime within your application, each with its own configuration. You may not have many uses for this in a generic Web application, but what if you're building a framework or a tool that may be embedded inside an application that's using Velocity? If your tool uses the `Velocity` helper class, it'll be sharing its configuration with the containing application, and nothing is stopping that application from modifying the configuration to such an extent that your tool will stop functioning. By using the `VelocityEngine` class, your tool can have its own isolated instance of the Velocity runtime that will not be affected by the configuration of any other Velocity runtimes in the containing application.

So, that's the theory; you'll now see this in action. In this example, I have two instances of the Velocity runtime that generate output from a template. Both instances use the same template name when obtaining an instance of the `Template` class, but they both have differently configured resource loaders, which will load the templates from different locations.

The first thing to do is to create two templates with the same name, but in different directories. Give each template different content so you'll be able identify the output from each template. My first template looks like this:

```
# template 1
This is template number 1
```

And the second one looks like this:

```
# template 2
This is template number 2
```

Next is the Java code to create the output from these templates. I want to create two instances of the Velocity runtime, each with a different resource loader configured. In this way, I can point each resource loader at one of the directories containing the templates, so even though the two instances will be looking for a template with same name, they'll be looking in different directories and thus retrieve different templates. Listing 2-7 shows the code in full; I'll explain each part in detail afterward.

Listing 2-7. Using `VelocityEngine`

```
package com.apress.pjv.ch2;

import java.io.StringWriter;
import java.io.Writer;
import java.util.Properties;

import org.apache.velocity.Template;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.VelocityEngine;

public class VelocityEngineTest {

    private static final String TEMPLATE_NAME = "VelocityEngineTest.vm";

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

        // create the properties for each engine
        Properties p1 = new Properties();
        p1.put("resource.loader", "file");
        p1.put(
            "file.resource.loader.class",
            "org.apache.velocity.runtime.resource.loader.FileResourceLoader");
        p1.put("file.resource.loader.path", "src/templates/ch2/ve1");
```

```
        Properties p2 = new Properties();
        p2.put("resource.loader", "file");
        p2.put(
            "file.resource.loader.class",
            "org.apache.velocity.runtime.resource.loader.FileResourceLoader");
        p2.put("file.resource.loader.path", "src/templates/ch2/ve2");

        //now create the engines
        VelocityEngine ve1 = new VelocityEngine();
        ve1.init(p1);

        VelocityEngine ve2 = new VelocityEngine();
        ve2.init(p2);

        // now get the output for each engine
        writeTemplateOutput(ve1);
        writeTemplateOutput(ve2);

    }

    public static void writeTemplateOutput(VelocityEngine ve) throws Exception {
        Template t = ve.getTemplate(TEMPLATE_NAME);

        Writer writer = new StringWriter();
        t.merge(new VelocityContext(), writer);

        System.out.println(writer.toString());
    }
}
```

The first part of the code is building the different configurations for the two runtime instances. Here, I've created the configuration in code rather than using property files so that the example is a bit simpler:

```
// create the properties for each engine
        Properties p1 = new Properties();
        p1.put("resource.loader", "file");
        p1.put(
            "file.resource.loader.class",
            "org.apache.velocity.runtime.resource.loader.FileResourceLoader");
        p1.put("file.resource.loader.path", "src/templates/ch2/ve1");

        Properties p2 = new Properties();
        p2.put("resource.loader", "file");
        p2.put(
            "file.resource.loader.class",
            "org.apache.velocity.runtime.resource.loader.FileResourceLoader");
        p2.put("file.resource.loader.path", "src/templates/ch2/ve2");
```

Notice that the only difference is the path in which the resource loader will search for resources. The names of the resource loaders are the same, since these two resource loaders won't exist in the same runtime instance. Next, I create the two runtime instances using the `VelocityEngine` class, like so:

```
//now create the engines
        VelocityEngine ve1 = new VelocityEngine();
        ve1.init(p1);

        VelocityEngine ve2 = new VelocityEngine();
        ve2.init(p2);
```

Notice that the configuration is applied using the `init()` method, just as with the `Velocity` class. Finally, the output for the templates is generated and written to the console, like so:

```
// now get the output for each engine
        writeTemplateOutput(ve1);
        writeTemplateOutput(ve2);
```

The `writeTemplateOutput()` method should look quite familiar; the `mergeTemplate()` and `getTemplate()` methods of the `Velocity` class have been used in all the previous examples. In the following code, they're used on the `VelocityEngine` instances but to the same effect:

```
public static void writeTemplateOutput(VelocityEngine ve) throws Exception {
        Template t = ve.getTemplate(TEMPLATE_NAME);

        Writer writer = new StringWriter();
        t.merge(new VelocityContext(), writer);

        System.out.println(writer.toString());
    }
```

Running this example gives the following output:

```
This is template number 1
This is template number 2
```

Even though each runtime instance is looking for a template with the same name, their resource loaders are configured differently, so they're looking in different directories and will therefore find the different templates.

Use `VelocityEngine` wherever you want an isolated runtime with its own configuration. You should consider creating your own singleton wrapper around the `VelocityEngine` if you need to use it so you can take advantage of the caching

functionality. Consider a Web application where you create a new instance of `VelocityEngine` for each request. Doing this will prevent resources from being cached between the different requests, as each request will have its own short-lived instance of the Velocity runtime.

Summary

This chapter introduced many new concepts that are core to the way the Velocity runtime works. It started by showing how to build the different Velocity JAR files and what differences these JAR files have. You'll now be able to decide which JAR file is suitable for you and build it from the source code. You've seen both the `Velocity` and `VelocityEngine` classes in action and taken an in-depth look at the similarities and differences between the two. A big part of this chapter focused on configuring the Velocity runtime to your liking. You looked at logging and the various different logging implementations Velocity can use, and you took a detailed look at the concept of resource loaders, how to configure them, and which ones are most suited to normal application development.

In the next chapter, you'll look at the templating language used by Velocity, VTL. The chapter will take you through every VTL construct included in the standard Velocity download and will also introduce some of the extra constructs in the VelocityTools project.

Please note that the chapters included here are in their "beta" form, and are subject to modification and correction before the final book ships

CHAPTER 5

Creating Stand-Alone Applications with Velocity

By now you should have a good grasp of how the underlying Velocity system functions and how you can best utilize Velocity in your applications. However, the examples I've presented so far have been trivial at best, so in this chapter and the next I'll show you how to build full applications with Velocity—for both the desktop and for the Web. In this chapter I'll focus on building an application with Swing and Velocity, and then in the next chapter I'll demonstrate how to use Velocity when building Web applications.

During this chapter I'll build a simple e-mail newsletter application that can be used to send out customized information to Apress customers. Using this application, a member of the marketing team at Apress could keep customers informed of the latest book releases, any new offers, or any upcoming seminars or conferences that Apress may be offering. I'll write the user interface of the application in Swing, and I'll use Velocity to generate the customized e-mail message for each subscriber.

You need to be aware of two main features of the e-mail generations.

First, subscribers can choose whether they want to receive the e-mail in plain text or HTML format. It's also conceivable that in the future Apress may want to offer newsletters that contain embedded Macromedia Flash content, so the software must be able to handle this gracefully.

Second, a subscriber will typically want to receive news on only certain categories of books. For instance, a Java programmer is likely to want to receive updates on Java books as well as books on databases and open source—that Java programmer is unlikely to want to receive updates on books about .NET or Visual Basic. So, the software must be able to generate customized e-mails that go beyond just putting the subscriber's name at the top.

It's important to note that the example in this chapter isn't a complete example of how to create an e-mail marketing tool. For instance, the example in this chapter uses a fixed list of subscribers whereas a real-life solution would most

likely load the subscriber list from a database. In addition, the application provides only rudimentary text editing for the e-mail content whereas a real-world solution would likely offer some kind of HTML-based editor and then derive the plain-text content from the HTML, perhaps using regular expressions to strip out the HTML tags. Of course, all these things are possible, and you could certainly extend the application in this chapter if you wanted to add any of these features.

Application Overview

Before starting to look at the code, you'll take a quick look at how the finished application looks (see Figure 5-1).

At the top of screen, the user can specify what text should appear in the subject of the message and what the sender address should be. These values default to Apress Newsletter and newsletter@apress.com, respectively.

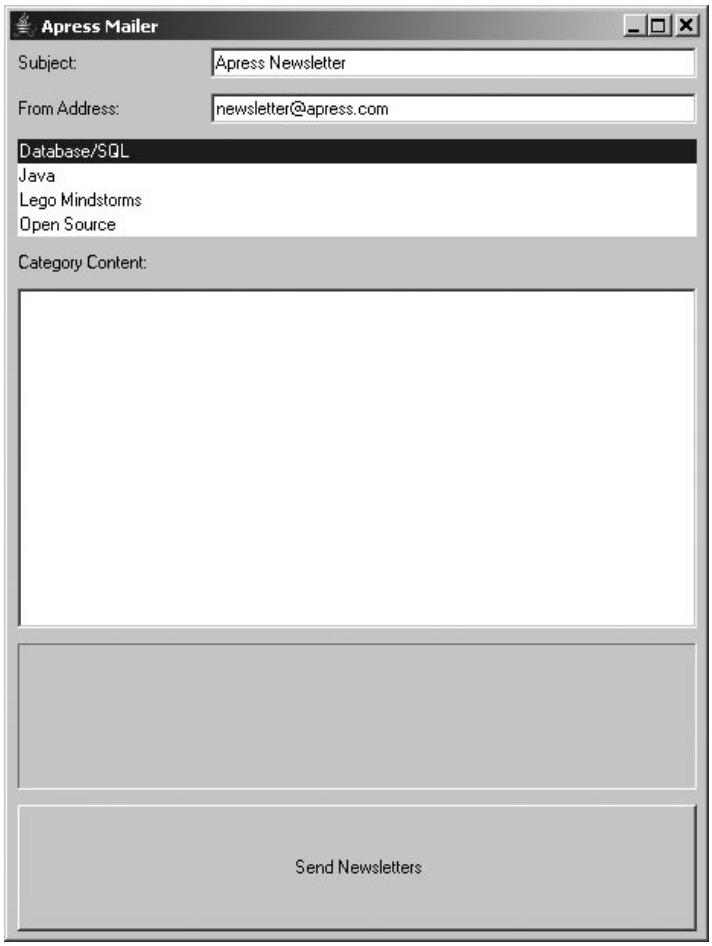

Figure 5-1. The finished application

Underneath this, you have the list of categories. Clicking a category will load the content for that category into the large text area underneath the list box. When the user moves from one category to another, the current content is stored for the category that the user is deselecting, and the stored content for the newly selected category is displayed. Finally, the progress bar and the send button are at the bottom of the screen.

Building the Application

So now that you've seen how the application works, you'll take a look under the hood and examine how it's built. I'll cover the following topics:

Understanding the Domain Object Model: The application needs an object model to represent the data it's manipulating, such as subscriber details and preferences and newsletter content.

Creating the user interface: You'll take a detailed look at how the Swing user interface is assembled and how events in the user interface are linked to actions in the code.

Sending mail with JavaMail: I'll demonstrate how to use the JavaMail API to create and send e-mails.

Interacting with Velocity: You'll see how to use the framework developed in Chapter 4 in the context of a nontrivial example.

Using the VTL templates: To finish the application code, I'll show you the VTL templates used to generate the e-mail content.

Testing the application: Finally, I'll talk you through running the application and taking it for a test drive.

Domain Object Model

The application functions by sending e-mails to subscribers based on their content preferences. Subscribers specify which category or categories of books they're interested in and in what format they want to receive the newsletter. The user of the application can specify content to be included with each category in the newsletter. The combination of the category data, which is fixed between newsletters, and the content for that category, which changes each time a newsletter is sent, forms one section of the newsletter. Depending on their preferences, subscribers will receive one or more of these sections in their preferred format.

The Subscriber Class

In the application, a `Subscriber` object represents each subscriber. The `Subscriber` class has no functionality; it's a simple domain object holding data about each subscriber (see Listing 5-1).

Listing 5-1. The `Subscriber` *Class*

```
package com.apress.pjv.ch5;

public class Subscriber {

    private String firstName = null;

    private String lastName = null;

    private String emailAddress = null;

    private Format preferredFormat = Format.PLAIN_TEXT;

    private Category[] subscribedCategories = null;

    public Subscriber(String firstName, String lastName, String emailAddress,
            Category[] subscribedCategories, Format preferredFormat) {

        this.firstName = firstName;
        this.lastName = lastName;
        this.emailAddress = emailAddress;
        this.subscribedCategories = subscribedCategories;
        this.preferredFormat = preferredFormat;
    }

    public String getEmailAddress() {
        return emailAddress;
    }

    public void setEmailAddress(String emailAddress) {
        this.emailAddress = emailAddress;
    }

    public String getFirstName() {
        return firstName;
    }
```

```
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }

    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }

    public Format getPreferredFormat() {
        return preferredFormat;
    }

    public void setPreferredFormat(Format preferredFormat) {
        this.preferredFormat = preferredFormat;
    }

    public Category[] getSubscribedCategories() {
        return subscribedCategories;
    }

    public void setSubscribedCategories(Category[] subscribedCategories) {
        this.subscribedCategories = subscribedCategories;
    }
}
```

As you can see from Listing 5-1, I've declared five JavaBeans properties for the `Subscriber` class: `FirstName`, `LastName`, `EmailAddress`, `PreferredFormat`, and `SubscribedCategories`. The first three should be fairly self-explanatory, but the other two use classes you haven't seen before, so a little more explanation is required.

The `PreferredFormat` property represents the format in which subscribers would prefer to receive their e-mail newsletter. As defined in the requirements, this can be either HTML or plain text, but you can add new formats later. It would've been simpler to use an `int` for each particular format and then use public static final fields to store the possible value. However, a problem arises with this, in that you need to know the MIME type for each format so that the e-mail message can be constructed appropriately. It'd be possible for the code sending the e-mail to decode each `int` value into the appropriate MIME type. Using this approach has a drawback, in that adding a new format would require changes to the class defining the constant `int` values and a change to the class that sends the e-mail message. A better approach is to create a `Format` class and to add a property, `ContentType`, to the class that returns

the appropriate MIME type for a particular format. Listing 5-2 shows the approach I used in the example.

Listing 5-2. The `Format` *Class*

```
package com.apress.pjv.ch5;

public class Format {

    public static final Format PLAIN_TEXT = new Format("text/plain");
    public static final Format HTML = new Format("text/html");

    private String contentType = null;

    private Format(String contentType) {
        this.contentType = contentType;
    }

    public String getContentType() {
        return contentType;
    }
}
```

You'll notice that the `Format` class has a private constructor and that the two available formats, HTML and plain text, are declared as static constants. The reason for this is twofold. First, it reduces the chance that the MIME type for one of the formats could be incorrectly specified because of a typing error. Second, this approach will be more efficient than allowing client code to create instance of the `Format` class. This approach prevents multiple instances of the `Format` object being created to represent the same actual format—since there are only two actual formats, there should only ever be two `Format` objects in the JVM (depending on classloader behavior).

The `SubscribedCategories` property returns an array of `Category` objects, each of which represents one of the categories about which the subscriber wants to receive information.

The Category Class

Listing 5-3 shows the code for the `Category` class.

Listing 5-3. The `Category` *Class*

```
package com.apress.pjv.ch5;

public class Category {
```

```
    public static final Category JAVA = new Category("Java",
            "http://www.apress.com/category.html?nID=32");

    public static final Category OPEN_SOURCE = new Category("Open Source",
            "http://www.apress.com/category.html?nID=28");

    public static final Category DATABASE_SQL = new Category("Database/SQL",
            "http://www.apress.com/category.html?nID=42");

    public static final Category LEGO_MINDSTORMS = new Category(
            "Lego Mindstorms", "http://www.apress.com/category.html?nID=46");

    private String name = null;

    private String webLink = null;

    private Category(String name, String webLink) {
        this.name = name;
        this.webLink = webLink;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getWebLink() {
        return webLink;
    }

    public void setWebLink(String webLink) {
        this.webLink = webLink;
    }

    public String toString() {
        return name;
    }

    public static Category[] getAllCategories() {
        return new Category[]{Category.DATABASE_SQL, Category.JAVA,
Category.LEGO_MINDSTORMS, Category.OPEN_SOURCE};
    }
}
```

The Category class has two properties: Name and WebLink. The WebLink property provides a hyperlink to that category on the Apress Web site. As with the Format class, Category has a private constructor, and individual instances of Category are declared as static constants. The static method getCategoryList() returns an array of Category objects, one for each category available. Since Apress has a small well-defined list of categories, this approach is acceptable. However, if the list of categories were larger or liable to change often, then it'd be wise to load the categories from a database or some other form of external storage and to use some form of object caching to ensure that only one instance existed for each category.

The SubscriberManager Class

Together the Subscriber, Category, and Format classes effectively describe subscriber and their preferences. However, none of those classes provides a way of getting access to the actual list of subscribers. For this I created the SubscriberManager class (see Listing 5-4).

Listing 5-4. The SubscriberManager *Class*

```
package com.apress.pjv.ch5;

import java.util.ArrayList;
import java.util.List;


public class SubscriberManager {

    public List getSubscribers() {
        List subscribers = new ArrayList();

        subscribers.add(new Subscriber("Rob", "Harrop",
                "rob@cakesolutions.net", new Category[] { Category.JAVA,
                        Category.DATABASE_SQL, Category.LEGO_MINDSTORMS},
                Format.HTML));
        subscribers.add(new Subscriber("Rob", "Harrop", "rob@cakesolutions.net",
                new Category[] { Category.JAVA, Category.DATABASE_SQL,
                        Category.OPEN_SOURCE}, Format.HTML));
        subscribers.add(new Subscriber("Rob", "Harrop", "robh@robharrop.com",
                new Category[] { Category.JAVA}, Format.PLAIN_TEXT));

        return subscribers;
    }
}
```

The implementation of `SubscriberManager` shown in Listing 5-4 is unrealistic, representing a finite list of subscribers. A more realistic implementation would load the subscribers from some kind of external storage, such as a database. However, for the sake of this example, that would be overly complex, so the simple implementation shown will suffice.

The NewsletterSection Class

The data represented by the `Category` class is persistent and will rarely change. Certainly you'd expect to see this data included in many different newsletters. However, if this were the only data included in the newsletter, then each one would be the same. The whole purpose of the example application is to communicate new information about the categories to the subscriber. Each newsletter is split into sections, with one section per category. These sections will contain persistent information about the category, such as its name and Web link, but will also contain information about the category that's specific to the particular newsletter. For this purpose, I created the `NewsletterSection` class, which represents a particular section in a newsletter (see Listing 5-5).

Listing 5-5. The `NewsletterSection` *Class*

```
package com.apress.pjv.ch5;

public class NewsletterSection {

    private Category category = null;
    private String content = null;

    public NewsletterSection(Category category, String content) {
        this.category = category;
        this.content = content;
    }

    public Category getCategory() {
        return category;
    }

    public void setCategory(Category category) {
        this.category = category;
    }

    public String getContent() {
        return content;
    }
```

```
    public void setContent(String content) {
        this.content = content;
    }
}
```

Each `NewsletterSection` class has a corresponding `Category` instance and a `String` instance containing the content for that `Category`.

With the code shown in this section, the application can now represent each subscriber and preferences within the JVM and get access to the list of categories and subscribers. The application also has an effective way of mapping the persistent category data to the transient category content that makes up the individual sections of a newsletter.

User Interface

So far, the application does little; after all, it has no entry point for the JVM to load the code, and it has no way for the user to interact with the application. In the earlier "Application Overview" section, I demonstrated the mailer application working and showed you the user interface. In this section, I'll show you the code behind the user interface.

NOTE *As you can no doubt tell by now, I'm not a user interface designer. A real-world application would probably have an interface designed by someone with a shred of graphic design skill and some understanding of what makes an application easy to use—and surprisingly that isn't a command-line interface!*

The main entry point to the application is the Mailer class (see Listing 5-6).

Listing 5-6. The `Mailer` *Class*

```
package com.apress.pjv.ch5;

import javax.swing.JFrame;
import javax.swing.SwingUtilities;

public class Mailer {

    public static void main(String[] args) {

        SwingUtilities.invokeLater(new Runnable() {
```

```
            public void run() {
                createAndShowGUI();
            }
        });
    }

    private static void createAndShowGUI() {
        //Make sure we have nice window decorations.
        JFrame.setDefaultLookAndFeelDecorated(true);

        //Create and set up the window.
        JFrame frame = new JFrame("Apress Mailer");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setContentPane(new MailerPanel());

        //Display the window.
        frame.pack();
        frame.setVisible(true);
    }

}
```

The first point of note in this class is the `main()` method itself. Rather than call `createAndShowGUI()` directly, the main method creates an anonymous class that implements the `Runnable` interface, implementing the `run()` method to call `createAndShowGUI()`. This anonymous class is then passed as an argument to the `SwingUtilities.invokeLater()` method. For those of you who aren't familiar with Swing, the reason for this is that all Swing applications have to be inherently thread safe. This means that the interface should be assembled on the same thread that dispatches the Swing events. The `SwingUtilities.invokeLater()` method provides a simple way of running a task on the Swing event dispatch thread.

TIP *Swing is a huge topic and is something that not all Java programmers have used. If you want to learn more about Swing, I recommend the fantastic* Java Swing*, Second Edition (O'Reilly, 2002).*

The `createAndShowGUI()` method creates a `JFrame` instance, sets the title and content pane, and then makes the `JFrame` visible. Most of the actual user interface is created by the `MailerPanel` class, an instance of which is set as the content pane for the main application window. The `MailerPanel` class contains a lot of code, so I'll show it piece by piece. To start with, the `MailerPanel` class imports all the Swing classes required to build the user interface (see Listing 5-7).

Listing 5-7. The First Part of the `MailerPanel` *Class*

```
package com.apress.pjv.ch5;

import java.awt.Cursor;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;
import java.awt.Insets;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.util.Iterator;
import java.util.List;

import javax.swing.JButton;
import javax.swing.JLabel;
import javax.swing.JList;
import javax.swing.JPanel;
import javax.swing.JProgressBar;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.JTextField;
import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

public class MailerPanel extends JPanel implements ActionListener {

    private JTextField subjectField = null;

    private JTextField fromAddressField = null;

    private JTextArea categoryContentField = null;

    private JList categoryList = null;

    private JButton sendButton = null;

    private JProgressBar progressBar = null;

    private int selectedCategory = -1;

    private Category[] categories = null;

    private String[] content = null;
```

Notice also that `MailerPanel` class extends the `JPanel` class and implements the `ActionListener` interface. The `JPanel` class is a container for Swing components; by extending this class, you can use the `MailerPanel` class as the main content pane of a `JFrame`, and you can take advantage of the basic implementation provided by `JPanel`. By implementing the `ActionListener` interface, the `MailerPanel` can receive notification of actions from different components—this is useful when you want to detect when the user clicks the send button.

The `MailerPanel` declares nine different private fields. The fields that store Swing components give the `MailerPanel` simple access to the components once they've been added to the content pane. The three remaining fields store the categories and their content and manage the category that the user has selected in the category list.

The actual user interface is assembled when the `MailerPanel` constructor is called (see Listing 5-8).

Listing 5-8. The `MailerPanel` *Constructor*

```
public MailerPanel() {
        super(new GridBagLayout());

        loadData();

        GridBagConstraints c = new GridBagConstraints();
        c.fill = GridBagConstraints.HORIZONTAL;
        c.weightx = 0.5;
        c.weighty = 0.5;
        c.insets = new Insets(5, 5, 5, 5);

        addFieldLabels(c);

        addSubjectField(c);

        addFromAddressField(c);

        addCategoryList(c);

        addCategoryContentsLabel(c);

        addCategoryContentsTextBox(c);

        addProgessBar(c);

        addSendButton(c);

    }
```

The first line in the constructor invokes the constructor on the superclass, `JPanel`, passing as an argument a new instance of `GridBagLayout`. This will set the layout of the `MailerPanel` to the `GridBagLayout`, which allows for precise, grid-based layout of components.

Next, the category list is loaded with a call to `loadData()`:

```
private void loadData() {
        categories = Category.getCategoryList();
        content = new String[categories.length];
    }
```

The `loadData()` method stores the result of the call to `Category.getCategoryList()` in the `categories` field and also initializes the `content` field to a `String[]` of the same size as the `categories` field. Back in the constructor, an instance of `GridBagConstraints` is created, like so:

```
GridBagConstraints c = new GridBagConstraints();
 c.fill = GridBagConstraints.HORIZONTAL;
 c.weightx = 0.5;
 c.weighty = 0.5;
 c.insets = new Insets(5, 5, 5, 5);
```

The `GridBagConstraints` class specifies the location and layout of components when adding them to the `GridBagLayout`. Now the constructor creates the user interface with calls to private methods, each of which configures a different piece of the user interface. Each of these methods is passed the `GridBagConstraints` instance. This allows the constructor to provide default settings, such as the insets and fill for the `GridBagConstraints`, but it also allows each method to supply the layout parameters for each component.

The first method called is the `addFieldLabels()` method, as follows:

```
private void addFieldLabels(GridBagConstraints c) {
        // labels
        c.gridx = 0;
        c.gridy = 0;
        add(new JLabel("Subject: "), c);

        c.gridx = 0;
        c.gridy = 1;
        add(new JLabel("From Address: "), c);
    }
```

This method adds the Subject and From Address labels to the top of the `MailerPanel`. Notice the use of the `gridx` and `gridy` fields of the `GridBagConstraints`

class. By setting these values and then passing the `GridBagConstraints` instance to the `add()` method along with each `JLabel`, I can specify which cell in the grid I want the component to reside. In this case, I'm saying that I want the Subject label to appear in the first row (0) and the first column (0) and that I want the Address label to appear in the second row (1) and the first column (0).

The next call from the constructor is to the `addSubjectField()` method, like so:

```
private void addSubjectField(GridBagConstraints c) {
    // text fields
    c.gridx = 1;
    c.gridy = 0;

    subjectField = new JTextField();
    subjectField.setText("Apress Newsletter");
    c.ipadx = 150;
    add(subjectField, c);
}
```

This method adds the text field used to enter the message subject to the `MailerPanel`. Notice that the instance of `JTextField` created is assigned to the `subjectField` field. This will give me easy access to the text field and its value later in the code.

Next up is the following call to `addFromAddressField()`:

```
private void addFromAddressField(GridBagConstraints c) {
    c.gridx = 1;
    c.gridy = 1;

    fromAddressField = new JTextField();
    fromAddressField.setText("newsletter@apress.com");
    c.ipadx = 150;
    add(fromAddressField, c);
}
```

This method is similar to the `addSubjectField()` method, so there's no need for any extra explanation. The next call the constructor makes is to the `addCategoryList()` method, like so:

```
private void addCategoryList(GridBagConstraints c) {
    c.gridx = 0;
    c.gridy = 2;
    c.gridwidth = 2;
    categoryList = new JList();
    categoryList.setListData(categories);
    categoryList.addListSelectionListener(new ListSelectionListener() {
```

```
            public void valueChanged(ListSelectionEvent event) {
                if (event.getValueIsAdjusting() == false) {
                    updateCategoryContents();
                }
            }
        });
        categoryList.setSelectedIndex(0);
        add(categoryList, c);
    }
```

Most of the code in this method will be familiar to you by now. It configures `GridBagConstraints`, creates an instance of `JList`, and assigns that instance to the `categoryList` field. However, before the `JList` is added to the `MailerPanel`, the `categories` array is set as the source data for the list using the `JList.setListData()` method. In addition, an anonymous class is created that implements the `ListSelectionListener` interface, and the `valueChanged` method is implemented to call the `updateCategoryContents()` method whenever the value of the list is actually changing, not just when the user is manipulating the list. Finally, before adding the `JList` to the `MailerPanel`, the first item is list is selected, providing a default selection when the user interface is displayed.

Next, the constructor calls `addCategoryContentsLabel()` and `addCategoryContentsTextBox()`, like so:

```
    private void addCategoryContentsLabel(GridBagConstraints c) {
        // Category Contents Label
        c.gridx = 0;
        c.gridy = 3;
        add(new JLabel("Category Content:"), c);
    }

    private void addCategoryContentsTextBox(GridBagConstraints c) {
        c.gridx = 0;
        c.gridy = 4;
        c.ipadx = 250;
        c.ipady = 100;
        categoryContentField = new JTextArea();
        categoryContentField.setLineWrap(true);
        categoryContentField.setWrapStyleWord(true);
        JScrollPane scroller = new JScrollPane(categoryContentField);
        add(scroller, c);
    }
```

Both of these methods should look pretty familiar; the only points you should note are the call to `JTextArea.setWrapStyleWord(true)` and the use of

`JScrollPane`. Without setting the line wrap style to word wrapping, the `JTextArea` will wrap a line in the middle of a word, which isn't very user-friendly. You probably noticed the `JTextArea` instance itself isn't added to the `MailerPanel` class; instead, an instance of `JScrollPane` is created with the `JTextArea` as the inner component and the `JScrollPane` instance is added to the `MailerPanel`. This prevents the `JTextArea` from expanding to take up the whole pane when the text inside it increases.

The final calls of the constructor are to `addProgressBar()` and `addSendButton()`, as follows:

```
private void addProgessBar(GridBagConstraints c) {
    c.gridx = 0;
    c.gridy = 5;
    c.ipady = 40;
    progressBar = new JProgressBar();
    add(progressBar, c);
}

private void addSendButton(GridBagConstraints c) {
    c.gridx = 0;
    c.gridy = 6;
    c.ipady = 30;
    c.ipadx = 100;
    sendButton = new JButton("Send Newsletters");
    sendButton.setActionCommand("send");
    sendButton.addActionListener(this);
    add(sendButton, c);
}
```

Both of these methods should be familiar to you by now; the only point of note is these two lines from `addSendButton()`:

```
sendButton.setActionCommand("send");
sendButton.addActionListener(this);
```

The call `setActionCommand()` gives the button a command name, and the call to `addActionListener()` sets the current `MailerPanel` instance as the object that will receive notifications when the button is clicked. Using the command name is especially useful when the same `ActionListener` is used for multiple buttons, as it provides a way to differentiate between the buttons.

The remaining methods in the `MailerPanel` class handle events from the user interface components. The `updateCategoryContents()` method is called whenever the value of the category list changes, as follows:

```
private void updateCategoryContents() {
        if (selectedCategory == -1) {
```

```
            // this is the first-time selection
            selectedCategory = categoryList.getSelectedIndex();
            return;
        }

        saveCurrentContent();

        // display the content for the newly selected category
        categoryContentField.setText(content[categoryList.getSelectedIndex()]);

        // store the new selection
        selectedCategory = categoryList.getSelectedIndex();
    }

    private void saveCurrentContent() {
        // store the content for the previously selected category
        content[selectedCategory] = categoryContentField.getText();
    }
```

The first time the `updateCategoryContents()` is called, the value of `selectedCategory` will be -1, so the method simply stores the newly selected index and returns control to the caller. However, after that, the value of `selectedCategory` will be greater than -1, so the rest of the method is executed. It's important to understand that this method is executed after the value. therefore, the selected index of the list has changed, and the value of the `selectedCategory` field will be the previously selected index. Using this value as the array index, the `saveCurrentContent()` method will store the current text from the `categoryContentField` `JTextArea` in the content array. In other words, if the `Category` object for Java is stored at index 1 in the `categories` array, then the content for the Java category will be stored at index 1 in the `content` array. Once the `saveCurrentCategory()` method has executed, the `updateCategoryContents()` method will get the text for the newly selected category from the content array and set that as the value for the `categoryContentField` `JTextArea`. Finally, the new index is stored in the `selectedCategory` field so that the next time this method is called, the correct category is flagged as the previously selected one.

To receive notifications when the send button is clicked, the `MailerPanel` class implements the `ActionListener` interface and is registered as an action listener for the send button. The `ActionListener` interface has one method, which is `actionPerformed()`, as follows:

```
public void actionPerformed(ActionEvent event) {
        if (event.getActionCommand().equals("send")) {
            new Thread(new Runnable() {
```

```
                public void run() {
                    sendNewsletters();
                }
            }).start();
        }
    }
```

The `actionPerformed()` method is passed an instance of the `ActionEvent` class by the Swing framework. Using this instance, the `MailerPanel` can determine which action is being performed by checking the value of the `ActionEvent.getActionCommand()` method. In this case, you're looking for the command `send`, which you'll remember was set as the action command for the send button. The `actionPerformed()` method is called from the Swing event dispatch thread, which means that any long-running tasks will interfere with the event processing in the application. To prevent this, I created a new thread to call the `sendNewsletters()` method.

The `sendNewsletters()` method handles the user interface updates before, during, and after the sending process, as well as controlling the sending process for each individual newsletter. The code for `sendNewsletters()` is quite long, so I'll present it in chunks:

```
private synchronized void sendNewsletters() {
    //disable controls
    switchControlState();
```

To start with, `sendNewsletters()` calls the `switchControlState()` method, which disables the send button and the category list if they're enabled and enables them if they're disabled.

```
private void switchControlState() {
    categoryList.setEnabled(!categoryList.isEnabled());
    sendButton.setEnabled(!sendButton.isEnabled());
}
```

Next, the `sendNewsletters()` method sets the cursor to the wait cursor and calls `saveCurrentContent()` to save any content changes for the currently selected category.

```
// show busy cursor
setCursor(new Cursor(Cursor.WAIT_CURSOR));

saveCurrentContent();
```

Next, the `sendNewsletters()` method gets the `List` of subscribers from the `SubscriberManager`. Using this `List`, the maximum and minimum values of the

progress bar are configured and then the message subject and from address are retrieved from the corresponding text fields.

```
List subscribers = (new SubscriberManager()).getSubscribers();

progressBar.setMinimum(1);
progressBar.setMaximum(subscribers.size());

String subject = subjectField.getText();
String fromAddress = fromAddressField.getText();
```

The next step in the process is to create an array of `NewsletterSection` objects, one for each `Category` object in the category list, like so:

```
NewsletterSection[] sections = new NewsletterSection[categories.length];

for (int x = 0; x < sections.length; x++) {
    sections[x] = new NewsletterSection(categories[x], content[x]);
}
```

Notice how the category and the corresponding content are loaded from the `categories` and `content` arrays using the same index. The next-to-last step for the `sendNewsletters()` method is to iterate over the `List` of `Subscriber` objects retrieved from the `SubscriberManager` and send a newsletter to each one using the `NewsletterManager` class, like so:

```
Iterator itr = subscribers.iterator();
NewsletterManager manager = new NewsletterManager(fromAddress, subject);

int count = 1;
while (itr.hasNext()) {
    Subscriber s = (Subscriber) itr.next();
    manager.sendNewsletter(sections, s);
    progressBar.setValue(count++);
}
```

Notice that each time a newsletter is sent, the progress bar value is incremented to give the user some visual feedback as to the progress of the sending process. The next section covers the `NewsletterManager` class in detail; for now, it's enough to know that the `NewsletterManager` will create the appropriate newsletter content for each subscriber and send it to each subscriber's e-mail address. The final part of the `sendNewsletters()` method restores the cursor to the default and switches the state of the send button and the category list back to enabled, like so:

```
        // restore cursor
        setCursor(new Cursor(Cursor.DEFAULT_CURSOR));

        // reactivate controls
        switchControlState();

    }
```

At this point, the application won't actually compile, because there's no implementation for the `NewsletterManager` class. However, you could provide a stub implementation of this class and run the example. If you do this, you should find that the user interface is fully operational—save for the fact that clicking the send button won't actually do anything—but you should be able to add content for each category and switch back and forth between the categories to ensure that the content for each category is being stored appropriately.

Sending the Newsletters

As you saw from the previous section, the logic for actually sending the newsletter resides in the `NewsletterManager.sendNewsletter()` method. The `NewsletterManager` class uses the JavaMail API to construct and send the e-mail message, so you'll need to download it before you can continue with the code. You can download the latest version of JavaMail from `http://java.sun.com/products/javamail/`; I used version 1.3.1 for this book. In addition to JavaMail, you need to download the JavaBeans Activation Framework (JAF), which is used by JavaMail to construct the mail messages. You contain obtain JAF from `http://java.sun.com/products/javabeans/jaf/`.

The `NewsletterManager` class declares three instance fields and one constant field (see Listing 5-9).

Listing 5-9. The `NewsletterManager` *Class*

```
package com.apress.pjv.ch5;

import java.util.Properties;

import javax.mail.Address;
import javax.mail.Message;
import javax.mail.MessagingException;
import javax.mail.Session;
import javax.mail.Transport;
import javax.mail.internet.AddressException;
import javax.mail.internet.InternetAddress;
import javax.mail.internet.MimeMessage;
```

```
public class NewsletterManager {

    private static final String SMTP_SERVER = "localhost";

    private Session session = null;

    private Address fromAddress = null;

    private String subject = null;
```

The SMTP_SERVER constant holds the address of the SMTP server used to send e-mails. It's likely that this would be a configurable parameter in a real-world application; however, for the sake of this example, a constant will suffice. The subject field will store the subject of the message, which, as you saw in MailerPanel.sendNewsletters(), is passed to the NewsletterManager in the constructor. The session field holds an instance of javax.mail.Session, which represents a communication session with the mail server. The fromAddress field stores the address to be used as the sender address on the newsletter e-mails.

The NewsletterManager constructor requires two arguments: the subject of the message and the sender address (see Listing 5-10).

Listing 5-10. The NewsletterManager *Constructor*

```
public NewsletterManager(String fromAddress, String subject)
            throws NewsletterException {
        try {
            this.fromAddress = new InternetAddress(fromAddress);
        } catch (AddressException ex) {
            throw new NewsletterException("Invalid from address", ex);
        }
        this.subject = subject;
    }
```

You'll notice that although the fromAddress field is declared as type javax.mail.Address, it's instantiated at type javax.mail.internet.InternetAddress. The reason for this is that the Address class is abstract and serves as the common base class for different kinds of addresses. The Address class has two concrete subclasses: InternetAddress and NewsAddress, used for e-mails and newsgroup addresses, respectively. You should also notice that the constructor for InternetAddress throws an AddressException if you supply an incorrectly formatted e-mail address. The NewsletterManager class catches this exception and wraps it in the NewsletterException class (see Listing 5-11).

Listing 5-11. The `NewsletterException` *Class*

```
package com.apress.pjv.ch5;

public class NewsletterException extends RuntimeException {

    public NewsletterException() {
        super();
    }

    public NewsletterException(String msg) {
        super(msg);
    }

    public NewsletterException(String msg, Throwable rootCause) {
        super(msg, rootCause);
    }

    public NewsletterException(Throwable rootCause) {
        super(rootCause);
    }
}
```

The `sendNewsletter()` method is where it all happens! In this method, the mail message is constructed, the recipients are configured, and the message is sent. The `sendNewsletter()` method is quite long, so I'll explain it in chunks. You'll recall from the previous section that the `MailerPanel.sendNewsletters()` method iterates over the list of subscribers and calls `sendNewsletter()` once for each subscriber, passing in the `Subscriber` object and an array of `NewsletterSection` object representing the newsletter content, like so:

```
public boolean sendNewsletter(NewsletterSection[] sections,
            Subscriber subscriber) throws NewsletterException {
```

The first step taken by the `sendNewsletter()` method is to create the actual newsletter content:

```
NewsletterTemplate template = NewsletterTemplateFactory.getInstance()
        .getNewsletterTemplate(subscriber.getPreferredFormat());

template.setSections(sections);
template.setSubscriber(subscriber);
```

This is where Velocity enters the equation. The `sendNewsletter()` method doesn't interact with Velocity directly; instead, it follows the pattern discussed in Chapter 4. It retrieves a template object, in this case an object that implements

the NewsletterTemplate interface, from the NewsletterTemplateFactory and appropriate properties on the NewsletterTemplate instance. The next section discusses these classes and interfaces in more detail.

Next up comes the code that sends the actual message. The JavaMail API throws quite a few different exceptions, so this block is wrapped in a try/catch block so that any JavaMail-specific exceptions can be wrapped in a NewsletterException. I start by creating an instance of MimeMessage and setting the content of the message, like so:

```
try {
    Message msg = new MimeMessage(getMailSession());
    msg.setContent(template.generate(), subscriber.getPreferredFormat()
            .getContentType());
```

The MimeMessage constructor is passed an instance of javax.mail.Session, which is obtained from the getMailSession() method. The code for getMailSession() is shown at the end of this section, but it simply ensures that the Session object is correctly configured and that all calls to sendNewsletter() on the same NewsletterManager instance use the same Session object. An important call in this block of code is the call to Message.setContent(). Here I use the generate() method of the template object to provide the content for the message, and I use the getContentType() method of the subscribers preferred Format object to set the content type.

The remaining code in the sendNewsletter() method simply configures the subject, from the address and recipient address of the mail message, and then sends it with a call to Transport.send(), as follows:

```
        msg.setSubject(subject);
        msg.setFrom(fromAddress);

        msg.addRecipient(Message.RecipientType.TO, new InternetAddress(
                subscriber.getEmailAddress()));

        Transport.send(msg);
        return true;
    } catch (AddressException e) {
        // invalid address - ignore
        e.printStackTrace();
        return false;
    } catch (MessagingException e) {
        e.printStackTrace();
        throw new NewsletterException("Unable to send newsletter", e);
    }
}
```

Notice that the catch block for the `AddressException` doesn't rethrow an exception; instead, it simply returns `false`, indicating that the message wasn't sent. The reason for this is that I don't want an invalid address to stop the entire process. Imagine what would happen if the user left the software to send 1,000 e-mails overnight, only for it to error out on the third one because of an invalid e-mail address. In a real-world application, it'd be likely that some kind of log of invalid addresses would be used so that the user could correct the invalid addresses and attempt to resend the newsletter to those addresses.

For completeness, the following is the code for the `getMailSession()` method:

```
private Session getMailSession() {
        if (session == null) {
            Properties props = new Properties();
            props.put("mail.smtp.host", SMTP_SERVER);
            session = Session.getDefaultInstance(props);
        }

        return session;
    }
```

The code for sending the e-mail messages is relatively short, but the messages are quite basic and don't involve any complex assembly, such as would be required for HTML messages with embedded images or messages with two alternative versions of the content. In the next section I'll show you last part of the puzzle: content generation using Velocity.

Generating the Newsletter Content

At this point you may be wondering if I've forgotten about Velocity completely. Well, the answer is certainly not. However, I did want to illustrate a point by leaving Velocity until the end of the chapter—the application I've built is completely decoupled from Velocity. I could provide any implementation of the `NewsletterTemplate` interface used by the `sendNewsletter()` method; I'm not just limited to using Velocity. This is good application design and doesn't add any additional complexity to the application. However, you aren't interested in another implementation; you're interested in Velocity, so you'll now see how you can provide an implementation of the `NewsletterTemplate` interface using the framework classes discussed in Chapter 4.

Template Object Model

Listing 5-12 shows the NewsletterTemplate interface.

Listing 5-12. The NewsletterTemplate *Interface*

```
package com.apress.pjv.ch5;

import com.apress.pjv.ch4.ContentTemplate;

public interface NewsletterTemplate extends ContentTemplate {

    public NewsletterSection[] getSections();
    public void setSections(NewsletterSection[] sections);

    public Subscriber getSubscriber();
    public void setSubscriber(Subscriber subscriber);
}
```

As you can see, the NewsletterTemplate interface is pretty basic, but it does inherit some methods from the ContentTemplate interface that were discussed in Chapter 4 (see Listing 5-13).

Listing 5-13. The ContentTemplate *Interface*

```
package com.apress.pjv.ch4;

import java.io.Writer;

public interface ContentTemplate {

    public String generate() throws TemplateException;
    public void generate(Writer writer) throws TemplateException;
}
```

If you recall the example from Chapter 4, it's a trivial job to implement the NewsletterTemplate to use Velocity using the AbstractVelocityContentTemplate class, which provides implementations of the generate() methods that use the Velocity template engine (see Listing 5-14).

Listing 5-14. The AbstractVelocityContentTemplate *Class*

```
package com.apress.pjv.ch4;

import java.io.StringWriter;
import java.io.Writer;
```

```
import org.apache.velocity.Template;

import org.apache.velocity.context.Context;

public abstract class AbstractVelocityContentTemplate {

    public void generate(Writer writer) throws TemplateException {
        try {
            VelocityManager.init("src/velocity.properties");

            Template t = VelocityManager.getTemplate(getResourceName());

            // create the context
            Context ctx = ContextFactory.getInstance();

            // populate with model data
            ModelBean model = getModel();
            ctx.put(model.getModelName(), model);

            t.merge(ctx, writer);

        } catch(Exception ex) {
            throw new TemplateException("Unable to generate output", ex);
        }
    }

    public String generate() throws TemplateException {
        Writer w = new StringWriter();
        generate(w);
        return w.toString();
    }

    protected abstract ModelBean getModel();
    protected abstract String getResourceName();
}
```

However, the difference between this example and the one from Chapter 4 is that I actually need two implementations of the `NewsletterTemplate` interface: one for the HTML newsletter and one for the plain-text newsletter. Both of these classes are going to share almost identical implementations—in fact, they only differ in their implementation of `AbstractVelocityContentTemplate.getResourceName()`. To save duplicating a bunch of code, I created the `AbstractNewsletterTemplate` class, which provides an implementation of the `NewsletterTemplate` interface and an implementation of `AbstractVelocityContentTemplate.getModel()` (see Listing 5-15).

Listing 5-15. The `AbstractNewsletterTemplate` *Class*

```
package com.apress.pjv.ch5;

import com.apress.pjv.ch4.AbstractVelocityContentTemplate;
import com.apress.pjv.ch4.ModelBean;

public abstract class AbstractNewsletterTemplate
            extends AbstractVelocityContentTemplate
         implements NewsletterTemplate {

    private NewsletterSection[] sections;
    private Subscriber subscriber;

    public NewsletterSection[] getSections() {
        return sections;
    }

    public void setSections(NewsletterSection[] sections) {
        this.sections = sections;
    }

    public Subscriber getSubscriber() {
        return subscriber;
    }

    public void setSubscriber(Subscriber subscriber) {
        this.subscriber = subscriber;
    }

    protected ModelBean getModel() {
        return new NewsletterModelBean(sections, subscriber);
    }
}
```

The `getModel()` method returns an implementation of the following `ModelBean` interface shown in Chapter 4:

```
package com.apress.pjv.ch4;

public interface ModelBean {

    public String getModelName();
}
```

Here, this implementation is given by the `NewsletterModelBean` class (see Listing 5-16).

Listing 5-16. The `NewsletterModelBean` *Class*

```
package com.apress.pjv.ch5;

import com.apress.pjv.ch4.ModelBean;

public class NewsletterModelBean implements ModelBean {

    private NewsletterSection[] sections = null;

    private Subscriber subscriber = null;

    public NewsletterModelBean(NewsletterSection[] sections,
            Subscriber subscriber) {
        this.sections = sections;
        this.subscriber = subscriber;
    }

    public String getModelName() {
        return "newsletter";
    }

    public NewsletterSection[] getSections() {
        return sections;
    }

    public void setSections(NewsletterSection[] sections) {
        this.sections = sections;
    }

    public Subscriber getSubscriber() {
        return subscriber;
    }

    public void setSubscriber(Subscriber subscriber) {
        this.subscriber = subscriber;
    }
}
```

The next step is to provide two subclasses of `AbstractNewsletterTemplate`: one for the HTML template and one for the plain-text template. The implementation for these classes is trivial, since most of the code is in the `AbstractNewsletterTemplate` and `AbstractVelocityContentTemplate` classes (see Listing 5-17).

Listing 5-17. `AbstractNewsletterTemplate` *and* `AbstractVelocityContentTemplate`

```
package com.apress.pjv.ch5;

public class HtmlNewsletterTemplate extends AbstractNewsletterTemplate {

    protected String getResourceName() {
        return "html/newsletter.vm";
    }
}

package com.apress.pjv.ch5;

public class PlainTextNewsletterTemplate extends AbstractNewsletterTemplate {

protected String getResourceName() {
        return "plainText/newsletter.vm";
    }
}
```

If you recall, the `NewsletterManager.sendNewsletter()` method obtains an implementation of the `NewsletterTemplate` from the `NewsletterTemplateFactory` class (see Listing 5-18).

Listing 5-18. The `NewsletterTemplateFactory` *Class*

```
package com.apress.pjv.ch5;

public class NewsletterTemplateFactory {

    private static NewsletterTemplateFactory instance;

    static {
        instance = new NewsletterTemplateFactory();
    }

    private NewsletterTemplateFactory() {
        // no-op
    }

    public static NewsletterTemplateFactory getInstance() {
        return instance;
    }

    public NewsletterTemplate getNewsletterTemplate(Format format) {
        if (format == Format.HTML) {
```

```
            return new HtmlNewsletterTemplate();
        } else {
            return new PlainTextNewsletterTemplate();
        }
    }
}
```

Notice that the `getNewsletterTemplate()` method returns an appropriate implementation based on the `Format` object supplied as the method argument. The `NewsletterTemplateFactory` class is the last class needed for the application. The class model that's in place for the template generation provides a model that's simple to code against but is also easy to extend. For instance, you'll remember that at the beginning of the chapter I talked about extending the application so that subscribers could receive mailings in Macromedia Flash format. Adding this support to the template generation would be easy; all it requires is a small change to the `NewsletterTemplateFactory` and a new class, `FlashNewsletterTemplate`, as follows:

```
public class FlashNewsletterTemplate extends AbstractNewsletterTemplate {

protected String getResourceName() {
        return "flash/newsletter.vm";
    }
}
```

Of course, I could quite easily rip out the Velocity support altogether and provide completely different implementations of the `NewsletterTemplate` interface.

Velocity Templates

With all the Java code complete, all that's left is to create the Velocity templates and run the software. The plain-text template is the simplest, so I will start with that one (see Listing 5-19).

Listing 5-19. The Plain-Text Template

```
Hi, $newsletter.Subscriber.FirstName $newsletter.Subscriber.LastName, ➥
and welcome to the
Apress Monthly Newsletter!

We have a great selection of new books for you this week:

#foreach($section in $newsletter.Sections)
#set($include = false)
```

```
#foreach($cat in $newsletter.Subscriber.SubscribedCategories)
#if($cat == $section.Category)
#set($include = true)
#end
#end
#if($include)
----------------------------------------
$section.Category.Name
----------------------------------------

$section.Content

View more details about $section.Category.Name at: $section.Category.WebLink.


#end
#end
----------------------------------------
To unsubscribe from this newsletter, visit: http://www.apress.com/unsubscribe/ ➥
$newsletter.Subscriber.EmailAddress
```

Most of this code will look familiar; the only part that may throw you is this:

```
#set($include = false)
#foreach($cat in $newsletter.Subscriber.SubscribedCategories)
#if($cat == $section.Category)
#set($include = true)
#end
#end
```

This code will run for each `NewsletterSection` object in the model and will check to see if the subscriber is subscribed to the category represented by the `NewsletterSection`. If so, the `$include` variable is set to true, and the content will be included; otherwise, `$include` is false and the content will be excluded. Another way of achieving this would have been to add an `isSubscribedToCategory(Category)` method to the `Subscriber` object and have Velocity call that—either way the outcome is the same.

For the most part, the HTML template is similar; the only real difference is the obvious one—layout is achieved using HTML tags (see Listing 5-20).

Listing 5-20. The HTML Template

```
<html>
<head>
<title>A P R E S S . C O M | Books for Professionals, by Professionals
...</title>
<base href="http://www.apress.com">
</head>
<body text="#000000" vLink="#333399" link="#333399" leftmargin="0"
background="/img/v1/bkgd.gif" topmargin="0"
marginheight="0" marginwidth="0">
<table cellspacing="0" cellpadding="0" width="780" border="0">
  <tbody>
    <tr valign="top" align="left">
      <td> <table cellspacing="0" cellpadding="0" width="166" border="0">
          <tbody>
            <tr valign="top" align="left">
              <td valign="top" align="left" width="166" height="109">
               <a href = "/">
                 <img src="/img/v1/aMod.gif" height="118"
                           width="166" border="0">
                </a><br />
               </td>
            </tr>
          </tbody>
        </table></td>
      <td> <table cellspacing="0" cellpadding="0" width="614" border="0">
          <tbody>
            <tr valign="top" align="left">
              <td valign="top" align="left" width="614" height="40">
                <img height="40" src="/img/v1/top.gif" alt=""
                          width="614" border="0"><br />
             </td>
            </tr>
          </tbody>
        </table>
        <table cellspacing="0" cellpadding="0" width="614" border="0">
          <tbody>
            <tr valign="top" align="left">
              <td valign=top align=left width="614" height="45">
                     <img height="45" src="/img/v1/mid.gif" alt=""
                               width="614" border="0">
```

```
                    <br />
                    </td>
                  </tr>
                </tbody>
              </table>
              <table cellspacing="0" cellpadding="0" width="614" border="0">
                <tbody>
                  <tr valign="top" align="left">
                    <td valign="top" align="left" width="614" height="24">
                    <img height="24" src="/img/v1/bot.gif" alt=""
                              width="614" border="0"><br /> </td>
                  </tr>
                </tbody>
              </table>
              <!-- Start of Newsletter Content -->
              <h1>Hi, $newsletter.Subscriber.FirstName
$newsletter.Subscriber.LastName,
and welcome to the Apress Monthly Newsletter!</h1>
              <h2>We have a great selection of new books for you this week:</h2>
              <table border="0">
                 #foreach($section in $newsletter.Sections)
                 #set($include = false)
                 #foreach($cat in $newsletter.Subscriber.SubscribedCategories)
                 #if($cat == $section.Category)
                 #set($include = true)
                #end
                #end

                #if($include)
                <tr>
                  <td style="text-weight:bold; text-decoration:underline" bgcolor>
                          $section.Category.Name
                    </td>
                </tr>
                <tr>
                  <td>$section.Content</td>
                </tr>
                <tr>
                  <td><br>
                    View more details about $section.Category.Name, click
                    <a href="$section.Category.WebLink">here</a>.
```

```
            <hr>
          </tr>
          #end #end
        </table>
        <h3>To unsubscribe from this newsletter, click
<a href="http://www.apress.com/unsubscribe/$newsletter.Subscriber.EmailAddress">
            here</a></h3>.
        <!-- End of Newsletter Content --> </td>
    </tr>
  </tbody>
</table>
</body>
</html>
```

Most of the HTML code has been taken from the Apress Web site; the only fragment of real interest is the bit between the `<!-- Start of Newsletter Content-->` and `<!-- End of Newsletter Content-->` comments.

That's all the code required for the application completed. Now all that remains is to test the application.

Running the Example

Now that you have all the code, you can test the application. Make sure that the SMTP server specified in `NewsletterManager` and the e-mail addresses specified in `SubscriberManager` are valid for your environment.

To run the application, Unix users should execute the following command:

```
java -cp "lib/mail.jar:lib/activation.jar:lib/velocity-dep-1.4.jar:build/." ➥
com.apress.pjv.ch5.Mailer
```

If you're using a Windows operating system, you should swap the colons separating the paths for semicolons.

When the application first loads, you get the main screen with Database/SQL option preselected in the category list, as shown in Figure 5-2.

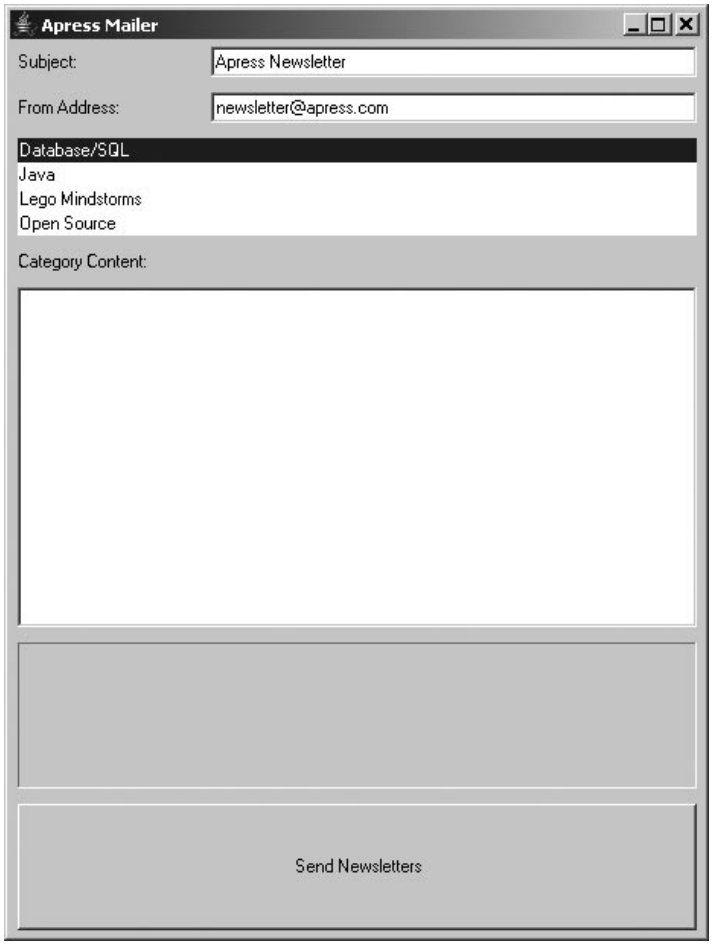

Figure 5-2. Application at startup

Enter some text in the text area, and then swap to another category. You should repeat this process until all the categories have some content, as shown in Figure 5-3.

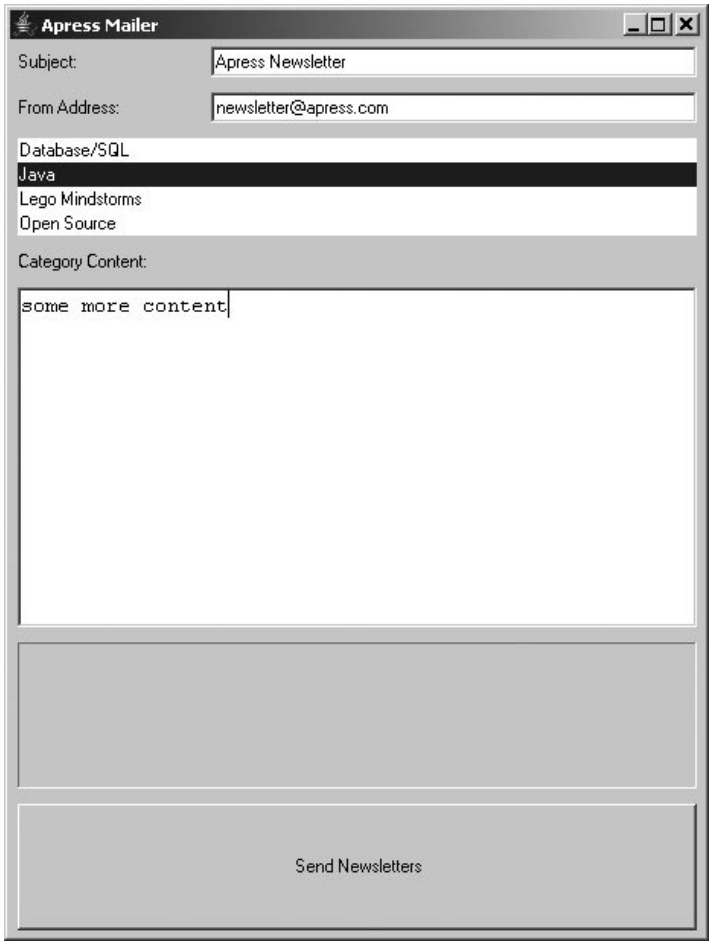

Figure 5-3. Adding category content

Now all that remains to do is click send and watch the progress bar as the mails are sent. When you receive the mail, you should see something like what's shown in Figure 5-4 for the plain-text mails.

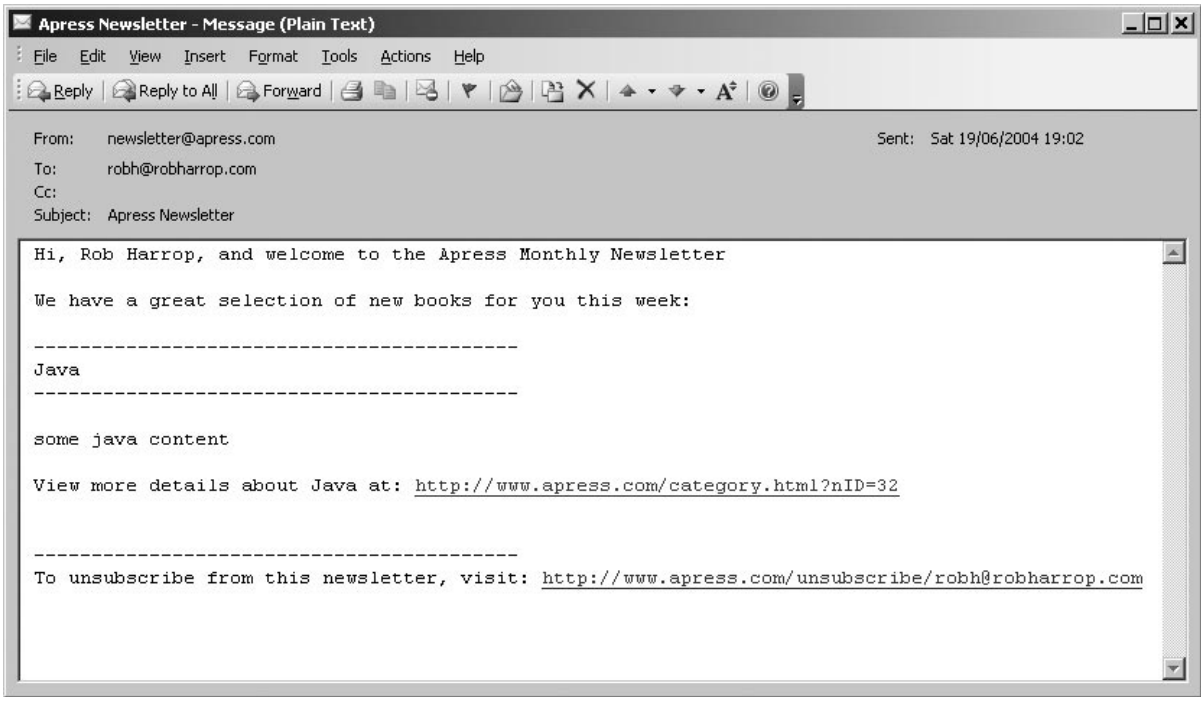

Figure 5-4. Plain-text e-mail

HTML mails look a bit more enticing, as shown in Figure 5-5.

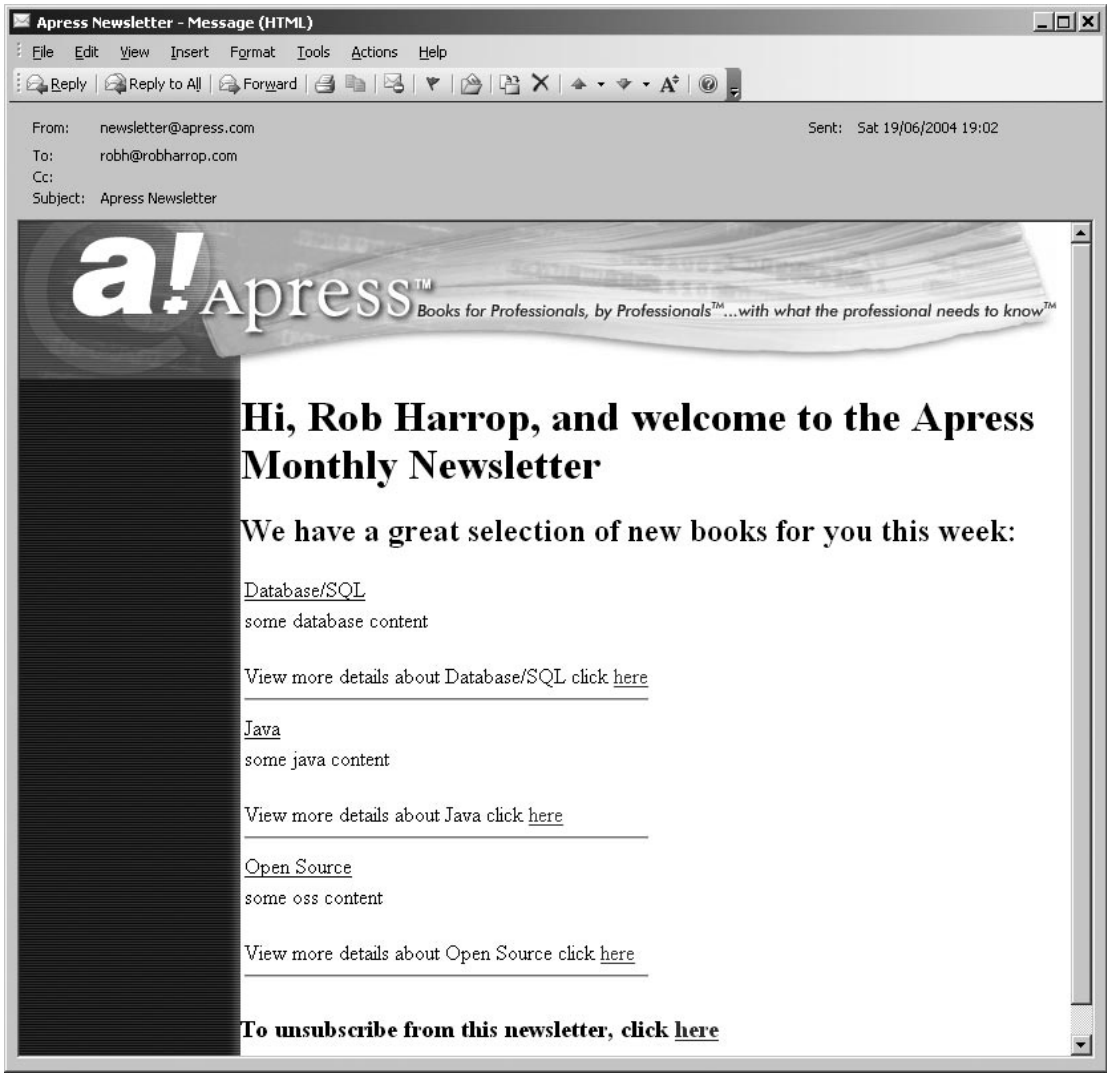

Figure 5-5. HTML e-mail

Summary

In this chapter you saw how you can use Velocity when building nontrivial applications. You haven't seen, however, a vast amount of Velocity code—this is mainly because of the framework discussed in Chapter 4. You saw how easy it is to extend the templating capabilities of the application to include new message formats and how it's possible to replace the templating implementation completely without affecting the application itself.

Applications that are built on top of Velocity should have little Velocity-specific code, reducing the application's dependency on the Velocity engine. In the example shown in this chapter, only one class, `AbstractVelocityContentTemplate`, contains Velocity-specific code. Even though the example application has only two templates, the benefits of this abstraction are clear—consider the benefits for an application with many more templates.

In the next chapter, you'll see how you can use Velocity to build Web-based applications both on its own and in conjunction with frameworks such as Struts and Spring.