Pro Jakarta Velocity: From Professional to Expert

ROB HARROP

Apress®

ISBN (pbk): 1-59059- 410-x

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Lead Editor: Steve Anglin

Technical Reviewer: Jan Machacek

Editorial Board: Steve Anglin, Dan Appleman, Ewan Buckingham, Gary Cornell, Tony Davis, Jason Gilmore, Chris Mills, Steve Rycroft, Dominic Shakeshaft, Jim Sumser, Gavin Wray

Project Manager: Tracy Brown Collins

Copy Edit Manager: Nicole LeClerc

Copy Editor: Kim Wimpsett

Production Manager: Kari Brooks

Production Editor: Laura Cheu

Compositor: Linda Weidemann, Wolf Creek Press

Proofreader: Linda Seifert

Indexer: Valerie Perry

Artist: Kinetic Publishing Services, LLC

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

Distributed to the book trade in the United States by Springer-Verlag New York, LLC, 233 Spring Street, 6th Floor, New York, NY 10013 and outside the United States by Springer-Verlag GmbH & Co. KG, Tiergartenstr. 17, 69112 Heidelberg, Germany.

In the United States: phone 1-800-SPRINGER, e-mail orders@springer-ny.com, or visit http://www.springer-ny.com. Outside the United States: fax +49 6221 345229, e-mail orders@springer.de, or visit http://www.springer.de.

For information on translations, please contact Apress directly at 2560 Ninth Street, Suite 219, Berkeley, CA 94710. Phone 510-549-5930, fax 510-549-5939, e-mail info@apress.com, or visit http://www.apress.com.

The source code for this book is available to readers at http://www.apress.com in the Downloads section.

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.

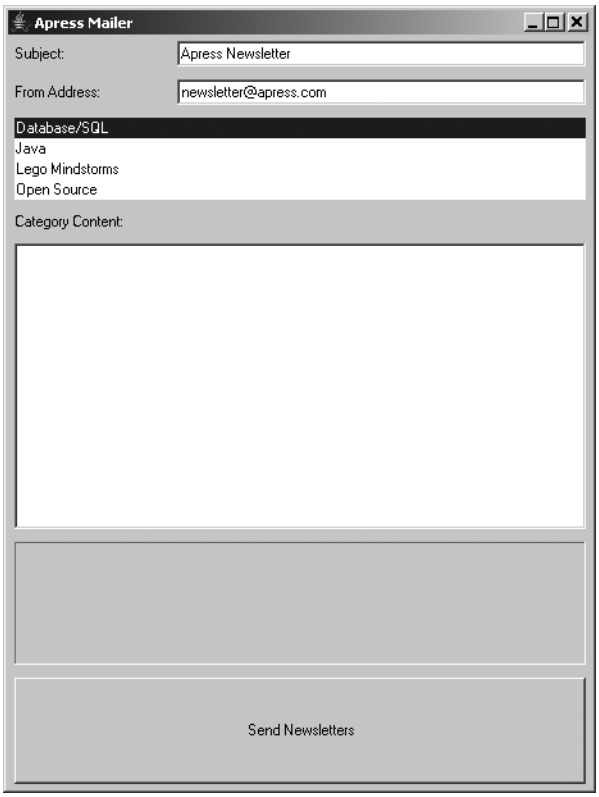

Figure 5-1. The finished application

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 an 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 his or her 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 very 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 in the 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 of the listbox has changed. 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 can 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 the 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.

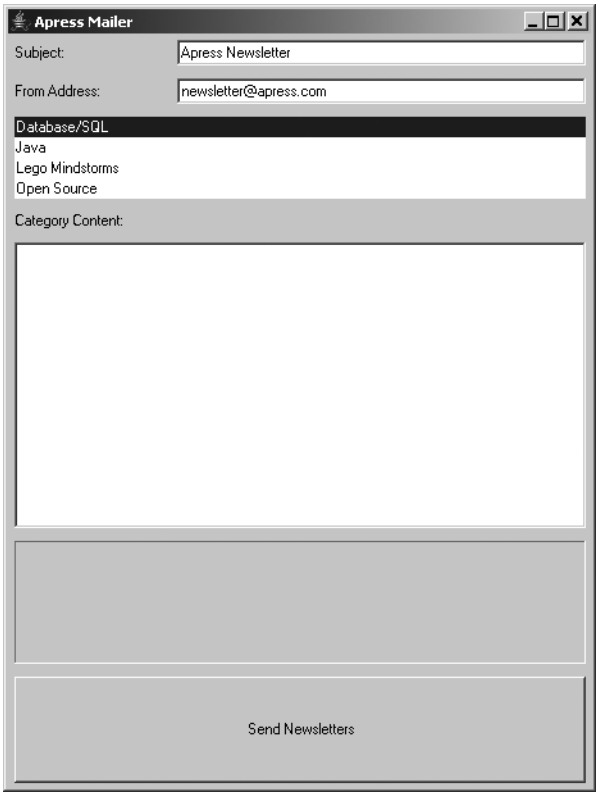

Figure 5-2. Application at startup

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.

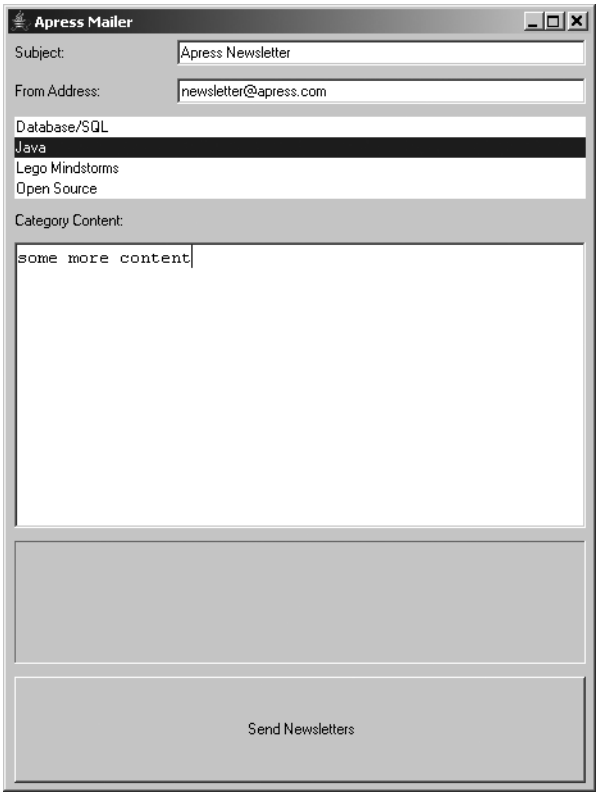

Figure 5-3. Adding category content

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.

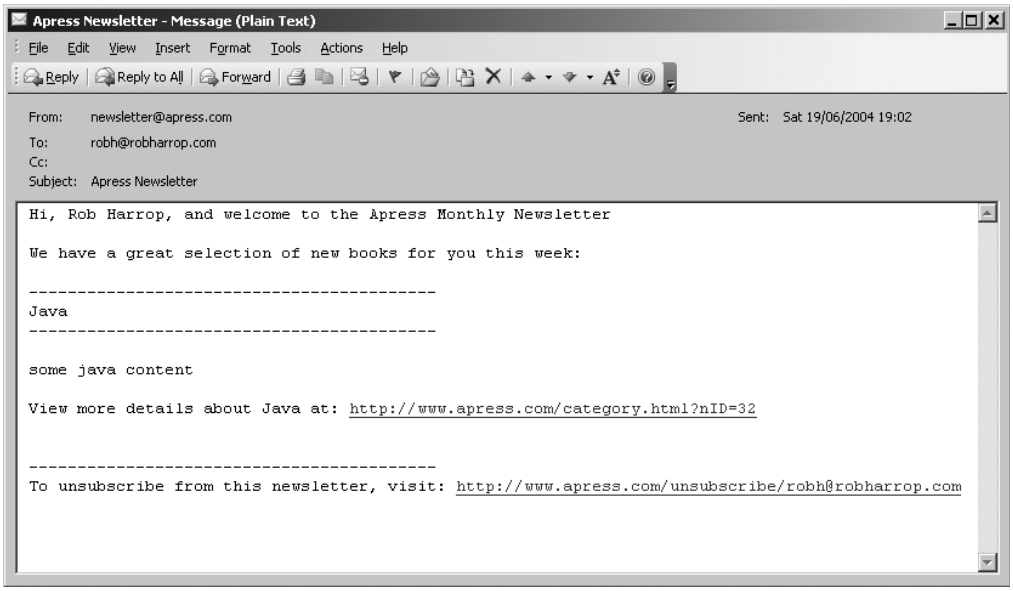

Figure 5-4. Plain-text e-mail

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

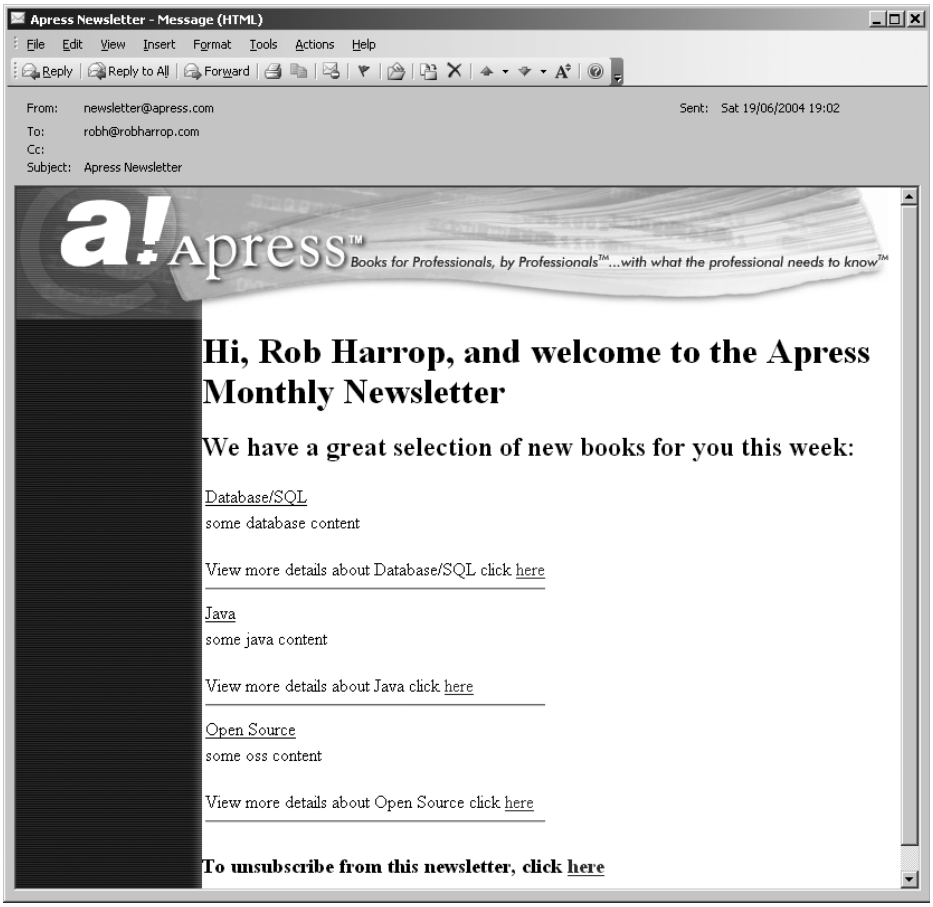

Figure 5-5. HTML e-mail

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.