THE UNIVERSITY OF WESTERN ONTARIO

Computer Science 2212b - Winter 2014
Introduction to Software Engineering

Lab 9 - Email with the JavaMail API

Part I
Preparing to Work with the JavaMail API

Estimated Time: 10-20 minutes

1 Introduction

The JavaMail API provides classes for both sending and receiving email in Java applications. It is bundled with the Java EE platform, but can be downloaded and used with Java SE. The API provides classes that can send email via an *SMTP server*, and supports encryption and authentication. Additionally, it provides the ability to retrieve email from *POP* and *IMAP* servers, thus allowing a Java program to retrieve mail from a remote server.

The *Simple Mail Transport Protocol (SMTP)*[1] is the standard protocol for transferring mail between hosts on the Internet. One sends an email to the SMTP server of one's ISP, and that server then routes it to the recipient's SMTP server. From there, the email is delivered into the mailbox of the recipient, who can then retrieve it through a *Post Office Protocol (POP)* server or, more commonly these days, an *Internet Mail Access Protocol (IMAP)* server. JavaMail makes switching between protocols very easy. If one writes a program to read mail from a POP server, for example, then changing the program to use an IMAP server largely requires changing only the name of the protocol in the code.

In this lab, we'll see how to setup a *dummy* SMTP server, allowing us to test the email functionality of our programs without having to actually send email to anyone. In general, it's a bad idea to test a program using a real SMTP server. Operators of SMTP servers take spam and abuse very seriously, and are quick to blacklist those who misuse their services, even if their intent was not malicious. Hence, we will send our email to a dummy SMTP server, but we will write the program such that we would only need to change the hostname of the SMTP server to that of a valid server in order to send "real" mail using our program.

Once we have a dummy server running, we'll write a program that can send email to the server in both HTML and text formats, and can attach files to the email it sends.

Finally, we'll see how to use the popular Apache Velocity templating engine to create email templates.

2 Getting Started

1. Change to your individual Git repository, create a `lab9` directory, and change to it:

```
$ cd ~/courses/cs2212/labs
$ mkdir lab9
$ cd lab9
```

2. Create a file `pom.xml` with the following elements:

   - `groupId`: `ca.uwo.csd.cs2212.USERNAME`
   - `artifactId`: `USERNAME-lab9`
   - `version`: `1.0-SNAPSHOT`

[1] An easy way to remember the purpose of an SMTP server is to use the mnemonic **S**end **M**ail **T**o **P**eople. After all, this is the purpose of an SMTP server!

Refer to lab 2 for the full details of creating a `pom.xml` (remember that you will also need a `modelVersion` tag). Of course, `USERNAME` should be replaced with your UWO username in **lower case**.

3. Edit the `pom.xml` file so that the JAR file that it produces is both executable and *fat* (refer to lab 2). Set the main class to `ca.uwo.csd.cs2212.USERNAME.App`.

4. Add the following dependency to your `pom.xml`:

groupId	artifactId	version
`javax.mail`	`mail`	`1.4.7`

5. Create the directory structure to host our project:

```
$ mkdir -p src/{main,test}/{java,resources}
```

6. Create the directory structure to house your Java package:

```
$ mkdir -p src/main/java/ca/uwo/csd/cs2212/USERNAME
```

   Again, `USERNAME` should be your UWO username in **lowercase**.

7. Download `App.java` from the following Gist and put it in the proper directory:
   https://gist.github.com/jsuwo/9224341.

   - Customize the package statement in the file.

3 Installing MailCatcher

As noted earlier, we will be using a *dummy* SMTP server to allow us to test email functionality without having to actually send mail to a real SMTP server. This is generally the fastest way to develop since there is latency associated with sending to a real server.

MailCatcher is a Ruby program that provides a dummy SMTP server, along with a web interface to view email sent to the server. When it is run, it will spawn an SMTP server on port 1025 (SMTP is normally run on port 25), along with a web server on port 1080. One can then send an email to the server and view the email by opening `http://localhost:1080` in a web browser. The mail can be addressed to any email address – MailCatcher will catch everything.

In this section, we will cover installing Ruby and the MailCatcher *gem* (the Ruby term for a packaged program or library). The instructions will differ depending on your operating system:

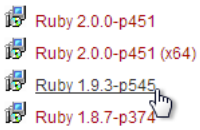
- Windows: see Section 3.1
- Linux: see Section 3.2
- OS X: see Section 3.3

3.1 Installing MailCatcher on Windows

If you are running Windows, you will need to install Ruby, along with a development kit. Once installed, you will be able to install MailCatcher. This section will detail the process.

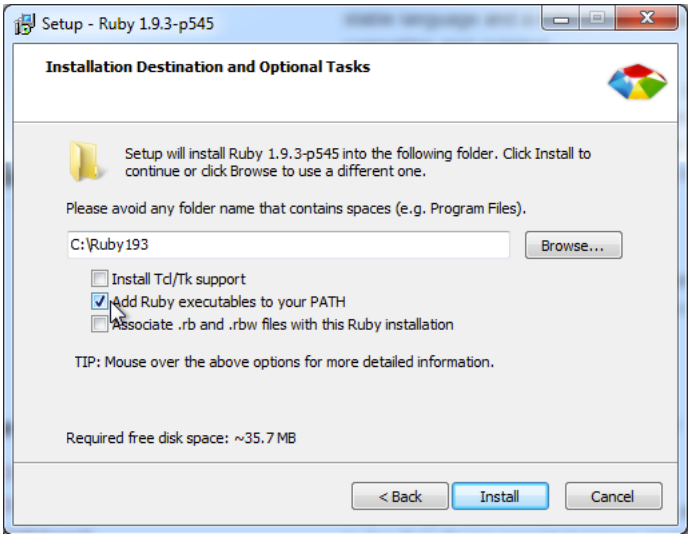
1. Go to http://rubyinstaller.org/downloads/
2. Download the latest version of Ruby 1.9.3 (currently Ruby 1.9.3-p545).

   - Do not download Ruby 2.0.0 as it is newly ported to Windows and not all libraries have been updated to work with it.

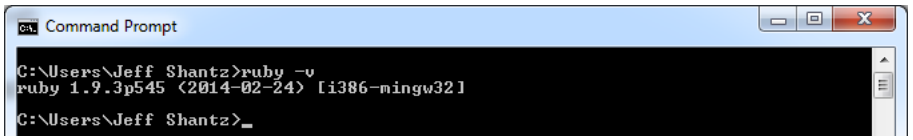


3. Run the installer and follow the prompts.

4. On the **Installation Destination and Optional Tasks** screen, be sure to check **Add Ruby executables to your PATH**.



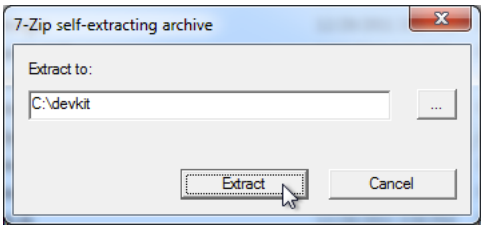
5. Open a terminal and type `ruby -v`. Ensure that it prints the version, meaning that Ruby is in your PATH.



6. Go back to http://rubyinstaller.org/downloads/

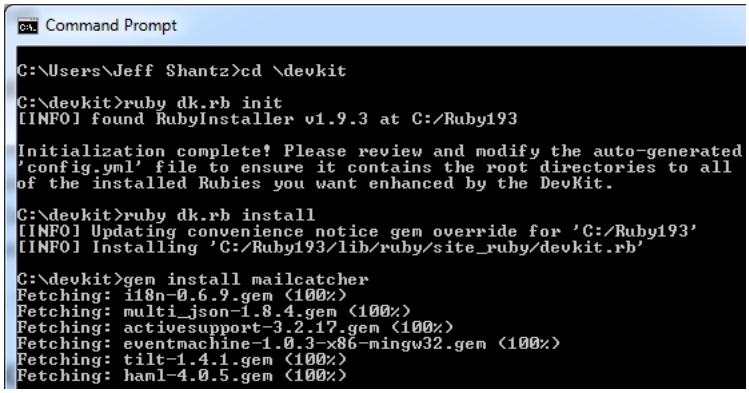
7. Scroll down to the **Development Kit** heading and download the kit for use with Ruby 1.9.3.

8. Run the executable, specify a path to which it should be extracted, such as `C:\devkit`, and click **Extract**.



9. Open a terminal and change to the directory where you extracted the development kit.
10. Run the following commands:

```
> ruby dk.rb init
> ruby dk.rb install
> gem install mailcatcher
```



11. Enter the command `mailcatcher` and press Enter.
12. Open a browser and go to http://localhost:1080. You should see the MailCatcher interface.

3.2 Installing MailCatcher on Linux

If you are running Linux, you may already have Ruby installed.

1. Open a terminal and type `ruby -v`. If you do not receive an error message, then Ruby is installed.
2. Otherwise, check your package manager and install Ruby 1.9.3. On Ubuntu 13.10, you can simply type `sudo apt-get install ruby`.
3. Once you have Ruby installed, type `sudo gem install mailcatcher`.
4. Enter the command `mailcatcher` and press Enter.
5. Open a browser and go to http://localhost:1080. You should see the MailCatcher interface.

3.3 Installing MailCatcher on OS X

If you are running OS X, you likely already have Ruby installed.

1. Open a terminal and type `ruby -v`. If you do not receive an error message, then Ruby is installed.
2. Otherwise, you can download and install MacRuby from http://macruby.org/.
3. Once you have Ruby installed, type `sudo gem install mailcatcher`.
    - If the `gem` command is not found, try replacing it with `macgem`.
4. Enter the command `mailcatcher` and press Enter.
5. Open a browser and go to http://localhost:1080. You should see the MailCatcher interface.

Part II
Sending Email with the JavaMail API

In this part, we'll build an application to send email using the JavaMail API. We will then iteratively improve it, adding support for multipart email, reading configuration from a `.properties` file, and adding file attachments to email sent by the application.

4 Getting Started with the JavaMail API

In Section 2, we added `App.java` to our project. This class provides a working example of using the JavaMail API to send an email. Let's take a few moments to explore the code.

```
52  public static void main(String[] args) throws Exception {
53
54    if (args.length == 0) {
55        System.out.println("Please specify a space-separated list of email addresses as arguments.");
56        System.exit(-1);
57    }
58
59    Session session = getSession();
60    sendMessage(session, args);
61  }
```

The program takes one or more email addresses as arguments. These addresses become the recipients of the email sent by the program. For instance, we might run the program as follows:

```
java -jar target/jarfile.jar jdoe@example.com juser@example.net ...
```

The program would then send an email to `jdoe@example.com` and `juser@example.net`.

Lines 54 – 57 perform some simple validation, ensuring that the user passed at least one email address to the program.

We then obtain a mail session object by calling `getSession`, and we send the message using the `sendMessage` method. Notice that `sendMessage` takes the `Session` object, along with the array of recipient email addresses passed as arguments to the program.

Let's look at the `getSession` method.

```
16  private static Session getSession() {
17
18    Properties props = new Properties();
19
20    props.put("mail.smtp.host", SMTP_SERVER);
21    props.put("mail.smtp.port", SMTP_PORT);
22    props.put("mail.smtp.auth", "true");
23    props.put("mail.smtp.starttls.enable", "true");
24
25    Session session = Session.getInstance(props,
26        new javax.mail.Authenticator() {
27            protected PasswordAuthentication getPasswordAuthentication() {
28                return new PasswordAuthentication(SMTP_USERNAME, SMTP_PASSWORD);
29            }
30        }
31    );
32
33    return session;
34  }
```

This method instantiates a `Properties` object that is used to specify the hostname of the SMTP server, and other details such as its port, and whether or not it requires authentication. MailCatcher does not require any authentication, and will simply ignore any authentication details passed to it. Nevertheless, authentication is set up in the code in order to demonstrate how you might use the code with another SMTP server, such as `smtp.gmail.com`.

On lines 25–31, we instantiate a new mail session, passing in the `Properties` object to the `Session.getInstance` method. We also instantiate and pass an `Authenticator`, which is used to pass a username and password to the SMTP server to authenticate with it.

```
36  private static void sendMessage(Session session, String[] recipients) throws Exception {
37
38    Message msg = new MimeMessage(session);
39
40    Address sender = new InternetAddress(SENDER_EMAIL, SENDER_NAME);
41    msg.setFrom(sender);
42
43    for (String address : recipients)
44        msg.addRecipient(Message.RecipientType.TO, new InternetAddress(address));
45
46    msg.setSubject("Hello");
47    msg.setText("Hello World");
48
49    Transport.send(msg);
50  }
```

Finally, we have the `sendMessage` method, which is passed the `Session` object, along with an array of recipient email addresses. We instantiate a new `MimeMessage` object on line 38, initializing it with the `Session` object so that the message effectively knows exactly how it is to be sent (e.g. the details of the SMTP server through which it will be sent). An `Address` object is created on line 40 to represent the sender of the email. This address is then set as the sender address of the message.

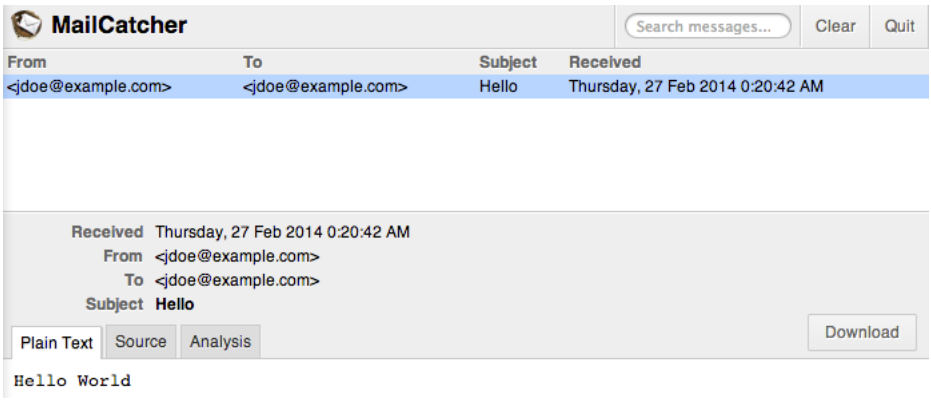
Next, on lines 43–44, we iterate over the array of recipient email addresses, adding them to the email message. Notice that we specify a type of `Message.RecipientType.TO` when calling the `addRecipient` message. We could also use `Message.RecipientType.CC` and `Message.RecipientType.BCC` here.

Finally, we set the subject and body of the email, and send the message using the `Transport.send` method.

1. Ensure that MailCatcher is running by issuing the `mailcatcher` command at a terminal.
2. Open a web browser and go to `http://localhost:1080`.
3. Package and run your program.

   ```
   java -jar target/jshantz4-lab9-1.0-SNAPSHOT-jar-with-dependencies.jar jdoe@example.com
   ```

4. You should see a message appear in the MailCatcher interface. If not, you are using an older browser that does not support websockets, and you will need to refresh the page.



That's it. We know how to send email. Lab done? Not quite.

5 Sending Multipart Email

In the last section, we saw a program capable of sending a basic plaintext email message. We frequently want to send *multipart* email, which contains both HTML and plaintext parts. This way, if one is using a modern email client, one can view an enhanced HTML version of the email, while if one is still using a text-based email reader like Pine or Mutt (yes, such people do still exist), one will see the plaintext version of the email. In this section, we'll see how to send multipart email.

1. Add the methods from `Multipart.java` from the following Gist to your `App` class: https://gist.github.com/jsuwo/9224341.
   - There is nothing particularly special about these methods. `getTextBody` returns a `String` containing a plaintext version of the email body, while `getHtmlBody` returns an HTML version.

2. Remove the `msg.setText()` call from `sendMessage` and modify the method to look as follows:

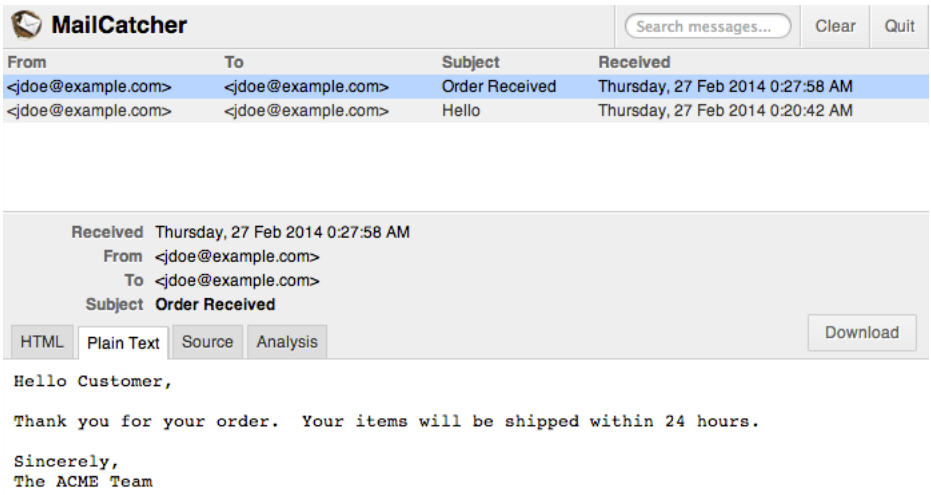
```
.
.
msg.setSubject("Order Received");

  Multipart multiPart = new MimeMultipart();

  MimeBodyPart textPart = new MimeBodyPart();
  textPart.setText(getTextBody(), "utf-8");

  MimeBodyPart htmlPart = new MimeBodyPart();
  htmlPart.setContent(getHtmlBody(), "text/html; charset=utf-8");

  multiPart.addBodyPart(textPart);
  multiPart.addBodyPart(htmlPart);

  msg.setContent(multiPart);

  Transport.send(msg);
```

   Here, we instantiate a new `Multipart` object, to which we add two `MimeBodyPart` objects. The first `MimeBodyPart` is initialized with the plaintext version of the email (obtained from `getTextBody`), while the second is passed the HTML version (obtained from `getHtmlBody`). We then add the body parts to the `multiPart` object and set the content of the message to be the `multiPart` object.

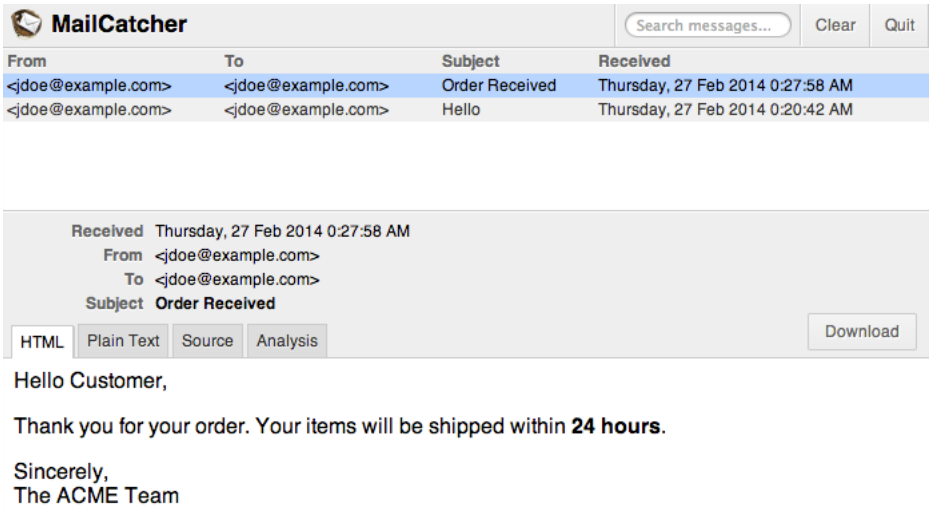
   Note that the order can sometimes be important here to some mail readers: add the plaintext part first, and then the HTML part.

3. Package and run your code.

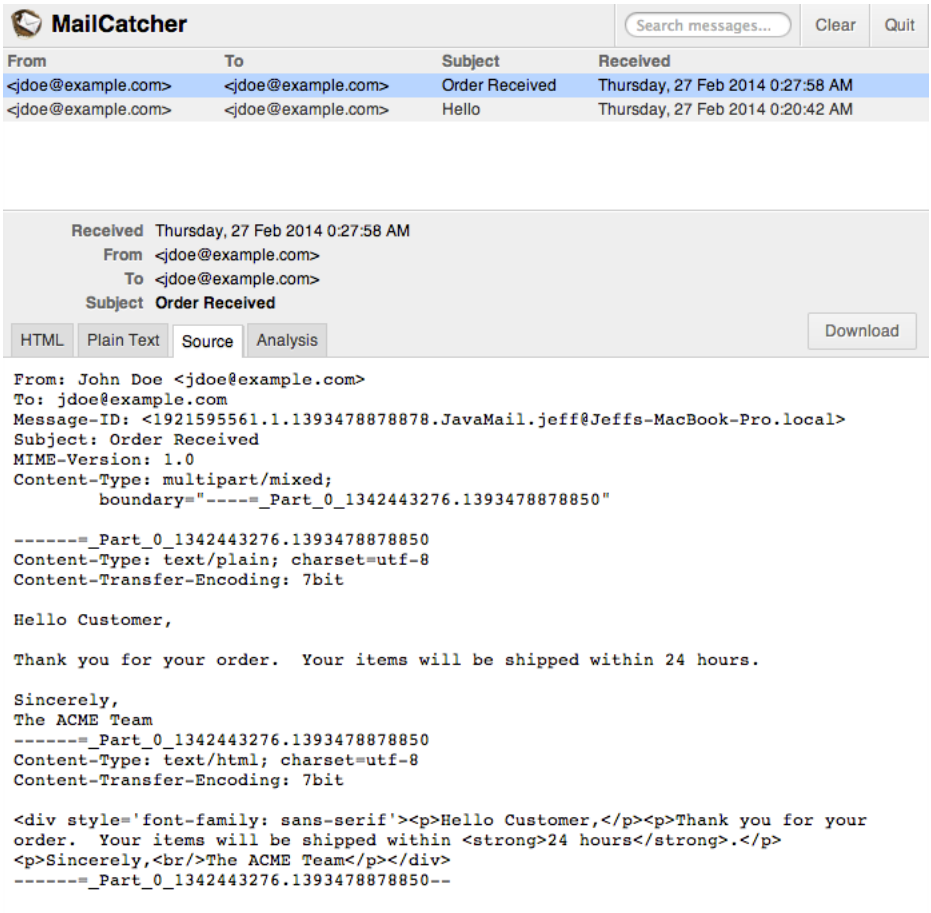
4. Open the message in MailCatcher and select the **Plain Text** tab. The plaintext version of the email should appear correctly.



5. Select the **HTML** tab. The HTML version should appear in a sans-serif font with bold text on **24 hours**.



6. Select the **Source** tab. Notice that the two parts of the message are visible in the message source, separated by a *boundary* line. The boundary is specified in the `Content-Type` *header* of the message and tells the email reader how the parts of the message are delineated. Notice that each part also has a `Content-Type` header, which tells the email reader the type of part it represents (e.g. `text/plain`, `text/html`). This way, an email reader that doesn't support HTML mail, for example, can skip any parts with a `Content-Type` of `text/html`, and simply display the `text/plain` parts.



6 Loading Configuration from a `.properties` File

We know how to send a message, but our code leaves much to be desired. We have values like the SMTP server hostname and our username and password hard-coded into the program. Normally, we would ask the user for these details and store them somewhere, perhaps using the Java Preferences API. For simplicity here, we're going to put all of our configuration in a `.properties` file that can then be stored in the JAR file and loaded at runtime. This way, we won't have to hard-code all of our configuration.

`.properties` files are simple text files commonly used to pass properties to Java programs. They simply consist of lines containing `key=value` statements. For instance, we might have a file `user.properties` with the following:

```
user.name=Jeff Shantz
user.course_taught=CS 2212b
```

We can then load and access properties in this file using the `java.util.Properties` class:

```
Properties properties = new Properties();
InputStream stream = new FileInputStream("user.properties");

// Load the properties from the file
properties.load(stream);

System.out.println("Name     : " + properties.getProperty("user.name"));
System.out.println("Teaching : " + properties.getProperty("user.course_taught"));
```

In this section, we'll similarly load our configuration from a `.properties` file, but instead of loading it from a file in the filesystem as above, we'll load it from a file contained in our JAR file.

1. Create a file `src/main/resources/mail.properties` with the following content:

   ```
   smtp.host=localhost
   smtp.username=username
   smtp.password=password
   mail.smtp.port=1025
   mail.smtp.auth=true
   mail.smtp.starttls.enable=true
   sender.name=John Doe
   sender.email=jdoe@example.com
   message.subject=Order Received
   ```

   Notice that we provide all of the SMTP server configuration in this file, along with other details such as the sender's name and email address, and the subject of the email.

2. Edit `App.java` and add the following method:

   ```
   private static Properties loadProperties() throws Exception {
     Properties properties = new Properties();
     InputStream stream = App.class.getClassLoader().getResourceAsStream("mail.properties");
     properties.load(stream);

     return properties;
   }
   ```

   Here, we declare a method `loadProperties` that loads the `mail.properties` file from the JAR file into a `Properties` object, and returns the object. **Note**: you will need to import `java.io.InputStream` in order for this to compile.

3. Edit the `main` method to look as follows:

   ```
     .
     .
     .
     Properties properties = loadProperties();
     Session session = getSession(properties);
     sendMessage(session, properties, args);
   ```

   We first load the properties, and then pass them when calling `getSession` and `sendMessage`.

4. Edit `getSession` to look as follows:

```
private static Session getSession(final Properties properties) {

    Session session = Session.getInstance(properties,
        new javax.mail.Authenticator() {
            protected PasswordAuthentication getPasswordAuthentication() {

                String username = properties.getProperty("smtp.username");
                String password = properties.getProperty("smtp.password");
                return new PasswordAuthentication(username, password);
            }
        }
    );

    return session;
  }
```

Notice that we no longer need to instantiate a `Properties` object in the method, since we're passing it in. Additionally, we can remove the `SMTP_USERNAME` and `SMTP_PASSWORD` constants, since we can simply load this information from the `properties` object.

One thing you might be wondering about is why the `properties` parameter is marked `final`. This is because we're accessing the `properties` object within the `Authenticator` class, which is an *inner class*. Java does not allow the use of a non-`final` variable from the parent class within an inner class.

5. Edit `sendMessage` to look as follows:

```
private static void sendMessage(Session session, Properties properties, String[] recipients) throws
     Exception {

    Message msg = new MimeMessage(session);
    String senderName = properties.getProperty("sender.name");
    String senderEmail = properties.getProperty("sender.email");

    Address sender = new InternetAddress(senderEmail, senderName);
    msg.setFrom(sender);

    for (String address : recipients)
        msg.addRecipient(Message.RecipientType.TO, new InternetAddress(address));

    msg.setSubject(properties.getProperty("message.subject"));

    Multipart multiPart = new MimeMultipart();

    MimeBodyPart textPart = new MimeBodyPart();
    textPart.setText(getTextBody(), "utf-8");

    MimeBodyPart htmlPart = new MimeBodyPart();
    htmlPart.setContent(getHtmlBody(), "text/html; charset=utf-8");

    multiPart.addBodyPart(textPart);
    multiPart.addBodyPart(htmlPart);

    msg.setContent(multiPart);

    Transport.send(msg);
  }
```

Once again, we remove the use of hard-coded constants, preferring to load the configuration from the `properties` object.

6. Remove all of the `final` variables at the top of the class, since we no longer need them, and bask in the glow of your newly-refactored, extensible code, which is devoid of hard-coded data.

```
public class App {

  // Remove these
  private final static String SMTP_USERNAME = "username";
  private final static String SMTP_PASSWORD = "password";
  private final static String SMTP_SERVER   = "localhost";
  private final static String SMTP_PORT     = "1025";
  private final static String SENDER_EMAIL  = "jdoe@example.com";
  private final static String SENDER_NAME   = "John Doe";
```

7. Package your code and run your program. Verify that you receive an email in MailCatcher and that nothing has changed from the last email received.

7 Adding a File Attachment

We often need to attach files to email sent from a program. In this section, we'll see how to add a file attachment to the email the program sends. In this case, we'll just attach a `.png` file of the company's logo.

1. Download `logo.png` from the following Gist and place it in your `src/main/resources` directory: https://gist.github.com/jsuwo/9224341

2. Edit the imports at the top of `App.java` to look as follows:

```
import java.io.*;
import javax.activation.*;
import java.net.URL;
import javax.mail.*;
import javax.mail.internet.*;
import java.util.*;
```

3. Edit the `sendMessage` method to look as follows:

```
    .
    .
    htmlPart.setContent(getHtmlBody(), "text/html; charset=utf-8");

    MimeBodyPart fileAttachmentPart = new MimeBodyPart();

    URL attachmentUrl = App.class.getClassLoader().getResource("logo.png");
    DataSource source = new URLDataSource(attachmentUrl);
    fileAttachmentPart.setDataHandler(new DataHandler(source));
    fileAttachmentPart.setFileName("logo.png");

    multiPart.addBodyPart(textPart);
    multiPart.addBodyPart(htmlPart);
    multiPart.addBodyPart(fileAttachmentPart);
    .
    .
```

   A file attachment is simply another part of the message body. We first obtain the URL of the file to attach from the JAR file. We then instantiate a `DataSource` object from the URL, and set the *data handler* of the body part accordingly. In a nutshell, these lines simply read the data from the logo file stored in the JAR and store the data in the file attachment body part.

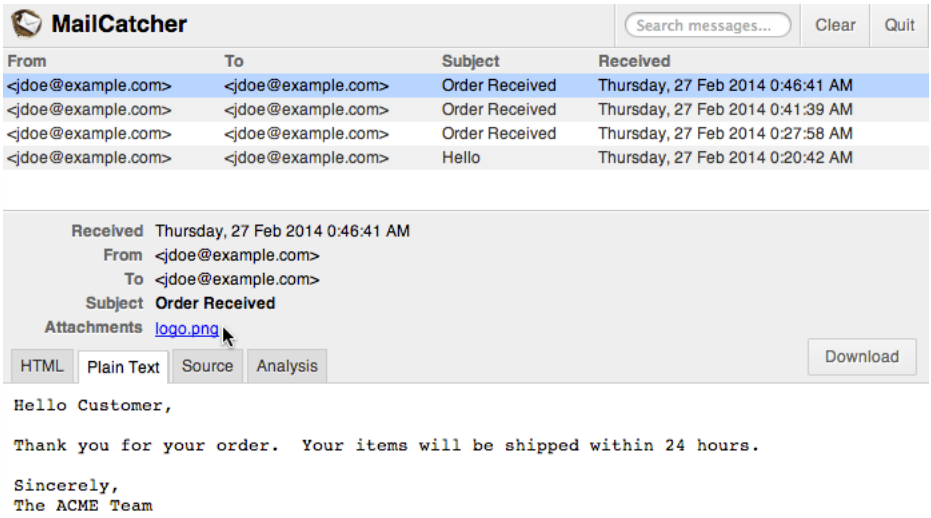
   We then set the filename of the attachment, which will be the name that is displayed in one's email client when viewing the email. Finally, we add the file attachment body part to the email, just as we did with the plaintext and HTML parts.

   Of course, here, we are loading the file from a JAR file. In many cases, we might be loading a file from the filesystem. In this case, instead of retrieving the URL of the file from the JAR and instantiating a `URLDataSource`, you would instead instantiate a `FileDataSource`, passing the filename directly to its constructor, e.g.,

```
    DataSource source = new FileDataSource("/home/jeff/logo.png");
    fileAttachmentPart.setDataHandler(new DataHandler(source));
    fileAttachmentPart.setFileName("logo.png");
```

4. Package your code and run your program.

5. Verify that `logo.png` has been attached to the email received in MailCatcher.



8 Email Templates with Apache Velocity

We've removed a lot of hard-coded configuration from our program, but we are still hard-coding the body of the email in the `getTextBody` and `getHtmlBody` methods. Ideally, we would like to store the email body outside the program so that we could change it in the future without having the recompile the code.

To do this, we could simply store the plaintext and HMTL bodies in simple text files that we could add to the `src/main/resources` directory of our project. This way, they would be embedded in the JAR file and we could simply read them in at run-time.

While this would work, we often require more sophistication. As an example, we might want to replace certain data in the email body to customize it according to the recipient of the email. For instance, we might want to customize the name in the email body (e.g. "Hi Joe,") or we might want to include a list of all the products that a particular customer has ordered.

One solution to this would be to put placeholders in the text files and then replace them at run-time, e.g.

```
Hi ##NAME##,

You have ordered the following products:

###PRODUCT_LIST###
```

We could then replace the placeholders with real data at run-time using regular expressions. However, this seems like a lot of work to do manually. What we need is some sort of *templating engine* that can handle all the replacement for us.

Fortunately, many such engines exist. Apache Velocity is a widely-used templating engine, allowing us to create templates and then populate them with data at run-time. While we will use Velocity for email templates, it can be used for many purposes such as generating web pages, text files, and more.

In this section, we'll extract our plaintext and HTML body parts out into Apache Velocity templates, and see how to load and populate them with data at run-time.

1. Add the following dependency to your `pom.xml`:

groupId	artifactId	version
`org.apache.velocity`	`velocity`	`1.7`

2. Download `email.text.vm1` and `email.html.vm1` from the following Gist and place them in your `src/main/resources` directory: https://gist.github.com/jsuwo/9224341

3. Rename the files to `email.text.vm` and `email.html.vm`.

4. Examine the templates. Notice that we have used one variable `$companyName` in the templates. We can specify a value for this variable at run-time when rendering the template.

```
Hello Customer,

Thank you for your order.  Your items will be shipped within 24 hours.

Sincerely,
The $companyName Team
```

   In practice, we would probably also use a variable for the customer's name rather than writing `Hello Customer`, but since we aren't using any customer objects in this lab, we will overlook this for the time being.

5. Remove the `getTextBody` and `getHtmlBody` methods from the `App` class, and once again relish in the glory and power of removing hard-coded data from your program, and therefore making it more extensible.

6. Add the following imports to `App.java`:

```
import org.apache.velocity.Template;
import org.apache.velocity.VelocityContext;
import org.apache.velocity.app.VelocityEngine;
import org.apache.velocity.runtime.RuntimeConstants;
import org.apache.velocity.runtime.resource.loader.ClasspathResourceLoader;
```

7. Add the following method to the `App` class:

```
17  private static String loadTemplate(String filename) {
18
19    VelocityEngine ve = new VelocityEngine();
20    ve.setProperty(RuntimeConstants.RESOURCE_LOADER, "classpath");
21    ve.setProperty("classpath.resource.loader.class", ClasspathResourceLoader.class.getName());
22
23    Template template = ve.getTemplate(filename);
24
25    VelocityContext context = new VelocityContext();
26    context.put("companyName", "ACME");
27
28    StringWriter out = new StringWriter();
29    template.merge(context, out);
30
31    return out.toString();
32  }
```

   The `loadTemplate` method takes the name of the file in which the template is stored. On line 19, we instantiate a `VelocityEngine` object, which we will use to load and render the template. Lines 20–21 are needed only because we are loading the templates from a JAR file and are otherwise of little interest.

   We then load the specified template on line 23. Next, we create a `VelocityContext` object, which will be used to pass data to the template. The context is essentially a `Map<String, Object>`, mapping `String` keys to `Object`s. The keys added to the context can then be used as variables in the template. For instance, on line 26, we add the key `companyName` to the template, specifying its value to be `ACME`. The template engine will then replace all instances of `$companyName` in the template with `ACME`.

   Finally, on lines 28–31, we render the template to a `StringWriter`, and return the rendered template as a `String`.

8. Edit the `sendMessage` method, and replace the declarations of `getTextBody` and `getHtmlBody` as shown below:

```
    .
    .
    MimeBodyPart textPart = new MimeBodyPart();
    textPart.setText(loadTemplate("email.text.vm"), "utf-8");

    MimeBodyPart htmlPart = new MimeBodyPart();
```

```
htmlPart.setContent(loadTemplate("email.html.vm"), "text/html; charset=utf-8");
.
.
```

Observe that, instead of calling `getTextBody` and `getHtmlBody`, we now make two calls to `loadTemplate` to load and render the plaintext and HTML templates.

9. Package your code and run your program. Observe that the `$companyName` variable has been replaced in the body of the message, as expected.

9 Iteration in Apache Velocity

Apache Velocity offers much more than just variable replacement. We can make use of conditional statements and loops, define macros, and call arbitrary Java methods from our templates.

In this section, we'll finish up the lab by adding a `Product` class to our program, which stores details of a product that a customer has purchased. We will then pass a `List` of `Product` objects to the Velocity engine and iterate over the products in our email templates.

1. Download `Product.java` from the following Gist and place it in the proper directory: https://gist.github.com/jsuwo/9224341
   - Customize the package statement in the file.
2. Edit the `loadTemplate` method in the `App` class to look as follows:

```
.
.
List<Product> productsOrdered = new ArrayList<Product>();

productsOrdered.add(new Product("Xbox 360", 299.0));
productsOrdered.add(new Product("The Saboteur", 26.23));
productsOrdered.add(new Product("Red Dead Redemption", 20.99));

VelocityContext context = new VelocityContext();
context.put("companyName", "ACME");
context.put("productsOrdered", productsOrdered);
.
.
```

We create a `List` of `Product` objects representing products that the customer has ordered. We then add this list to the `context`.

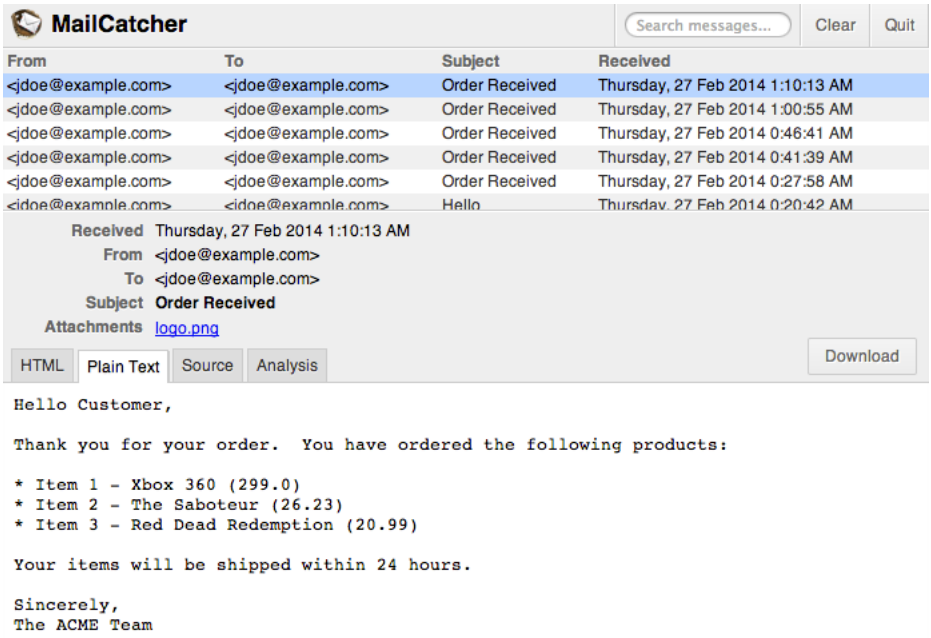
3. Download `email.text.vm2` and `email.html.vm2` from the following Gist and replace the contents of `email.text.vm` and `email.html.vm` with their contents: https://gist.github.com/jsuwo/9224341
4. Take a moment to examine the templates.

```
Hello Customer,

Thank you for your order.  You have ordered the following products:

#foreach($product in $productsOrdered)
* Item $velocityCount - $product.getName() ($product.getPrice())
#end

Your items will be shipped within 24 hours.

Sincerely,
The $companyName Team
```

Observe that we are using the `#foreach` keyword to iterate over the `$productsOrdered` list. For each product, we print a bullet (an asterisk in the plaintext template) along with the product's name and price. Notice that we are calling Java methods from the template.

The `$velocityCount` variable is a special variable provided by Velocity within a `foreach` loop. It gives us the current iteration number and will therefore yield `1` on the first iteration, `2` on the second iteration, and so on.

5. Package your code and run your program. You should now have a list of products ordered in the email received.



We're nearly finished, but there's one problem that has arisen: the price is not formatted properly. To wrap things up, we'll use a Velocity extension to format the price so that it has a currency symbol and two decimal places.

6. Add the following dependency to your `pom.xml`:

groupId	artifactId	version
`velocity-tools`	`velocity-tools`	`1.4`

7. Add the following import to the `App` class:

```
import org.apache.velocity.tools.generic.NumberTool;
```

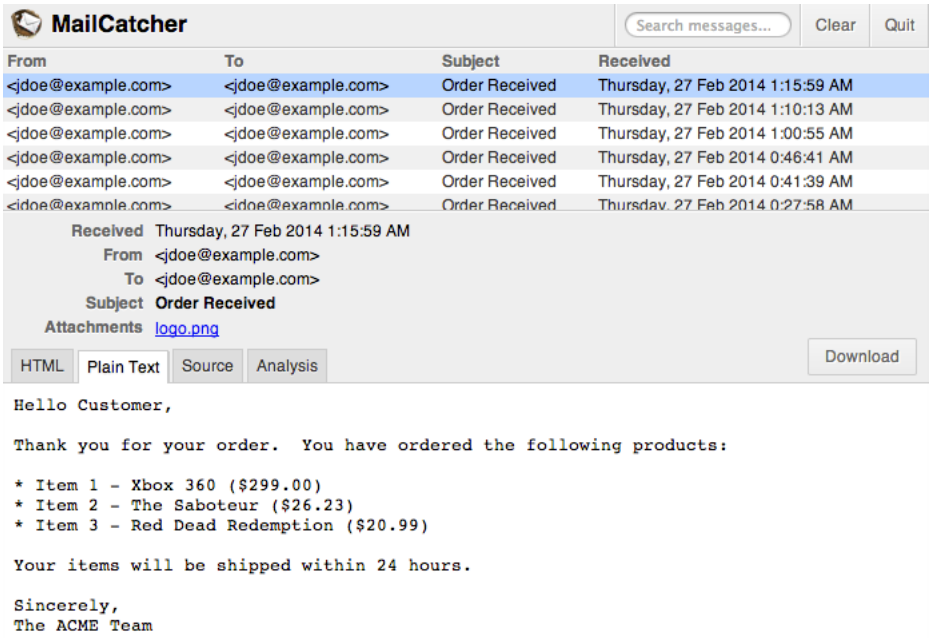
8. Edit the `loadTemplate` method to add an additional variable to the `context`:

```
context.put("number", new NumberTool());
```

9. Edit the `email.text.vm` and `email.html.vm` templates and change the `$product.getPrice()` calls to the following:

```
$number.format("currency", $product.getPrice())
```

10. Package your code and run your program. The prices should now be properly formatted in both the plaintext and HTML templates.



10 JavaMail and Apache Velocity Resources

Both JavaMail and Apache Velocity have a great deal of functionality that is beyond the scope of this lab. For more information, see the following resources:

- http://www.oracle.com/technetwork/java/javamail/index-141777.html
- http://www.oracle.com/technetwork/java/javamail-1-149769.pdf
- https://velocity.apache.org/engine/devel/user-guide.html

11 Submitting Your Lab

Commit your code and push to GitHub. To submit your lab, create the tag `lab9` and push it to GitHub. For a reminder on this process, see Lab 1.