4

Isolating Locale-Specific Data with Resource Bundles

Error messages, status messages, graphic images, and even sound clips are perfect examples of candidate elements for localization. For instance, all text messages displayed to users throughout your program first need to be translated, or localized, for the locales that you're going to support. This task can be very time consuming and costly. After completing the task, how do you package these messages together so the proper locale-specific information gets displayed to the user?

Java's `ResourceBundle` class provides the functionality for you to retrieve locale-specific resources. You can use resource bundles in your application to keep text messages, formatting conventions, images, sound files, etc., in a "package" targeted to different locales. It is possible to use the `ResourceBundle` class by providing your own subclass; however, two concrete implementations of `ResourceBundle` exist: `PropertyResourceBundle` and `ListResourceBundle`. These two classes provide additional functionality over the base `ResourceBundle` class.

Why Use Resource Bundles?

Applications need to provide locale-sensitive information to users. One approach is to hardcode locale-sensitive information or messages within the application. However, this approach is inefficient because it makes your application larger, complicates your application logic when you decide which messages to display, and requires that you change or release a new version of your application when locale-specific resources change. Another approach, Java resource bundles, provides facilities for packaging and managing collections of locale-specific elements, such

as feedback or status messages. These elements may also be more complex datatypes, such as graphic images or sound files. The resource bundle facilities provide a convenient mechanism for you to separate your program code from your resource data.

Let's outline some general resource bundle properties. These properties are illustrated in Figure 4-1. Resource bundles:

- Provide facilities for storage and retrieval of all locale-specific information.
- Allow you to support multiple locales in a single application.
- Allow you to support additional locales easily in the future by simply adding more resource bundles.

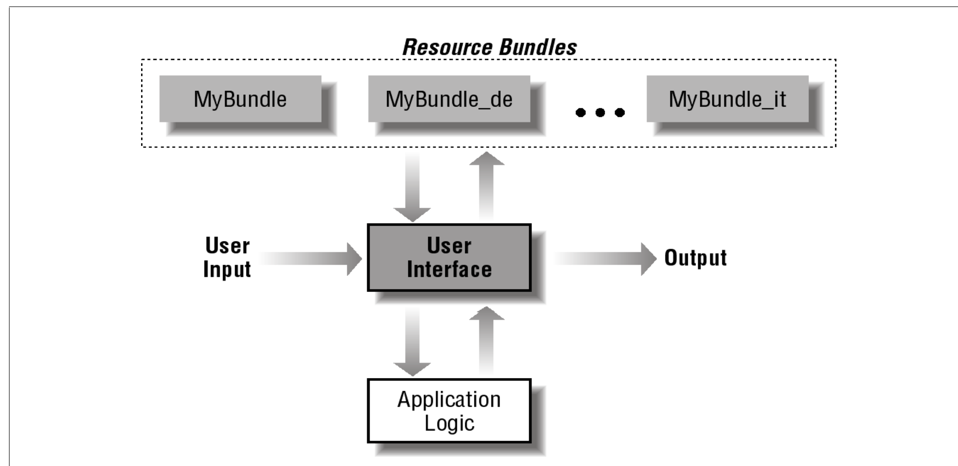

Figure 4-1. How resource bundles fit into application development

The ResourceBundle Class

The `ResourceBundle` class is an abstract class in the `java.util` package. You can provide your own subclass of `ResourceBundle` or use one of the subclass implementations, as in the case of `PropertyResourceBundle` or `ListResourceBundle`. We discuss each of these two classes in later sections of this chapter. For now, let's discuss some details of `ResourceBundle`.

Resource bundles inherit from the `ResourceBundle` class and contain localized elements that are stored external to an application. Resource bundles share a base name. The base name `SimpleResourceBundle` might be selected because of the

resources it contains. We talk about design considerations for resource bundles in the last section of this chapter. Locale information further differentiates a resource bundle. For example, `SimpleResourceBundle_it` means that this resource bundle contains locale-specific items for Italian. The exact naming convention for resource bundles is detailed in the next subsection.

Example 4-1 is a simple subclass of `ResourceBundle`.

Example 4-1. Simple Resource Bundle

```
import java.util.Enumeration;
import java.util.ResourceBundle;
import java.util.StringTokenizer;

public class SimpleResourceBundle extends ResourceBundle {

  private String keys = "AMMessage PMMessage";

  public Object handleGetObject(String key) {
    if (key.equals("AMMessage")) return "Good morning";
    if (key.equals("PMMessage")) return "Good evening";

    return null;
  }

  public Enumeration getKeys() {
    StringTokenizer keyTokenizer = new StringTokenizer(keys);

    return keyTokenizer;
  }
}
```

Example 4-2 is a subclass of `SimpleResourceBundle` and contains resources specific to Italian.

Example 4-2. Simple Resource Bundle Containing Localized Strings in Italian

```
public class SimpleResourceBundle_it extends SimpleResourceBundle {

  public Object handleGetObject(String key) {
    if (key.equals("AMMessage")) return "Buona mattina";
    if (key.equals("PMMessage")) return "Buona sera";

    return null;
  }
}
```

A resource bundle is a container for key/value pairs. The key is used to identify a locale-specific resource in a bundle. If that key is found in a particular resource bundle, its value is returned. Two abstract methods must be implemented in a subclass of `ResourceBundle`: `getKeys` and `handleGetObject`. The `handleGetObject(String key)` method takes a `String` as its argument. This argument is used to retrieve a particular value from the resource bundle. In Example 4-1, if a key of `AMMessage` is passed into the `handleGetObject` method, a value of `Good morning` is returned. This method returns `null` to indicate that no value could be found for the given key.

The `getKeys` method returns an enumeration containing all keys that a particular resource bundle knows about. In Example 4-1, we only manage two keys, `AMMessage` and `PMMessage`. We return a `StringTokenizer` object, which implements the `Enumeration` interface, that parses the `keys` string for the `SimpleResourceBundle` class.

We had to override only the `handleGetObject` method in `SimpleResourceBundle_it` because only our values had changed. The keys used to retrieve those values do not change. If only the `AMMessage` value had changed, we would have included only the line `if (key.equals("AMMessage")) return Buona mattina`.

Three methods in the `ResourceBundle` class retrieve the value for a given key. Two of these methods are convenience methods that save you from having to cast the object returned to you. The first method, `getObject(String key)`, takes a `String` used as the key and tries to retrieve a value from the resource bundle. The second method, `getString(String key)`, also takes a `String` used as the key and tries to retrieve a value from the resource bundle and return the object casted to a `String`. The third method, `getStringArray(String key)`, exhibits the same behavior as `getObject` and `getString`; however, this method attempts to cast the object returned to you as a `String` array (i.e., `String []`).

Example 4-3 uses `SimpleResourceBundle` and `SimpleResourceBundle_it` and demonstrates how to retrieve values from them.

Example 4-3. Demonstrating the Use of the ResourceBundle Class

```
import java.util.Enumeration;
import java.util.ResourceBundle;
import java.util.MissingResourceException;

public class SimpleResourceBundleExample {

  public static void main(String [] argv) {
    try {
      ResourceBundle rb = ResourceBundle.getBundle("SimpleResourceBundle");
```

Example 4-3. Demonstrating the Use of the ResourceBundle Class (continued)

```
      System.out.println(rb.getString("AMMessage"));
      System.out.println(rb.getString("PMMessage"));

    } catch (MissingResourceException mre) {
      mre.printStackTrace();
    }
  }
}
```

Here is the output produced by Example 4-3:

```
C:\>java SimpleResourceBundleExample

Good morning
Good evening

C:\>java -Duser.language=it SimpleResourceBundleExample

Buona mattina
Buona sera
```

Our next section describes how to retrieve resource bundles for use in an application and how resource bundles are named. We've already seen this process in Example 4-3 by using the `getBundle` method in the `ResourceBundle` class and by providing a localized resource bundle for Italian.

How Resource Bundles Are Discovered and Named

You must make resource bundles explicitly available in your applications. You can achieve this by using one of three static `getBundle()` methods in `ResourceBundle` that allow you to retrieve a resource bundle:

```
public final static getBundle(String baseName)
public final static getBundle(String baseName, Locale locale)
public final static getBundle(String baseName, Locale locale,
                              ClassLoader loader)
```

Using `getBundle(String baseName)` returns a resource bundle using the default locale for the running JVM. This method is useful if you allow users to select or set their locale before runtime while not allowing them to change the locale during runtime. However, we recommend that you always use the static method, `getBundle(String baseName, Locale locale)`. This method is an easy way to eliminate debugging headaches. If your application uses its own class loader, you

can use the `getBundle(String baseName, Locale locale, ClassLoader loader)` method and pass in the class loader used by your application.

The `ResourceBundle` class uses a particular lookup mechanism to find the individual resource bundles you request through one of the `getBundle` methods. Resource bundles can be thought of as having parents and children. The resource bundle naming hierarchy forms a tree structure. The name given to resource bundles is a combination of the base name and locale information. The base name you've picked for your bundles is at the root of the tree. In Example 4-1, this name was `SimpleResourceBundle`.

You further differentiate at the next level with language specific resources. We provided such a bundle in Example 4-2 with `SimpleResourceBundle_it`, to identify resources specific to Italian. Below that level, bundles are differentiated by country, and beyond that by the locale variant. Therefore, we could have provided `SimpleResourceBundle_it_IT` to localize Italian language resources for the country of Italy. Finally, we could have provided `SimpleResourceBundle_it_IT_Capri` to identify localized resources in Italian that are specific to the island of Capri.

The `getBundle` method searches for a particular resource bundle in the following order:

1. The desired or requested locale passed to the `getBundle` method
2. The default locale for the running JVM
3. The root resource bundle, or `baseName`, that you supply

The search proceeds in the following order from a lower, more specific level in the resource bundle name hierarchy to a higher, more generic level. The order is as follows:

1. baseName + "_" + desiredLocale.language + "_" + desiredLocale.country + "_" + desiredLocale.variant
2. baseName + "_" + desiredLocale.language + "_" + desiredLocale.country
3. baseName + "_" + desiredLocale.language
4. baseName + "_" + defaultLocale.language + "_" + defaultLocale.country + "_" + defaultLocale.variant
5. baseName + "_" + defaultLocale.language + "_" + defaultLocale.country
6. baseName + "_" + defaultLocale.language
7. baseName

Suppose that someone running Example 4-3 is interested in obtaining the SimpleResourceBundle for the locale "fr_FR." Further, suppose that the default locale is "en_US." The resource bundle lookup will search for the following classes:

```
SimpleResourceBundle_fr_FR
SimpleResourceBundle_fr
SimpleResourceBundle_en_US
SimpleResourceBundle_en
SimpleResourceBundle
```

The resource bundles you use in your application must be available from a directory in the classpath of the JVM. If not, the bundles will not be found and an exception will be thrown when one of the getBundle methods is called.

The ResourceBundle class also provides a convenience method called getLocale(). This method should be called when you want to explicitly check the locale of the resource bundle that is returned to you after calling one of the getBundle() methods. This way, you can determine whether the returned resource bundle matches the desired locale exactly or if one of the fallback resource bundles is in the bundle hierarchy.

Package names also affect how resource bundles are found. The name you supply as baseName in the getBundle() methods is the name of the class containing the resources, and this name must be fully qualified. Therefore, you must also supply the package name when retrieving a resource bundle that exists in a package. Imagine the class hierarchy illustrated in Figure 4-2.

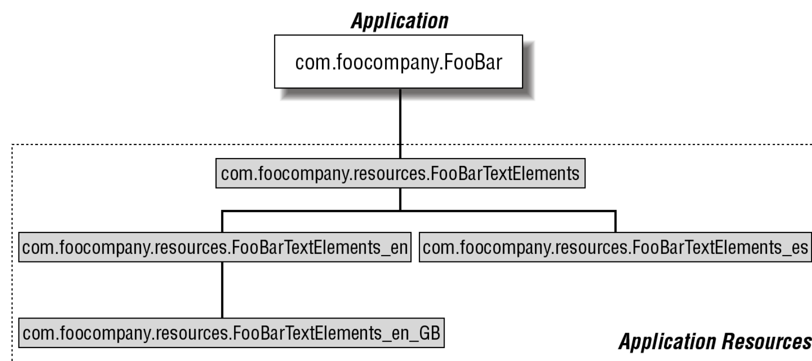

Figure 4-2. Resource bundle hierarchy using packages

Your application contains a class called `FooBar` in the `com.foocompany` package. This class uses the `getBundle(String baseName, Locale locale)` method to obtain resource data from the `FooBarTextElements` class for different locales. You've also provided a number of resource bundles for text elements in the `com.foocompany.resources` package. If the method call in the `FooBar` class looks like `ResourceBundle.getBundle("FooBarTextElements", userLocale)` a `MissingResourceException` is going to be thrown because the fully qualified name (package name and class name) needs to be passed into the `getBundle()` method.

The proper way to get the resource bundle in this situation is to use `ResourceBundle.getBundle("com.foocompany.resources.FooBarTextElements", userLocale)`.

Example 4-4 illustrates this point.

Example 4-4. Proper Retrieval of a Resource Bundle in a Package

```
package com.foocompany;

import java.util.Locale;
import java.util.ResourceBundle;
import java.util.MissingResourceException;

public class FooBar {

  public static void main(String [] argv) {

    ResourceBundle textResources = null;

    try {
      textResources = ResourceBundle.getBundle("FooBarTextElements",
          Locale.getDefault());
      System.out.println(textResources.getString("CompanyName"));
    } catch (MissingResourceException e) {
      System.out.println("Unqualified resource bundle name");
    }

    try {
      textResources = ResourceBundle.getBundle(
            "com.foocompany.resources.FooBarTextElements",
            Locale.getDefault());
      System.out.println(textResources.getString("CompanyName"));
    } catch (MissingResourceException e) {
    }
  }
}
```

Example 4-5 illustrates the resource bundle used in Example 4-4.

Example 4-5. Resource Bundle Used in Example 4-4

```
package com.foocompany.resources;

import java.util.Enumeration;
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.StringTokenizer;
import java.util.MissingResourceException;

public class FooBarTextElements extends ResourceBundle {

  private String keys = "CompanyName";

    public Object handleGetObject(String key) {
      if (key.equals("CompanyName")) return "Foo Bar Company";

      return null;
    }

    public Enumeration getKeys() {
      StringTokenizer keyTokenizer = new StringTokenizer(keys);

      return keyTokenizer;
    }
}
```

Running Example 4-4 produces the following results:

```
C:\>java com.foocompany.FooBar
Unqualified resource bundle name
Foo Bar Company
```

You do not need to replicate elements in a resource bundle if they exist in another bundle within the resource bundle hierarchy. Example 4-6 demonstrates the lookup of the key "`Hello`." Examples 4-7, 4-8, and 4-9 are the resource bundles used by Example 4-6.

Example 4-6. Demonstrates That You Do Not Have to Replicate Keys and Values in All Resource Bundles

```
import java.util.Enumeration;
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.MissingResourceException;

public class HelloResourceBundleExample {
```

Example 4-6. Demonstrates That You Do Not Have to Replicate Keys and Values in All Resource Bundles (continued)

```
  public static void main(String [] argv) {
    try {
      Locale frenchLocale = new Locale("fr", "FR");
      ResourceBundle rb =
        ResourceBundle.getBundle("HelloResourceBundle", frenchLocale);

      System.out.println(rb.getString("Hello"));
      System.out.println(rb.getString("Goodbye"));

    } catch (MissingResourceException mre) {
      mre.printStackTrace();
    }
  }
}
```

Example 4-7. Fallback Resource Bundle

```
import java.util.Enumeration;
import java.util.ResourceBundle;
import java.util.StringTokenizer;

public class HelloResourceBundle extends ResourceBundle {

  private String keys = "Hello Goodbye";

    public Object handleGetObject(String key) {
      if (key.equals("Hello")) return "Hello";
      if (key.equals("Goodbye")) return "Goodbye";

      return null;
    }

    public Enumeration getKeys() {
      StringTokenizer keyTokenizer = new StringTokenizer(keys);

      return keyTokenizer;
    }
}
```

Example 4-8. French Resource Bundle Containing the Key "Hello"

```
import java.util.Enumeration;
import java.util.ResourceBundle;
import java.util.StringTokenizer;
```

Example 4-8. French Resource Bundle Containing the Key "Hello" (continued)

```
public class HelloResourceBundle_fr extends HelloResourceBundle {

  public Object handleGetObject(String key) {
    if (key.equals("Hello")) return "Bonjour";

    return null;
  }
}
```

Example 4-9. French Resource Bundle Containing the Key "Goodbye"

```
import java.util.Enumeration;
import java.util.ResourceBundle;
import java.util.StringTokenizer;

public class HelloResourceBundle_fr_FR extends HelloResourceBundle_fr {

  public Object handleGetObject(String key) {
    if (key.equals("Goodbye")) return "Au Revoir";

    return null;
  }
}
```

Example 4-9 uses the locale "`fr_FR`," so the search for a resource matching the key of "`Hello`" starts in `HelloResourceBundle_fr_FR`. The key does not exist in this bundle, so the search proceeds up the hierarchy. The key is found in `HelloResourceBundle_fr`, and the value is printed to the screen. Likewise, the search for a resource matching the key of "`Goodbye`" starts in `HelloResourceBundle_fr_FR`. The search begins and ends here, since the key exists in this resource bundle.

The first time you search for a particular resource bundle, Java starts a lookup process and either returns the specific resource bundle you requested or throws a `MissingResourceException` if the resource is not found. Searching for a particular resource bundle (after that bundle was found) results in the bundle being cached in memory. Java stores the resource bundle in a hashtable for quick lookup. Requests made for a particular resource bundle at a later point in your application result in use of the cached bundle. Had such design considerations not been made, this situation could have been a huge bottleneck in programs that rely on resource bundle facilities. Think of how slow programs would be if each time you request a resource from a bundle, the system must go through the steps of searching the resource bundle hierarchy, loading the bundle from disk, and returning the data for a particular resource.

Property Resource Bundles

A property resource bundle is a collection of text elements in a properties file format stored in a *.properties* file. The `PropertyResourceBundle` class provides all code necessary for you to access these resources in a locale-dependent manner.

Example 4-10 illustrates basic property file formatting. The `Properties` class in the `java.util` package handles reading and writing properties files. Both the keys to lookup a value and the values returned are Java `Strings`. Example 4-10 shows a simple properties file.

Example 4-10. A Properties File

```
# Sample properties file for demonstrating PropertyResourceBundle class
#
# Text that will appear in the title bar of our application
ApplicationTitle=Demonstrating the use of PropertyResourceBundle
#
# Text that will be displayed on OK buttons
OKButtonLabel=OK
#
# Text for display on a button to cancel current operation
CancelButtonLabel=Cancel
#
#
```

A property in a properties file is specified in the following format:

```
Key=Value
```

In most properties files, you see that the separator between key and value is the equals sign (=) character. Java also allows use of the colon (:) character as a separator between key and value, as in:

```
Key:Value
```

We won't go into all the details about how Java deals with properties files, but here are a few general comments on how properties files are structured: lines starting with an exclamation (!) or a pound (#) character are ignored when the properties file is read. These characters indicate that the line is a comment line. Keys are case sensitive. Finally, you can specify values in a properties file that span multiple lines by using a backslash (\) to indicate line continuation. Example 4-11 illustrates this point.

Example 4-11. Properties File Where Values Span More Than One Line (AnimalResources.properties)

```
# Sample properties file with keys whose values span more than one line
#
Animals=Cat, Dog, Giraffe, \
    Shark, Dolphin, Bird, \
    Bear, Moose
```

If you retrieved the "Animals" key, you'd get back the value `Cat, Dog, Giraffe, Shark, Dolphin, Bird, Bear, Moose`. All the leading whitespace on any continuation line is discarded. You could then, for example, break up the value returned to you by using the `java.util.StringTokenizer` class.

Example 4-12 uses the properties file from Example 4-11 to print a list of animal names.

Example 4-12. Printing a List of Animals Stored in a Properties Resource File

```
import java.util.Locale;
import java.util.ResourceBundle;
import java.util.MissingResourceException;

public class Animals {

  public static void main(String [] argv) {

    ResourceBundle animalResources;

    try {
      animalResources = ResourceBundle.getBundle(
         "AnimalResources", Locale.getDefault());
      System.out.println(animalResources.getString("Animals"));
    } catch (MissingResourceException mre) {
      mre.printStackTrace();
    }
  }
}
```

Here is the output produced after running Example 4-12:

```
C:\>java Animals

Cat, Dog, Giraffe, Shark, Dolphin, Bird, Bear, Moose
```

List Resource Bundles

The `ListResourceBundle` class takes its name from the fact that it manages a list of resources. Those resources are stored in a class file, allowing you to provide local-

ized resources for any datatype. Resources in a ListResourceBundle are not limited to text strings, as are the resources in a PropertyResourceBundle. For example, images are one example of resources that can be localized, but can't be stored as text strings. The company logo is one common image important to most companies. Companies pride themselves on, and can earn hefty profits from, logo recognition alone. Logos can be associated with a company or a particular product.

ListResourceBundle, an abstract subclass of ResourceBundle stores its keys and values in a two-dimensional Object array. This mechanism allows you to create resources of arbitrary datatypes unlike the PropertyResourceBundle, in which both the keys and values must be strings. When you use a ListResourceBundle, the only restriction is that the keys must be specified as strings.

When you create a subclass of ListResourceBundle, you must override the getContents() method that returns a two-dimensional Object array. The first element is a key (a String) used to access the value. The second element of the pair can be of any datatype, such as an AWT Button, an Integer number, or a String. Example 4-13 shows a simple ListResourceBundle called SampleResourceBundle and illustrates how to create a subclass of ListResourceBundle.

Example 4-13. A Subclass of ListResourceBundle Called SampleResourceBundle

```
import java.awt.*;
import java.util.*;

public class SampleResourceBundle extends ListResourceBundle {

  public Object [][] getContents() {
    return contents;
  }

  static final Object [][] contents = {
      {"okButton", new Button("OK")},
      {"negativeInteger", new Integer(-1)},
      {"textString", "Thank you for reading our book"}
  };
}
```

The ResourceBundle class provides a method called getObject(String key). When you retrieve a value from your resource bundle, you must cast it to its proper type. To retrieve the key okButton from the resource bundle in Example 4-13, we would pass the okButton string to the getObect method. This method would return an Object, which would then be cast to Button to use the Button object.

Example 4-14 illustrates the power of the `ListResourceBundle` class. Examples 4-15 and 4-16 show a resource bundle for this applet and the HTML in which the applet is embedded. We display the O'Reilly & Associates logo to the user, but the logo displayed depends on the locale. If the parameter `url` is set to `yes` in the applet's HTML file, the logo will contain the URL for that country's O'Reilly website. We don't show all resource bundles for this example because they're so large; instead, we show a relevant portion of the German resource bundle in Example 4-15.

Example 4-14. Applet Used to Display O'Reilly & Associates Logo Depending on the Set Locale

```
import java.applet.*;
import java.awt.*;
import java.util.*;

public class OReillyLogoApplet extends Applet {
  private ImageArea imageArea;
  private Image logo;
  private Button okButton;
  private ListResourceBundle appletElements;
  private Locale logoAppletLocale;
  private boolean displayURLLogo;
  public void init()   {

    // Get the parameters to set the locale for the applet
    logoAppletLocale = new Locale(getParameter("language"),
                                  getParameter("country"));
    if (getParameter("url") != null &&
        getParameter("url").equalsIgnoreCase("yes"))
      displayURLLogo = true;
    else
      displayURLLogo = false;

    try {

      setLayout(new BorderLayout());

      imageArea = new ImageArea();
      imageArea.setBackground(Color.white);
      imageArea.setSize(getSize().width, getSize().height);

      appletElements = (ListResourceBundle)ResourceBundle.
         getBundle("OReillyResources", logoAppletLocale);
      if (displayURLLogo)
        logo = Toolkit.getDefaultToolkit()
               .createImage((byte [])appletElements
               .getObject("OReillyLogoURL"));
      else
```

Example 4-14. Applet Used to Display O'Reilly & Associates Logo Depending on the Set Locale (continued)

```
        logo = Toolkit.getDefaultToolkit()
               .createImage((byte [])appletElements
               .getObject("OReillyLogo"));

      add("North", imageArea);

      imageArea.displayImage(logo);

    } catch (MissingResourceException e) {
    }
  }
}

class ImageArea extends Canvas {

  Image image;

  public void displayImage(Image image) {
    this.image = image;
    repaint();
  }

  public void paint(Graphics g) {
    if (image != null)
      g.drawImage (image, 0, 0, Color.lightGray, this);
  }
}
```

Example 4-15. A Partial Listing of the Resource Bundle for Example 4-14 for Germany

```
import java.util.ListResourceBundle;

public class OReillyResources_de extends ListResourceBundle {

    public Object [][] getContents() {
        return contents;
    }

    static byte [] OReillyLogo = {
     (byte)71,
     (byte)73,
     (byte)70,
     (byte)56,
     (byte)57,
     (byte)97,
```

Example 4-15. A Partial Listing of the Resource Bundle for Example 4-14 for Germany (continued)

```
        ...
        (byte)59
        };

    static final Object [][] contents = {
        {"OReillyLogo", OReillyLogo},
        {"OReillyLogoURL", OReillyLogoURL}
    };
}
```

You can change the value in the `language` parameter in HTML to see a different logo displayed in the applet, as shown in Example 4-16. If the value for the `url` parameter is not `yes`, no URL is displayed in the logo. For example, to create Figure 4-3, set the `language` parameter to `fr` and the `url` parameter to `no`. To create Figure 4-4, set the `language` parameter to `de` and the `url` parameter to `yes`.

Figures 4-3 and 4-4 show the applet running in the French language (no URL) and the German language (with URL), respectively.

Figure 4-3. Logo applet using French language (no URL)

Figure 4-4. Logo applet using German language (with URL)

Example 4-16. HTML Used to Run the O'Reilly Logo Display Applet

```
<HTML>

<HEAD>
<TITLE>O'Reilly Logo Applet</TITLE>
</HEAD>

<BODY>

<APPLET CODE="OReillyLogoApplet" WIDTH=270 HEIGHT=115>
<PARAM NAME="country" VALUE="">
<PARAM NAME="language" VALUE="fr">
<PARAM NAME="url" VALUE="yes">
</APPLET>

</BODY>

</HTML>
```

Example 4-17 shows a utility we've used in this chapter to create a `ListResourceBundle` class automatically, using data from an image or a sound file. You can use any data file you like. You'll find that adding this type of data to your resource bundles, specifying each byte manually, is very time consuming! This program takes three arguments: the data file to include in the resource bundle; the key used to retrieve the resource from the bundle; and the name of the class

you subclass from `ListResourceBundle`. Specify the class name with no *.java* extension. The proper *.java* source file is created from the name you supply.

Here's how you might use this utility:

```
C:\>java ListResourceBundleCreator oreilly_logo_germany.gif
        OReillyLogoGerman
        OReillyLogoResources_de
```

Example 4-17. Utility Program to Create a ListResourceBundle Using Data from a User-Supplied Data File

```
import java.io.*;
import java.util.Vector;

public class ListResourceBundleCreator {

  public static void main(String [] argv) throws Exception{
    FileInputStream inputFileReader = new FileInputStream(argv[0]);
    DataInputStream dis = new DataInputStream(inputFileReader);
    long fileSize;

    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    Vector myVector = new Vector();

    while (true) {
      try {
        myVector.addElement(new Integer(dis.readUnsignedByte()));
      } catch (EOFException e) {
        break;
      }
    }

    fileSize = myVector.size();
    FileWriter outputFileWriter = new FileWriter(argv[2] + ".java");
    outputFileWriter.write("import java.util.ListResourceBundle;\n\n");
    outputFileWriter.write("public class " + argv[2] +
                           " extends ListResourceBundle {\n\n");
    outputFileWriter.write("    public Object [][] getContents() {\n");
    outputFileWriter.write("        return contents;\n");
    outputFileWriter.write("    }\n\n");

    outputFileWriter.write("    static byte [] " + argv[1] + " = {\n");
    for (int i = 0; i < fileSize; i++) {
        outputFileWriter.write("\t(byte)");
        outputFileWriter.write(((Integer)myVector.elementAt(i)).toString());
        if (i < fileSize - 1)
            outputFileWriter.write(",\n");
    }
    outputFileWriter.write("\n\t};\n\n");
```

Example 4-17. Utility Program to Create a ListResourceBundle Using Data from a User-Supplied Data File (continued)

```
    outputFileWriter.write("    static final Object [][] contents = {\n");
    outputFileWriter.write("      {\"" + argv[1] + "\", " + argv[1] + "}\n");
    outputFileWriter.write("    };\n");
    outputFileWriter.write("}\n");
    outputFileWriter.close();
  }
}
```

Resource Bundle Caveats

This section details information you need to consider when using resource bundles in your application.

What to Localize in a Resource Bundle

Certain strings should not be localized and placed into a resource bundle in any application. Consider the strings that specify the region that a component is placed in for the `BorderLayout` class in the `java.awt` package. Java allows you to specify five regions on the border layout, namely, "North," "South," "East," "West," and "Center." Strings like these are used as *program identifiers* and must not be localized. Localizing would create programs that either do not work at all or behave incorrectly. This situation also holds true for keys used to retrieve a specific element from a resource bundle; you would not localize keys used to retrieve values from a `PropertyResourceBundle`.

Character Encoding Issues for Resource Bundles

Java source files are most commonly written and saved as ASCII text files with escape sequences for non-ASCII characters. File editors used to create your source files may also allow you to use a regional character encoding. The `javac` compiler supports the `-encoding` directive, which allows you to specify the regional encoding of the file you're trying to compile. Appendix B, *Character Encodings Supported by Java*, contains a complete listing of all supported languages and converters that can be specified for your specific regional encoding. Chapter 6, *Character Sets and Unicode*, explains encodings in more detail.

You need to be aware of the character encoding when you use a ListResourceBundle. In a ListResourceBundle, your resources are stored in a class file, which is created by compiling a *.java* source file. Say we compile a source file using a Polish character encoding. We'd compile our resource bundle as follows:

```
C:\>javac -encoding iso8859_2 ProgramResources.java
```

Using the PropertyResourceBundle class, we must also worry about the character encoding of the individual properties files. Properties files must use an ISO 8859-1 encoding or contain Unicode escape sequences to represent Unicode characters. Unicode escape sequences are specified as \uxxxx, for which xxxx represents the hexadecimal value of the specific character.

Java provides a utility called native2ascii, which takes a file written in a given character encoding and converts non-ASCII characters to the appropriate Unicode escape sequences. native2ascii also takes an -encoding option like javac, through which you can specify the character encoding of the file to be converted. If this option is not specified, it uses the default character encoding of the platform on which the program is running. The command-line option -reverse allows you to convert a file containing Unicode escape sequences to a native character encoding. You must specify an input file and you may also specify an optional output file where the converted file will be written. If the output file is not specified, standard output is used:

```
C:\>native2ascii -encoding SJIS ProgramResources_ja.java
    ProgramResources_ja.java.converted
```

Property Resource Bundle Caveats

We should point out a few things to watch for now that you're aware of how to use PropertyResourceBundles.

Using the PropertyResourceBundle class is very convenient when all you have to localize are textual elements. If you have other datatypes that you'd like to localize in a resource bundle, the ListResourceBundle class allows you that flexibility. In the next subsection of this chapter, we'll discuss a problem that could arise when using both PropertyResourceBundles and ListResourceBundles.

Unlike the ListResourceBundle class, which must be subclassed, the PropertyResourceBundle class is a concrete subclass of ResourceBundle. Therefore, to add more resources using a PropertyResourceBundle, simply create the appropriate properties files. You do not have to compile these resource bundles, as you must with a ListResourceBundle.

Properties files that are found by one of the `getBundle` methods must end with a *.properties* extension. If one of your properties files is missing this extension, it will not be found; this situation causes problems when performing a resource bundle lookup. The *.properties* extension is added automatically for you during the lookup process, so you do not need to add one to the name passed to the `getBundle` method.

Resource Bundle Lookup Caveats

An age-old philosophical musing states, "Which came first, the chicken or the egg?" A similar, and maybe equally puzzling, question arises when we speak about using class files or properties files for our resource bundles. It becomes an issue when you think about mixing the two types of bundles within an application.

Let's rephrase our philosophical question in resource bundle terms, "When looking for resource bundles, which is found first, a class file or a properties file?" If both a class file and a properties file exist at the same level in the resource bundle hierarchy, any calls to retrieve resources are retrieved from the class file. In this situation, the class file "hides" the properties file.

Example 4-18 demonstrates this behavior; Examples 4-19 through 4-23 are the resource bundles used with this example.

Example 4-18. Demonstration of Finding a Class Resource File Before a Properties Resource File

```
import java.util.ResourceBundle;
import java.util.MissingResourceException;

public class WhichBundleComesFirstExample {

  public static void main(String [] argv) {
    try {
      ResourceBundle resources =
        ResourceBundle.getBundle("WhichBundleComesFirstResources");
      System.out.println(resources.getString("Message3"));
      System.out.println(resources.getString("Message2"));
      System.out.println(resources.getString("Message1"));
    } catch (MissingResourceException mre) {
      mre.printStackTrace();
    }
  }
}
```

Example 4-19. WhichBundleComesFirstResources_en_US in ListResourceBundle Format

```
import java.util.ListResourceBundle;

public class WhichBundleComesFirstResources_en_US extends
    ListResourceBundle {

    public Object [][] getContents() {
        return contents;
    }

    static final Object [][] contents = {
        {"Message3",
         "Message 3: From WhichBundleComesFirstResources_en_US \
         (ListResourceBundle)"}
    };
}
```

Example 4-20. WhichBundleComesFirstResources_en_US in PropertyResourceBundle Format (WhichBundleComesFirstResources_en_US.properties)

```
# Properties file for WhichBundleComesFirstResources_en_US
#
# English language in the United States
#
Message3=Message 3: From WhichBundleComesFirstResources_en_US \
  (Properties file)
```

Example 4-21. WhichBundleComesFirstResources_en in PropertyResourceBundle Format (WhichBundleComesFirstResources_en.properties)

```
# Properties file for WhichBundleComesFirstResources_en
#
# English language in the United States
#
Message2=Message 2: From WhichBundleComesFirstResources_en (Properties file)
```

Example 4-22. WhichBundleComesFirstResources in ListResourceBundle Format

```
import java.util.ListResourceBundle;

public class WhichBundleComesFirstResources extends ListResourceBundle {

  public Object [][] getContents() {
    return contents;
  }

  static final Object [][] contents = {
```

Example 4-22. WhichBundleComesFirstResources in ListResourceBundle Format (continued)

```
      {"Message1",
       "Message 1: From WhichBundleComesFirstResources (ListResourceBundle)"}
  };
}
```

Example 4-23. WhichBundleComesFirstResources in PropertyResourceBundle Format (WhichBundleComesFirst.properties)

```
# Properties file for WhichBundleComesFirst
#
# English language in the United States
#
Message1=Message 1: From WhichBundleComesFirstResources (Properties file)
```

The following is outputted when Example 4-18 is executed:

```
Message 3: From WhichBundleComesFirstResources_en_US (ListResourceBundle)
Message 2: From WhichBundleComesFirstResources_en (Properties file)
Message 1: From WhichBundleComesFirstResources (ListResourceBundle)
```

As you can see, the messages are retrieved from the class file resource bundle for Message3 and Message1, which are contained in both a class file resource bundle and a properties file resource bundle.

Why even bring this topic up? If you wish to use the `ListResourceBundle` class to localize objects such as GUI widgets and images, and if you use a `PropertyResourceBundle` to handle all of the text messages displayed to users using your application. You could retrieve your images and widgets correctly, but a lot of `MissingResourceExceptions` would be thrown when your application tries to retrieve any text messages.

Example 4-24 is a utility that converts a properties file into a `ListResourceBundle` class. This utility can convert your existing `PropertyResourceBundles` to `ListResourceBundles`, given that `ListResourceBundles` are found before `PropertyResourceBundles`. The utility takes one command-line parameter, the name of the properties file, to convert to a `ListResourceBundle`.

Example 4-24. PropertyToListResourceBundle Utility

```
import java.io.FileInputStream;
import java.io.FileWriter;
import java.io.IOException;
import java.io.OutputStreamWriter;
import java.util.Enumeration;
import java.util.Locale;
import java.util.Properties;
```

Example 4-24. PropertyToListResourceBundle Utility (continued)

```
import java.util.MissingResourceException;

public class PropertyToListResourceBundle {

  public static void main(String [] argv) {
    if (argv.length < 1) {
      System.exit(1);
    }

    try {
      String bundleName = argv[0] + "_" + Locale.getDefault().toString();
      Properties propertiesFile = new Properties();
      propertiesFile.load(new FileInputStream(bundleName + ".properties"));
      FileWriter fw = new FileWriter(bundleName + ".java");
      String key;

      fw.write("import java.util.ListResourceBundle;\n\n");

      fw.write("public class " + bundleName +
               " extends ListResourceBundle {\n\n");
      fw.write("    public Object [][] getContents() {\n");
      fw.write("        return contents;\n");
      fw.write("    }\n\n");

      fw.write("    static final Object [][] contents = {\n");

      Enumeration e = propertiesFile.propertyNames();
      while (e.hasMoreElements()) {
        key = (String)e.nextElement();
        fw.write("        {\"" + key + "\", " + "\""
                  + propertiesFile.getProperty(key) + "\"},\n");
      }

      fw.write("        {\"PropertyToListResourceBundleCreator\", "
               + "\"O'Reilly\"}\n");
      fw.write("    };\n");
      fw.write("}\n");

      fw.close();
    } catch (MissingResourceException mre) {
      mre.printStackTrace();
    } catch (IOException ioe) {
      ioe.printStackTrace();
    }
  }
}
```

We demonstrate the utility on the `PropertyResourceBundle` in Example 4-25.

Example 4-25. Test Resource Bundle

```
# Test properties file (TestProperties_en_US.properties)
#
Key1=This is the value for Key 1
Key2=This is the value for Key 2
Key3=This is the value for Key 3
```

The utility is run as follows:

```
C:\>java -Duser.language=en -Duser.region=US PropertyToListResourceBundle TestProperties
```

Example 4-26 shows the `ListResourceBundle` created by the utility.

Example 4-26. ListResourceBundle Created by the PropertyToListResourceBundle Utility

```
import java.util.ListResourceBundle;

public class TestResources_en_US extends ListResourceBundle {

    public Object [][] getContents() {
        return contents;
    }

    static final Object [][] contents = {
        {"Key3", "This is the value for Key 3"},
        {"Key2", "This is the value for Key 2"},
        {"Key1", "This is the value for Key 1"},
        {"PropertyToListResourceBundleCreator", "O'Reilly"}
    };
}
```

Deploying Resource Bundles with Applets

Applets can use resource bundles just like other applications. In this section, we show you how to package an entire applet (both program code and resource files) into a single JavaARchive (JAR) file for deployment. Two advantages of using a JAR file for deploying applets are:

- A single HTTP request is made to the web server hosting the applet to retrieve the program code and resources, which reside in a single JAR file.
- A JAR can compress the files it contains, making the request size for an application and its resources smaller than if the files were uncompressed.

There's really not much difference to deploying an applet packaged in a JAR file. We only have to specify the JAR file that contains all required *.class* and *.properties*

files that hold the necessary resources required to run your applet. When running your applet, the browser tries to pull the resource files from the JAR file instead of pulling the resource from the web server. This feature helps save download time for your users and makes your application self-contained.

Like resource bundles, JAR is a convenient bundling mechanism for Java code. You can use JAR files to package and deploy applets and applications. JAR files (ending with a *.jar* extension) are similar to the ubiquitous ZIP file (ending with a *.zip* extension), a popular compressed file format. JAR and ZIP files differ because JAR files can contain an optional `Manifest` entry. A manifest entry is used to specify meta-information about the entire JAR file or individual files in the JAR file.[1]

JAR files are created using the `jar` command-line utility provided with the Java Development Kit. The syntax is fairly straightforward and the utility is very easy to use. If you have all your *.class* and *.properties* files in a single directory, run the jar utility by entering:

```
C:\>jar cvf DemonstrationApplet.jar *.class *.properties
```

The `c` option tells the jar program that you're *creating* a JAR file. The `v` option specifies *verbose* output while creating the archive file. The `f` option lets the program know that you're specifying the archive *file name* as the next parameter. Once you created a JAR file successfully, how do you deploy your applet? This step should require few modifications to the *.html* file containing your applet's `<APPLET></APPLET>` tag. Suppose your HTML file looked like Example 4-27.

Example 4-27. HTML File for Running an Applet Not Contained in a JAR File

```
<HTML>

<HEAD>
<TITLE>My Test Applet</TITLE>
</HEAD>

<BODY>

Here is my applet…<P>
<APPLET code="MyTestApplet.class" width=300 height=400>
<PARAM name="User" value="Pamela Ann">
</APPLET>

</BODY>
</HTML>
```

---

[1] You can read more about the manifest format online at ***http://java.sun.com/products/jdk/1.3/docs/guide/jar/manifest.html***

Let's assume that you've called the JAR file containing this applet and its associated *.properties* files *MyTestApplet.jar*. The HTML used to run your applet would now look like Example 4-28.

Example 4-28. HTML File for Running an Applet Contained in a JAR File

```
<HTML>

<HEAD>
<TITLE>My Test Applet</TITLE>
</HEAD>

<BODY>

Here is my applet…<P>
<APPLET archive="MyTestApplet.jar" code="MyTestApplet.class" width=300 height=400>
<PARAM name="User" value="Pamela Ann">
</APPLET>

</BODY>
</HTML>
```

Design Considerations for Resource Bundles

The resource bundle facilities in Java allow you to decouple user-displayed messages from your actual program code and user interface. That is, resources like messages are not hardcoded directly into the application. You may also want to differentiate your resources further by packaging similar types of resources together. You do not need to restrict yourself to using only one set of resource bundles. Examples of these considerations might be:

```
UserExceptionResources
UserExceptionResources_de
UserExceptionResources_de_DE
UserExceptionResources_es
StatusMessageResources
StatusMessageResources_es
StatusMessageResources_es_ES
... and so on
```

Another good practice is to give your resource bundles meaningful names. They should be easily identified as being specific to your application. If you decided to develop a spreadsheet application, starting out with `SpreadsheetApplicationResources` would be a good choice for naming your bundles. A base name such as

`ProgramResources` for your resource bundles is not bad, but from a development standpoint, it does not adequately identify the resources as being specific to your spreadsheet application.

One advantage of using resource bundles is that you can support new locales by adding more resource bundles. If you have taken the proper steps to internationalize your application, you just need to make new resource bundles available. Your users don't have to download a new version of the entire application. They just have to install the new resource bundles in the proper location.