Enterprise Java for SAP

AUSTIN SINCOCK

Apress™

Enterprise Java for SAP
Copyright © 2003 by Austin Sincock

Technical Reviewers: Rob Castenada, Scott Babbage

Editorial Board: Dan Appleman, Craig Berry, Gary Cornell, Tony Davis, Steven Rycroft, Julian Skinner, Martin Streicher, Jim Sumser, Karen Watterson, Gavin Wright, John Zukowski

Assistant Publisher: Grace Wong

Copy Editor: Rebecca Rider

Production Manager: Kari Brooks

Production Editor: Janet Vail

Composition, Proofreading, and Illustration: Kinetic Publishing Services, LLC

Indexer: Valerie Perry

Cover Designer: Kurt Krames

Manufacturing Manager: Tom Debolski

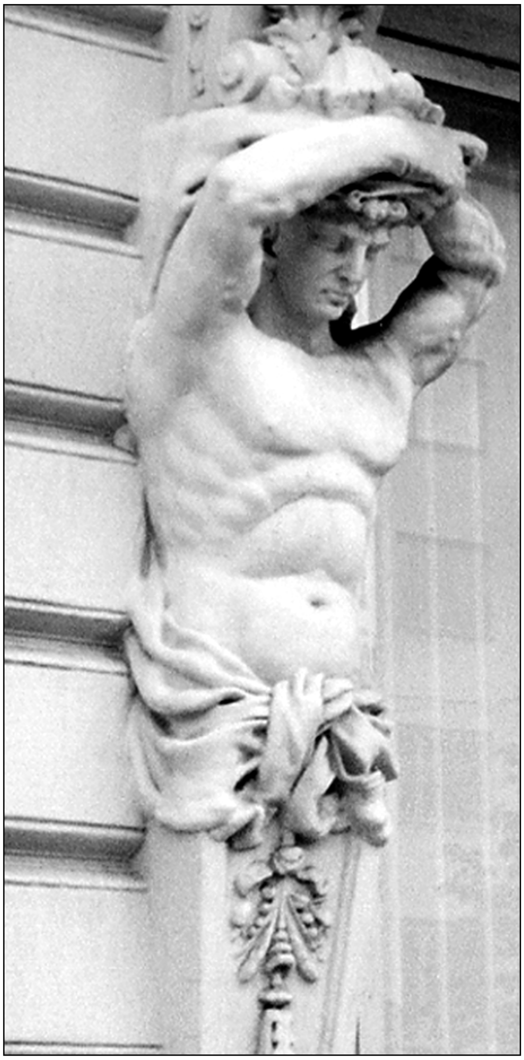
Library of Congress Cataloging-in-Publication Data

Sincock, Austin, 1974–
Enterprise Java for SAP / Austin Sincock.
p. cm.
ISBN 1-59059-098-8 (hardcover : alk. paper)
1. Java (Computer program language) 2. SAP R/3. I. Title.
QA76.73.J38S56 2003
005.13'3—dc21
2003013230

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

CHAPTER 5

Building a Desktop Interface to SAP



In the previous chapter, you finally got the chance to develop a Java application for SAP. Granted, the program executes from the command line and has little use in the real world, but it does demonstrate the fundamental concepts involved in using the Java Connector (JCo). However, this tutorial comprised only a fraction of the overall development effort required to create applications for everyday use.

This chapter focuses on Java technologies that extend an application to the end user's desktop. Specifically, this chapter looks at the following:

- Using configuration files to handle system variables
- Providing object-level access to your existing development
- Developing a graphical desktop client to access SAP

Using Configuration Files

Let's assume that you have decided to reuse your existing Java application development. In this case, your first step is to ensure that an end user can configure the program without having to edit and recompile any source code.

The last section of Chapter 4 showed you how to move the system variables from the method level to the class level in order to reduce redundancy in the code and create more visibility for the SAP variables. Unfortunately, a Java developer still needs to modify and recompile this code in order to make any changes to these system variables. Unless you are willing to provide a Java Software Development Kit (JSDK) and appropriate training to each and every user, this is not a viable solution.

Luckily, the Java 2 specification provides a simple-to-use set of libraries that allow you to read data in from a plain text file and use that data as variables within a Java application. Known as *resource bundles,* these text files contain single or multiple text strings associated with keywords. Once you have created them, you can read these resource bundles into an application using the `ResourceBundle` class library that can be found in the `java.util` package.

Internationalizing Java Applications

Java-based resource bundles are designed to house locale-specific information, such as internationalized text strings. This allows developers to build applications in a single written language, which can then be translated over time into any other required language. How is this possible?

Different programming languages and operating systems often provide native methods for passing country- or region-specific information within an application. Java is no exception; it allows the developer to use the `java.util.Locale` class to represent the region where the end user resides. This localization occurs through the combination of language key and country code, both as listed by the International Organization for Standardization (ISO).

NOTE *The ISO claims that "ISO" is not an abbreviation or acronym. ISO is a word that is derived from the Greek* isos, *which means "equal." The rationale for using this Greek word is to keep the organization's name standard throughout the world. So, despite language and translation differences, the International Organization for Standardization remains ISO in every country across the globe.*

Once the developer determines where the user is from and what language he speaks (usually by asking at the start of the application), the developer creates an instance of the `Locale` class. This language-specific `Locale` instance is then used throughout the program wherever the developer needs locale-based formatting or text.

Resource bundles fit into this scheme by providing a plain text file format that houses locale-specific text and can automatically retrieve that text without the developer needing to intervene further. To use a resource bundle, the developer creates a text file with a file extension of `*.properties` and saves this file somewhere within the Java `CLASSPATH`. Assuming the default language is English, the properties file is called `MyResources.properties` and looks something like this:

```
main.welcome=Welcome to the new OpenSource Guru site.
main.articles=Check out the latest in Java technical articles.
main.projects=See the newest OSG projects.
main.contact=Contact OpenSource Guru at:
```

But what if you needed to internationalize this application? If you had designed your application using the `Locale` class, then you could simply add a new properties file to the Java `CLASSPATH` that contained the translated language texts.

Assume that you needed to translate the application into German. To do so, instead of changing any of your application code, you would simply create a properties file with the name `MyResources_de.properties`. Notice that "_de" has been added to the name of the properties file. The two-letter key, `DE` (meaning

Deutschland), is the code established by the ISO to indicate that German is being used. The new properties file would look like this:

```
main.welcome= Willkommen zum neuen OpenSource Guruaufstellungsort.
main.articles=Überprüfen Sie aus dem spätesten Java in den
                          technischen Artikeln.
main.projects=Sehen Sie die neuesten OSG Projekte.
main.contact=Kontakt OpenSource Guru an:
```

If DE is supplied as the language key within the Locale instance, the ResourceBundle class automatically looks for a properties file named MyResource followed by de, rather than using the default properties file, MyResources.properties.

Perhaps the best way to understand multilingual resource bundles is to develop a quick application that relies on the end user to define the program's language text.

```
import java.util.ResourceBundle;
import java.util.Locale;
import java.util.Date;
public class LanguageApp {

  public static void main(String[] args){

   String langKey;
   Date currDate = new Date();
   if (args.length > 0)
    langKey = args[0];
   else
    langKey = "";

   Locale currLocale = new Locale(langKey);

   ResourceBundle myResources =
       ResourceBundle.getBundle("MyResources", currLocale);
   System.out.println("<<<***---------------------------***>>>");
   System.out.println(myResources.getString("main.welcome"));
   System.out.println("<<<***---------------------------***>>>" +
                              System.getProperty("line.separator"));
   System.out.println(myResources.getString("main.articles") +
                              System.getProperty("line.separator"));
   System.out.println(myResources.getString("main.projects") +
                              currDate.toString() +
                              System.getProperty("line.separator"));
    System.out.println(myResources.getString("main.contact") +
                                     " guru@opensourceguru.com" +
                                     System.getProperty("line.separator"));
  }
}
```

NOTE *This code will not work in anything older than JDK1.4. As of 1.4, the default constructor used to create a new* `Locale` *instance requires only a language key. In earlier releases, in order to utilize this class, this constructor needed both a language key and a country code. If you are using an older JDK, simply hard-code the value "US" as a second parameter in the* `new Locale()` *statement.*

To get this application to work, ensure that the `MyResources.properties` and `MyResources_de.properties` files are located in your development directory. Next, build the preceding application in your text editor, and save it with the file name `LanguageApp.java`. Compile the application and run it from the command line:

```
java LanguageApp
```

By not passing in any parameters on the command line, the `currLocale` instance was initialized with an empty string, "". The `getBundle()` method treats this locale as the default, retrieves the `MyResources.properties` file, and uses it to display the requested fields. You can pass a value for the language key to the Java application by entering it on the command line. Try entering a value of `DE` on the command line.

```
java LanguageApp DE
```

Your application initializes the `currLocale` instance using the ISO language key for German. The `getBundle()` method attempts to retrieve a properties file with "de" at the end of the file name, as in `MyResources_de.properties`, and the German translation displays to the console screen.

Try creating new properties files with the same naming convention, `MyResources_`*`XX`*`.properties`, where *`XX`* is any standard ISO language key. Then rerun your application, passing in that language key from the command line. This demonstrates how easy it is to support additional languages without having to modify the application code. A list of additional ISO language codes can be found at `http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt`.

The sample application also shows you how to mix static information, such as display formatting, and the email address with the more flexible content of the properties files.

```
System.out.println(myResources.getString("main.contact") +
                   " guru@opensourceguru.com" + "\n");
```

As you can see, this fairly straightforward approach allows you to add new language texts and translations to an application over time, rather than limiting the program to only those language requirements you know of at the time of development. With a little forethought to where language text translations might be appropriate, you can make your Java application an international superstar, or at least you can make it look like one, overnight.

Configuring the Application with Resource Bundles

Having said all of that, I am now going to tell you that you are going to use resource bundles for a slightly different purpose. Instead of needing to add new language translations to your Java application, you need to allow end users to configure the SAP R/3 system variables to suit each local install of the program. You can use the flexibility demonstrated in your properties application to provide configuration files for any Java application. These configuration files are easily modified in any standard text editor and are reinitialized each time the application is started.

First, define the contents of the properties file based on the system variables you are using in the existing application. Those variables include SAP username, password, and language key, as well as R/3 host name, client, and system number.

Now, create a text file called `sapdata.properties` and save this file in your base development directory. Add the following lines to this file:

```
jco.client.user=username
jco.client.passwd=password
jco.client.ashost=sapserver
jco.client.client=100
jco.client.sysnr=00
jco.client.langu=EN
```

As you can see, each line consists of a name/value pair separated by the equal sign (=). You use the name or key attribute within the Java application to specify the value you want retrieved. In this properties file, the key is made up of two terms separated by a period (.). This is not required for the properties file, but it is a convention that developers often use to group similar value sets together. You are not even limited to using a single period because common data groups can take on a more hierarchical structure, as shown here:

```
SAP.system.user=username
SAP.system.password=password
SAP.SALES.salesorg=3000
SAP.SALES.distrchan=10
USER.color=blue
USER.scheme=cornfield
```

In this sample properties file, the SAP-related data has been further broken down into `system` versus `SALES`, in addition to the non-SAP fields used to set the color and interface scheme for the end user. This further clarifies the use of various components within the properties file and makes it easier for an end user to configure.

NOTE *The* `sapdata.properties` *is used in later development to automatically populate the* `JCO.Client` *object with system values. Because this class relies on specific naming conventions in the properties file, you should stick to the* `jco.client.XXX` *for the key names in each line.*

Using the `sapdata.properties` file created earlier, you must now modify the Java program to read in this file and use the values specified as R/3 system variables.

```
import com.sap.mw.jco.*;
import java.util.ResourceBundle;
```

Along with the JCo libraries imported in the first line, the application needs to import the `java.util.ResourceBundle` class provided by the Java 2 application libraries. The `ResourceBundle` class gives you the methods you need to read in and extract data from the `sapdata.properties` file. In the following code snippet, you can see this file read into a `ResourceBundle` from the file system:

```
import com.sap.mw.jco.*;
import java.util.ResourceBundle;
public class MaterialList1 {
  static ResourceBundle sapProperties = ResourceBundle.getBundle("sapdata");
...
}
```

Now you need to instantiate a new instance of ResourceBundle by calling the getBundle() method on the ResourceBundle class. The getBundle() method requires the name of the requested properties file to be passed in as a text string. Notice that the *.properties file extension is not required here. The ResourceBundle class looks for any file in the Java CLASSPATH that matches the text string parameter passed to getBundle() that has the *.properties file extension.

Once the application has an instance of ResourceBundle initialized with the correct properties file, you can pull out String variables based on the keys associated with each value.

```
  public static final String SAP_CLIENT =
         sapProperties.getString("jco.client.client");
  public static final String USER_ID =
         sapProperties.getString("jco.client.user");
  public static final String PASSWORD =
         sapProperties.getString("jco.client.passwd");
  public static final String LANGUAGE =
         sapProperties.getString("jco.client.langu");
  public static final String HOST_NAME =
         sapProperties.getString("jco.client.ashost");
  public static final String SYSTEM_NUMBER =
         sapProperties.getString("jco.client.sysnr");
```

Instead of hard-coding every system variable, you need to initialize each String instance using the getString() method of sapProperties. The getString() method requires a single text string parameter that indicates the key set to a given value within the sapdata.properties file. Remember, this key is the full text string to the left of the equal sign (=), which separates the name/value pairs in your properties file. In this case, your keys include the "." separator used to group similar pairs together, as in SAP.client.

That's all it takes to implement a configuration file for your Java application. This way, the end user can easily utilize his own R/3 user seat to be authorized for use of SAP. Likewise, should the user wish to run your application against a different R/3 system or client, he can make this change simple by modifying the sapdata.properties file.

Modify this sapdata.properties file to suit your specific SAP system:

```
jco.client.user=your user name
jco.client.passwd=your password
jco.client.ashost=your sap hostname
jco.client.client=your sap client
jco.client.sysnr=your sap system number
jco.client.langu=your logon language
```

CAUTION *Be aware of the fact that these properties files are plain text. This means that files such as these can be read and changed by anyone who has access to a text editor and your local computer. You can find the best example of this in the* `SAP.password` *variable set in the* `sapdata.properties` *file, which can be used to determine that R/3 user's name and password. Properties files can be encrypted and the Java 2 specification provides libraries for doing so. However, Java cryptography is outside the scope of this book. For more information on cryptography and general security, check out* Java Security *(2nd Edition), by Scott Oaks (O'Reilly & Associates, 2001).*

As you begin building more complex Java applications, especially integrating with SAP, keep in mind the power and flexibility of resource bundles, not only for configuration but also for internationalization. In fact, SAP has already done much of the translation work for you. By using a language-specific user seat to authorize a user in SAP, the R/3 system automatically formats and translates numeric and text fields into the language designated by the user's language key. Your job is to simply translate any remaining text fields that exist solely within your Java application and provide those translations through locale-based properties files.

Providing Object-Level Access

Up until this point, your application development has combined SAP business logic with execution and display logic (the `main` method). However, object-oriented programming encourages you to decouple these application components into discreet elements that can be reused in a variety of ways. The `MaterialList` class that you have been working on up to this point has very few applications in the real world. It is a standalone Java class that executes from the command line and provides very limited display feedback. `MaterialList` has one very basic feature, and other Java classes that require access to the R/3 system cannot even use it.

Just as you have used Java classes and libraries to easily add new functionality to your application, so too can you create your own class libraries that contain discrete functional components you can reuse throughout your development. By building on the `MaterialList` class, you can create a new Java class, called `InterfaceCaller`, which cannot be run by itself but is an integrated component of the application framework.

Creating the Initial Java Class

Start off by creating a new Java class called `InterfaceCaller.java`. You can use the same imports from the `MaterialList` class, including the new line that imports `java.util.ResourceBundle`.

```
import com.sap.mw.jco.*;
import java.util.ResourceBundle;
import java.util.Hashtable;
public class InterfaceCaller {
  ...
}
```

NOTE *The third import statement allows your application to use the Java* `Hashtable` *class. In Java, the* `Hashtable` *allows data to be easily stored and retrieved by your application. This is discussed in more detail later in this chapter.*

Likewise, you need to add the SAP system variables and the initialized `ResourceBundle` so that the application has access to the `sapdata.properties` file you created earlier.

```
  ResourceBundle sapProperties = ResourceBundle.getBundle("sapdata");
  public static final String SAP_CLIENT =
        sapProperties.getString("jco.client.client");
  public static final String USER_ID =
        sapProperties.getString("jco.client.user");
  public static final String PASSWORD =
        sapProperties.getString("jco.client.passwd");
  public static final String LANGUAGE =
        sapProperties.getString("jco.client.langu");
  public static final String HOST_NAME =
        sapProperties.getString("jco.client.ashost");
  public static final String SYSTEM_NUMBER =
        sapProperties.getString("jco.client.sysnr");
  private JCO.Client aConnection;
  private IRepository aRepository;
```

The difference between the variables you initialized here and those in `MaterialList` is that here the `static` declarations have been removed. Throughout the remainder of `InterfaceCaller`, you will see that all `static` declarations have been taken out because they will soon prove to be more of a hindrance than a help.

As you recall, in the previous chapter's development, you needed to declare each method `static` so that the application could call them without needing to instantiate an instance of the `MaterialList` class. You had to make these methods `static` because your program needed to call these methods from within the `main` method of the application's executable class. The new `InterfaceCaller` class does not have a `main` method. Instead, if a class needs to use `InterfaceCaller` it must first instantiate an instance of the `InterfaceCaller` class in order to gain access to its methods and attributes.

Ultimately, the `InterfaceCaller` class is a nonexecutable Java object, designed strictly to house the SAP business logic and JCo connectivity. By removing this class's `main` method, you require another Java class to act as the effective application executable, instantiating and calling methods on `InterfaceCaller` to provide SAP functionality to the end user.

The practical effect of removing all these `static` declarations is that another application can no longer directly access the methods of `InterfaceCaller` without first instantiating an instance of the class. In order to instantiate a Java class, you must make sure that it has at least one default constructor. Remember, you can use the default constructor to create an empty instance of a Java class, and you can then use its methods and attributes as a native part of the application. The simplest default constructor uses the name of the Java class as the method name and doesn't take any parameters.

In the case of the `InterfaceCaller` class, the default constructor would look like this:

```
public InterfaceCaller() {...}
```

However, the default constructor is a great place to put code that must be executed when the class is instantiated. Examples of this type of code include initialization logic required by the methods in this class or variables that are set using the parameters (if any) required to call the default constructor.

If you look at the original `MaterialList` class, you find just such logic in the form of the JCo connection and repository initialization. Whereas before your program called the methods to create both instances at the class level of the application, you can now position them more appropriately within the default constructor. This guarantees that the `InterfaceCaller` attempts to establish a connection and retrieve a repository from SAP before any of the methods that call R/3 BAPIs can be executed.

In the following code, the `createConnection()` and `retrieveRepository()` methods have been moved to the end of the `InterfaceCaller` class. The beginning of the class should look like this:

```
import com.sap.mw.jco.*;
import java.util.Hashtable;
import java.util.ResourceBundle;
public class InterfaceCaller {
  ResourceBundle sapProperties = ResourceBundle.getBundle("sapdata");
  public static final String SAP_CLIENT =
         sapProperties.getString("jco.client.client");
  public static final String USER_ID =
         sapProperties.getString("jco.client.user");
  public static final String PASSWORD =
         sapProperties.getString("jco.client.passwd");
  public static final String LANGUAGE =
         sapProperties.getString("jco.client.langu");
  public static final String HOST_NAME =
         sapProperties.getString("jco.client.ashost");
  public static final String SYSTEM_NUMBER =
         sapProperties.getString("jco.client.sysnr");

  private JCO.Client aConnection;
  private IRepository aRepository;

  public InterfaceCaller() {
    createConnection();
    retrieveRepository();
  }
...
}
```

Implementing the getMaterialList() Method

Next, you need to add a new method, called `getMaterialList()`, to `InterfaceCaller` that implements the SAP logic to call `BAPI_MATERIAL_GETLIST`. When you implement this method, you will see that it looks somewhat different from the methods you developed in Chapter 4. Here is what the opening and closing of this new method looks like:

```
import com.sap.mw.jco.*;
import java.util.Hashtable;
import java.util.ResourceBundle;
public class InterfaceCaller {
...
  public Hashtable getMaterialList(String searchParam) {
    ...
    return returnHash;
  }
}
```

This line declares the method to be `public` and to return a `Hashtable` object:

```
public Hashtable getMaterialList(String searchParam)
```

In addition, it declares the method name and a single text string parameter, `searchParam`. By declaring a return object for this method (as opposed to the `void` seen in earlier methods), using an instance of the `InterfaceCaller` class, you tell the application to expect a given Java object to be returned.

In this case, the return is an object of type `java.util.Hashtable`, which is a standard Java class used to store name/value pairs. The last statement in the `getMaterialList()` method, `return returnHash;`, indicates that an instance of the `Hashtable` class populated by this method is returned to the originating Java instance.

The Java `Hashtable` is one of the most widely used mechanisms for quickly storing and retrieving data. It provides very simple methods for setting and getting name/value pairs and can store a wide variety of Java objects. Unlike arrays, the `Hashtable` is not limited to a set number of elements when it is initialized. In Java, when an array is initialized, you have to declare the number of elements in that array, otherwise known as its boundaries. With a `Hashtable`, you simply append new entries to the current instance; this creates additional elements that can be referenced by a key value and modified later in the application.

Understanding HashMaps

Introduced in JDK 1.2, HashMaps are part of the Java Collections Framework. Similar to Hashtables, HashMaps provide highly optimized concrete implementations of the `Map` interface. This simply means that an instance of the `HashMap` class takes advantage of the advanced data structures and algorithms provided by the Java Collections Framework. The major difference is how each class accesses the data stored in it.

Hashtables rely on the `Enumeration` class to retrieve stored values lists, whereas HashMaps return `Collection` views of stored data. These views provide failsafe access to the data via an `Iterator` instance, which has the advantage of allowing elements to be removed during the course of iteration and offers more clarified method names.

The Java Collections Framework deserves a great deal more explanation than can be provided in this book. For more information, check out the excellent tutorials at `http://java.sun.com/docs/books/tutorial/collections/`.

Aside from using the key to retrieve a value from the `Hashtable`, this class provides several methods to pull out a list or enumeration of either key names or values. Table 5-1 lists the various methods in the `Hashtable` class along with a brief explanation of each.

Table 5-1. Synopsis of Methods in Hashtable Class

METHOD NAME	RETURN OBJECT	DESCRIPTION
`put(`*`key`*`, `*`value`*`)`	*`Object`*	Stores Java *`Object`*, *`value`*, in a hashtable mapped to a specified key.
`get(`*`key`*`)`	*`Object`*	Returns a *`value`* reference by *`key`* as Java *`Object`*. *`Object`* should be cast.
`isEmpty()`	Boolean	Tests whether a hashtable has no *`key`*s or *`value`*s.
`elements()`	*`Enumeration`*	Returns a Java *`Enumeration`* containing all *`value`*s in the hashtable.
`keys``()`	*`Enumeration`*	Returns a Java *`Enumeration`* containing all *`key`*s in the hashtable.
`contains(`*`value`*`)`	Boolean	Tests whether the hashtable contains a given Java *`Object`*, *`value`*.
`contains`*`Key`*`(`*`key`*`)`	Boolean	Tests whether the hashtable contains a given Java *`Object`*, *`key`*.

This is not an exhaustive list of all methods in the `Hashtable` class, but it encompasses some of the most commonly used.

So why such a heavy focus on the `Hashtable` class? In Chapter 4, your application used the JCo table structures to retrieve and display data back to the end user. Although these Java classes did get the job done, they are totally reliant on the JCo Java class libraries. In order to build more extensible Java applications, I highly recommended that you allow a certain amount of decoupling to occur between objects and systems in the application framework. Simply put, this means that the Java classes you developed to present an interface to the end user do not need to have specific knowledge of the underlying SAP connector architecture. If you rely on the JCo libraries to display information to the end user, you create a reliance on the SAP connector where none needs to exist.

By using the standard Java `Hashtable` class to store information you want to present to the end user, you create a marked delineation between the various levels of your application. That way, your presentation layer, the desktop client, doesn't need to understand anything about connecting to the R/3 system. This will make more sense as you continue development of the `InterfaceCaller` class.

Adding JCo Logic to getMaterialList()

This part is fairly straightforward because you have already developed much of the functionality for this method in the previous chapter.

```
import com.sap.mw.jco.*;
import java.util.Hashtable;
import java.util.ResourceBundle;
public class InterfaceCaller {
...
  public Hashtable getMaterialList(String searchParam) {
//The getFunction() method is a new method in this class
    JCO.Function function = this.getFunction("BAPI_MATERIAL_GETLIST");
    JCO.ParameterList tabParams = function.getTableParameterList();
    JCO.Table materials = tabParams.getTable("MATNRSELECTION");

    materials.appendRow();
    materials.setRow(0);
    materials.setValue("I", "SIGN");
    materials.setValue("CP", "OPTION");
    materials.setValue(searchParam, "MATNR_LOW");
    aConnection.execute(function);

    JCO.ParameterList resultParams = function.getExportParameterList();
    JCO.Table materialList =
                    function.getTableParameterList().getTable("MATNRLIST");

    Hashtable returnHash = new Hashtable();
    Hashtable rowHash = new Hashtable();
    int i = 0;
    if (materialList.getNumRows() > 0) {
        do {
            for (JCO.FieldIterator fI = materialList.fields();
                 fI.hasMoreElements();)
              {
               JCO.Field tabField = fI.nextField();
               rowHash.put(tabField.getName(),tabField.getString ());
             }
              returnHash.put("line" + i, rowHash);
              rowHash = new Hashtable();
              i++;
            }
            while(materialList.nextRow() == true);
    }
    else {
          System.out.println("Sorry, couldn't find any materials");
    }
    return returnHash;
  }
  ...
}
```

The first difference between this code and the Chapter 4 application is in the statement that follows the method declaration.

```
JCO.Function function = this.getFunction("BAPI_MATERIAL_GETLIST");
```

This line calls another method within the `InterfaceCaller` class, as indicated by the implied class name `this`. Using `this` as the instance against which to call a method tells the Java Virtual Machine (JVM) to look for that method, in this case `getFunction()`, within the current class. The `getFunction()` method provides generic access to the JCo logic required to create a `JCO.Function` object. This method retrieves a `JCO.FunctionTemplate` based on the name of an RFC and then returns a new instance of the `JCO.Function` class. Here is that method:

```
  public JCO.Function getFunction(String name) {
    try {
       return aRepository.getFunctionTemplate(name.toUpperCase()).getFunction();
    }
    catch (Exception ex) {
     ex.printStackTrace();
      return null;
    }
  }
```

If you review Chapter 4, you will see that the JCo logic used to initialize the `function` instance has been extracted and added to the `getFunction()` method. Encapsulating this common technical logic in a method allows you to reuse code throughout the additional methods of your `InterfaceCaller` class.

Likewise, you will notice that the array used to retrieve a search text string from the command line has been removed. Instead, the method uses the `searchParam` text string passed as a parameter when the method is called.

A few lines further down, you see the first use of that much-heralded `Hashtable` class, in the form of two variables that are initialized using the `Hashtable`'s default constructor.

```
    Hashtable returnHash = new Hashtable();
    Hashtable rowHash = new Hashtable();
```

The rowHash is used to represent individual rows in the table returned from SAP, and returnHash stores all of those rows as a representation of the R/3 table.

Since you have already dealt with using the JCo libraries to pull data out of Java-based SAP tables, now you need only to understand how to put that information into the respective Hashtables. The following code snippet demonstrates how the application populates a Java Hashtable:

```
        do {
            for (JCO.FieldIterator fI = materialList.fields();
                 fI.hasMoreElements();)
              {
                JCO.Field tabField = fI.nextField();
//This line uses the field name and value to populate a hashtable
                rowHash.put(tabField.getName(),tabField.getString ());
              }
            returnHash.put("line" + i, rowHash);
            rowHash = new Hashtable();
            i++;
        }
        while(materialList.nextRow() == true);
```

Instead of writing the materials records out to the console display using println() statements, the above code snippet adds those field names and values to the temporary rowHash instance, then it appends rowHash to the returnHash Hashtable.

```
rowHash.put(tabField.getName(),tabField.getString());
```

As noted previously, the JCO.FieldIterator provides a representation of all the field names and values in a given table row. Using the getName() and getString() methods on the tabField instance, the application maps the methods' returns to the key and value parameters of the returnHash.put() method. Mapping data to a standard Java object serves to create a Hashtable representation of that same JCO.FieldIterator table row. This new Hashtable record can then be stored in the returnHash instance through the following statement:

```
                returnHash.put("line" + i, rowHash);
```

Having created an integer, i, to act as a counter, the rowHash instance is stored as a value mapped to key "item*X*," where *X* is the current value of the counter. The rowHash instance is then reinitialized to a new Hashtable so that it can be used as the next table row, and the line item counter is incremented.

```
rowHash = new Hashtable();
i++;
```

NOTE *The Java programming syntax provides a number of shortcuts for writing expressions. The expression* `i++;` *can also be written as* `i = i + 1;`*. The outcome of either is the same—namely the value of* `i` *is incremented by 1—but the former is preferred for its more widely recognized clarity.*

Once the do...while loop has completed, the getMaterialList() method returns the newly populated returnHash instance, thus finishing the method call.

TIP *Because these examples are meant strictly for educational purposes, much of the recommended error and exception handling that should be part of a more robust development effort has been left out. In the previous example, you would likely add logic to determine whether the call to SAP was successful and to check whether any material records were returned. If the call to SAP was unsuccessful, you would want the* `getMaterialList()` *method to return either a Java exception or a* `null` *value rather than an empty* `Hashtable` *instance. The type of return or exception would depend on how this class needed to interact with your overall application framework.*

Your InterfaceCaller class should now look something like this:

```
import com.sap.mw.jco.*;
import java.util.Hashtable;
import java.util.ResourceBundle;
public class InterfaceCaller {

  ResourceBundle sapProperties = ResourceBundle.getBundle("sapdata");
  public static final String SAP_CLIENT =
         sapProperties.getString("jco.client.client");
  public static final String USER_ID =
         sapProperties.getString("jco.client.user");
  public static final String PASSWORD =
         sapProperties.getString("jco.client.passwd");
  public static final String LANGUAGE =
            sapProperties.getString("jco.client.langu");
    public static final String HOST_NAME =
            sapProperties.getString("jco.client.ashost");
    public static final String SYSTEM_NUMBER =
            sapProperties.getString("jco.client.sysnr");

    private JCO.Client aConnection;
    private IRepository aRepository;

    public InterfaceCaller() {
      createConnection();
      retrieveRepository();
  }

    public Hashtable getMaterialList(String searchParam) {
      JCO.Function function = this.getFunction("BAPI_MATERIAL_GETLIST");
      JCO.ParameterList tabParams = function.getTableParameterList();
      JCO.Table materials = tabParams.getTable("MATNRSELECTION");

      materials.appendRow();
      materials.setRow(0);
      materials.setValue("I", "SIGN");
      materials.setValue("CP", "OPTION");
      materials.setValue(searchParam, "MATNR_LOW");
      aConnection.execute(function);

      JCO.ParameterList resultParams = function.getExportParameterList();
      JCO.Table materialList =
          function.getTableParameterList().getTable("MATNRLIST");

      Hashtable returnHash = new Hashtable();
      Hashtable rowHash = new Hashtable();
      int i = 0;
      if (materialList.getNumRows() > 0) {
          do {
              for (JCO.FieldIterator fI = materialList.fields();
                   fI.hasMoreElements();)
                {
                  JCO.Field tabField = fI.nextField();
                  rowHash.put(tabField.getName(),tabField.getString());
             }
              returnHash.put("line" + i, rowHash);
              rowHash = new Hashtable();
              i++;
          }
```

```
            while(materialList.nextRow() == true);
        }
        else {
               System.out.println("Sorry, couldn't find any materials");
        }
        return returnHash;
      }

      public Hashtable checkPassword(String username, String password) {

        JCO.Function function = getFunction("BAPI_CUSTOMER_CHECKPASSWORD");
        JCO.ParameterList listParams = function.getImportParameterList();
        listParams.setValue(username, "CUSTOMERNO");
        listParams.setValue(password, "PASSWORD");

        aConnection.execute(function);

        JCO.ParameterList resultParams = function.getExportParameterList();
        Hashtable returnHash = new Hashtable();
        returnHash.put("RETURN.TYPE",extractField("RETURN","TYPE",resultParams));
        returnHash.put("RETURN.CODE",extractField("RETURN","CODE",resultParams));
        returnHash.put("RETURN.MESSAGE",
                        extractField("RETURN","MESSAGE",resultParams));
        return returnHash;
        }

      public String extractField(String structure,String field,
                                          JCO.ParameterList parameterList)
      {
          return ((JCO.Structure)parameterList.getValue(structure)).getString(field);
      }

      public JCO.Function getFunction(String name) {
        try {
             return
    aRepository.getFunctionTemplate(name.toUpperCase()).getFunction();
        }
        catch (Exception ex) {}
          return null;
        }
```

```
private void createConnection() {
  try {
    aConnection = JCO.createClient(SAP_CLIENT,
                                   USER_ID,
                                   PASSWORD,
                                   LANGUAGE,
                                   HOST_NAME,
                                   SYSTEM_NUMBER);
    aConnection.connect();
    }
    catch (Exception ex) {
      System.out.println("Failed to connect to SAP system");
    }
  }

private void retrieveRepository() {
  try {
    aRepository = new JCO.Repository("SAPRep", aConnection);
    }
    catch (Exception ex) {
      System.out.println("Failed to retrieve SAP repository");
    }
  }
}
```

Run the Java compiler and you should get back a new Java class, `InterfaceCaller.class`. In order to test `InterfaceCaller`, you need to build a quick little test class. This class contains simply a `main` method that calls a specific method on `InterfaceCaller`. The following class uses the `getMaterialList()` method to demonstrate that `InterfaceCaller` can retrieve materials from SAP.

```
  import java.util.Hashtable;
  import java.util.Enumeration;
  public class SapTest {

  public static void main(String[] args) {
  //Change this variable to narrow search criteria
    String materialSearch = "*";
    InterfaceCaller iCaller = new InterfaceCaller();
    Hashtable resultHash = iCaller.getMaterialList(materialSearch);
    Hashtable tempRow;
```

```
      for (Enumeration e = resultHash.elements(); e.hasMoreElements();) {
        tempRow = (Hashtable)e.nextElement();
        System.out.println("Material: " +
            (String)tempRow.get("MATERIAL") +
            System.getProperty("line.separator"));
        System.out.println("Material description: " +
            (String)tempRow.get("MATL_DESC") +
            System.getProperty("line.separator"));
      }
    }
  }
```

Save and compile this file as `SapTest.java` in the same directory as your compiled `InterfaceCaller` class. Execute `SapTest` and your results should look something like this:

```
Material: MATERIAL1

Material description: Material 1
Material: MATERIAL2
Material description: Material 2
Material: MATERIAL3
Material description: Material 3
Material: MATERIAL4
Material description: Material 4
```

The next section details how to use `InterfaceCaller` to provide SAP connectivity from a graphical user interface.

Developing a Graphical User Interface

The final section in this chapter deals with developing a desktop client application that can be deployed across different computers within your company's local area network (LAN). The graphical user interface (GUI) refers to the visual representation of information and how a user interacts with it through accepted metaphors, such as buttons, menus, and scroll bars. In contrast, your earlier Java application required the user to enter and view all application data through a command line, relying solely on the keyboard for input and text characters for output.

Times have changed—green screens and dumb terminals are a thing of the past. Most end users are not hardened terminal-heads and they expect applications to adhere to general windows-like conformity. Certainly, as you develop new Java classes, you conduct initial unit testing and debugging from a command line interface. In fact, I recommend that you become more comfortable with the command line because it provides a very ready and incredibly useful way to quickly build prototype applications or experiment with new programming techniques.

However, you must always keep end users in mind; regard them as your ultimate customers, upon whose satisfaction your position likely rests. Visual interface or GUI development can often be very frustrating, particularly when you are dealing with the Java 2 programming specification. This is due more to the current lack of available strong visual development tools, including the list of integrated development environments (IDEs) mentioned in Chapter 4. On the positive side, Java 2 has (re)introduced a strong set of application libraries that ease the development effort around GUI programming and, with a little practice, can even be fun to work with.

Even if you don't think GUI development is in your near future, you will find that having experience with these Java application libraries is invaluable. Although you are very likely eager to move on to Web application programming, which seems to be the stuff of many high profile corporate projects these days, you need to cover the basics first. In the process, you will very likely need some level of GUI development, even if it just means providing a user-friendly interface for configuring the properties files.

Managing the Graphical User Interface

Developing an application in Java Swing may seem daunting at first, but you can break it down into several manageable pieces. In order to grasp the overall picture, you must understand the various pieces in a Swing application.

NOTE *Sun provides the Java Swing API as part of its Java Foundation Classes (JFC). The JFC is used to support graphical interface design and functionality across a variety of operating system platforms. You can find more information on JFC at Sun's Java Foundation Classes Website* (`http://java.sun.com/products/jfc/`).

At the most basic level, Swing, and most GUI application development, is comprised of two major framework components—windows and buttons. You use windows to take data from the end user and display it back throughout the process of the application. Buttons provide interactivity—they allow the end user to tell the program when to begin certain processes, and they provide access to the underlying functions of the application.

Windows can be display only, entry only, or both. The most basic window displays text strings to the end user, commonly in the form of error, warning, or informational dialogs. These types of windows require little underlying logic and provide no interaction other than to close the dialog box, that is. However, every windows-based application inherits from the very simple premise of a dialog window and expands on it until you are left with overly complicated, multilayered, button-laden applications that require a master's degree just to get through the user license during installation. Of course, your new Java GUI application for SAP suffers under no such onus.

Unbeknownst to the end user, most GUI applications comprise a series of nested windows, visible or otherwise, that can be used to build a cohesive application front end. Grouping tabbed windowpanes within a larger window frame is a very common mechanism for providing quick access to a wide range of information, which can be bundled logically within each individual pane. Although not a subject in this book, Java Swing provides some very sophisticated windowing mechanisms that have been used to build commercial grade desktop applications. To learn more, visit the excellent tutorial on Java Swing development at `http://java.sun.com/docs/books/tutorial/uiswing/`.

Windows can also be input fields, which are designed to allow text-based entry of field data. Similar to the example provided later in "Building a Java GUI for SAP," the material search parameter provides a window in which the user can enter a full or partial material identifier to search against SAP's material master database. Windows can also be text areas, such as the white space in a text editor where you enter text to be saved to a file in the file system. You will also see an example of a text area in the following application, specifically used as a display mechanism, rather than as a means to take data entry from the end user.

Figure 5-1 depicts a diagrammatic user interface and represents some of the most commonly used elements in UI design.

Buttons, in their many forms, allow the user to interact with data on the screen and describe how that data can be manipulated. The general class of buttons we deal with here go far beyond the more usual idea of the vaguely rectangular, slightly gray, text- or picture-enhanced icons that you see in everything from Web pages to the SAP GUI. Buttons can be radio style (multiple, small, and circular), check boxes, highlighted text, menu bars, and so forth.

The definition of a button is really just a visual representation of underlying application logic. Better representations lead to easier-to-use applications, which lead to happier end users. Of course, the major challenge surrounding GUI development is not the language syntax or your conceptual understanding of it, but the ultimate implementation of the user interface, as evidenced by the myriad of painfully designed Websites you are forced to visit every day.

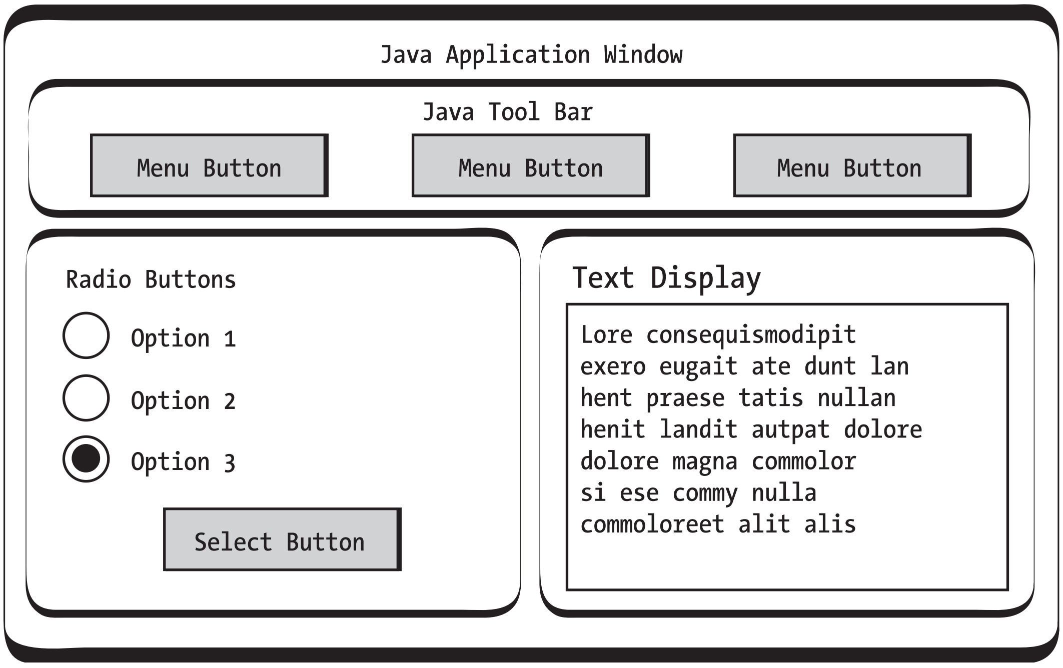


Figure 5-1. Overview of GUI windowing components

The button can be the user's best friend or her worst enemy. As the developer, your goal is to make your application as simple and user friendly as possible, a prospect that Java Swing does not always aid.

Building a Java GUI for SAP

Now that you have a better idea of the conceptual basics that underlie the JFC and the Swing application libraries, it is time to start building the application.

This section demonstrates a different type of user interface and Java class definition than the command line interface introduced in Chapter 4. Although your new class still relies on a `main` method, much of the actual GUI implementation resides in a default constructor.

Introducing the MaterialGui Class

Start off by creating a new Java class called `MaterialGui.java` and save this file into your development directory.

Next, import the appropriate Swing application libraries along with the standard Java libraries you need to support interaction with the `InterfaceCaller` class.

Remember, you need the `java.util` package to return the `Hashtable` that the methods of `InterfaceCaller` send back.

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;

public class MaterialGui implements ActionListener {
  ...
}
```

Similar in appearance to previous class declarations, the `MaterialGui` class implements an interface that allows your program to receive action events and process them according to your internal application logic. The `ActionListener` interface provides you with a key method that you need to implement within the application. Where and why this method is used becomes clear very shortly.

Like the `InterfaceCaller` class, certain variables in `MaterialGui` are declared at the class level to make them easier to use throughout the application. In order to develop a graphical interface in Java, the developer must use key objects to build the visual framework and provide the end user with interaction and display.

```
...
public class MaterialGui implements ActionListener {
  JFrame sapFrame;
  JPanel sapPanel;
  JTextField matSearch;
  JButton retrieveList;
  JTextArea resultText;
  JPanel selectPane;
  JPanel displayPane;
  ...
}
```

Table 5-2 gives a quick overview of each instance and describes its responsibility.

Table 5-2. Java Swing Application Components

COMPONENT NAME	**DESCRIPTION**
`JFrame` *`sapFrame`*	Overall window frame. Contains all subsequent GUI components.
`JPanel` *`sapPanel`*	Top-level panel used to house and organize content panels.
`JTextfield` *`matSearch`*	Single text field that allows user to enter material search criteria.
`JButton` *`retrieveList`*	Clickable button that initiates a call to SAP.
`JTextArea` *`resultText`*	Scrollable text area that displays a list of materials that match search criteria.
`JPanel` *`selectPane`*	Panel that organizes a text field and a button.
`JPanel` *`displayPane`*	Panel that provides layout for a text area display.

This table does not describe all of the possible Swing components or their various uses within a graphical development framework. Rather, it describes the components you will use in application developed in this chapter and how they specifically relate to the functionality provide by the `MaterialGui` class.

In order to execute the `MaterialGui` class from a command line, you must implement the `main` method to house the application start-up logic.

```
public class MaterialGui implements ActionListener {
  ...
   public static void main(String[] args) {
    try {
      UIManager.setLookAndFeel(
      UIManager.getCrossPlatformLookAndFeelClassName());
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    MaterialGui matGui = new MaterialGui();
  }
}
```

This is a very basic implementation of the `main` method that could be used in GUI development, but you would likely want to expand or refactor it before you used it in a real-world application. For now, it demonstrates the basic requirements that surround setting up the application's look and feel, as well as calling a second method in your class to provide the visual components for the end user.

Defining Look and Feel with the UIManager

The `UIManager` is part of the `javax.swing.*` package and its purpose is to define and organize the look and feel of your application. Generally speaking, Java Swing development allows your application to instantly conform to a variety of cross-platform visual interfaces. This means that your application can run just as effectively within a UNIX environment as it can within the MS Windows operating system.

The *look and feel* of an application describes the visual properties such as the Maximize, Minimize, and Close buttons, in addition to overall button, field, and display formatting. That Java 2 is a platform-independent language means that your application tries to determine the default look and feel of the operating system and use those visual elements to conform to that system's standard graphical representation.

Normally, you can allow the `UIManager` to determine the default look and feel from a properties file found in the Java installation. If this file cannot be found, the UIManager uses the look and feel defined by the Java 2 specification, otherwise known as the Java Look and Feel (JLF). In this case, the statement

```
UIManager.getCrossPlatformLookAndFeelClassName());
```

tells the application to use the JLF by default and not to worry about determining or allowing the user to define the application's look and feel.

The last line of the `main` method creates a new instance of the `MaterialGui` class (yes, the same class that is currently being executed) using its default constructor.

```
    MaterialGui matGui = new MaterialGui();
```

You will recall that using the default constructor is a great way to execute application logic as soon as the class is instantiated, and it does not require any further developer interference. Using the default constructor is appropriate when you are creating a GUI application because you must create the visual components before any user interaction can take place.

Interacting with the User Through ActionListener

Before you jump into using the default constructor, however, there is one other method that you must implement in order to provide SAP functionality for the end user. This is the method mentioned earlier as part of the ActionListener interface; you need it to translate the user's interaction with the GUI into an actual call to your InterfaceCaller class.

```
public class MaterialGui implements ActionListener {
  ...
  public void actionPerformed(ActionEvent event) {
  InterfaceCaller iCaller = new InterfaceCaller();
  Hashtable returnHash = iCaller.getMaterialList(matSearch.getText());
  Hashtable rowHash;
  for (Enumeration e = returnHash.elements(); e.hasMoreElements();) {
    rowHash = (Hashtable)e.nextElement();
    resultText.append("Material: " +
       (String)rowHash.get("MATERIAL") + "\n");
    resultText.append("Material description: " +
       (String)rowHash.get("MATL_DESC") + "\n\n");
    }
  }
  ...
}
```

The actionPerformed() method is automatically called when the user interacts with a visual component, such as a button, that the application has registered as an action listener. In this case, your application has only one such component, the retrieveList button, which is registered as an action listener within the MaterialGui class's default constructor. The actionPerformed() method takes a single parameter, an ActionEvent object. This object simply tells the actionPerformed() method about the event and its source, details that are hidden in the Swing library classes so that the developer doesn't need to worry about the code-level implementation.

Within the actionPerformed() method, the first step is to instantiate a new instance of your InterfaceCaller class. Remember that all of the SAP connection logic buried in that class's default constructor? That is handled automatically through the simple instantiation of InterfaceCaller and requires no specific knowledge of SAP, the JCo connector, or even the type of ERP system used on the backend.

```
  InterfaceCaller iCaller = new InterfaceCaller();
  Hashtable returnHash = iCaller.getMaterialList(matSearch.getText());
  Hashtable rowHash;
```

The only knowledge of InterfaceCaller the developer needs at this point is the name of method, the parameter(s) required, and the type of Java object returned. Currently, InterfaceCaller has only one method to call SAP, getMaterialList(). This method takes a single text string as its sole parameter and returns a standard Java Hashtable. Notice that using InterfaceCaller does not require the use of any JCo application libraries or any specific knowledge of the underlying SAP connectivity. By encapsulating the R/3 data returns in a standard Java Hashtable, you effectively release the developer using InterfaceCaller from having to learn the JCo connector, thereby making any future development around this class much simpler.

Populating Data in a Hashtable

The returnHash instance is populated by the object returned from the call to the getMaterialList() method on the iCaller instance of InterfaceCaller. In the following line, you initialize a Hashtable called rowHash, which is used as temporary storage for each individual row contained in the returnHash instance.

```
for (Enumeration e = returnHash.elements(); e.hasMoreElements();) {
  rowHash = (Hashtable)e.nextElement();
  resultText.append("Material: " +
            (String)rowHash.get("MATERIAL") + "\n");
  resultText.append("Material description: " +
            (String)rowHash.get("MATL_DESC") + "\n\n");
  }
```

Once the application has a populated instance of returnHash, you can use the for loop to loop through the Hashtable and write the results back to the JTextArea, resultText. The for loop instantiates a new Enumeration, e, using the elements() method of returnHash. This Enumeration instance contains strictly a list of Hashtable elements, which are the values in the name/value pairs contained within returnHash.

At this point, you are not concerned with the order in which the line items are returned by SAP, so using the elements() method to retrieve an unordered list of the values in returnHash will suffice. The for loop also specifies the termination expression e.hasMoreElements(), which tests whether the enumeration contains additional elements, and ends the loop when this method returns a Boolean value of FALSE.

Ordering Data in a Hashtable

Retrieving data from a `Hashtable` using the `elements()` method does not guarantee a list ordered in the way the data was originally stored. If you need to pull out the elements in the same order in which they were stored, that is, the line item order returned by SAP, you must retrieve each value individually, using the name stored as a key.

In the Chapter 4 code examples, these keys were stored as `item`*`XX`*, where *`XX`* represents the individual line number of each item. You could increment an integer variable, then use the `Hashtable.get()` method, and pass in a String parameter of ““item” + integer” to incrementally retrieve the elements of that `Hashtable`. Here is an example of that code:

```
int i = 0;
while (i < 100) {
  (String)myHash.get("item" + i);
  i++;
}
```

Additionally, the Java Collections Framework offers abstract and concrete implementations of sort and store algorithms to provide ordered list functionality. See the “Understanding HashMaps” sidebar earlier in this chapter for more information.

The first line in the `for` loop sets the value of `rowHash` equal to the result of a call to the `nextElement()` method on the `Enumeration`, `e`. Again, you must cast the Java object returned by this method as a `Hashtable`, otherwise the JVM is unable to determine how to treat this object.

```
resultText.append("Material: " +
          (String)rowHash.get("MATERIAL") + "\n");
resultText.append("Material description: " +
          (String)rowHash.get("MATL_DESC") + "\n\n");
```

TIP *As you saw in the previous chapter, escape characters can be used to provide limited character formatting in the text strings. Escape characters, specified as a backslash (\) followed by a reserved character, are a simple mechanism to make the output text more legible to the end user. Try using the tab escape,* `\t`*, to line up the field name columns, or* `\"` *if you need to place printed quotes around a text string. Likewise, you can add additional text to better delineate the difference between material records, such as dashes, asterisks, and hash marks.*

As stated earlier, the `rowHash` instance is used as temporary storage for a given SAP table row retrieved from `returnHash`. The `rowHash` instance now contains a list of name/value pairs, each representing a different field name and field value of a given row in the table. For this simple application, you need to display only the `MATERIAL` and `MATL_DESC` fields contained in `rowHash`, so using the `get()` method on this instance and passing in the field name as key parameter is sufficient.

Displaying Information Using JTextArea

In order to display this information to the end user in a text area, the application uses the `resultText` instance initialized as a class-level variable. Remember, by using a class-level variable, any logic you build into your application can access and modify this instance. Through this mechanism, you do not need to worry about passing variables to the `actionPerformed()` method, which you cannot do, regardless, because `ActionEvent` is the only valid parameter that can be passed to this method.

By using the `append()` method of `resultText`, you can easily push strings of text back to the text area. This is a very basic approach to displaying lists of data back to the end user, but it suffices for this relatively simple application. As you explore the Java Swing libraries, you will find much more advanced mechanisms to providing more creative and interactive display results to the end user.

The `JTextArea` class provides several different methods that are used to set the way text characters are displayed and how the user interacts with the text area. Table 5-3 shows a few of these methods and provides a brief description for each. Feel free to try out these methods on the `resultText` instance and see how they affect your application.

Table 5-3. Methods of `JTextArea`

METHOD NAME	DESCRIPTION
`append(String text)`	Adds the *text* string specified to the end of the current `JTextArea` document.
`insert(String text, int pos)`	Inserts the specified *text* string at the position of the integer value.
`replaceRange(String text, int start, int end)`	Replaces the characters specified by the *start/end* integers with the value of the *text* string.
`setFont(Font f)`	Uses a Java `Font` instance to set the font display; check out the Java documentation on the API class `java.awt.Font` for more information.
`setLineWrap(boolean wrap)`	Sets the text area to wrap text based on a boolean value of `TRUE` or `FALSE`
`setTabSize(int size)`	Sets the number of characters to which the tab key expands.
`new(String text, int rows, int cols)`	Uses the default constructor to set the initial *text* string entry in text area. This is in addition to the text area set using integer values for rows and columns.

After implementing the `actionPerformed()` method, you are ready to build the GUI interface and use the `InterfaceCaller` logic built into this method, as shown here:

```
public class MaterialGui implements ActionListener {
  ...
  public MaterialGui() {
//Initialize and set visual framework components
    sapFrame = new JFrame("Material Search for SAP R/3");
    sapPanel = new JPanel();
    sapPanel.setLayout(new BorderLayout());
    selectPane = new JPanel();
    displayPane = new JPanel();
//Initialize and set input and output components
    matSearch = new JTextField(20);
    resultText = new JTextArea(10,30);
    resultText.setEditable(false);
    JScrollPane resultPane = new JScrollPane(resultText,
                                       JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,

JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
//Implement button with keyboard access and set listener
    retrieveList = new JButton("Get materials...");
    retrieveList.setMnemonic(KeyEvent.VK_ENTER);
    retrieveList.addActionListener(this);
//Add individual panels to overall application frame
    selectPane.add(matSearch);
    selectPane.add(retrieveList);
    displayPane.add(resultPane);
//Define panel layout within frame
    sapPanel.add(selectPane, BorderLayout.NORTH);
    sapPanel.add(displayPane, BorderLayout.SOUTH);
    sapFrame.getContentPane().add(sapPanel);
    sapFrame.getRootPane().setDefaultButton(retrieveList);
//General cleanup for close and frame implementation
    sapFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    sapFrame.pack();
    sapFrame.setVisible(true);
  }
  ...
}
```

Housing Visual Components with JFrame

You are finally ready to implement the default constructor that houses the application logic for building the visual interface. As seen previously, `public MaterialGui() {...}` declares the constructor using the name of the Java class, `MaterialGui`.

Next, you initialize the various GUI components that were declared as class-level variables.

```
//Initialize and set visual framework components
    sapFrame = new JFrame("Material Search for SAP R/3");
    sapPanel = new JPanel();
    sapPanel.setLayout(new BorderLayout());
    selectPane = new JPanel();
    displayPane = new JPanel();
```

The `sapFrame` instance is a `new` instance of the `JFrame` class, to which you pass the title displayed above the menu bar in the application's window. Similarly, you instantiated a `new` instance of `JPanel`, called `sapPanel`, which houses the subpanels of the GUI. Through the `setLayout()` method of `sapPanel`, the application is able to determine the method by which subsequent components are displayed in this panel. Java Swing provides a number of different layout mechanisms, each serving a slightly different purpose for displaying components.

`BorderLayout` gives the panel five distinct areas in which components can be placed: north, south, east, west, and center. This is a very straightforward means to layout components, enabling you to use commonly known directional terms to specify component placement. You only need to pass a newly instantiated instance of the `BorderLayout` class to the `setLayout()` method, with no parameters, so using the expression `new BorderLayout()` will suffice.

The last two lines instantiate the content panels, `selectPane` and `displayPane`, that are to be contained within the main panel, `sapPanel`. Nesting panels is a very common practice in Swing because it allows the developer more exact control over the placement of various components within the graphical user interface. Figure 5-2 shows the various user interface components as represented by their Java class names.

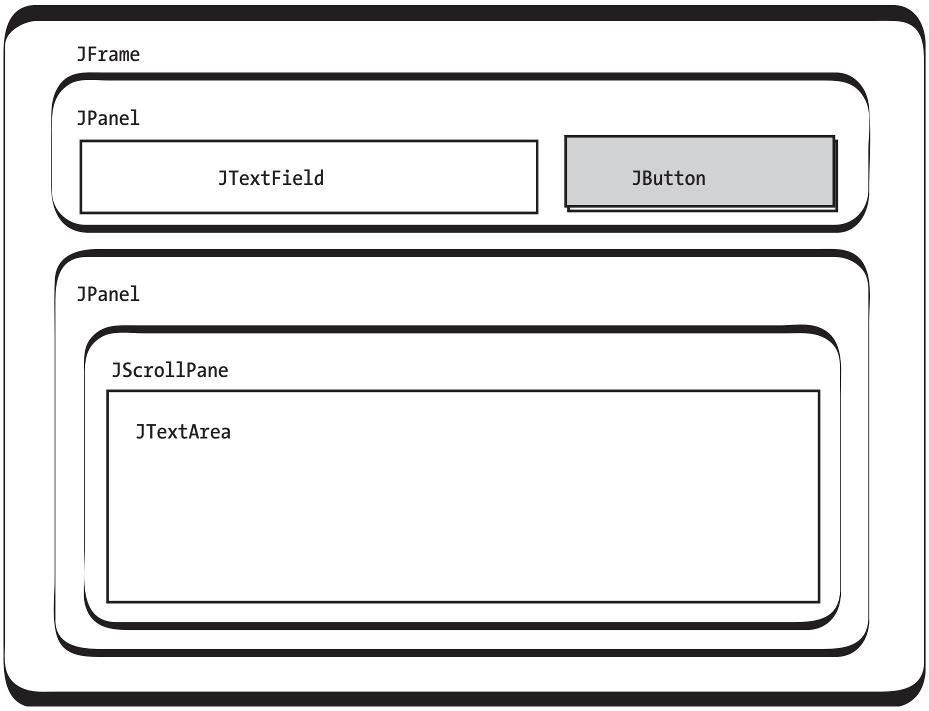


Figure 5-2. Java Swing layout used by the `MaterialGui` *class*

The next block of code initializes the various Java UI variables that were declared previously:

```
//Initialize and set input and output components
    matSearch = new JTextField(20);
    resultText = new JTextArea(10,30);
    resultText.setEditable(false);
    JScrollPane resultPane = new JScrollPane(resultText,
                                       JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,

JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);
```

The application initializes a new `JTextField` called `matSearch`, which serves as a means for gathering input from the end user. Recall that this instance is used within the `actionPerformed()` method to retrieve the text string the user entered and pass it to the `getMaterialList()` method of `InterfaceCaller`. The single parameter used by `JTextField`'s default constructor is an integer value that specifies the display length of the field box in the GUI. This value does not limit the number of characters that can be entered into the `JTextField` instance, `matSearch`, rather, it limits the length of the field box displayed to the end user.

The subsequent statement instantiates a new `JTextArea` instance, called `resultText`, that is used to display the material list return from SAP. Like the `matSearch` instance, you have also seen `resultText` accessed in the `actionPerformed()` method, in this case to display text results back to the end user. This instance is also created using two integer values to limit the display size of the text area. The first value specifies the number of rows, and the second specifies the number of columns. As with `JTextField`, these values do not limit the amount of data displayed, only the amount of screen real estate used by the text area.

You will also likely want to restrict the text area display so that the end user cannot edit the list of materials that SAP returns. Regardless, this editing ability cannot affect SAP, only the user's ability to edit the display on the local computer. By passing a parameter of `FALSE` to the `setEditable()` method of `resultText`, you make it impossible for the user to change the text of the material list. The default behavior for a `JTextArea` allows the user to modify any text displayed here, so be sure you use the `setEditable()` method to turn this capability off.

Lastly, in order to make the display contents of `resultText` scrollable, you must instantiate an instance of `JScrollPane`, passing in `resultText` and two field attributes. The field attributes define certain characteristics of the scroll bar, such as whether or not to use horizontal and vertical scroll bars. Looking at the application library documentation for the `JScrollPane` class gives you a more comprehensive look at what fields are available for use in the default constructor.

```
//Implement button with keyboard access and set listener
    retrieveList = new JButton("Get materials...");
    retrieveList.setMnemonic(KeyEvent.VK_ENTER);
    retrieveList.addActionListener(this);
```

In this code snippet, the application instantiates a new `JButton` instance, `retrieveList`, passing in the text string displayed on the button's face as the sole parameter. `JButton` also has several methods that the application calls on the `retrieveList` instance. The `setMnemonic()` method creates a keyboard shortcut so that the button can be clicked using ALT-*XX* from the keyboard, where *XX* is a predefined key. In this case, the application passes a field attribute, `KeyEvent.VK_ENTER`, which sets the shortcut to ALT-Enter on the keyboard.

Finally, the `retrieveList` instance uses its `setActionListener()` method to register this object as an action listener on the class `this`. Remember, `this` is a reserved keyword in Java that points to the current class. In effect, you are telling the `setActionListener()` method to look in the `MaterialGui` class for the appropriate methods to handle an action event sent from the `retrieveList` button. By implementing the `ActionListener` class within the class declaration for `MaterialGui`, you allow `MaterialGui` to be registered as an action listener. Furthermore, by implementing the `actionPerformed()` method of the `ActionListener` class as part of `MaterialGui`, when you click the Get Materials button, any logic built into this method is automatically invoked.

```
//Add individual panels to overall application frame
    selectPane.add(matSearch);
    selectPane.add(retrieveList);
    displayPane.add(resultPane);
```

You are now ready to add the newly populated content components to their respective application panels. The text field and button need to be housed in the `selectPane`, which is the panel displayed at the top of the application. Using the `add()` method on `selectPane`, the application passes in `matSearch` and `retrieveList` as two different calls to this method. Likewise, the pane containing the display text area `resultPane` needs to be added to `displayPane` using that instance's `add()` method.

```
//Define panel layout within frame
    sapPanel.add(selectPane, BorderLayout.NORTH);
    sapPanel.add(displayPane, BorderLayout.SOUTH);
    sapFrame.getContentPane().add(sapPanel);
    sapFrame.getRootPane().setDefaultButton(retrieveList);
```

Now things get a little more complicated. Recall that the application has used the setLayout() method on sapPanel to set the layout to an instance of BorderLayout. In order to use the layout management features of the BorderLayout class, you must specify one of five areas in which to place a given component: north, south, east, west, and center. As you add each pane to sapPanel, you must also tell the instance in which quadrant to locate that component. Using the add() method of sapPanel, you pass two parameters: the JPanel instance that maintains the application's content components, and a field attribute that specifies where to place that pane. In the first line or this code, selectPane is being added to sapPanel and placed in the north quadrant of that pane. Then similar to selectPane, on the next line, displayPane is also added to sapPanel; however, it is being placed in the south quadrant.

Once the content panes have been populated, you are ready to add the main pane into the window frame. Using a straightforward nested method call series, you retrieve the sapFrame instance's contentPane object using the getContentPane() method. Then, you call the add() method on this object, passing in the sapPanel instance populated earlier. Ultimately, these methods allow you to push content panes out to the main JFrame instance, sapFrame, which then displays those panes and their content components back to the end user.

```
//General cleanup for close and frame implementation
    sapFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    sapFrame.pack();
    sapFrame.setVisible(true);
```

Now that the panes and frame have all been populated, formatted, and set, the final detail is to tell the Java Swing libraries to kick them off to the end user. However, before you do that, you might also want to set a close operation so that the user can exit cleanly from the application.

The setDefaultCloseOperation() sets the default action that occurs when the user clicks the Close button on the current window. Because this is a single window application, you probably want the application to exit completely when the user clicks the Close button. By passing the field attribute JFrame.EXIT_ON_CLOSE to this method, you cause the JVM to exit the application and shut down.

Calling the pack() method on sapFrame causes the window to be sized appropriately and made displayable if it is not already. The setVisible() method simply tells the JVM whether to make the window visible or not based on the value of the boolean passed as a parameter. In this case, you set this value to TRUE so that the window and all of its components are visible to the end user.

Bringing all of that together, your application should look something like this:

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
```

```
public class MaterialGui implements ActionListener {
  JFrame sapFrame;
  JPanel sapPanel;
  JTextField matSearch;
  JButton retrieveList;
  JTextArea resultText;
  JPanel selectPane;
  JPanel displayPane;

  public MaterialGui() {
    sapFrame = new JFrame("Material Search for SAP R/3");
    sapPanel = new JPanel();
    sapPanel.setLayout(new BorderLayout());
    selectPane = new JPanel();
    displayPane = new JPanel();

    matSearch = new JTextField(20);
    resultText = new JTextArea(10,30);
    resultText.setEditable(false);
    JScrollPane resultPane = new JScrollPane(resultText,
                                      JScrollPane.VERTICAL_SCROLLBAR_ALWAYS,

JScrollPane.HORIZONTAL_SCROLLBAR_ALWAYS);

    retrieveList = new JButton("Get materials...");
    retrieveList.setMnemonic(KeyEvent.VK_ENTER);
    retrieveList.addActionListener(this);

    selectPane.add(matSearch);
    selectPane.add(retrieveList);
    displayPane.add(resultPane);

    sapPanel.add(selectPane, BorderLayout.NORTH);
    sapPanel.add(displayPane, BorderLayout.SOUTH);
    sapFrame.getContentPane().add(sapPanel);
    sapFrame.getRootPane().setDefaultButton(retrieveList);
    sapFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    sapFrame.pack();
    sapFrame.setVisible(true);
  }
  public void actionPerformed(ActionEvent event) {
    InterfaceCaller iCaller = new InterfaceCaller();
    Hashtable returnHash = iCaller.getMaterialList(matSearch.getText());
    Hashtable rowHash;
    for (Enumeration e = returnHash.elements(); e.hasMoreElements();) {
      rowHash = (Hashtable)e.nextElement();
      resultText.append("Material: " +
                (String)rowHash.get("MATERIAL") + "\n");
      resultText.append("Material description: " +
                (String)rowHash.get("MATL_DESC") + "\n\n");
    }
  }
  public static void main(String[] args) {
    try {
      UIManager.setLookAndFeel(
      UIManager.getCrossPlatformLookAndFeelClassName());
    }
    catch (Exception e) {}
    MaterialGui matGui = new MaterialGui();
  }
}
```

For the sake of brevity, all of the comments used to describe the code snippets have been removed. Remember, single-line comments are indicated with double forward slashes, `//`, and are totally ignored by the Java compiler and executable.

Compile the new `MaterialListGui` application and execute it from the command line as you have done previously with the `MaterialList` class:

```
java MaterialListGui
```

Figure 5-3 depicts a screenshot of the `MaterialListGui` after the end user has retrieved a list of materials from SAP and selected one for display.

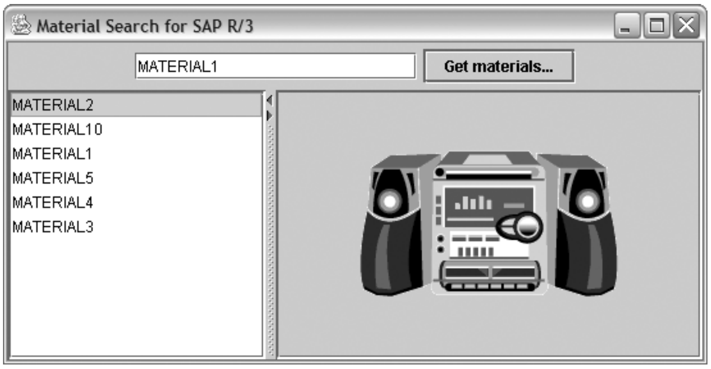


Figure 5-3. `MaterialListGui` *screenshot*

Summary

Having worked through the `MaterialGui` application tutorial, you should now be familiar with the fundamental concepts surrounding the development of a desktop client using a visual interface. Likewise, this chapter demonstrated how to use properties files to configure the system variables of your SAP application; it also helped you broaden your understanding of internationalization in Enterprise Java.

Here are a few points to keep in mind:

- Be very careful when you are hard-coding information within your applications; consider using properties files or other methods unless hard-coding is absolutely required.
- Using Java locales and resource bundles to specify static text strings in your application makes it much simpler to internationalize, should the need to do so ever arise.
- Graphical desktop clients provide an intuitive way for users to work with your SAP functionality; however, if you must deploy to a large number of users, consider the Web application development proposed in the next two chapters.