Programmer's Guide

iPlanet™ Process Manager

Version 6.5

816-6353-10
April 2002

Contents

List of Figures

List of Tables

List of Code Examples

Preface

About This Manual

This manual, *iPlanet Process Manager Programmer's Guide*, is intended for programmers who want to extend iPlanet™ Process Manager functionality, either for developing Process Manager applications or for interacting with clusters and deployed applications.

This manual assumes you are familiar with using Process Manager and with using the Java language. This manual does not attempt to teach Java.

To get started learning Java, see the online Java Tutorial at:

`http://java.sun.com/docs/books/tutorial/`

Developing iPlanet Process Manager Applications

For many applications, Process Builder allows flexibility when building applications to control the flow of processes. You can create process maps that route the flow of control of a process from assignee to assignee. The process can include tasks that are performed manually or automatically, in parallel or sequentially. In many cases, you never need to go outside the Process Builder to build your application.

However, in some cases you may need to tweak applications further. You may want your process to use a data field that is different from any of the built in data fields. You may want to define activities in Java that integrate with external data sources. In these cases, you can create custom data fields and custom activities in Java, and then bring them into the Process Builder to use when building an application.

Relevant chapters are:

- Chapter 1, "Writing Custom Activities," discusses how to write custom activities in Java and bring them into the Process Builder.
- Chapter 2, "Writing Custom Data Fields," discusses how to write custom data fields in Java and bring them into the Process Builder.
- Chapter 3, "Advanced Office Setup Application," discusses a custom data field and custom activity that are provided with the AdvancedOfficeSetup sample application.

Interacting with Clusters and Deployed Applications

The standard way for users to interact with deployed applications is through the Process Express. However, there may be situations when you need to interact programmatically with deployed applications or you want to programmatically perform cluster administration tasks, for example if you want to embed Process Manager functionality inside another application. In this case you can write Java applications that use the Process Manager Engine and Cluster API. In other situations, you might want to write your own front-end to the Process Manager Engine rather than have your users use the Process Express. You can also create web services clients that access Process Manager applications using the iPM Web Services API.

Relevant chapters are:

- Chapter 4, "Cluster Management," discusses the classes for programmatically interacting with Process Manager clusters.
- Chapter 5, "Deployment Manager," discusses the classes for programmatically accessing deployment descriptors.
- Chapter 6, "Working with Applications, Process Instances and Work Items," discusses the classes for programmatically interacting with applications, process instances and work items.
- Chapter 7, "Web Services," discusses how to create Apache SOAP clients and WSDL SOAP clients that can access web services exposed by Process Manager applications.

Conventions Used in This Manual

File and directory paths are given in Windows format (with backslashes separating directory names). For Unix versions, the directory paths are the same, except slashes are used instead of backslashes to separate directories.

This guide uses URLs of the form:

`http://`*`server`*`.`*`domain`*`/`*`path`*`/`*`file`*`.html`

In these URLs, *`server`* is the name of server on which you run your application; *`domain`* is your Internet domain name; *`path`* is the directory structure on the server; and *`file`* is an individual filename. Italic items in URLs are placeholders.

This guide uses the following font conventions:

- The *`monospace`* font is used for sample code and code listings, API and language elements (such as function names), file names, path names, directory names, and HTML tags.
- *Italic* type is used for book titles, emphasis, variables and placeholders, and words used in the literal sense.

Viewing Documentation

For your convenience, iPlanet Process Manager manuals are provided in both PDF and HTML formats. You can access the documentation from the Help menu of each Process Manager component. You can access context-sensitive documentation by clicking a Help button or link in each Process Manager component.

The location of the documentation in your distribution is at:

`iPM_Install_Dir``/builder/manual`

The documentation is also available at the Sun documentation web site:

`http://docs.sun.com/`

For More Information

Table 1 summarizes the tasks involved in using Process Manager and describes where to go for more information about each one.

Table 1 Summary of Process Manager Components

Do What?	Which Process Manager Component?	Comments
Install Process Manager	Installation component	For more information about installing Process Manager, see the *Process Manager Installation Guide*
Build a process application	Process Builder	Process Builder is a graphical user interface for building process applications. For more information, read the *Process Builder's Guide*.
Perform the steps in a process	Process Express	Process Express is a web-based interface used by the people who perform tasks in a process. It includes a customized worklist for each qualified user as well as a web-based form for each task. For more information about Process Express, see the *Process Express User's Guide*.
Administer a process application	Process Administrator and Business Manager	Process Administrator and Business Manager are a set of web-based interfaces for performing administration tasks such as: creating, deleting, and joining Process Manager clusters, and administering process applications deployed to clusters. For more information, read the *Process Administrator's and Business Manager's Guide*.
Build Java custom data fields and activities in Java. Also use the Java API to programmatically interact with Process Manager.	Process Manager Java classes and API which are available in a JAR file.	Java programmers can build custom data fields and custom activities in Java that can be imported into the Process Builder. Programmers can also create Java applications that embed Process Manager functionality or present customized front ends to Process Manager. Additionally, programmers can create SOAP clients that access web services APIs exposed by Process Manager applications. For more information, see For more information, see the *Process Manager Programmer's Guide*.

NOTE Process Manager runs on top of iPlanet Application Server (iAS). For more information about iAS and other iPlanet products, see the Sun documentation web site at http://docs.sun.com/.

Chapter 1

Writing Custom Activities

This chapter describes how to write and use custom activities. The sections in this document are:

Introduction

Process Manager lets you create custom activities as Java classes and bring them into your process definitions.

Custom activities are useful when you want to do more than can easily be done in an automation script, such as when the programming logic or data resides outside of Process Manager. For example, you might build a custom activity to interface with external applications and databases. Custom activities might also run local applications and then interact with mail gateways or FAX servers.

Comparison to Automated Activities

Custom activities are similar to automated activities. In both cases:

- You place them on the process map by dragging and dropping them from the Palette.
- They can have verification and completion scripts.
- They are triggered as soon as the process instance reaches the activity, unless the activity is deferred. A deferred activity is triggered at its specified date and time.

Automated and custom activities have one main difference: an automated activity is carried out by an automation script, whereas a custom activity is carried out by a user-defined Java class.

Overview of Creating a Custom Activity

Creating and using a custom activity involves the following major steps:

1. Write and compile a Java class that implements the `ISimpleWorkPerformer` interface.
2. Define an XML description file for the activity.
3. Package the Java class and the XML description file as a zip or jar file.
4. Bring the custom activity into an application.

Implementing ISimpleWorkPerformer

The first step in creating a custom activity is to write a Java class that implements `ISimpleWorkPerformer`, an interface in the package `com.netscape.pm.model`.

`ISimpleWorkPerformer` defines a custom activity that:

1. gets values, typically data field values, as input
2. performs some task

3. sets data field values as output

NOTE You can find the ISimpleWorkPerformer class in the `pm65classes.jar` file. If you have installed the Process Manager Builder, you can find this jar file in the directory *builder-root*`\support\sdk`. You may also be able to find it on the CD.

This section describes the following topics:

Methods of ISimpleWorkPerformer

`ISimpleWorkPerformer` has three methods:

- The init( ) method is called when the application starts.
- The perform( ) method is called each time the custom activity is executed. This method must be thread-safe.
- The destroy( ) method is called when the application is unloaded or removed.

The init( ) method

```
public void init (Hashtable environment) throws Exception
```

The `init()` method performs initialization tasks that the custom activity requires when the application starts. For example, use `init()` to set up database connections that are shared by all instances of the activity, or use `init()` to define variables that are constant across all instances of the activity.

The `init()` method does *not* execute each time a custom activity is created in a process instance. Instead, this method is called only once—when the application starts.

As its input argument, `init()` takes a hashtable of environment variables. A hashtable is a Hashtable object that contains a series of parameter-value pairs. The parameters in the environment hashtable are defined in the ENVIRONMENT section of an XML description file.

A process designer sets the values of the hashtable parameters while creating the process map.

For example, suppose a `Language` parameter is defined in the environment hashtable of a custom activity. In Process Builder, the `Language` parameter would appear as a property for the custom activity (you would open the Inspector window and view the Properties tab).

In your Java class, define the `init()` method to perform the desired initialization tasks. Then, to obtain the value of a parameter in the environment hashtable, call the `get()` method on the environment hashtable. The `get()` method returns either the value of the parameter, or `null` if the parameter doesn't exist.

The perform( ) method

```
public void perform (Hashtable in, Hashtable out) throws Exception
```

The `perform()` method executes whatever tasks must be done for the activity. This method takes two Hashtable arguments. The input hashtable contains values taken from data fields, and the output hashtable contains values to put into data fields.

The parameters in the input and output hashtables are defined in the INPUT and OUTPUT sections, respectively, of an XML description file.

The Input Hashtable

To obtain the value of a parameter in the input hashtable, call the `get()` method on the input hashtable. The `get()` method returns either the value of the parameter, or `null` if the parameter doesn't exist. Note that the `get()` method returns a Java object, so you must cast this object to the object class type that your custom activity is expecting. For example:

```
String sizeOrder = (String) input.get("order");
```

The Output Hashtable

To set data field values, the `perform()` method must put values into the output hashtable by calling `put()` on the output hashtable. When the `perform()` method finishes executing, you then assign the values to the corresponding data fields.

The destroy( ) method

```
public void destroy()
```

The `destroy()` method is called when the application that uses the custom activity is unloaded or removed. Typically, you use the `destroy()` method to clean up resources that were used by the `init()` method.

Sample Java Class

The following code samples are from `HelloWorldPerformer.java`, the class that implements the HelloWorld custom activity. HelloWorld is included in Process Manager as a sample custom activity, so you can view the source code directly.

HelloWorld constructs a welcome message in either French or English. The message value is derived from two things: the value of the `customerName` data field in the process instance, and the Language property of the `HelloWorld` activity instance. The HelloWorld activity puts the welcome message in the `greeting` data field.

Creating HelloWorldPerformer.java

Using your favorite Java editor and compiler, create and compile a Java class that implements the `ISimpleWorkPerformer` interface. When you use Process Builder to add a custom activity, Process Manager automatically places the custom activity's class file in the server's class path when the application is deployed.

NOTE Don't define any constructors in classes implementing ISimpleWorkPerformer, because Process Manager does not use them. A Java exception will be thrown. Defining a class without any constructors is the same as defining one with just a default constructor.

Here are the steps for creating `HelloWorldPerformer.java`:

1. Define a package for your class:

   ```
   package com.netscape.pm.sample;
   ```

2. Import the required standard Java packages:

   ```
   import java.lang.*;
   import java.util.*;
   ```

3. Define the class HelloWorldPerformer to implement `com.netscape.pm.model.ISimpleWorkPerformer`, as follows:

```
public class HelloWorldPerformer
    implements com.netscape.pm.model.ISimpleWorkPerformer
{
```

4. Define two variables to hold the English and French parts of the greeting. Define another variable to hold the complete greeting when it has been derived (such as "Bonjour Nikki.")

```
// Greeting Messages
public static final String GREETING_FRENCH  = "Bonjour";
public static final String GREETING_ENGLISH = "Hello";

// Holds the greeting message once the language is specified
String mGreeting;
```

5. Define the `init()` method to get the value of the Language environment variable and to set the language-specific part of the greeting. In addition, throw an exception if the language is not provided, or if the language is neither English nor French. For example:

```
/**
* The HelloWorld custom activity knows to generate both French
* and English greetings. The Language argument defines which
* language should be used.
*/

public void init( Hashtable env ) throws Exception
{
  String lang = (String) env.get( "language" );

  if( lang == null )
  {
      throw new Exception( "-- language not defined.") ;
  }

  else if ( lang.equalsIgnoreCase("French") )
  {
      mGreeting = GREETING_FRENCH;
  }

  else if ( lang.equalsIgnoreCase("English") )
  {
      mGreeting = GREETING_ENGLISH;
  }

  else
  {
      throw new Exception( "-- Unknown language:"+ lang +
      ". We currently support English or French--" ) ;
  }
}
```

Later, you will set the exact value of the Language environment. You'll do this in Process Builder, when you set up the custom activity in a process definition.

6. Define the `perform()` method to construct a welcome message consisting of the language-specific part of the greeting and the user's name, for example "Hello Billy." The value of the `userName` parameter is derived later—from a data field in a process instance that uses the custom activity.

   Use the `get()` method on the input parameter to get the value of an input parameter.

```
/**
* Reads the userName element of the input hashtable,
* generates greetings, and sets the Greeting element of out.
*/

public void perform( Hashtable input, Hashtable output )
  throws Exception
{
  // Read the userName attribute from the input hashtable
  String userName = (String) input.get( "userName" );

  if( userName == null )
  {
      throw new Exception("userName is not initialized!");
  }

  // Generate greetings
  String msg = mGreeting + " " + userName;

  /* Use the put() method on the output parameter to set
  * the value of an output parameter.
  */
  // Put the greeting into the welcomeMsg parameter of
  // the output hashtable.
  output.put( "welcomeMessage" , msg );
}
```

7. Finally, define the `destroy()` method, which is invoked when the application is unloaded from the application server. In this case, the method does nothing because no resource cleanup is needed.

```
public void destroy( )
{
}
// End of class
}
```

8. Compile `HelloWorldPerformer.java` to get a class file, `HelloWorldPerformer.class`.

Writing the XML Description File

After you write and compile the Java class that implements ISimpleWorkPerformer, the next step is to define an XML description file for the class. This XML file specifies the environment, input, and output parameters that the class uses. In addition, the XML file specifies some optional design parameters. Design parameters control the custom activity's appearance in Process Builder.

This section describes the following topics:

File Format

The XML description file starts with a tag indicating the XML version, such as:

```
<?XML version = "1.0" ?>
```

The body of the description is contained between an opening <WORKPERFORMER> tag and a closing </WORKPERFORMER> tag. Within the WORKPERFORMER section you define four sections, as summarized in the following table.

XML Section	What this section describes
ENVIRONMENT	Environment hashtable used by `init()` method.
INPUT	Input hashtable used by `perform()` method.
OUTPUT	Output hashtable used by `perform()` method.
DESIGN	Appearance of custom activity icons in Process Builder.

Here is the structural overview of an XML description file:

```
<?XML version = "1.0" ?>
<WORKPERFORMER >
<ENVIRONMENT>
  <PARAMETER> ... </PARAMETER> ...
</ENVIRONMENT>
<INPUT>
  <PARAMETER> ... </PARAMETER> ...
</INPUT>
<OUTPUT>
  <PARAMETER> ... </PARAMETER> ...
</OUTPUT>
<DESIGN>
  <PARAMETER> ... </PARAMETER> ...
</DESIGN>
</WORKPERFORMER>
```

WORKPERFORMER Tag

The <WORKPERFORMER> tag has four attributes: TYPE, NAME, CLASS_ID, and VERSION.

- TYPE is the full package name for the Java class for this type of activity. For a simple custom activity, TYPE is always this:

  com.netscape.pm.model.ISimpleWorkPerformer

- NAME is the name of the custom activity (which is the same as the name of the XML description file and the jar file that contains the custom activity). This name is not currently used anywhere.
- CLASS_ID is the full package name for the Java class that implements the custom activity.
- VERSION is the version of the custom activity. VERSION is currently unused, but you could use it to keep version information about the description file.

Here is a sample <WORKPERFORMER> tag:

```
<WORKPERFORMER
  TYPE="com.netscape.pm.model.ISimpleWorkPerformer"
  NAME="HelloWorld"
  CLASS_ID="com.netscape.pm.sample.HelloWorldPerformer"
  VERSION="1.1">
```

ENVIRONMENT Section

The <ENVIRONMENT> tag defines environment parameters that are constant within all instances of the custom activity. For example, suppose that in an application named HelloWorld, you set the value of the Language environment parameter to French. Then, the value is always French in every process instance of that application.

The ENVIRONMENT section contains embedded <PARAMETER> tags. Each <PARAMETER> tag describes a parameter in the environment hashtable—the argument used by the init() method. The <ENVIRONMENT> tag has a corresponding closing </ENVIRONMENT> tag , and each <PARAMETER> tag has a closing </PARAMETER> tag.

When you add the custom activity to the process map in Process Builder, each parameter in the <ENVIRONMENT> tag appears as a field in the Inspector Window.

Here's a sample ENVIRONMENT section:

```
<ENVIRONMENT>
  <PARAMETER NAME="Language">"French"</PARAMETER>
</ENVIRONMENT>
```

CAUTION Parameter values (such as "French" in the example above) are actually JavaScript expressions, so you can supply the value as a string, integer, or function. However, be sure to quote any string expression. Note that `French` (without quotes) and `"French"` (with quotes) mean different things.

For details on the syntax of the <PARAMETER> tag, see the section "PARAMETER Tag."

INPUT Section

The <INPUT> tag contains embedded <PARAMETER> tags. Each <PARAMETER> tag specifies a JavaScript expression that returns a value for the input hashtable to be used as the argument to `perform()`. The<INPUT> tag has a corresponding closing </INPUT> tag , and each <PARAMETER> tag has a closing </PARAMETER> tag.

The <PARAMETER> can specify any JavaScript expression as the parameter.

To use the value of a data field in the process instance as an input parameter, embed a call to `getData()` in the <PARAMETER> tag. For example, the following code sets the value of the `userName` parameter in the input hashtable to the value of the `customerName` data field in the process instance.

```
<INPUT>
  <PARAMETER
      NAME="userName"
      DISPLAYNAME="User Name"
      TYPE="java.lang.String"
      DESCRIPTION="Last Name">
  getData("customerName")
  </PARAMETER>
</INPUT>
```

For details on the syntax of the <PARAMETER> tag, see the section "PARAMETER Tag."

The corresponding code in your Java class file uses the `perform()` method to get the value of the `userName` parameter. Within the `perform()` method, you call the `get()` method. Here is a code fragment:

```
public void perform( Hashtable input, Hashtable output )
  throws Exception
{
  // Read the userName attribute from the input hashtable
  String userName = (String) input.get( "userName" );

  if( userName == null )
  {
      throw new Exception("userName is not initialized!");
  }

  // Generate greetings
  String msg = mGreeting + " " + userName;
```

OUTPUT Section

The <OUTPUT> tag contains embedded <PARAMETER> tags. Each <PARAMETER> tag specifies a JavaScript statement that defines what to do with parameter in the output hashtable, the output argument of the `perform()` method. The <OUTPUT> tag has a corresponding closing </OUTPUT> tag, and each <PARAMETER> tag has a closing </PARAMETER> tag.

Use the `mapTo()` JavaScript function to specify that the value of a parameter of the output hashtable is to be automatically installed in a data field in the process instance. For example, the following code specifies that when the `perform()` method has finished executing, the value of the `welcomeMsg` parameter in the output hashtable is automatically installed in the `greeting` data field in the process instance.

```
<OUTPUT>
  <PARAMETER
      NAME="welcomeMsg"
      DISPLAYNAME="Welcome Message"
      TYPE="java.lang.String"
      DESCRIPTION="Greeting for the user">
  mapTo("greeting")
  </PARAMETER>
</OUTPUT>
```

For details on the syntax of the `<PARAMETER>` tag, see the section "PARAMETER Tag."

The corresponding code in your Java class file uses the `perform()` method to put a value in the `welcomeMsg` parameter of the output hashtable. Within the `perform()` method, call the `put()` method:

```
output.put( "welcomeMessage" , msg );
```

PARAMETER Tag

The `<PARAMETER>` tag specifies a JavaScript statement or expression that defines a parameter for the input hash table when used in the INPUT section.

The `<PARAMETER>` tag specifies a JavaScript statement that defines what to do with a parameter in the output hash table when used in the OUTPUT section.

The `<PARAMETER>` tag has the attributes as summarized in the following table. When you define parameters within the DESIGN section of the XML description file, only the NAME and DESCRIPTION attributes apply. However, within the ENVIRONMENT, INPUT, or OUTPUT sections, all of the attributes apply.

Attribute	Meaning
NAME	Name of the parameter.
DESCRIPTION	The text for the tool tip (also called bubble help) that appears when you place the mouse over the item in Process Builder.
TYPE	The Java object class of the parameter. This attribute is optional. The value can be given as a complete class name, such as `java.lang.String` or `com.netscape.pm.ShoppingCart`.
VALUESET	A comma-delimited list of possible values for this parameter. These values appear as a pop up menu in the Inspector Window. This attribute is optional.
EDITOR	The type of editor window to use. For example, use this attribute to set a Browse button, text area, drop down list, dialog box. This attribute is optional.
EDITABLE	A boolean that determines whether the parameter value can be edited in the Inspector Window. The default is true. This attribute is optional.

DESIGN Section

The `<DESIGN>` tag contains embedded `<PARAMETER>` tags. The `<DESIGN>` tag has a corresponding closing `</DESIGN>` tag, and each `<PARAMETER>` tag has a closing `</PARAMETER>` tag.

Use the DESIGN section to define the custom activity's user interface within Process Builder. In the DESIGN section, the `<PARAMETER>` tag accepts two attributes: NAME and DESCRIPTION.

By setting the NAME attribute, you define a particular aspect of the custom activity's user interface. The following table summarizes the available values for the NAME attribute:

NAME Attribute	Meaning
Icon	The image file to use for the icon in the custom palette.
Label	A text label that appears under the icon.
BubbleHelp	The text for the tool tip that appears when the mouse pointer is over the icon.
HelpUrl	The URL for the online help for this custom activity, accessible from a right-click.
MapIcon	The image file to use for the icon in the process map. In typical usage, this is the same as Icon.
SelectedMapIcon	The image file to use for the icon in the process map, when the activity is selected.
TreeViewIcon	The file to use for a small image that represents the activity in the Application Tree View.

Sample XML Description File

The following code defines a file called `HelloWorld.xml` . Things to note are:

- This file specifies `userName` as a parameter in the input hash table. However, the value of this parameter is obtained from the `customerName` data field in the process instance.
- Similarly, the file specifies `welcomeMsg` as a parameter in the output hashtable, and maps its value back into the `greeting` data field in the process instance.

Here is the entire code for the `HelloWorld.xml` description file:

```
<?XML version = "1.0" ?>
<WORKPERFORMER
  TYPE="com.netscape.pm.model.ISimpleWorkPerformer"
  NAME="HelloWorld"
  CLASS_ID="com.netscape.pm.sample.HelloWorldPerformer"
  VERSION="1.1">
<ENVIRONMENT>
  <PARAMETER
    NAME="Language"
    VALUESET="'English','French'"
    TYPE="java.lang.String">
  'English'
  </PARAMETER>
</ENVIRONMENT>

<INPUT>
  <PARAMETER
    NAME="userName"
    DISPLAYNAME="User Name"
    TYPE="java.lang.String"
    DESCRIPTION="Last Name">
  getData("customerName")
  </PARAMETER>
</INPUT>
<OUTPUT>
  <PARAMETER
    NAME="welcomeMsg"
    DISPLAYNAME="Welcome Message"
    TYPE="java.lang.String"
    DESCRIPTION="Greeting for the user">
  mapTo("greeting")
  </PARAMETER>
</OUTPUT>
```

```
<DESIGN>
  <PARAMETER
    NAME="Icon"
    DESCRIPTION="A 32x32 icon that is placed on the palette">
    drap_uk2.gif
  </PARAMETER>
  <PARAMETER
    NAME="Label"
    DESCRIPTION="The DISPLAYNAME for this palette element.">
  Hello World
  </PARAMETER>

  <PARAMETER
    NAME="BubbleHelp"
    DESCRIPTION="Bubble help for the palette element">
  HelloWorld - A simple work performer Custom Activity.
  </PARAMETER>

  <PARAMETER
    NAME="HelpURL"
    DESCRIPTION="URL explaing this palette element">
  http://people.netscape.com/michal/
  </PARAMETER>

  <PARAMETER
    NAME="MapIcon"
    DESCRIPTION="Icon for the process map (48x48)">
  drap_uk2.gif
  </PARAMETER>

  <PARAMETER
    NAME="SelectedMapIcon"
    DESCRIPTION="Icon for the process map (48x48)">
  drap_fr2.gif
  </PARAMETER>

  <PARAMETER
    NAME="TreeViewIcon"
    DESCRIPTION="Icon for the tree view (48x48)">
  mailer_tree_view.gif
  </PARAMETER>
</DESIGN>
</WORKPERFORMER>
```

Packaging a Custom Activity

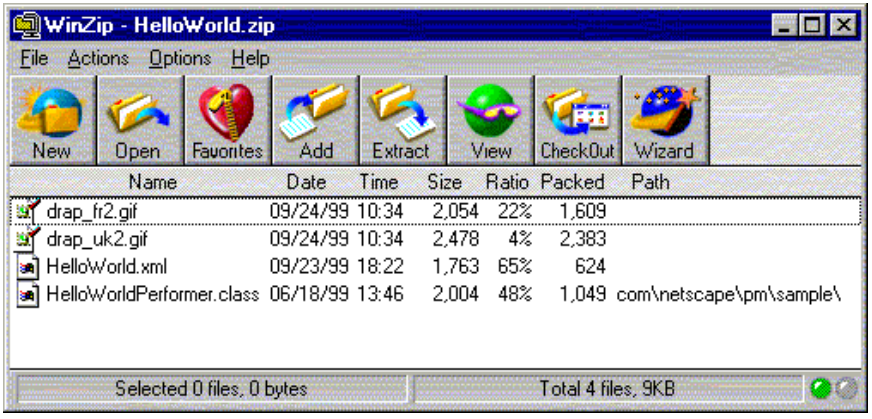
After you create the Java class file and the XML description file, the next step is to package the custom activity. A custom activity consists of the following files:

- One or more Java classes. At least one of these classes must implement `ISimpleWorkPerformer`.
- An XML description file.
- Optional image files to use as icons in Process Builder.

Create a zip or jar archive that contains these files. The archive must have the same root name as the XML file. For example, if the XML file is `HelloWorld.xml`, then name the zip file `HelloWorld.zip`.

As you create the archive, check that the directory structure reflects the package structure of the class. For example, the `HelloWorldPerformer` class is in the package `com.netscape.pm.sample`. Therefore, the class file must be in the directory `com/netscape/pm/sample`, as shown in Figure 1-1. The `HelloWorld.xml` file must be at the top level.

Figure 1-1 Directory structure for the HelloWorld activity



Note the two image files, `drap_fr2.gif` and `drap_uk2.gif`. These images will be used by Process Builder in the process map. The images, shown in Figure 1-2, will correspond to the selected state of the Language property, either French or English.

Figure 1-2 Image files in the HelloWorld activity


Adding a Custom Activity to the Process Map

There are two ways to add a custom activity to the process map:

- In one case you create a custom palette. This approach is useful if you intend to use a custom activity often, either within a single application or across several applications.
- In the other case, you don't create a custom palette, and you simply use the Custom Activity icon provided with Process Builder. This approach might be better if you rarely use custom activities, and you don't want to create a custom palette for them.

Adding a Custom Activity from a Custom Palette

To use a custom activity from a custom palette, do the following:

1. In the palette, right-click the area below the title bar, and choose "Add custom palette," as shown in Figure 1-3.

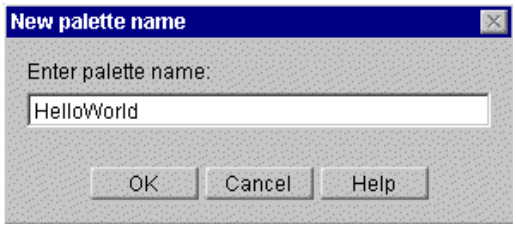
   This adds a new tab to the palette.

Figure 1-3 Adding a custom palette

2. Type the label for the new tab in the "New palette name" dialog box as shown in Figure 1-4.

   In this example, the palette name "HelloWorld" has been entered.

Figure 1-4 New Palette Name Dialog



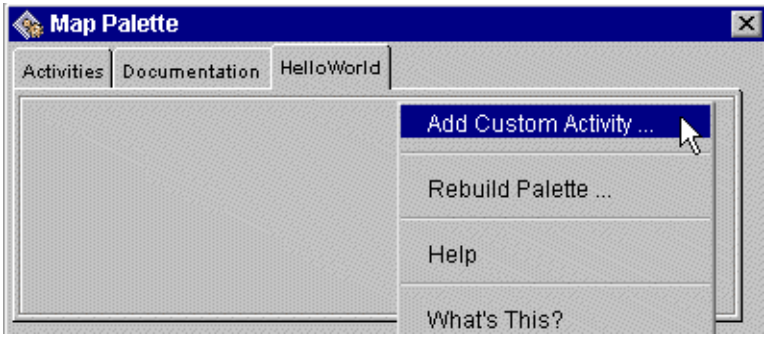
   A new tab is added to the palette.

3. Select your new custom tab to make it active.

   Note that the area contains no icons.

4. Right-click in the empty area under the tabs, and select "Add Custom Activity…", as shown in Figure 1-5.

Figure 1-5 Add a custom activity to the palette

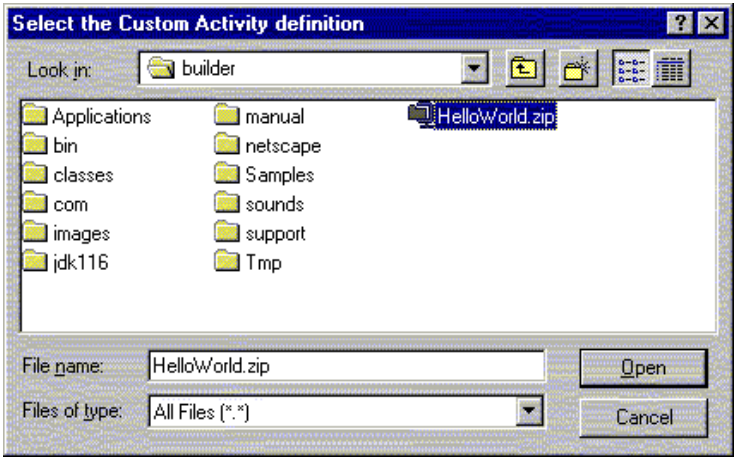


   A file selection window appears.

5. Using the file selection window, locate the archive file that represents the custom activity, and select the file.

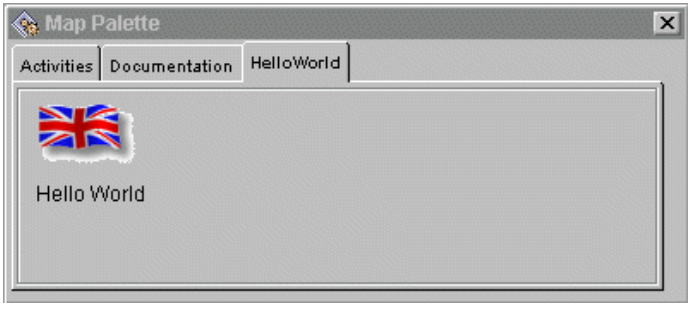
   For example, Figure 1-6 shows the selection of `HelloWorld.zip`.

Figure 1-6 Select the file that represents a custom activity



The custom activity is added to your new palette. For example, as shown in Figure 1-7, the HelloWorld activity appears on the palette like this:

Figure 1-7 A custom activity icon appearing on the HelloWorld custom palette



Note that the custom activity's appearance in Process Builder is controlled by the DESIGN section of the XML file. In the HelloWorld tab pictured above, you see the effects of setting the Icon, Label, and BubbleHelp parameters in the DESIGN section.

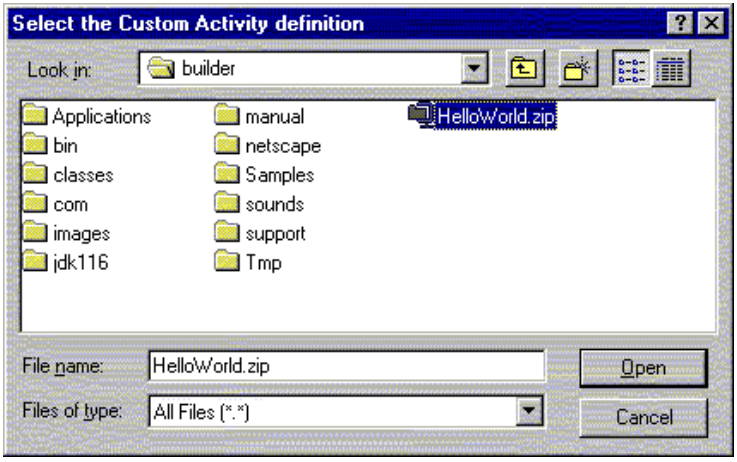
6. To add the activity to your application, drag the icon from the custom palette to the process map.

Adding a Custom Activity without Using a Custom Palette

If you don't have a custom palette icon or don't want to create one, you can add a custom activity as follows:

1. In the palette, drag the Custom Activity icon  to the process map.
2. Select the custom activity and open the Inspector window.
3. On the Properties tab of the Inspector, locate the property named Custom Activity.
4. Click the Browse button to bring up a file selection window, and locate the zip or jar file that represents the custom activity. An example is shown in Figure 1-8.

Figure 1-8 Setting the Custom Activity property

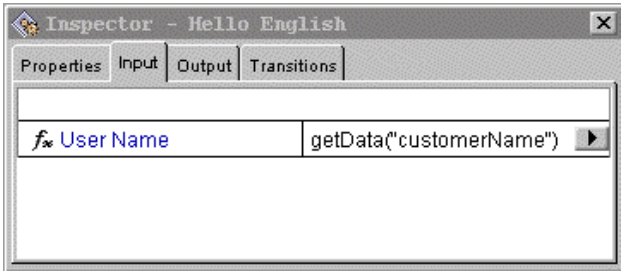


5. Click Open to associate the selected file with the Custom Activity icon. The Custom Activity icon now has the characteristics defined by the file.

Working with a Custom Activity

After you place a custom activity on the process map, you can view or set its properties in the Inspector window. For example, Figure 1-9 shows the Inspector window's Input tab for HelloWorld.

Figure 1-9 Input properties for a custom activity



The Input tab shows the parameter names in the input hashtable, and shows how the value for each parameter is derived. In this case, the value for the input parameter `userName` is derived by getting the value of the `customerName` datafield.

The INPUT section of the XML description file determines the appearance of the Input tab in the Inspector window. For example, note that the `userName` parameter displays as "User Name," which was specified through the DISPLAYNAME attribute in the XML file.

Similarly, the Output tab shows the parameter names in the output hashtable, and shows how the value for each parameter is mapped back into the process instance. In this case, the value for the output parameter `welcomeMsg` is put in the `greeting` data field.

As you design the process, be sure to add the data fields that are used by the custom activity. For example, the HelloWorld activity uses two Textfields: `greeting` and `customerName`.

Implementation Tips

This section describes some design tips you should consider as you create and implement a custom activity.

Avoid Instance Data

A custom activity, like a custom data field, is a stateless entity. In effect, there is only one copy of each occurrence of a custom activity per application. All the process instances in an application effectively share the same custom activity instance for each occurrence of the custom activity class in the application. Because of this, it's recommended that you avoid using instance data in a class that implements a custom activity, particularly if the `perform()` method is likely to change this data. If you can't avoid using instance data, be sure to synchronize the data. With unsynchronized data, a variable set during one request might not exist for the next request.

For example, consider an application that employees use to request vacation days. Let's suppose this application has a custom activity that updates the corporate database with the new vacation balance.

The following code, which uses an instance variable called `vacationBalance`, shows how NOT to implement the custom activity:

```
// This is the WRONG way to implement a custom activity!!

public class RequestVacationPerformer
  implements com.netscape.pm.model.ISimpleWorkPerformer
{

int vacationBalance;

public void init (Hashtable environment) throws Exception
{
  ...
  // Get the employee's vacation balance from the database
  // Store it temporarily as instance data
  vacationBalance = getVacBalance(employeeID);
  ...
}

public void perform( Hashtable input, Hashtable output )
  throws Exception
{
  // Read the employee ID attribute from the input hashtable
  String employeeUD = (String) input.get( "employeeID" );

  // Get the num of requested vac days from the input hash table
  String vacRequested = (String) input.get("vacDaysRequested");

  // Update the vacationBalance instance variable
  vacationBalance = vacationBalance - vacationRequested;

  // Change the vacation balance in the database
  updateVacationInfo(employeeID, vacationBalance)
  ....
```

Fred is the first person to request a vacation using this Process Manager application and he wants to go river rafting for 2 days. Fred's `init()` method gets his vacation balance, which is 3 days, (Fred has already been scuba diving in Hawaii this year) and stores it in the `vacationBalance` instance variable. Fred's `perform()` method calculates his updated vacation balance, which is 1 day, and stores it in the `vacationBalance` instance variable.

Now Bob comes online and requests a vacation of 8 days. Bob has been saving his vacation days for a long time for his dream trip to climb Everest. However, since the application has already been initialized, `init()` does not run again. Bob's `perform()` method ends up accessing the `vacationBalance` that was set by Fred's `perform()` method, thus Bob ends up having a vacation balance of only 1 day, which is hardly enough time to fly to Nepal, let alone climb Everest and fly home again.

It is OK to use instance variables for data that is constant across all occurrences of a custom activity within an application. For example, the custom activity in the vacation request application might use a global variable that represents the number of company holidays per year. This number does not change from employee to employee, so it is OK to store this as an instance variable.

Another example of a situation where it is OK to use instance variables is if the custom activity needs the iAS engine context to call out to a connector such as an SAP connector. In this case, you could set the context inside `init()` and then re-use it inside `perform()`. The key thing to remember is that objects such as the context are considered to be immutable and hence will only be used, not changed, inside `perform()`.

An application can contain multiple occurrences of a custom activity class. For example, an application might have custom activities called `CheckVacationBalance` and `CheckVacationAccrual`, which are both instances of the `CheckVacationInfo` custom activity class. When the application is running, these two activities operate completely independently. If the activities use instance data, that data would not be shared between them. For example, if the activities use an instance variable called `DBTableName`, the `CheckVacationBalance` instance could set it to `VacBalTable` while the `CheckVacationAccrual` could set it to `VacAccTable`, and there would be no confusion between the two.

Use Consistent Data Types

Watch for consistent data typing. Make sure that the data types you specify in the XML file are consistent with the corresponding values you pass to the input and output hashtables. Although Process Manager performs some basic data matching for you, inconsistent data is likely to generate an error.

Avoid Non-default Constructors

In classes that implement `ISimpleWorkPerformer`, avoid defining non-default constructors (meaning constructors with non-zero arguments). Otherwise, you may encounter problems during dynamic loading. The problem may arise because Process Manager dynamically loads the class that implements your custom activity. In other words, Process Manager has no prior awareness of non-default constructors and therefore cannot call them.

Use One Implementation of a Java Class Per Server

When an application is deployed, the Java classes it uses are deployed to the appropriate folder in the class path on the engine. This class path is shared by all applications running on the engine. Every application that uses a particular Java class *uses the same implementation of that class*. For example, suppose application A and application B both use a Java class `SharedClass1`. When application A is deployed, its version of `SharedClass1` is deployed to the class path. When application B is deployed, its version of `SharedClass1` is deployed to the class path, overwriting the implementation deployed previously by application A.

Thus if multiple applications running on a Process Manager engine use the same custom activity, they should all use exactly the same implementation of the custom activity, since each time the custom activity is deployed to the engine, it overwrites the previous implementation.

If you want multiple applications to use a custom activity that is basically the same but differs slightly from application to application, make sure that the name of the activity Java class is different in each application.

When to Use a Custom Activity

Custom activities are useful when you want to integrate an existing legacy process into a Process Manager process through a well-defined interface. For example, use a custom activity in a Process Manager process that exchanges data with external resources such as a CORBA server, a CICS system, or the business logic in an EJB component.

By contrast, custom activities are not a good solution if you must represent a complex data structure from an external source. For example, to represent result sets or other data types from Oracle databases or SAP R/3 systems, you are better off using a custom field. Reserve custom activities for situations where data can be easily parsed and stored (either directly in a data field or in the content store).

Example Custom Activity

The AdvancedOfficeSetup sample application that ships with the Process Builder includes an example of a custom activity.

The AdvancedOfficeSetup application has a custom activity that automatically schedules a new employee to attend a company orientation training.

The day of the training depends on which department the employee is joining and what day they start work at the company. For more details, see Chapter 3, "Advanced Office Setup Application."

Chapter 2

Writing Custom Data Fields

This chapter describes how to write custom fields for use in iPlanet Process Manager.

This chapter includes the following sections:

Introduction

A data field contains information relevant to a process instance, such as the maximum value of the budget or the name of a document. Process Builder offers a set of predefined data field classes, such as `Radio Buttons` and `Textfield`. The predefined data fields store a single value per data field.

In situations where you need behavior that is not provided by any of the predefined data field classes, you can define your own custom data field. Such situations include the need for:

- supporting data types that are more complex than the data types available with built-in fields.
- representing multi-dimensional values, or other high-level data objects, in a process. For example, custom fields can represent a "shopping cart," an account, or a service order.
- accessing data objects that are stored in resources external to Process Manager, such as PeopleSoft or CICS.
- displaying the data field differently in an entrypoint and a workitem

iPlanet Process Manager allows you to define your own classes of data fields. Custom data fields are sometimes known as entity fields.

Steps for Creating a Custom Field

The main steps for creating a custom field are as follows:

- Create a JavaScript bean (JSB) file to specify the field properties that will be visible in Process Builder. For details, see "Defining Field Properties in a JSB File" on page 53.
- Write a Java class to determine the presentation and data management capabilities of the custom field. At a minimum, you must implement two interfaces, IDataElement and IPresentationElement. For details, see "Writing the Java Classes" on page 58.
- Optionally, create images to depict the data field in the Process Builder interface. For details, see "Specifying Images for Use in Process Builder" on page 77.
- Package the JSB and Java classes into a zip or jar archive. For details, see "Packaging a Custom Field" on page 77.
- In Process Builder, insert a data field and add the archive file as a new class. For details, see "Adding a Custom Field to an Application" on page 79.

Defining Field Properties in a JSB File

You need to write a JSB file that defines which of the custom field's properties can be set at design time in Process Builder. In Process Builder, these properties are visible through the field's Inspector window. For each property shown in the Inspector window, a corresponding property must be defined in the JSB file.

To create a JSB file for a new custom field class, you can copy an existing JSB file and modify it to suit your needs. For example, you can copy the JSB files for Process Builder's predefined data fields, or you can copy a template JSB file. These files are located in the following path of your Process Builder installation:

builder-root`\com\netscape\workflow\fields`

CAUTION Do not modify the original JSB files for predefined data fields. If you do, the data fields may no longer work

The JSB file and the custom field class must have the same name. For example, a custom field class named `ShoppingCartField.class` must have a JSB file named `ShoppingCartField.jsb`.

A JSB file has the following general structure:

```
<JSB>
   <JSB_DESCRIPTOR ...>
   <JSB_PROPERTY ...>
   <JSB_PROPERTY ...>
   ...
</JSB>
```

The file is surrounded by an opening `<JSB>` tag and a closing `</JSB>` tag. The other two tags are described in the following sections:

- JSB_DESCRIPTOR Tag
- JSB_PROPERTY Tag

JSB_DESCRIPTOR Tag

The <JSB_DESCRIPTOR> tag specifies the name, display name, and a short description of the data field class.

For example, ShoppingCartField.jsb uses the following <JSB_DESCRIPTOR> tag:

```
<JSB_DESCRIPTOR
   NAME="com.netscape.pm.sample.ShoppingCartField"
   DISPLAYNAME="Shopping Cart Field"
   SHORTDESCRIPTION="Shopping Cart Field">
```

The NAME attribute is the full path name for the data field class, using a dot (.) as the directory separator.

The DISPLAYNAME attribute is the name that Process Builder uses for the field, such as the field's name in the Data Dictionary.

The SHORTDESCRIPTION attribute is a brief description of the field.

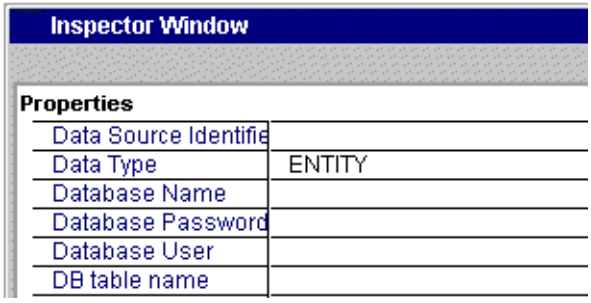
JSB_PROPERTY Tag

The JSB file contains a series of <JSB_PROPERTY> tags, one for each property that appears in the Inspector window. The following code shows some example <JSB_PROPERTY> tags. In this case, the Inspector window shows properties for dsidentifier, dbuser, dbpassword, dbuser and dbtablename as shown in Figure 2-1.

```
<JSB_PROPERTY NAME="dsidentifier"
  TYPE="string"
  DISPLAYNAME="Data Source Identifier"
  SHORTDESCRIPTION="DS Identifier">

<JSB_PROPERTY NAME="dbname"
  TYPE="string"
  DISPLAYNAME="Database Name"
  SHORTDESCRIPTION="DB Type">

<JSB_PROPERTY NAME="dbpassword"
  TYPE="string"
  DISPLAYNAME="Database Password"
  SHORTDESCRIPTION="DB Password">
```

```
<JSB_PROPERTY NAME="dbuser"
  TYPE="string"
  DISPLAYNAME="Database User"
  SHORTDESCRIPTION="DB User">

<JSB_PROPERTY NAME="dbtablename"
  TYPE="string"
  DISPLAYNAME="DB table name"
  SHORTDESCRIPTION="DB User">
```

Figure 2-1 Inspector Window shows properties defined in the JSB file



The `JSB_PROPERTY` attributes and required property names are described in the next two sections.

JSB_PROPERTY Attributes

The attributes for the `JSB_PROPERTY` tag are shown in Table 2-1:

Table 2-1 Attributes for the JSB_PROPERTY Tag

Attribute Name	Purpose
`NAME`	The name of the property.
`DISPLAYNAME`	The display name for this property, as it appears in the Inspector window.
`SHORTDESCRIPTION`	A short description of the property.
`DEFAULTVALUE`	The default value of the property. This attribute is optional.

Table 2-1 Attributes for the JSB_PROPERTY Tag *(Continued)*

Attribute Name	Purpose
VALUESET	A comma-delimited list of the possible values for this property. These values appear as a pop up menu on the property in the Inspector window. This attribute is optional.
TYPE	The type of the data field column in the application table in the database.
ISDESIGNTIMEREADONLY	When specified, this attribute indicates that the property cannot be changed in Process Builder. This attribute is optional. By default, a property value can be changed any time. This attribute does not have an attribute=value specification. You simply give the value, for example: `<JSB_PROPERTY NAME="myname" ISDESIGNTIMEREADONLY>`
ISEXPERT	When specified, this attribute indicates that the property can be changed in Process Builder while the application is in design mode. This attribute is optional. By default, a property value can be changed any time. This attribute does not have an attribute=value specification. You simply give the value, for example: `<JSB_PROPERTY NAME="myname" ISEXPERT>`

Required Data Field Properties

Each data field must have the properties listed in Table 2-2:

Table 2-2 Standard Data Field Properties

Property Name	Default Display Name	Purpose
cn	Name of this field	The common name of the data field instance. (Note this is not the name of the data field class.) The name is set when you create the data field in Process Builder.
description	Short Description	A description of the data field.

Table 2-2 Standard Data Field Properties *(Continued)*

Property Name	Default Display Name	Purpose
`prettyname`	Display Name	The field's display name which is the name that Process Builder uses for the field.
`help`	Help Message	A help message for the field.
`fieldclassid`	Field Class ID	This is the package name of the data field class. This is used to ensure that each data field type is unique. This value uses the same convention as the Java naming convention for packages. For example, if `ShoppingCartField` is stored in `\com\netscape\pm\sample`, then its `fieldclassid` is: `com.netscape.pm.sample.ShoppingCartField`
`fieldtype`	Data Type	The datatype that the field uses when it is stored in the Process Manager database. The value must be ENTITY.

In addition to these required properties, each data field can have properties that are specific to itself. For example, a Textfield has properties for size and length; a radio button data field has a property for options; and so on.

When you define the properties for a custom field, consider the purpose of the field. For example, if the custom field must access an external database, you may want to define connection properties. These properties might include the database type (ORACLE, SYBASE), a username and password, or a connection string.

Not all properties you define in a JSB file will necessarily be used. It depends on how your Java class interprets these properties. For example, the JSB file could contain a `color` property that is totally ignored in the Java class. In this case, no matter what color the designer specifies for the field, it has no effect.

Writing the Java Classes

To write your Java classes, you must know something about the data these classes will work with. Consider the following questions:

- What data types do you want the custom field to accept? For example, in what format will the data be? This could well depend on where the data is coming from.
- What data sources will the custom field be required to access? For example, will the custom field access a PeopleSoft application? an SAP R/3 application? a relational database?

Custom data fields are stateless, that is, you cannot use them to store information about a process instance from one workitem to another. Think of your custom data field as being a data manager. When a process instance arrives at a work item, the data field gets its data from an external data store. The data can be any Java object. When the process instance leaves the work item, the data field saves its data to an external store. The important idea is that the custom fields specify only the logic to manage the data, not the data itself.

Define a Subclass of BasicCustomField

To implement a new data field class, create a Java subclass of BasicCustomField. This class provides methods that enable iPlanet Process Manager to treat your custom field just like any other data field.

The BasicCustomField class implements the IPresentationElement and IDataElement interfaces. The IPresentationElement interface specifies the presentation methods for the data field, which are `display()` and `update()`. The IDataElement interface specifies the methods for managing the loading and storing of data, which are `create()`, `load()`, `store()` and `archive()`. Your new subclass needs to provide definitions for each of the methods in these interfaces.

Note: You can find all the necessary classes in the `pm65classes.jar` file. If you have installed the Process Manager Builder, you can find this jar file in the directory *builder-root*`\support\sdk`. You may also be able to find it on the CD.

Before looking at the methods in detail, here is a discussion of how and when the methods are called.

When a form is displayed in an entrypoint or a workitem the following happens:

- The `display()` method displays the data field.

  If the form is displayed in an entrypoint, the process instance does not yet exist, therefore `display()` cannot access information about it. When the form is displayed in a workitem, the process instance exists, therefore the `display()` method can access information on it, such as the value of other data fields.

- If the `display()` method of a work item calls `getData()` to get the value of the data field, the `load()` method is invoked.

When an entrypoint or workitem form is submitted, the following happens:

- If the process instance is at an *entrypoint*, the system automatically calls `create()` on every data field, regardless of whether the field appears on the entry point form. The `create()` method initializes the value of the data field.

  If the process instance is at a *workitem*, the process instance already exists, so the `create()` method is not called.

- If the form displayed the field is displayed in EDIT mode, the field's `update()` method is called to update the data field on the process instance. The `update()` method typically calls `setData()` to put the changed data into the data field.

- If the field's data was modified by a call to `setData()` (which might happen in the `load()`, `create()` or `update()` methods) the system calls the `store()` method to store the data.

A JavaScript script (for example, an automation script, assignment script or completion script) can use the JavaScript functions `getData()` and `setData()` to get and set the data objects of a custom field. In this case, the invocation order is as follows:

- When `getData()` is called, the `load()` method is invoked to fetch the data unless it has already been loaded for the current activity.

- The `load()` method typically uses `setData()`to load the fetched data in to the data field.

- Whenever `setData()` is performed, the `store()` method is invoked when the process instance is ready to store itself. As a result, the `store()` method may be called even if the field's data has not changed.

The next sections discuss in detail the methods that your new data field class must implement.

Implementing Data Field Methods

Define the following methods on your subclass of BasicCustomField:

- About the loadDataElementProperties Method

  Processes the properties for the data field that were set in the Builder.

- About the display Method

  Determines how the data field is displayed in an entrypoint or work item form.

- About the update Method

  Processes form element values when an entrypoint or workitem form is submitted.

- About the create Method

  Initializes the data field's value when the process instance is created.

- About the load Method

  Loads the value of the data field when an attempt is made to retrieve the value of a data field for which no value has been set yet in the current workitem.

- About the store Method

  Stores the data field value externally.

- About the archive Method

  Archives the data field value.

About the loadDataElementProperties Method

Define this method to process the properties that were set in the Builder during development of the process definition. The syntax is:

```
protected void loadDataElementProperties( Hashtable entry )
throws Exception
```

This method is passed a hashtable of all the properties that can be set in the Builder (which correspond to the properties that are defined in the JSB file). Call the `get()` method on the hashtable to get the values of specific properties.

Typically, the `loadDataElementProperties()` method sets default values that are needed by the data field. For example, if the data field displays a table, `loadDataElementProperties()` might read the background color of the table from the properties hash table. If the data field uses an external database, the method might read the database table name from the properties hash table.

For example, suppose your JSB file contains properties for `dbTableName` and `bgColor`, as follows:

```
<JSB_PROPERTY
  NAME="dbTableName"
  TYPE="string"
  DISPLAYNAME="Database Table Name"
  SHORTDESCRIPTION="DB table where this df stores its data"
  ISEXPERT>

<JSB_PROPERTY  NAME="bgColor"
  TYPE="string"
  DISPLAYNAME="Table Background Color"
  DEFAULTVALUE="white"
  SHORTDESCRIPTION="Table Background Color">
```

The `dbTableName` value is used to access a database, and the `tableBackground` color is used in an HTML `TABLE` tag when the data field is displayed.

Given the previous JSB code, the following Java code implements the `loadDataElementProperties()` method. This method reads the `dbTableName` property and sets a variable accordingly. If no value was specified in the Builder for this property, the method throws an exception. The `loadDataElementProperties()` method also gets the value for `bgColor` that was specified in the Builder. If the `bgColor` was not specified in the Builder, the method sets a default value for the corresponding variable.

```
public class myCustomField
  extends BasicCustomField
{
  // Database table name
  String MY_DB_TABLE;
  // background color for table displaying this data field
  String bgColor;

public myCustomField( )
{
  super();
}

/** Get the values that were set in the Builder */
protected void loadDataElementProperties( Hashtable entry)
  throws Exception
{
```

```
   String tableBackground = (String) entry.get( "bgColor" );
   if( tableBackground == null)
       tableBackground = "white");

   // Get the database table name
   MY_DB_TABLE = (String) entry.get( "dbTableName");
   if( MY_DB_TABLE == null )
       throw new Exception( "DB Table not specified");
}
```

About the display Method

The `display()` method determines how the data field displays itself in an HTML form in an entrypoint or a work item.

This method is invoked when the process instance reaches an activity that displays an HTML form, it is not invoked when the process instance reaches an automated or custom activity.

This method has two definitions -- one for an entrypoint and one for a workitem. When the user views an entrypoint form, the process instance does not exist yet, thus the `display()` method cannot access information on the process instance. When the user views a workitem form, the process instance does exist, therefore the `display()` method can access information about the process instance.

The syntax for an entrypoint is:

```
public void display(
    IHTMLPage html,
    int displayMode,
    String displayFormat ) throws Exception
```

The syntax for a workitem is:

```
public void display(
    IProcessInstance pi,
    IHTMLPage html,
    int displayMode,
    String displayFormat ) throws Exception
```

Define the `display()` method to write the HTML code for displaying the data field. For example, if the field is to be displayed as a table, define the `display()` method to write the `<TABLE>` and `</TABLE>` tags, as well as the tags for the table rows and table cells. Call the `write()` method on the `IHTMLPage` input parameter to write to the HTML page.

Attributes that affect the appearance of the data field (such as the background color of a table) can be defined in the JSB file and set by the process designer in the Builder. The `loadDataElementProperties()` method can retrieve them and set them as values of instance variables to make them available to the `display()` method.

The `display()` method should consider whether the data field is in view, edit, or hidden mode, and display the data field accordingly. For view mode, it should display the value in such a way that the user cannot change it. For edit mode, it should display the value in some kind of form element, such as a text field or check box, so that the user can edit the value. There is no need to write the <FORM> and </FORM> tags -- these are written automatically.

The `display()` method should write a form element for every value that is associated with the data field, even if the user is not allowed to change the value. You can use hidden form elements to transmit data values that the user does not need to see or is not allowed to edit.

The workitem version of `display()` can retrieve the value of a data field by calling the `getData()` method on the process instance. This method gets the value out of the process instance if it has already been set for the current workitem, or loads it from external storage by calling the `load()` method if it has not already been set.

As far as the `display()` method is concerned, however, all it needs to do to get the value is to call `getData()` on the process instance, specifying the name of this data field. To get the name of this data field, use the `getName()` method as follows:

```
myDataObject myObject = (myDataObject) pi.getData( getName() );
```

When a process instance is loaded into a work item, the value of a data field can be any kind of object. The `display()` method might, for example, get the values of several instance variables on an object and display each one in a separate text field.

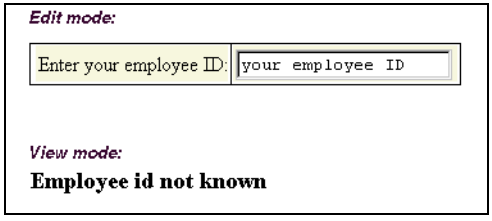
Example display() method

This example discusses a data field that manages information about employees, such as their name, phone number and email address. Each employee is uniquely identified by their employee ID.

The data field presents itself as a table, as illustrated in Figure 2-3. Some of the table attributes, such as background color, can be specified in the Builder. When the data field is displayed in an entrypoint form, it does not know which employee it is associated with. The intent is that the employee would enter their employee ID number in the entrypoint. Given the employee ID number, the data field can retrieve information about the employee from the employee database. When the data field is displayed in a subsequent workitem, it has access to information about the employee, such as their name and phone number.

The code for the entrypoint version is shown here. At an entrypoint, the only thing that the user can enter is their employee ID number. This number is needed to uniquely identify the employee in the database. Figure 2-2 shows the data field in edit and modes in an entry point form.

Figure 2-2 Example data field in an entrypoint

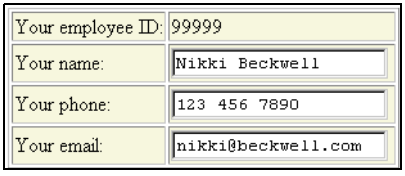



```
public void display( IHTMLPage htmlpage, int displayMode,
  String displayFormat ) throws Exception
{
  // Create a string buffer to hold the HTML code
  StringBuffer buffer = new StringBuffer();

  // Write the code to display the data field data in a table
  switch( displayMode)
  {
  case MODE_EDIT:
      // Write HTML text to display the field in edit mode
      buffer.append("<TABLE BORDER=1 BGCOLOR=" + bgColor + ">";
      buffer.append("<TR><TD>Enter your employee ID:</TD>");
      buffer.append("<TD><INPUT TYPE=TEXT NAME=idEntrypointFE"
        + " VALUE=your_employee_ID> </TD</TR></TABLE>");
      break;

  case MODE_VIEW:
      // In an entrypoint, this data field should not
      // be shown in view mode
      buffer.append("<P>Employee id not known.</P>");
      break;
```

```
  case MODE_HIDDEN:
  default:
      // Do nothing
        }

  // Write the contents to the HTML page
  htmlpage.write( buffer.toString() );
  // end class
  }
```

When the data field appears in a work item form, it shows the ID number, name, phone number and email address for the employee. The user is not allowed to change the value of the ID once the process instance has progressed beyond the entrypoint, but they are allowed to change their name and phone number when the data field is in edit mode. Figure 2-3 shows the data field in edit mode in a work item.

Figure 2-3 Example data field in a workitem



The code for the work item version of `display()` is shown here:

```
public void display( IProcessInstance pi, IHTMLPage htmlpage,
  int displayMode, String displayFormat ) throws Exception

{
  StringBuffer buffer = new StringBuffer();

  // Get the value of this data field as an object.
  // Use getName() to get the name of this data field
  myDataObject myObject =(myDataObject) pi.getData(getName() );

  // Get the employee id, name, phone number and email
  String employeeED = myObject.employeeID;
  String name = myObject.employeeName;
  String phone = myObject.phone;
  String email = myObject.email;
```

```
// Write the code to display the values of
// the data field in a table
switch( displayMode)
{

case MODE_EDIT:
    // Write HTML text to display the field in edit mode
    // Display a table that contains editable text fields
    buffer.append("<TABLE BORDER=1 BGCOLOR=" + bgColor + ">");

    // Display the employee ID as plain text
    // so it is not editable
    buffer.append("<TR><TD>Your employee ID:</TD>");
    buffer.append("<TD>'" + employeeID + "'> </TD</TR>");

    // Add a hidden element to represent the employee id
    // so that update() can access the employee id number
    buffer.append("<INPUT TYPE=HIDDEN NAME=idFE VALUE='" +
      + employeeID + "'>");

    // Display the name, phone, and email as text fields
    buffer.append("<TR><TD>Your name:</TD>");
    buffer.append("<TD><INPUT TYPE=TEXT NAME=nameFE" +
      " VALUE='" + name + "'> </TD</TR>");

    buffer.append("<TR><TD>Your phone:</TD>");
    buffer.append("<TD><INPUT TYPE=TEXT NAME=phoneFE" +
      " VALUE='" + phone + "'> </TD</TR>");

    buffer.append("<TR><TD>Your email:</TD>");
    buffer.append("<TD><INPUT TYPE=TEXT NAME=emailFE" +
      " VALUE='" + email + "'> </TD</TR></TABLE>");
    break;


case MODE_VIEW:
    // Write HTML text to display the field in
    // a table in view mode
    buffer.append("<TABLE BORDER=1 BGCOLOR=" + bgColor + ">");
    buffer.append("<TR><TD>Your employee ID:</TD>");
    buffer.append("<TD>" + employeeID + "</TD</TR>");

    buffer.append("<TR><TD>Your name:</TD>");
    buffer.append("<TD>" + name + "</TD</TR>");

    buffer.append("<TR><TD>Your phone:</TD>");
    buffer.append("<TD>" + phone + "</TD</TR>");

    buffer.append("<TR><TD>Your email:</TD>");
    buffer.append("<TD>" + email + "</TD</TR></TABLE>");
    break;

case MODE_HIDDEN:
default:
```

```
        // Do nothing
    }
    // Write the contents to the HTML page
    htmlpage.write( buffer.toString() );
}
```

For more information about `display()`, see the discussion of `display()` in the Class Reference.

About the update Method

This method updates the value of the data field on the process instance when a form is submitted in an entrypoint or workitem. This method is not invoked when the process instance finishes an automated or custom activity.

The syntax is:

```
public void update(
    IProcessInstance pi,
    IPMRequest rq ) throws Exception
```

Despite its name, the `update()` method is not the place where you update external databases when the value associated with the data field has changed. The `update()` method does not store values for persistence from one workitem to another, it just updates the process instance for the current work item only. The `store()` method stores the data field value in an external data store to make it persistent between workitems. You can define `store()` to store the data in whatever way you wish, for example in a database table of your choosing.

Typically, the `update()` method translates the name/value parameters sent by the form submission into an appropriate kind of data object for the field. If you do not define `update()`, all changes relevant to this data field that the user makes in the form are discarded when the form is submitted.

When a form is submitted, the value and name of every form element (such as text field or checkbox) on the form is packaged into a query string. One of the arguments to `update()` is an IPMRequest object that has a method for extracting individual values from the submitted parameter string. You can use the `getParameter()` method to get the value of a named parameter. You can use the `isParameterDefined()` method to test for the existence of a parameter before attempting to get its value.

When defining `update()`, you do not need to worry about whether the data field was in edit, view or hidden mode. The `update()` method is only called if the field was displayed in edit mode.

The `update()` method needs to know the names of the form elements that the `display()` method writes to the HTML page. For example, if `display()` displays a textfield called `idFE`, `update()` can access the value of that form element as follows:

```
String employeeID = rq.getParameter("idFE");
```

Conversely, if `update()` needs to receive a value from the form, it is the responsibility of the `display()` method to write an appropriately named form element to the page, even if the user is not allowed to change the value. You can use hidden form elements to transmit data values that the user does not need to see or is not allowed to edit.

CAUTION Do not use variables on the data field itself to hold values that are specific to a process instance, since all process instances effectively share a single instance of the data field.

In the following example, four form elements are used to represent the data field when it is in edit mode. These form elements are `idFE`, `nameFE`, `phoneFE`, and `emailFE`, which are form elements whose values specify employee ID number, employee name, phone, and email respectively.

This `update()` method creates a new instance of myDataObject. It extracts the values of the `idFE`, `nameFE`, `phoneFE` and `emailFE` form elements, and puts the values into corresponding variables on the object. Finally it calls `setData()` to put the object as the value of the data field on the process instance.

```
public void update( IProcessInstance pi, IPMRequest rq )
  throws Exception
{
  // Create an instance of myDataObject and set its
  // employeeID, employeeName, phone and email variables.
  myDataObject myObject = new myDataObject();

  if (rq.isParameterDefined("idFE"))
      myObject.employeeID = rq.getParameter("idFE");

  if (rq.isParameterDefined("nameFE"))
      myObject.employeeName = rq.getParameter("nameFE");

  if (rq.isParameterDefined("phoneFE"))
      myObject.phone = rq.getParameter("phoneFE");

  if (rq.isParameterDefined("emailFE"))
      myObject.email = rq.getParameter("emailFE");

  // Put myObject into the data field on the process instance
  pi.setData( getName(), myObject);
}
```

About the create Method

This method sets the default value for a data field when the process instance is created. The syntax is:

```
public void create (IProcessInstance pi) throws Exception
```

The purpose of `create()` is to set a default value for a data field when the process instance is initialized in case the data field's value is not set in an entrypoint form. If the user sets the value of the data field in an entrypoint form, the user-specified value overrides the value set by the `create()` method (assuming that `update()` handles the user-specified value appropriately).

When a process instance is initialized, the `create()` method is called on all data fields, regardless of whether they appear in the entry point form or not.

The `create()` method puts values into the process instance which is created when the entrypoint completes. The `store()` method takes the values out of the process instance and stores them externally to make them persistent until the process instance reaches the next work item.

Typically, you would define the `create()` method to create a default value and put it in the process instance through a call to `setData()`. However, not all custom fields require these actions. This decision is up to the process designer. If a default value does not need to be set, it is recommended that you define `create()` to do nothing:

```
public void create( IProcessInstance pi )
   throws Exception
{
}
```

Default values can be defined in the JSB file and set by the process designer in the Builder. The `loadDataElementProperties()` method can retrieve them and set them as values of instance variables to make them available to the `create()` method.

The following code shows a generic example of `create()`:

```
/** Initialize the data field with the default value */
public void create( IProcessInstance pi )
  throws Exception
{
  // Assign a default value for this field.
  customObject object1 = new customObject();
  object1.value1 = default_value1";
  object1.value2 = default_value2;
  pi.setData( getName(),object1);
)
```

About the load Method

This method loads the value of the data field when the process instance is at a work item.

```
public void load( IProcessInstance pi) throws Exception
```

This method is invoked when the `getData()` method of the process instance attempts to get the value of the data field but the value has not yet been set at the current work item. If the value has been set already at the current work item, then `getData()` returns the value -- it does not call `load()` again. Each time the process instance moves to a new work item, the first call to `getData()` at the new work item causes the value to be loaded again by a call to `load()`.

Note that built-in fields are loaded whenever the process instance is loaded, but custom fields are loaded only when their data value is explicitly asked for. This behavior is called lazy loading.

Define the `load()` method to retrieve the data field's value from wherever it is stored by the `store()` method. It might, for example, retrieve a set of values from a database (maybe address, phone number and employee id) and create an object that stores those values as instance variables.

If the data is loaded from an external database, you might need a key to access the database tables. The `store()` method should set this key if necessary when the entry point or work item is completed.

If a key is needed (and has been set by the `store()` method), the `load()` method can retrieve it by calling `getEntityKey()` on the process instance, specifying the data field name. This method returns a String:

```
String myKey = pi.getEntityKey(getName());
// now that you know the key, you can access the database
```

To load the value into the data field, define the `load()` method to call `setData()` on the process instance, specifying the name of the data field and the value for it.Use `getName()` to get the name of this data field:

```
pi.setData( getName(), value );
```

NOTE Do not call `getData()` on this data field from within `load()` or you will end up in an infinite loop, since `getData()` invokes `load()`.

The following example shows the basic structure for defining `load()`. The real work of extracting the values from the database is carried out in this case by the user-defined function `retrieveMyData()`.

```
public void load( IProcessInstance pi ) throws Exception
{
  // Load the data from wherever it is stored
  // and put it in the PI
  // Get the entity key
  String thisID = (String) pi.getEntityKey(getName());

  // retrieveMyData() is a user-defined function ( not a system
  // one) that interrogates an external database to get the data
  myDataObject myObject =(myDataObject) retrieveMyData(thisID);

   // Put the value in the data field
  pi.setData( getName(), myObject );
}
```

The following is a simple example of a function that extracts data from a database and returns an object that encapsulates that data. In this case, `retrieveMyData()` gets the name, phone number and email for an employee, given the key which is the employee's ID number.

This example assumes that `MY_DB_TABLE` has been defined as the database table name and that `myDataSource` has been bound to a data source for a database.

```
myDataObject retrieveMyData(String employeeID)throws Exception
{
  myDataObject myDataObject = new myDataObject();

  // Database-related variables
  Connection c = null;
  PreparedStatement myStatement = null;
  ResultSet myResultSet = null;

  // String MY_DB_TABLE; -- already defined
  // DataSource myDataSource; -- already defined

  try {
  // Connect to the database.
  // Database parameters are specified by myDataSource
  c = myDataSource.getConnection();
```

```
  // Create a query string to get the name, phone and email from
  // the MY_DB_TABLE database table
  // (for example EMPLOYEE_TABLE)
  String MY_QUERY_DATA = "SELECT name, phone, email " +
      " FROM MY_DB_TABLE WHERE employee_id = " + employeeID;

  // Prepare and execute the query statement
  myStatement = c.prepareStatement( MY_QUERY_DATA );
  myResultSet = myStatement.executeQuery();

  // Process the results
  while( myResultSet.next() )
  {
      String name = myResultSet.getString("name");
      String phone = myResultSet.getString("phone");
      String email = myResultSet.getString("email");
      myDataObject.employeeName = name;
      myDataObject.phone = phone;
      myDataObject.email = email;
  }
  }
  catch( Exception e ) {
  throw new Exception( "Cannot load " + getName() +
      " because: " + e );
  }
  return myDataObject;
}
```

For more information about `load()`, see the discussion of `load()` in the Class Reference.

About the store Method

This method stores the data associated with the custom field to make it persistent from one workitem to another.

`public void store (IProcessInstance pi) throws Exception`

This method stores the data associated with the custom field to make it persistent from one workitem to another. This method is automatically invoked during completion of an activity if `setData()`was previously called. Typically, the `setData()` method is called during completion of an entrypoint or workitem in which the data field appeared in the form, but it can also be called in other situations such as by a completion script, an automated activity or a custom activity.

The `setData()` method is typically called in the following cases:

- by `create()` to initialize the value of the data field when the process instance is created
- by `load()` to load the value into the data field for a workitem. Often the value is loaded and displayed in a form in a workitem. In such cases, `store()`is called when the work item completes even if the value did not change.
- by `update()` to update the process instance when the user enters relevant data in a form in an entrypoint or workitem

It's up to the designer of the custom field to decide where and how to externally store the data. Note that data from a custom field cannot be stored in the application-specific table where built-in data fields are stored. That is, you cannot define `store()` to just "do the default thing" and save the value in the default way as done by the builtin data fields.

Do not define `store()` to save state by storing values in instance variables. The reason for this restriction is that for each instance of a data field in a process definition, iPlanet Process Manager creates one instance of that class when the application is deployed. This instance is shared by all the process instances that use that process definition. For example, suppose you create and deploy a process definition whose data dictionary contains one instance of a custom data field, called `employeeInfo`. Three employees, Ann, Ben, and Carol, start processes instances. You might think that Ann, Ben and Carol's process instances would each create their own instance of `employeeInfo`, but you would be wrong. There is only a single instance of `employeeInfo` that is shared by all three process instances.

If the data field stores its data in an external database, it may need to use a key to identify the relevant data in the database. If applicable, defined the `store()` method to store a key to the database by calling `setEntityKey()` on the process instance. The `setEntityKey()` method stores the key with the process instance. Later, when the `load()` method needs to load the data, it can get the key to identify the data it is looking for by calling `getEntityKey()`.

Currently, iPlanet Process Manager does not support global transactions. If the custom field stores its data in an external datasource that is both XA-compliant and managed by a resource manager, the custom field could participate in a global transaction. However, transactions initiated by iPlanet Process Manager are not made through an XA resource manager, so they cannot be a part of the larger transaction.

The following example shows the basic structure for the `store()` method. In this example, the real work of storing the data gets done in the user-defined function `storeMyData()`.

```
public void store( IProcessInstance pi )throws Exception
{
  myDataObject myObject =(myDataObject) pi.getData(getName());
  String myKey = myObject.employeeID;

  // Set the key so we can get it back when needed
  pi.setEntityKey(getName(), myKey);

  // Store the data. storeMyData is a user-defined function
  // not a system one
  storeMyData(myKey, myObject);
}
```

The following is a simple example of a user-defined method that serializes a data field value to an external database. In this case, the `name`, `phone` and `email` values are stored in corresponding columns in a database table. The key is the `employeeID`.

This example assumes that `MY_DB_TABLE` has been defined as the database table name and that `myDataSource` has been bound to a data source for a database.

```
protected void storeMyData(String myKey, myDataObject myObject)
  throws Exception
{
  Connection c = null;
  PreparedStatement myStatement = null;

  // String MY_DB_TABLE; -- previously defined
  // DataSource myDataSource; -- previously defined
try {
  // Create the SQL statement for updating the database
  String SQL_UPDATE_DATA = "UPDATE "  +
      MY_DB_TABLE +
      " SET name = ? , phone = ?, email = ?" +
      " WHERE id = myKey";

  // Connect to the database.
  // Database parameters are specified by myDataSource
  c = myDataSource.getConnection();
  // Prepare and execute the SQL statement
  myStatement = c.prepareStatement( SQL_UPDATE_DATA );
  // do any other necessary preparation work
  ...
  // Update the database
  try {
      myStatement.executeUpdate();
  }
catch( Exception e ){
      throw new Exception( "Cannot save data for data field: "
        + getName() + " because: " + e );
  };
  c.commit();
  }
```

For more information about the `store()` method, see the discussion of `store()` in the Class Reference.

About the archive Method

When an archive operation is initiated from the administration pages, the `archive()` method for each data element associated with the process instance writes its data value to an output stream. The syntax for `archive()` is:

```
public void archive(
   IProcessInstance pi,
   OutputStream os) throws Exception
```

Built-in data elements archive themselves simply by writing their values out as bytes. By contrast, you can determine how custom fields write their data to an output stream. For example, you can stream bytes or encapsulate the values in XML.

For more information, see the discussion of `archive()` in the Class Reference.

Specifying Images for Use in Process Builder

You can optionally create images to represent data fields in the Process Builder. Name the image that represents the data field in edit mode as *dataFieldName*`-EDIT.gif`, and name the image that represents the data field in view mode as *dataFieldName*`-VIEW.gif`.

For example, for the `myNewCustomField` data field, the edit mode image is `myNewCustomField-EDIT.gif`, and the view mode image is `myNewCustomField-VIEW.gif`.

Packaging a Custom Field

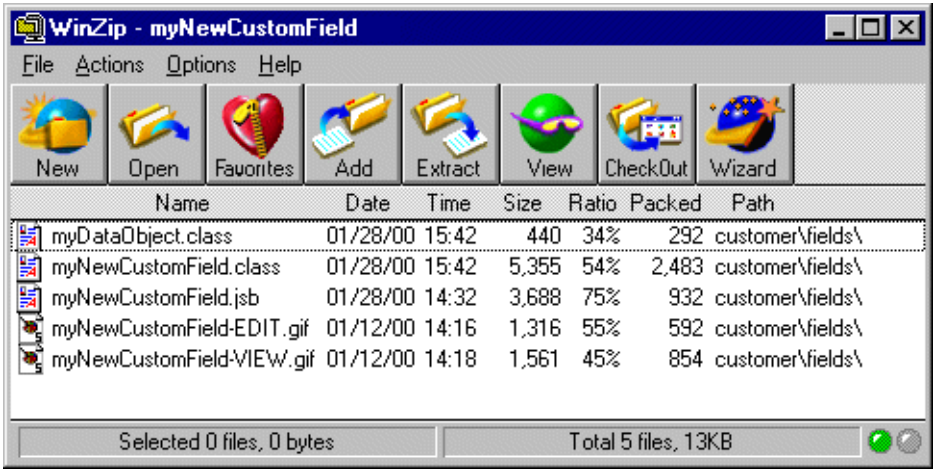
After you have compiled your custom field Java classes, defined the JSB file and optionally created images to represent the data field in the builder, the next step is to package these files into a zip or jar archive. Include any additional classes that your custom field uses in the archive.

Figure 2-4 shows an example archive file for a custom data field called `myNewCustomField`. In this case, the data field is in a package `customer.fields`. The archive contains the following files:

- `myNewCustomField.jsb` is the JSB file for this custom field.
- `myNewCustomField.class` is the class file for this custom field.

- `myDataObject.class` is the class of data objects that are used as the dat field values.
- `myNewCustomField-EDIT.gif` and `myNewCustomField-VIEW.gif`. are GIF image files that are used as icons to represent the data field in edit and view mode in the Builder.

Figure 2-4 Archive file for myNewCustomField



Note that the archive file, JSB file, and custom field class must all have the same root name. In the example shown in Figure 2-4, this name is `myNewCustomField`.

As you create the archive, check that the directory structure reflects the package structure of the class. For example, if the class files are in the package `customer.fields`, the class files must be in the directory `customer/fields`, as shown in Figure 2-4. The JSB file must be at the same level as the class files.

NOTE When you use the `jar` command to create an archive, a file named `manifest.mf` is automatically created by default. This file contains information about the other files within the archive. The `manifest.mf` file has no effect on the custom field.

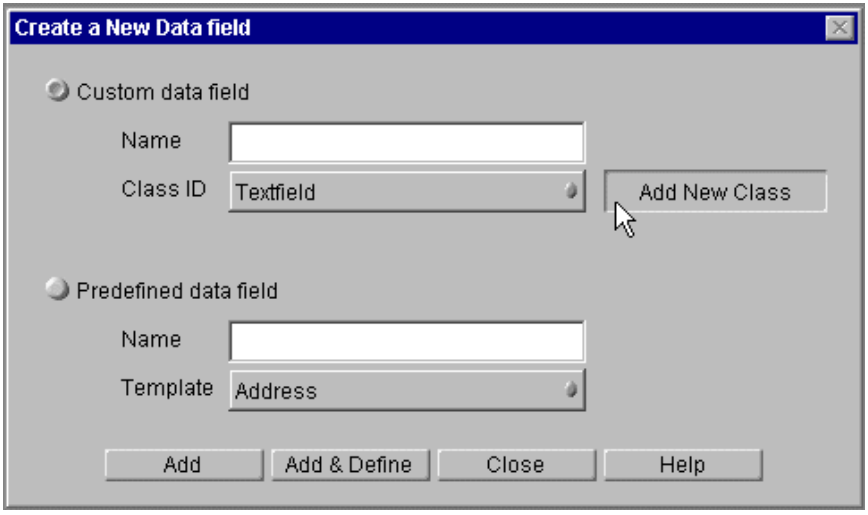
Adding a Custom Field to an Application

After you package a custom field as an archive file, you can add the field in Process Builder, as described in this section.

The specific steps for adding a custom field are as follows:

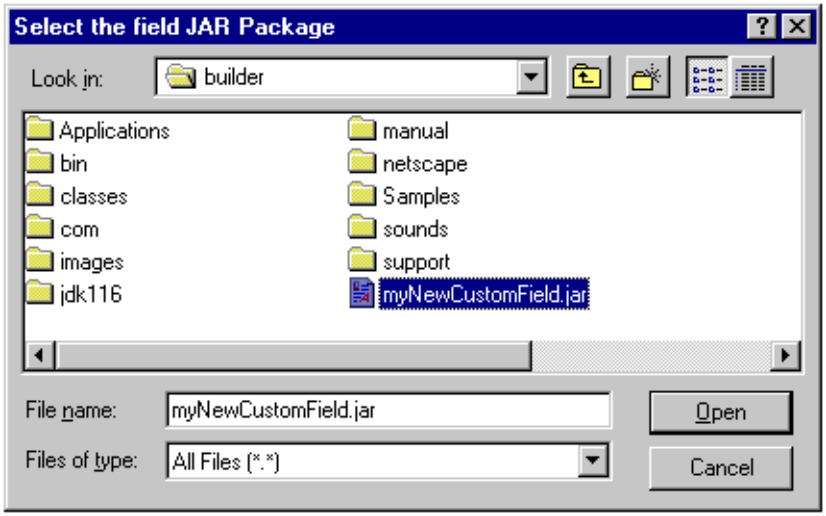
1. From the Insert menu, choose Data Field.
2. In the "Create a New Data Field" dialog box click Add New Class. An example is shown in Figure 2-5:

Figure 2-5 Creating a data field from a new class



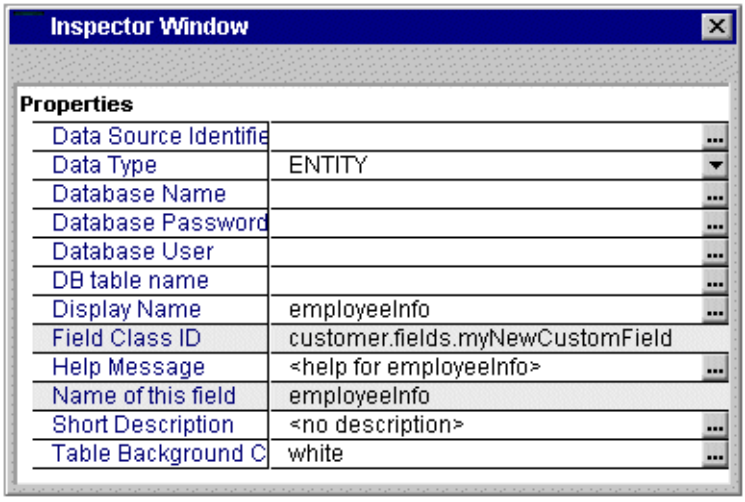
3. In the "Select the field JAR Package" dialog box, select the archive that represents your custom field class, then click Open. An example is shown in Figure 2-6:

Figure 2-6 Selecting the archive that represents a custom field class



4. In the Name field, enter the name of the new field.
5. Add the field to the Data Dictionary in either of two ways:
   - Click Add to add the field without setting its properties first. The dialog box remains open, and you can add more items.
   - Click Add & Define to add the field and set its properties immediately. The Inspector window appears for the data field you added, as shown in Figure 2-7.

Figure 2-7 Setting properties for the new custom field



6. Set the properties and close the window when you are done.

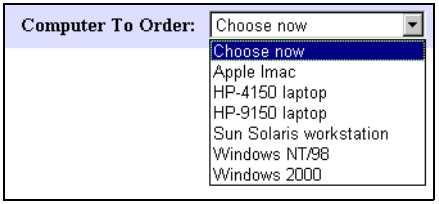
The new data field, with the properties you defined, now appears in the Data Dictionary folder in the application tree view. You can now use the data field as you would use a typical data field in Process Builder.

Example Custom Data Field

The AdvancedOfficeSetup sample application that ships with the Process Builder includes an example of a custom data field.

The advanced office setup application uses a custom data field called dfComputerChoice that presents a pop-up menu of computers that can be ordered for a new employee, as shown in Figure 2-8. This data field dynamically generates the list of computers every time it is displayed in edit mode. It gets the list by reading an XML file containing the choices. Whenever the company's list of approved computers changes, all the administrator needs to do is to change the list in the XML file -- there's no need to redeploy the Process Manager application.

Figure 2-8 Pop-up menu of computers



For full details of this custom data field, see Chapter 3, "Advanced Office Setup Application."

Development Hints and Tips

This section gives some hints and tips for developing and debugging custom fields:

- "Avoid Non-default Constructors"
- "Avoid Instance Data"
- "Use Entity Keys"
- "Deploy the Custom Field to Test It"
- "Develop and Test on a Server Where iPlanet Process Manager is Installed"
- "Use One Implementation of a Java Class Per Server"
- "Debugging Hints"

Avoid Non-default Constructors

In classes that extend `BasicCustomField`, do not define non-default constructors (meaning constructors with non-zero arguments). Process Manager has no prior awareness of non-default constructors and therefore cannot call them. Thus if you define non-default constructors, your class may encounter problems during loading.

Avoid Instance Data

Custom fields, like custom activities, are stateless entities. In effect, there is only one copy of each occurrence of a custom data field per application. All the process instances in an application effectively share the same custom data field instance for each occurrence of the custom data field class in the application. Because of this, it's recommended that you avoid using instance data in a class that implements a custom data field. If you can't avoid using instance data, be sure to synchronize the data. With unsynchronized data, a variable set during one request might not exist for the next request.

For example, consider a custom data field class called `DynamicList` that dynamically generates a list of things (such as computers), displays the list as a SELECT menu in HTML and stores the selected item in a text file.

Suppose the custom data field erroneously uses an instance variable to keep track of the file name where the value is stored, as shown in the following code:

```
// This is the WRONG way to track the name of the file
// where data field values are stored!!

public class UpdatableList extends BasicCustomField implements
IPresentationElement ,IDataElement
{
  // The name of the file where the value is saved
  public String thisFile;
protected void loadDataElementProperties(Hashtable entry )
throws Exception
{

    // Generate the file name
    thisFile = generateFileName();
}
```

```
public void store(IProcessInstance pi) throws Exception
{
  // Store the value
  thisValue = pi.getData(getName());

  storeItNow (thisValue, thisFile);
  ...
}
public void display(IProcessInstance pi, IHTMLPage html,
  int displayMode, String displayFormat ) throws Exception
{
  String selectedOption = pi.getData(getName());
  // code to display the selected option
  ...
}
public void load(IProcessInstance pi) throws Exception
{
   // Get value out of the file
  thisValue = readDataFromFile (thisFile);
  return thisValue;
}
```

Suppose Carol starts a process instance that uses the `ChooseComputer` data field, which is an instance of this data field class. The `loadDataElementProperties()` method sets the value of the `thisFile` variable to `CarolsValue.txt`. She chooses an HP laptop computer. Her `store()` method stores the value in the file.

Later, Alice starts a process instance. The `loadDataElementProperties()` method sets the value of `thisFile` to `AlicesValue.txt`. Alice chooses a Compaq Pro computer, which is saved to the file `AlicesValue.txt`.

Carol's process instance reaches a work item where the `ChooseComputer` data field is displayed again, perhaps to confirm the choice of computer. This time, when the `load()` method is invoked by the `display()`, it reads the value from `thisFile`, which is now `AlicesValue.txt`. Thus the data field displays Alice's choice, not Carol's, which does not please Carol at all.

For an example of the correct way to record the names of files where data field values are stored, see Chapter 3, "Advanced Office Setup Application" for a discussion of the example data field class, `updatableList`, that is available in the AdvancedOfficeSetup application.

It is OK, however, to use instance variables to store read-only information that is common across all process instances, such as the name of the external database or file where the data is stored. For example, the custom data field discussed previously might use a global variable to record the file name of an XML file that contains the choices to be displayed in the SELECT list (for example, IBM ThinkPad, Apple iMac, Sun Solaris workstation and so on) . These choices do not change from employee to employee, so it is OK to store the name of the file containing the choices as an instance variable. (Note that the file name containing the selected choice *does* vary from process instance to process instance, whereas the one containing the choices for the menu does *not* vary from process instance to process instance.)

An application can contain multiple occurrences of a custom data field class. For example, an application might have custom data fields called `ComputerChoices` and `ChairChoices`, that are both instances of the `DynamicList` class. When the application is running, these two fields operate completely independently. If the fields use instance data, that data would not be shared between them. For example, if the data fields use an instance variable called `myChoicesFileName`, the `ComputerChoices` data field could set it to `computerChoices.xml` while the `ChairChoices` field could set it to `chairChoices.xml`, and there would be no confusion between the two.

Use Entity Keys

When a custom field loads data from an external data source, the custom field might need a key to identify the data it is looking for. This key, known as an entity key, can be stored with the process instance.

To work with entity keys, use the following two methods on the `IProcessInstance` interface:

- `getEntityKey(fieldName)`

  This call returns the key for the custom field whose name is fieldName.

- `setEntityKey(fieldName, key)`

  Specifies `key` as the key for the custom field whose name is `fieldName`.

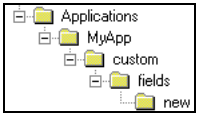
Deploy the Custom Field to Test It

To test and debug a custom data field, you need to import it into a Process Manager application in the Builder, deploy the application and then test it. During the development process, you will make changes to your Java source files and recompile the classes. Whether you need to redeploy your application or not depends on where Process Manager is running.

If the Process Manager engine is running on the same computer where you do your Java development work, you do not need to redeploy the application from the Builder but you do need to restart the Application Server, at least the KJS component. See the next section for more details.

If the Process Manager engine is running on a remote computer, you need to redeploy the application each time you make changes to the Java classes for the custom data field. Before redeploying, make sure the changes have been copied into the appropriate places (as discussed next) in the `Applications` folder hierarchy.

When you use the Builder to bring a zip or jar file for a custom data field into a Process Manager application, the Builder unzips the zip or jar file and then creates the folders needed for the package structure for the custom data field. For example, if the application name is `myApp` and the custom data field is in the package `custom.fields.new`, the Builder creates a folder called `new` in a folder called `fields` in a folder called `custom` in the `myApp` folder in the `Applications` directory, as illustrated in the following image:



The Builder places the unzipped files into the appropriate folders, for example, it places the JSB and Java class file for the data field in the `new` folder.

After making changes to the Java files, copy the compiled class files into the appropriate folder beneath the Applications directory. If you make changes to the JSB file, make sure the changes are also copied to the appropriate folder beneath the Applications directory.

To test the changes, redeploy the application.

When you're done making changes, make a new zip or jar file so that the finished data field can be imported into other Process Manager applications.

Develop and Test on a Server Where iPlanet Process Manager is Installed

If your server is on your development machine, you can develop and test your custom field without redploying after each modification.

1. To start with, develop and compile your Java classes in your preferred Java development environment.
2. Create the JAR file, import it into the Process Builder, and then build and deploy an application that uses the custom field.
3. When you need to make changes to the Java file that defines the custom data field, edit and compile your Java classes in your preferred Java development environment as before.
4. If your output directory is not in the classpath for the Process Manager engine (BPMCLASSPATH) , copy the class file into the Process Manager classpath.
5. Restart the kjs on the Application Server to update your deployed applications to use the newly compiled data field class. (For information on restarting the kjs, see the section "Print Debugging Information.")

   New and existing process instances will use the new definition for the custom data field.
6. When you have completely finished debugging your custom field, create a new jar file for importing into the Process Builder.

Use One Implementation of a Java Class Per Server

When an application is deployed, the Java classes it uses are deployed to the appropriate folder in the class path on the engine. This class path is shared by all applications running on the engine. Every application that uses a particular Java class *uses the same implementation of that class*. For example, suppose application A and application B both use a Java class `SharedClass1`. When application A is deployed, its version of `SharedClass1` is deployed to the class path. When application B is deployed, its version of `SharedClass1` is deployed to the class path, overwriting the implementation deployed previously by application A.

Thus if multiple applications running on a Process Manager engine use the same custom data field, they should all use exactly the same implementation of the custom data field, since each time the custom data field is deployed to the engine, it overwrites the previous implementation.

If you want multiple applications to use a custom data field that is basically the same but differs slightly from application to application, make sure that the name of the data field Java class is different in each application.

Debugging Hints

Print Debugging Information

To get debugging information, your Java methods can use `System.out.println("Helpful debugging info goes here")` to display debugging information in the kjs console on the Application Server if you start it from a DOS command prompt.

Note: To run the Application Server from the command line, use the interface to stop the application server. Then open a command prompt and type `kxs`. When the kxs process blocks, type `kjs` in another command prompt to start the kjs.

Send Error Messages to the Process Manager Logs

You can also log debugging information to the iPlanet Process Manager logs by using the `log()` method on the IPMCluster bean or the `log()` method on the IPMApplication bean. Use the `getPMApplication()` method of `BasicCustomField` to get the IPMApplication.

For example:

```
IPMApplication myapp = getPMApplication();

myapp.log(IPMApplication.LOG_INFORMATION, "log info here", null);
```

You can view the logs in the Process Administrator at `http://`*`yourPMserver:port`*`#/Administrator.apm`.

Errors in store()

If the `store()` method has a problem, it might result in `store()` being invoked twice while the system tries to restore the process instance to its previous state. Thus if you have print statements that record entry into and exit from `store()`, and you see that `store()` is invoked twice or seems to be entered recursively, or you see `load()` being called from inside `store()` when `getData()` is called, this is an indication that there might be a problem with the definition of your `store()` method.

Class Reference

The remainder of this chapter provides a class and method reference for the classes needed for implementing a custom data field. The classes are:

- "BasicCustomField"
- "IPMRequest"

Several methods on these classes take an `IProcessInstance` object as an input argument. This class has methods for interacting with the process instance, such as getting the values of other data fields, getting the creation date and finding out who created the process instance. For details of the methods you can use to work with process instances, see the section "IProcessInstance" on page 155.

BasicCustomField

This class is the superclass for all custom data fields. To define a custom data field, create a subclass of BasicCustomField.

```
import com.netscape.pm.model.BasicCustomField;

public class myCustomField
  extends BasicCustomField
{
  ...
}
```

The BasicCustomField class provides methods that enable iPlanet Process Manager to treat your custom field just like any other data field. Most of the methods in BasicCustomField are predefined and are used internally by iPlanet Process Manager. However, you need to provide a definition for the following methods in your custom data field class:

- `archive()`
- `create()`
- `display()`
- `load()`
- `loadDataElementProperties()`
- `store()`
- `update()`

BasicCustomField also implements the `getPMApplication()` method that returns the IPMApplication that contains the field. Two other methods implemented by BasicCustomField are `getName()` and `getPrettyName()` which are specified by the interface `IPMElement` . Your custom data field can use these methods to get the name of the field itself.

- `getName()`
- `getPMApplication()`
- `getPrettyName()`

archive()

Summary Writes the data associated with the custom field to an output stream. When you create a custom data field, you should define this method if you want your data field to be archivable. This method is specified by the IDataElement interface.

Syntax

```
public void archive (IProcessInstance pi,OutputStream os) throws
Exception
```

Arguments

- pi — Object representing the process instance.
- os — The output stream to write the data to.

Return Value None.

Description When an archive operation is initiated from the administration pages, the data elements associated with the process instance write their data values to an output stream. Built-in data elements archive themselves, simply by writing their values out as bytes. By contrast, you can determine how custom fields write their data to an output stream. For example, you can stream bytes or encapsulate the values in XML.

If a call to this method fails, you can throw a `java.lang.Exception` at any time to signal an error. The error message will be displayed to the administrator.

For more information about this method, see the section "About the archive Method" on page 77.

create()

Summary Initializes a newly created process instance with a default value for the custom field. When you create a custom data field, you must implement this method if you want your custom data field to have a default value in cases where it does not appear on an entry point form. This method is defined by the IDataElement interface.

Syntax

```
public void create( IProcessInstance pi) throws Exception
```

Arguments

- pi — Object representing the process instance.

Return Value None.

Description Most of the time, the `create()` method creates a default value and stores it in the process instance through a call to `setData()`. However, not all custom fields require these actions. This decision is up to the process designer.

If a default value does not need to be set, it is recommended that you do not implement the `create()` method. Leave it blank instead. The `store()` method for custom fields is called only when `setData()` has been performed on the field.

The `create()` method for all fields, whether predefined or custom fields, is called when the user initiates a process instance from the entry point.

If a call to this method fails, you can throw a `java.lang.Exception` at any time to signal an error. The error message will be displayed to the user, and the process instance will not be created.

For more information and an example, see the section "About the create Method" on page 69.

display()

Summary Displays the custom data field in the HTML page. When you create a custom data field, you must implement both versions of this method. This method is specified in the IPresentationElement interface.

Syntax 1 This version displays the field when the user is viewing the entry point form, in which case the process instance does not yet exist.

```
public void display( IHTMLPage html, int displayMode, String
displayFormat ) throws Exception
```

Syntax 2 This version displays the field in a workitem form, in which case the process instance does exist.

```
public void display( IProcessInstance pi,IHTMLPage html, int
displayMode, String displayFormat ) throws Exception
```

Arguments

- pi — Object representing the process instance.
- html — Object representing the HTML page to be returned to the user.
- displayMode — Mode that the field should be displaying itself in. Possible values are MODE_EDIT, MODE_VIEW and MODE_HIDDEN.
- displayFormat — Additional formatting information available to the field. This value is specified from the "Display Format" property of the Inspector window of the field when it is placed in the form. This value is specific to a process designer. One possible use is to distinguish between a secure viewing mode and a non-secure viewing mode, such as for credit card information. In such a case, the display format could contain either the value "secure" or "not secure."

Return Value None.

Description The version of `display()` shown in Syntax 1 will be called after the process instance has been created. In other words, it is called everywhere but the entry point node. The process instance will contain the data that is associated with your custom field; your implementation of `display()` will need to fetch the data object via the `getData()` method of the process instance class before displaying it.

The `displayMode` and `displayFormat` arguments are defined by the process designer through the Inspector window.

If a call to this method fails, you can throw a `java.lang.Exception` at any time to signal an error. The error message will be displayed to the user.

For more information about this method, see the section "About the display Method" on page 62.

For an example, see the section "Example display() method" on page 63

getName()

Summary Returns the name of the current element. This method is defined by the IPMElement interface.

Syntax

```
public String getName()
```

Arguments None.

Return Value A String object representing the name of the current element.

Description The returned name is used to access the field's primary key and data value.

Example The following code uses `getName()` inside the `create()` method:

```
public void create( IProcessInstance pi )
  throws Exception
{
  // Assign a default value for this field.
  // Just an empty shopping cart...
  pi.setData( getName(), new ShoppingCart() );
}
```

getPMApplication()

Summary Returns the IPMApplication for the application containing this data field.

Syntax

```
public String getPMApplication()
```

Arguments None.

Return Value An IPMApplication bean representing the application containing this data field.

Description Use this method to get access to the application containing the custom data field. IPMApplication has many useful methods for accessing other information about the application

Example The following code gets the pathname for the application containing this data field.

```
// Inside store(), get the application path
String path;
try {
   path = getPMApplication().getHomePath();
   }
catch (Exception e) {
   System.out.println("Exception getting app path" + e);
}
```

getPrettyName()

Summary Returns the "pretty name" of the current element. This method is defined by the IPMElement interface.

Syntax

```
public String getPrettyName()
```

Arguments None.

Return Value A String object representing the pretty name of the current element.

Description In previous releases of iPlanet Process Manager, every element had a name as well as a "pretty name," the display name of the element. In the current release, an element's pretty name and its name are equivalent.

load()

Summary Loads the data associated with the custom field. When you create a custom data field, you must implement this method. This method is specified by the IDataElement interface.

Syntax

```
public void load( IProcessInstance pi) throws Exception
```

Arguments

- pi — Object representing the process instance.

Return Value None.

Description The `load()` method is invoked whenever the data value associated with the custom field is accessed through `getData()` off the process instance. Note that built-in fields are loaded whenever the process instance is loaded, but custom fields are loaded only when their data value is explicitly asked for. This behavior is called lazy loading.

Warning: Within the `load()` method, do not call `getData()` on the custom field. The `load()` method is already invoked as a result of a call to `getData()`. As a result, a call to `getData()` within the `load()` method causes an infinite loop.

If a user script accesses or modifies the data associated with a custom field, the script must implicitly know the object's data type. For example, a script would need to know the API for objects such as `ItemSet` and `Item` in a shopping cart custom field.

If a call to `load()` fails, you can throw a `java.lang.Exception` at any time to signal an error. If the current action is to display a form, an error message will be displayed to the user. If the user has completed a work item, an exception work item will be created.

For more information and an example, see the section "About the load Method" on page 70

loadDataElementProperties()

Summary Loads the design-time properties for the field specified in Process Builder's Inspector window. Specified by BasicCustomField. Custom data fields should implement this method.

Syntax

```
protected void loadDataElementProperties(Hashtable entry ) throws
Exception
```

Arguments

- entry — The hashtable containing property/value pairs for the properties of this field that can be set in Process Builder .

Return Value None.

Description This method is called after the custom field has been created (while the application is being initialized). The hashtable entry parameter contains the field's configuration information, as it is stored in the LDAP repository. This information includes the properties you specified in the custom field's JSB file (which are the properties that appear in the inspector window in Process Builder).

If a call to this method fails, it can throw a `java.lang.Exception` at any time to signal an error. The error message will be displayed to the user, and the application will stop being initialized.

For more information about this method, see the section "About the loadDataElementProperties Method" on page 60.

Example Suppose your JSB file contains the following entry:

```
<JSB_PROPERTY
  NAME="dbidentifier"
  TYPE="string"
  DISPLAYNAME="External DB Identifier"
  SHORTDESCRIPTION="Local alias for connecting to external DB"
  ISEXPERT>
```

Given the previous JSB code, the following Java code implements the `loadDataElementProperties()` method. The method will first read the property and then, based on the value, set the instance variable `mDBIdentifier`.

```
protected void loadDataElementProperties( Hashtable entry )
  throws Exception
{
  String dbIdentifier = (String) entry.get( "dbidentifier" );
  if( dbIdentifier == null )
      throw new Exception( "DB Identifier not specified" );
  else
      mDBIdentifier = dbIdentifier;
}
```

store()

Summary Stores the data associated with the custom field to a persistent resource. When you create a custom data field, you must implement this method. This method is defined by the IDataElement interface.

Syntax

```
public void store( IProcessInstance pi) throws Exception
```

Arguments

- pi Object representing the process instance.

Return Value None.

Description It's up to the designer of the custom field to decide which external persistent data store will store the custom field data. Note, however, that data from a custom field cannot be stored in the application-specific table, where built-in data fields are stored.

The custom field is responsible for storing the data, whereas iPlanet Process Manager is responsible for storing the custom field's primary key. This key is stored in the application-specific database table.

The `store()` method is called only if the field's value has been modified, through a call to `setData()`. Note that the `load()` method typically calls `setData()`. As a result, the `store()` method is called whenever `load()` is called.

Currently, iPlanet Process Manager does not support global transactions. If the custom field stores its data in an external datasource that is both XA-compliant and managed by a resource manager, the custom field could participate in a global transaction. However, transactions initiated by iPlanet Process Manager are not made through an XA resource manager, so they cannot be a part of the larger transaction.

If a call to this method fails, you can throw a `java.lang.Exception` at any time to signal an error. The current work item is converted to an exception work item, and all data field values are reset to their values prior to the request.

For more information and an example, see the section "About the store Method" on page 73.

update

Summary Determines how the HTML representation of a custom data field is processed when a form is submitted. Typically this method translates the form element value into the usual data object associated with the field. When you create a custom data field, you must implement this method. This method is specified by the IPresentationElement interface.

Syntax

```
public void update( IProcessInstance pi, IPMRequest rq ) throws
Exception
```

Arguments

- pi — Object representing the process instance.
- html — Object representing the HTTP request.

Return Value None.

Description The `update()` method is called after the user has submitted a request to the iPlanet Process Manager server. Since all requests take the form of an HTTP GET or HTTP POST, this method translates the form parameters of the request into the usual data object associated with your custom field. For example, suppose the form includes values for an item ID and an item quantity. The `update()` method would convert the item quantity to a numeric value, and the method would create an `Item` object out of the item ID. The `Item` object could then be bound to the process instance via `setData()`.

If a call to this method fails, you can throw a `java.lang.Exception` at any time to signal an error. The error message will be displayed to the user.

For more information and an example, see the section "About the update Method" on page 67.

IPMRequest

The IPMRequest class represents requests sent by the browser to the iPlanet Process Manager engine when a form is submitted. These requests contain the values of the form elements in the form. An IPMRequest object is automatically passed to the `update()` method of a custom data field class. The `update()` method can access the IPMRequest object to extract form element values and to find the authenticated user.

The IPMRequest class has the following methods:

- `getAuthenticatedUserId`
- `getParameter`
- `isParameterDefined`

getAuthenticatedUserId

Summary Gets the ID of the authenticated user who made the request.

Syntax

```
public String getAuthenticatedUserId( ) throws XInvalidRequest;
```

Arguments None.

Return Value A String of the name of the authenticated user.

Description Gets the ID of the authenticated user who made the request

getParameter

Summary Gets the value of a parameter in the request string. Typically, the parameter is the name of a form element in the form that was submitted.

Syntax

```
public String getParameter(String parameter) throws XInvalidRequest;
```

Arguments

- parameter The name of the parameter whose value is to be retrieved.

Return Value A String of the value of the parameter.

Description Gets the value of a parameter in the request string. Typically, the parameter is the name of a form element in the form that was submitted. This method is typically invoked by the `update()` method of a custom data field to extract form element values.

Example The section, "About the update Method" on page 67, contains an example of using the `getParameter` method.

isParameterDefined

Summary Returns true if a parameter is defined in the query string sent by a form submission, otherwise returns false.

Syntax

```
public boolean isParameterDefined(String parameter)
```

Arguments

- parameter — The name of a parameter whose existence is being tested

Return Value A Boolean indicating whether the named parameter exists or not.

Description Returns true if a parameter is defined, otherwise returns false. The `update()` method can use this method to test for the existence of a parameter before attempting to retrieve its value. For example, if an entrypoint form displays different data fields than a work item form, `update()` can test for the existence of particular data fields to determine if the form came from an entry point or a work item.

Example The section, "About the update Method" on page 67, contains an example of using the `isParameterDefined` method.

Advanced Office Setup Application

This chapter discusses the AdvancedOfficeSetup sample application that is provided with Process Builder. This application, which is an advanced version of the OfficeSetup sample application, uses both a custom data field and a custom activity.

The sections in this chapter are:

Changes in the Advanced Office Setup Application

The advanced office setup sample application basically achieves the same goal as the simple office setup application, which is to get an office ready for a new employee. However, the advanced version has been fine-tuned to improve the process.

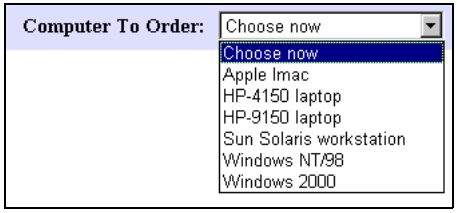
The differences between the two versions are:

- The advanced version uses a custom data field to dynamically generate the list of computers that can be purchased for the new employee.

  The intent here is to limit the choice a pre-defined selection of computers that have been approved for corporate use. Previously, the computer choice was represented as a text field, thus the administrative assistant could enter any computer they wanted in this field, from a Palm Pilot to a Cray supercomputer.

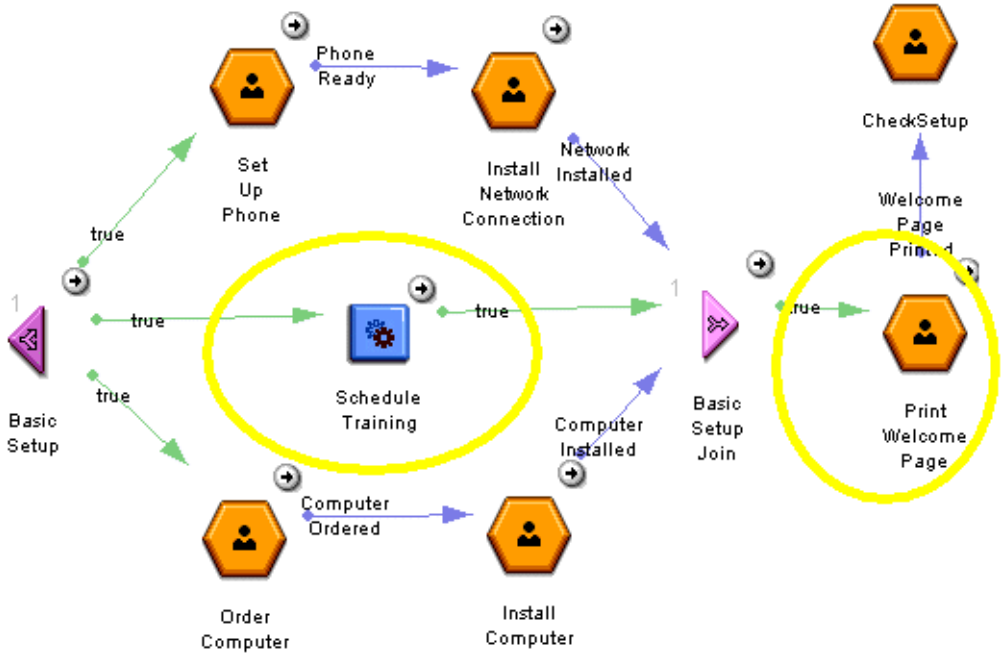
  The figure below shows the pop-up menu of computer choices.

Figure 3-1 Pop-up menu of computers



- The advanced version has a custom activity that automatically schedules the new employee to attend a company orientation training.

  The day of the training depends on which department the employee is joining and what day they start work at the company. The following figure shows the custom activity in the process map, as well as a new user activity for printing the information about the training.



- The advanced version has an additional manual activity which requires the administrative assistant to print a page that tells the employee what day to attend company orientation training. This page is written by the custom activity.

The Custom Data Field

This section discusses the custom data field and has the following subsections:

Overview

The advanced office setup application uses a custom data field called `dfComputerChoice` that presents a pop-up menu of computers that can be ordered for the new employee, as shown in Figure 3-1.

This data field dynamically generates the list of computers every time it is displayed in edit mode. It gets the list by reading an XML file containing the choices. An example of the XML file is:

```
<xml version="1.0" encoding="us-ascii">
<ITEMSET>
  <ITEM>Apple Imac</ITEM>
  <ITEM>HP-4150 laptop</ITEM>
  <ITEM>HP-9150 laptop</ITEM>
  <ITEM>Sun Solaris workstation</ITEM>
  <ITEM>Windows NT/98</ITEM>
  <ITEM>Windows 2000</ITEM>
</ITEMSET>
```

This XML file resides in the same folder as the application. When the process designer deploys the application from the Builder, the XML file is automatically copied to the correct location on the server. After the application has been deployed, users can modify the file whenever the company's computer purchase policy changes. The changes take effect immediately.

The selected value is stored externally as an object that is serialized to a file. The file UpdatableList.java, available in Appendix A, "Scripts", contains the entire source code.

The Code in Detail

This section discusses the code for the methods on the custom data field.

- loadDataElementProperties ()
- display()
- update()
- store()
- load()
- Helper Functions
- Complete Source Code

loadDataElementProperties ()

This method reads the properties that were set in the Builder. In this case, it reads the value of the `xmlfile` property and stores it in a global variable, `myFileName`.

Although as a general rule you should not store data in instance variables, in this case it is OK because this file name is constant for all process instances in the application -- it never changes. (The contents of the file might change, but the file name itself never changes).

```
// Method from BasicCustomField that loads
// properties that were set in the Builder
protected void loadDataElementProperties (Hashtable entry)
  throws Exception
{
    // Get the XML File name from the Builder properties
    myFileName = (String)entry.get("xmlfile");
}
```

display()

When a form containing the custom data field is displayed in an HTML page, the field's `display()` method is invoked. If the process is at an entry point, the entry point version of `display()` is invoked; if it is at a work item then the work item version is used.

The purpose of this data field is to present a list of choices, store the selection, and retrieve the selection the next time the data field is displayed. Therefore, in an entry point, there is no point displaying the field in view mode, since there is no prior selection to view.

The data field shows a menu of computers, which is displayed as a SELECT list, for example:

```
<SELECT size="1" name="dfComputerChoice" >
  <OPTION selected>Choose now</OPTION>
  <OPTION value="Apple Imac">Apple Imac</OPTION>
  <OPTION value="HP-4150 laptop">HP-4150 laptop</OPTION>
  <OPTION value="HP-9150 laptop">HP-9150 laptop</OPTION>
  <OPTION value="Sun Solaris workstation">
    Sun Solaris workstation</OPTION>
  <OPTION value="Windows NT/98">Windows NT/98</OPTION>
  <OPTION value="Windows 2000">Windows 2000</OPTION>
</SELECT>
```

At an entry point in edit mode, the data field reads the XML file and displays all the choices. At a work item in edit mode, the data field checks if there is a previously selected value. If a value has been chosen previously, the data field displays it as the current selection. If a value has not been chosen previously, the data field displays the default initial value.

The data field has no meaning when used in view mode at an entry point, thus is displayed as a simple warning. The data field is displayed as plain text showing the current selection in view mode at a work item.

Here is a discussion of the `display()` method for a work item. The code is similar but simpler for the a work item, since it does not need to consider whether there is a previously selected value or not.

display() at a Work Item

In edit mode, the `display()` method starts by calling `getData()` to get the value of the data field. If the value is already loaded, `getData()` simply returns it, otherwise `getData()` invokes `load()` to load the value. In this case, `load()` gets the value by reading it from a file. The retrieved value is an object that has a variable, `myvalue`, which indicates the current value. For example, if the hiring manager has previously selected Apple Imac as the computer for the new employee, then `myvalue` would be bound to `Apple Imac`.

The setting of the `myvalue` variable happens in the `store()` method, which we will worry about later. For now, it's enough to know that the `myvalue` variable holds the selected option.

```
public void display(IProcessInstance pi, IHTMLPage html,
  int displayMode, String displayFormat ) throws Exception
{

StringBuffer buffer = new StringBuffer();
String selectedOption = null;

// Get the value of the data field
// If the value is not loaded, getData invokes load()
myObject myobj = (myObject) pi.getData(getName());

// If an object is found, set the selected option
// to the value of the object's myvalue variable.
if (myobj != null) {
  selectedOption = myobj.myvalue;
}
```

The `display()` method writes the HTML code to display a SELECT menu. Each menu item is embedded in an <OPTION> tag. The selected menu item is indicated by <OPTION SELECTED>.

The `display()` method reads all the menu items from the appropriate XML file. If no menu item has been previously selected, it uses a default value for the selected option, which in this case is <OPTION SELECTED>Choose now</OPTION>. Then the method writes <OPTION> tags for all the menu items.

```
switch(displayMode){
// In edit mode, display the data field as a SELECT menu
// The menu options are stored in an xml file

case MODE_EDIT:
  // Get the option names from the xml file and store
  // them in the vector optionNames.
  Vector optionNames = fetchDataFromXML();

  // Write the opening <SELECT> tag.
  // The name is the same as the data field name.
  buffer.append("<select size=1 name=" + getName() + " >");

  // If the option was not selected previously show default
  String optName = "";
  if (selectedOption==null)
  {
    buffer.append("<option selected>Choose now</option>");

    // For each option in the vector optionNames
    // write <OPTION> value="optionName"</OPTION>
        for(int i=0; i<optionNames.size(); i++)
    {
      optName = (String)optionNames.elementAt(i);
      buffer.append("<option value=\"" + optName + "\">");
      buffer.append(optName);
      buffer.append("</option>");
    }
  }
```

If a menu item has been previously selected, the `display()` method writes the appropriate <OPTION SELECTED> tag. Then it takes each menu item in turn, checks if it is the selected item, and if not, writes an <OPTION> tag for it.

```
// Else write <OPTION SELECTED> value=selectedOption</OPTION>
// and the rest of the options below that

else
{
  buffer.append("<option selected>" + selectedOption +
    "</option>");

  // For each option in the vector optionNames, check if this
  // option is the selected one. If it is, ignore it since we
  // already wrote the HTML code for the selected option.
  // If it is not the selected one,
  // write <OPTION> value="optionName"</OPTION>
  for(int i=0; i<optionNames.size(); i++)
  {
    optName = (String)optionNames.elementAt(i);
    if(! optName.equals(selectedOption))
    {
      buffer.append("<option value=\""+ optName +"\">");
      buffer.append(optName);
      buffer.append("</option>");
    }
  }
}
// End the Select list
buffer.append("</select>");
break;
}
```

In view mode, the data field is displayed as plain text since it is not editable.

```
case MODE_VIEW:
// In View  mode, display the selected option as a string
// The user cannot change the value in View mode
buffer.append(" "+ selectedOption);
break;
}
```

Finally, the method writes the entire buffer to the HTML page.

```
// Write the contents to the HTML page
html.write(buffer.toString());
}
```

display() at an Entry Point

The `display()` method for the entry point is similar but simpler. It does not have any of the conditional code used in edit mode to check for an existing value, since there can be no existing value. In view mode, the data field displays a warning since there is no good reason to ever use this data field in view mode in an entry point.

See the source code for UpdatableList.java for the definition for the `display()` method at an entry point.

update()

When a form containing the custom data field is submitted, the field's `update()` method is invoked. In this case, `update()` creates an object and sets the value of its `myvalue` variable, then puts the object into the process instance. Later, the `store()` method gets the value of the data field back out of the process instance and saves it to a file as a serialized object. The next time the data field value needs to be retrieved, the `load()` method reads the object from the file and puts it into the process instance.

It may seem like overkill to create and save an object to store a single value, but the purpose of this example is to provide the groundwork for building your own custom data fields. You can use the same paradigm to store objects with multiple values, for example, if the data field needed to store the price and SKU of the chosen computer as well as just the name, it could use an object with three variables instead of one. The mechanism for saving the object to an external file would be exactly the same. The mechanism for storing the object is implemented by the `store()` method which is discussed later.

Code Discussion

The `update()` method parses the form parameters when the HTML form is submitted. An `IPMRequest` object containing all the values of the form elements is sent to the `update()` method. In this example, the `update()` method extracts the value of the form element that has the same name as the data field. (This form element was created by the `display()` method).

Then the `update()` method creates a new instance of `myObject` and sets it `myvalue` variable to the extracted value. Finally it puts the new object into the process instance.

```
public void update(IProcessInstance pi, IPMRequest rq )
  throws Exception
{

try {
  // Get the value of the form element
  String thisValue = rq.getParameter(getName());

  // Create a new myObject to hold the results
  myObject obj1 = new myObject();

  // Put the value into the object
  obj1.myvalue = thisValue;

  // put the object into the pi
  pi.setData(getName(), obj1);
  }
catch (Exception e) {
  System.out.println("Problem translating form values: " + e);
  }
}
```

store()

This data field stores its value externally as a serialized object. (The object is created by the `update()` method.) The job of the `store()` method is to get the data field value out of the process instance and store it in a persistent storage. In this example, the `store()` method saves the value, which is an object, by serializing it to a file using standard object serialization techniques.

Code Discussion

The method generates a unique file name, consisting of the name of the data field plus the process instance ID.

```
public void store(IProcessInstance pi) throws Exception
{
  // Get the data field name
  String thisID = getName();

  // Get the process instance ID
  long procID = pi.getInstanceId();

  // Concatenate the data field name with the PID
  // to keep the name unique across all process instances
  thisID = thisID + procID;
  String thisFileName = thisID + ".txt";

  // Get the application directory
  String appdir = getMyApplicationsPath();

  // Generate the full path to the file
  // where the value will be stored
  String fullPath = appdir + "\\" + thisFileName;
```

When the data field value is needed in the future, the `load()` method retrieves it from the external storage. The `load()` method needs a key to help it find the data. The store method saves the key by calling `setEntityKey()`, and the load method retrieves the key by calling `getEntityKey()`. The `load()` method needs to know which file to access, thus the `store()` method saves the name of the file as the entity key.

```
  // Store the file name as the entity key
  pi.setEntityKey(getName(), thisFileName);
```

Next, the `store()` method gets the value of the data field out of the process instance.

```
// Get the value of the data field from the pi.
// The value is an instance of myObject
myObject myobj = (myObject) pi.getData(getName());
```

Now to the task of storing the value. In this case, `store()` saves the object to a file using standard Java object serialization techniques.

```
 // Write the object to a file
try {
  FileOutputStream fileout = new FileOutputStream(fullPath);
  ObjectOutputStream objout = new ObjectOutputStream(fileout);
  objout.writeObject(myobj);
}
catch (Exception e) {
  System.out.println("Error while saving field value to file:"
  + e);
}
// end store
}
```

load()

When an attempt is made to access the value of the data field when it has not been loaded, the `load()` method is called. This happens, for example, when the data field is being displayed in an HTML form or when an automated activity calls `getData()` to get the value of the data field.

The task of the `load()` method is to retrieve the data field value from external storage and put it in the process instance. In this case, the value is stored as an object in a file.

Code Discussion

The first thing this `load()` method needs to do is to find out which file to access. The name of the file is stored as the entity key, thus `load()` starts off by getting the entity key.

```
public void load(IProcessInstance pi) throws Exception
{
  // Get the name of the file where the value is stored.
  // The file name is saved as the entity key.
  // An example is thisfield123.txt
  String thisFileName = (String) pi.getEntityKey(getName());
```

If the entity key is a file name, the next thing to do is to get generate the full path to the file. The `load()` method uses the user-defined `getMyApplicationsPath()` method to find the path to the directory where the application is stored. The file is in that directory.

```
if (thisFileName != null)
{
  try {
    // getMyApplicationsPath is a user-defined function
    // that returns the path to the dir for the application
    String myPath = getMyApplicationsPath();

    // Get the full path to the file in the Applications dir
    // eg rootdir\Applications\myApp\thisfield123.html
    thisFileName = myPath + "\\" + thisFileName;
```

Now comes the task of loading the value. In this case, the value is an instance of `myObject` that has been serialized to a file. The `load()` method uses standard Java techniques for reading the file and unserializing the object.

```
// Get a file reader and read in the object
FileInputStream filein = new FileInputStream(thisFileName);
ObjectInputStream objectin = new ObjectInputStream(filein);
myObject newobj = (myObject) objectin.readObject();
```

The `load()` method puts the retrieved value into the data field on the process instance, where it is now available for access by all comers (such as the `display()` method).

```
// Put the object in the data field in the process instance
pi.setData(getName(), newobj);
```

Finally, the `load()` method closes the try clause, writes the catch clause, and takes account of the situation where `getEntityKey()` did not return a value.

```
// end try clause
}
catch (Exception e)
{
  System.out.println("Error while reading value from file: "
    + e);
}
// end if (thisFileName != null)
}
else {
  pi.setData(getName(), null);
  }
// end load
}
```

Helper Functions

This class uses several helper methods:

- getMyApplicationsPath()
- fetchDataFromXML()
- parseForItemTag()

getMyApplicationsPath()

This method returns the directory where the current application resides.

```
// Returns the path to the folder where the application is saved
String getMyApplicationsPath ()
{ String path = "";
   try {
     path = getPMApplication().getHomePath();
   }
   catch (Exception e) {
     System.out.println("Exception while getting app path"
      + e);
   }
return path;
}
```

fetchDataFromXML()

This method reads the contents of an XML file that contains a series of items. The method adds each item to a vector and then returns the vector.

To start with, the method creates an empty vector and gets the name of the file to read.

```
// Fetch the set of menu options from the XML file
public Vector fetchDataFromXML()
{
  Vector optionNames = new Vector();

  try {
    // Get the path for the xml file
    // myFileName is a global variable
    // It is the same for all process instances
    String Path = getMyApplicationsPath();
    Path = Path + "\\" + myFileName;
```

Next, the method reads the file into a string.

```
// Get a file reader
java.io.File f = new java.io.File(Path);
FileReader fr = new FileReader(f);
BufferedReader in = new BufferedReader(fr);

// Create variables in preparation for reading the file
int MAX_LENGTH = 2000;
char xml[] = new char[MAX_LENGTH];

// Read the entire xml file into the array xml
int count =0;
count = in.read(xml, count, MAX_LENGTH);

// Create a string of the content and get its length
String charSet = new String(xml);
int charSetLength = charSet.length();
```

Next, the method iterates through every character in the content string contained in the `charSet` variable, looking for items. It uses the `parseForItemTag()` method to find items and add each one to the vector.

Finally the method returns the vector.

```
  count = 0;
  for(; count < charSetLength; count++)
  {
   parseForItemTag(count, charSetLength, charSet, optionNames);
  }
// end try
}
catch(Exception e){
  System.out.println("Error while getting data from xml file: "
    + e);
  }

// return the vector of option names
return optionNames;
}
```

parseForItemTag()

This method iterates over a string, looking for substrings embedded between <ITEM> and </ITEM> tags. Each substring, or item, is added to a vector.

```
/// This method parses an array of characters
// to extract the items embedded in <ITEM>...</ITEM> tags
public void parseForItemTag (int count, int charSetLength,
  String charSet, Vector optionNames)
{
  String temp;
  Object tempobj;
  // Looking for "<" character
  if(charSet.charAt(count) == '<' )
  {
    // Read characters between "<" and ">" into temp string
    temp = "";
    for(; charSet.charAt(count) != '>'; count++)
    {
      temp = temp + charSet.charAt(count);
    }
    temp = temp + charSet.charAt(count);
    count++;
    // Check if the temp string is <ITEM>
    if(temp.equalsIgnoreCase("<ITEM>"))

    // if so, empty out temp and then read
    // the characters between ">" and "<" into temp
    {
    for (temp = ""; charSet.charAt(count) != '<' ;
        temp = temp + charSet.charAt(count++))
    // We now have an item
    // Convert the string temp to an object and
    // add the object to the vector of options
    tempobj = (Object) temp;
    optionNames.addElement(tempobj);
    }
  }
// end of method
}
```

Complete Source Code

For the complete source code, click the following links:

- `UpdatableList.java`
- `UpdatableList.jsb`
- `myObject.java`
- `MenuOptions.xml`

The Custom Activity

This section discusses the custom data field and has the following subsections:

Overview

The advanced office setup application uses a custom activity called `employeeTrainingPerformer` to schedule each new employee to attend a company orientation. The activity writes a web page telling the employee when to attend the orientation. The day that the employee should attend orientation depends on what department they are joining, for example, engineers attend on Mondays while marketing personnel attend on Tuesdays. The custom activity schedules the employee to attend training on the first appropriate day after they start work. Trainings are held at 2 pm so they can go to training on their start date if necessary.

For example, the training day for marketing personnel is Tuesday. So if a marketing in the marketing department starts on Monday, they are scheduled for training the next day. If they start on Tuesday, they are scheduled for training on their first day. If they start on Wednesday, they are scheduled for training the following Tuesday.

The custom activity reads the training schedule from a file called `trainingDays.xml`. An example is:

```
<xml version="1.0" encoding="us-ascii">
  <DEPT>Engineering</DEPT>
  <DAY>monday</DAY>
  <DEPT>Marketing</DEPT>
  <DAY>tuesday</DAY>
  <DEPT>Human Resource</DEPT>
  <DAY>wednesday</DAY>
  <DEPT>Sales</DEPT>
  <DAY>thursday</DAY>
```

This XML file resides in the same folder as the application. The process designer must manually copy this file into the correct place in the Builder directory. When the process designer deploys the application from the Builder, the XML file is automatically copied to the correct location on the server. After the application has been deployed, users can modify the file whenever the training schedule changes. The changes take effect immediately.

To see the entire source code file, click `EmployeeTrainingPerformer.java`.

The Code in Detail

This section discusses the code for the following.

- `EmployeeTrainingPerformer.xml`
- `perform()`
- `Helper Functions`

EmployeeTrainingPerformer.xml

A custom activity receives input data from an xml file that has the same base name as the activity. This xml file defines the elements in an input hashtable that is passed to the custom activity automatically. Typically, the elements in the input hashtable are data field values, but they can be any JavaScript expression. In this case, the input hashtable puts the value of the `dfEmpname` data field value into the `Emp_name` element, the `dfDeptName` data field value into the `Dept` element, and the `dfStartDate` data field value into the `Start_Date` element.

```
<?xml version = "1.0" ?>
<WORKPERFORMER TYPE="com.netscape.pm.model.ISimpleWorkPerformer"
  NAME="EmployeeTrainingPerformer"

CLASS_ID="customer.activities.EmployeeTrainingPerformer"
  VERSION="1.1">
<ENVIRONMENT>
</ENVIRONMENT>

<INPUT>
  <PARAMETER NAME="Emp_Name" DESCRIPTION="Employee Name">
    getData("dfEmpName")
  </PARAMETER>

  <PARAMETER NAME="Dept" DESCRIPTION="Dept. Name">
    getData("dfDeptName")
  </PARAMETER>

  <PARAMETER NAME="Start_Date" DESCRIPTION="Start Date">
    getData("dfStartDate")
  </PARAMETER>
</INPUT>
```

The `EmployeeTrainingPerformer.xml` file also puts the applications path into the element `path` and the process instance ID into the element `id`. The custom activity uses the applications path to identify where the training schedule resides and it uses the process instance id to generate a unique file name for the welcome page.

```
<PARAMETER NAME="path" DESCRIPTION="Applications path">
  getApplicationPath()
</PARAMETER>

<PARAMETER NAME="id" DESCRIPTION="Process Instance ID">
  getProcessInstance().getInstanceId()
</PARAMETER>
```

A custom activity can put elements into an output hashtable. The same xml file that defines the input parameters also defines what happens to the elements in the output hashtable when the custom activity is completed. Typically, values are saved into data fields on the process instance. In this case, the output hashtable contains a filename which is saved into the `dfWelcomePage` data field.

```
<OUTPUT>
   <PARAMETER NAME="welcomePage"
    DESCRIPTION="Greeting for New Employeer">
    mapTo("dfWelcomeURL")
    </PARAMETER>
</OUTPUT>

<EXCEPTIONS></EXCEPTIONS>
<DESIGN></DESIGN>

</WORKPERFORMER>
```

perform()

The `perform()` method of a custom activity executes the activity's task. In this case, it generates an HTML page telling the new employee when to attend training. The method reads the employee's start date, name and department from the input hash table. It also gets the pathname where the application resides, as well as the process ID which is a number that uniquely identifies the process instance.

```
// The perform() method defines what the custom activity does.

public void perform(Hashtable input, Hashtable output)
{

// Get the employee's start date, name and department
// from the input hashtable.
Date startDate = (java.sql.Date)input.get("Start_Date");
String dept = (String) input.get( "Dept" );
String empName = (String) input.get( "Emp_Name" );

// Get the application path from the input hashtable
String appPath = (String)input.get("path");

// Get the process instance id from the input hashtable
int thisID = ((Double)input.get("id")).intValue();
```

Then it calls the `readSchedule()` method to read the `TrainingDays.xml` file, which resides in the applications directory.

```
// Read the schedule from the TrainingDays.xml file
Hashtable trainingDays = readSchedule(appPath);
```

Next, it calls the `scheduleTraining()` method to figure out which day the employee must attend training.

```
// Figure out what day of the week the employee goes to training
Date trainingDate = scheduleTraining(startDate, dept,
  trainingDays);
```

The `perform()` method then calls the `writeWelcomePage()` method to write an HTML page that informs the employee when to attend company orientation training.

```
// Write a welcome page containing the training info
String filename = writeWelcomePage(empName, thisID, appPath,
  trainingDate);
```

Finally, `perform()` puts the filename for the welcome page into the output hash table so that it can be saved into a data field on the process instance.

```
// Put the file name for the HTML page in the output hashtable
output.put("welcomePage", filename);
}
```

Helper Functions

The `perform()` method uses the following helper methods:

- `readSchedule()`
- `scheduleTraining()`

  which uses the helper method

  - `IncrementForDayOfWeek()`
- `writeWelcomePage()`

readSchedule()

This function reads the training schedule and returns a hashtable containing key:value pairs of dept:day. The method creates a hashtable to hold the results. It gets the full path to the file containing the training schedule.

```
 private Hashtable readSchedule(String appPath)
{

Hashtable trainingDays = new Hashtable();

try{
  // Get the full name for the training schedule
  String Path = appPath + "\\" + "TrainingDays.xml";
```

Then it creates a file reader.

```
// Create a file reader
java.io.File f = new java.io.File(Path);
FileReader fr = new FileReader(f);
BufferedReader in = new BufferedReader(fr);
```

It reads the entire contents of the file into the String variable `charSet`.

```
 // Read the entire file into the String "charSet"
int MAX_LENGTH = 500;
char xml[] = new char[MAX_LENGTH];
int count = 0;
count = in.read(xml, count, MAX_LENGTH);
String charSet = new String(xml);
```

It gets some variables ready for parsing the file.

```
// Create variables in preparation for parsing the String "xml"
int charSetLength = charSet.length();
String temp = new String();
String dept = new String();
String day = new String();
count = 0;
```

The method start parsing the string. First it looks for the substring <DEPT>. When it's found that, it reads the substring between the end of <DEPT> and the starting "<" in </DAY>. It stores this substring in the variable dept.

```
for(; count < charSetLength; count++)
{
  if(charSet.charAt(count) == '<' )
  {
    temp = "";
    for(; charSet.charAt(count) != '>'; count++)
    { temp = temp + charSet.charAt(count);
    }
    temp = temp + charSet.charAt(count);
    count++;

    // When temp = <DEPT>, find the name of the department
    if(temp.equalsIgnoreCase("<DEPT>"))
    {
      for(dept = ""; charSet.charAt(count) != '<' ;
        dept = dept + charSet.charAt(count++));
    }
```

The parser has just reached the end of </DEPT> in the string. Now it looks for <DAY>, then reads the substring between the end of <DAY> and the starting "<" in </DAY> and stores the substring in the variable day. It puts an element representing the dept and day into the trainingDays hashtable.

```
  // Now we know the current DEPT. Find the day.
  if(temp.equalsIgnoreCase("<DAY>"))
  {
    for(day = ""; charSet.charAt(count) != '<' ;
      day = day + charSet.charAt(count++));

    trainingDays.put(dept, day);
  }
  }
 }
// end of function
}
```

scheduleTraining()

This method figures out what date the new employee is to attend orientation training. To do this, it figures out what day of the week the employee starts work, finds out what department the employee is joining, looks up the training day for that department in the trainingDays hashtable, and then calculates the date for the training.

First, the method gets the day of week, day of month, month and year that the employee is starting work.

```
public Date  scheduleTraining (String startDate, String dept,
  Hashtable trainingDays)
{
  // Get info about the start date
  date = new Date(startDate);
  int thisDay = date.getDay();
  int dayOfMonth = date.getDate();
  int month = date.getMonth();
  int year = date.getYear();
```

Then the method looks in the `trainingDays` hashtable to see what day of the week the employee goes to training, which depends on what department they are joining. The `scheduleTraining()` method then calls a helper function to find the date of the first appropriate day of the week on or after the start date. For example, if the employee should attend training on Monday, the helper function returns the start date if it is a Monday or else finds the date of the first Monday following the start date.

```
// Using the dept as the key, get the value of the
// training day from the the trainingDays hashtable
if(((String)trainingDays.get(dept)).equals("monday") ){
  dayOfMonth=IncrementForMonday(thisDay, dayOfMonth);
  }
else if(((String)trainingDays.get(dept)).equals("tuesday") ){
  dayOfMonth=IncrementForTuesday(thisDay, dayOfMonth);
  }
else if(((String)trainingDays.get(dept)).equals("wednesday") ){
  dayOfMonth=IncrementForWednesday(thisDay, dayOfMonth);
  }
else if(((String)trainingDays.get(dept)).equals("thursday") ){
  dayOfMonth=IncrementForThursday(thisDay, dayOfMonth);
  }
else if(((String)trainingDays.get(dept)).equals("friday") ){
  dayOfMonth=IncrementForFriday(thisDay, dayOfMonth);
  }
if(((String)trainingDays.get(dept)).equals("saturday") ){
  dayOfMonth=IncrementForSaturday(thisDay, dayOfMonth);
  }
else if(((String)trainingDays.get(dept)).equals("sunday") ){
  dayOfMonth=IncrementForSunday(thisDay, dayOfMonth);
}
Date trainingDate = new Date(year, month, dayOfMonth);
return trainingDate;
}
```

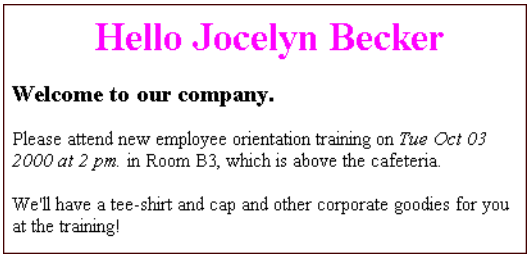
IncrementForDayOfWeek()

The `scheduleTraining()` method uses helper functions to find the first appropriate day of the week on or after the start date. All these methods have the same basic structure. Here is an example for `IncrementForMonday()`, which takes a week day and a day of the month. It returns the day of the month unchanged if the given week day is Monday, else returns the day of the month for the first Monday following the day of the month that was passed in.

```
// helper functions to find training date
private int IncrementForMonday(int thisDay, int dayOfMonth ){
if(thisDay == SUN)
  // for Monday increment 1 from Sunday
  dayOfMonth = dayOfMonth+1;
if(thisDay == TUE)
  // for Monday increment 6 from Tue
  dayOfMonth = dayOfMonth+6;
if(thisDay == WED)
  // for Monday increment 5 from Wed
  dayOfMonth = dayOfMonth+5;
if(thisDay == THU)
  // for Monday increment 4 from Thursday
  dayOfMonth = dayOfMonth+4;
if(thisDay == FRI)
  // for Monday increment 3 from Friday
  dayOfMonth = dayOfMonth+3;
if(thisDay == SAT)
  // for Monday increment 2 from Saturday
dayOfMonth = dayOfMonth+2;

return dayOfMonth;
}
```

writeWelcomePage()

This method generates an HTML page that informs the employee when to attend training, for example:



The method starts by formatting the Date string to make it more readable and generating a unique pathname for a new file in the applications folder.

NOTE For the code for `formatDateString()`, see the source code for `EmployeeTrainingPerformer.java`.

```
public String writeWelcomePage(String employeeName, int thisID,
  String appPath, Date trainingDate)
{
  // Format the date string to remove "00:00:00 PDT/PST"
  String finalDate = formatDateString(trainingDate);

  // File name is Employee name + ProcessInstance
  String fileName = employeeName + thisID + ".html";

  // Remove all white spaces from the filename
  // URLS cannot have white spaces
  fileName = fileName.replace(' ', '_');

  // Get the pathname to the file in the Application's folder
  String thisPath = appPath + fileName;
```

Then it creates a file with the name it has just derived, and writes a welcome message for the new employee into the file. This message includes the date of the employee's orientation training.

```
// Make a file in this Application's folder
try {
  RandomAccessFile HTMLfile = new RandomAccessFile (
    thisPath, "rw");

  HTMLfile.writeUTF("<HTML><HEAD>");
  HTMLfile.writeUTF("<TITLE>New Employee Training</TITLE>");
  HTMLfile.writeUTF("</HEAD>");
  HTMLfile.writeUTF("<BODY>");
  HTMLfile.writeUTF("<CENTER>");
  HTMLfile.writeUTF(<H1><FONT COLOR=MAGENTA>Hello <I> ");
  HTMLfile.writeUTF(employeeName);
  HTMLfile.writeUTF("</I></FONT></H1></CENTER>");
  HTMLfile.writeUTF("<H3>Welcome to our company. </H3>");
  HTMLfile.writeUTF("<P> Please attend new employee ");
  HTMLfile.writeUTF("orientation training on ");
  HTMLfile.writeUTF("<I>" + finalDate + " at 2 pm.</I>");
  HTMLfile.writeUTF("in Room B3 above the cafeteria.</P>");
  HTMLfile.writeUTF("<P>We'll have a tee-shirt and cap and ");
  HTMLfile.writeUTF("other corporate goodies for you");
  HTMLfile.writeUTF("at the training!</P>");
  HTMLfile.writeUTF("</BODY>");
  HTMLfile.writeUTF("</HTML>");
  HTMLfile.close();
}
```

Complete Source Code

For the complete source code, click the following links:

- EmployeeTrainingPerformer.java
- EmployeeTrainingPerformer.xml
- TrainingDays.xml

Chapter 4

Cluster Management

This chapter discusses the Java classes used for performing cluster administration tasks. This chapter has the following sections:

NOTE You can find all the necessary classes in the `pm65classes.jar` file. If you have installed the Process Manager Builder, you can find this jar file in the directory *builder-root*`\support\sdk`. You may also be able to find it on the CD.

Introduction

A Process Manager cluster contains the following components:

- a corporate user LDAP directory service
- a configuration LDAP directory service that stores the application definitions
- a relational database for user data
- one or more application servers
- an iPlanet Web Server
- a mail server for notifications

When deploying an application from Process Builder, application developers must identify the cluster on which to deploy it. All applications in a cluster share the same common database and directories. They access the same Directory Server for their process definitions and they use the same set of cross-application tables in the database, as well as the same corporate users and groups directory.

Using the Process Administrator interface, you can interactively modify a cluster.

You can build Java applications that programmatically perform Process Manager cluster administration tasks. Process Manager has one cluster manager bean, IPMClusterManager. For each separate cluster, there is an IPMCluster bean.

You can use the cluster manager bean to perform administrative tasks such as creating, deleting, joining and unjoining clusters. You can also use the cluster manager bean to get handles to individual clusters beans.

You can use the cluster bean to change properties of the cluster, to access and write to cluster logs, and to retrieve information about the cluster such as getting installed applications, getting path information, getting the finder and getting the deployment manager.

Given an individual cluster bean, you can get individual application beans. Given an application bean, you can access any process instance or work item so long as you have the key, such as the process instance ID.

For details of IPMClusterManager and IPMCluster beans see:

- IPMClusterManager
- IPMCluster

Clusters have properties. These are represented as IPMClusterProperty objects. Many of the methods for working with clusters take IPMClusterProperty objects as arguments. To create cluster property objects, you use IPMClusterPropertyFactory. For details see:

- IPMClusterProperty
- PMClusterPropertyFactory

At the end of this chapter, there is a coded example of creating a cluster.

- "Code Samples"

IPMClusterManager

The IPMClusterManager bean can be thought as a manager of all the cluster beans in the application server instance.

The following code shows how to access the cluster manager:

```
// Get the cluster manager
IPMClusterManager myClusterManager = null;

try {
   javax.naming.Context cxt = new javax.naming.InitialContext();
   IPMClusterManagerHome clManagerHome = (IPMClusterManagerHome)
      cxt.lookup(IPMClusterManager.JNDI_ROOT);
   myClusterManager = clManagerHome.create();
   }

catch( Exception e )
   {
   System.out.println("Cluster manager creation failed" + e);
   }
```

The IPMClusterManager bean is a stateless session bean that has methods for creating, joining, accessing, unjoining and deleting clusters, and checking LDAP and database connections.

IPMCluster

IPMCluster objects represent individual clusters. For operations such as creating, deleting, joining and unjoining clusters, you call methods on the cluster manager class, IPMClusterManager, rather than calling methods on the cluster itself.

To get access to an IPMCluster method, call the `getCluster()` method on the IPMClusterManager as illustrated in the following code sample:

```
// Get the cluster manager
IPMClusterManager myClusterManager = null;

try {
   javax.naming.Context cxt = new javax.naming.InitialContext();
   IPMClusterManagerHome clManagerHome = (IPMClusterManagerHome)
      cxt.lookup(IPMClusterManager.JNDI_ROOT);
   myClusterManager = clManagerHome.create();
   }

catch( Exception e )
   {
   System.out.println("Cluster manager creation failed" + e);
   }

// Get the default cluster in a single cluster implementation
IPMCluster myCluster = myClusterManager.getCluster(
   IPMCluster.DEFAULT);

//Get a specific cluster in a multiple cluster implementation
IPMCluster myCluster = myClusterManager.getCluster(
   IPMCluster.NAME_OF_CLUSTER);
```

The IPMCluster bean has methods for changing cluster properties, accessing logs and retrieving information about the cluster.

For details for the IPMCluster bean methods, see the javadocs, which can be found in the support\sdk\docs directory of the Process Manager Builder installation.

IPMClusterProperty

The IPMClusterProperty interface is used in various cluster bean APIs. It is used to set and get cluster properties.

When creating an IPMCluster, you first need to create an IPMClusterProperty and then use the `setValue()` method to set the cluster properties, as listed in the section "Setting Properties When Creating Clusters" on page 139. The section, "Other Cluster Properties" on page 142, lists cluster properties that specify URLs for accessing various Process Manager components as well as other cluster properties and pre-defined variables.

Getting and Setting Property Values

To retrieve or set properties, use the methods:

- getValue
- setValue
- getProperties

For information on the property value settings, see the javadocs, which can be found in the support\sdk\docs directory of the Process Manager Builder installation.

Setting Properties When Creating Clusters

This section describes how to set properties on creation of a cluster, and then lists the properties that must be set.

To create an IPMClusterProperty, use the `create()` method on IPMClusterPropertyFactory as follows:

```
IPMClusterProperty prop1 = PMClusterPropertyFactory.create ( );
```

To get the existing IPMClusterProperty for a cluster, call `getClusterProperty()` on the cluster, as follows:

```
IPMClusterProperty myProp = myCluster.getClusterProperty();
```

NOTE All properties listed in the following tables are `public static int`

Table 4-1 Cluster and Configuration Properties

Property	Description
`CLUSTER_DN`	Distinguished name (DN) of the cluster entry
`CLUSTER_NAME`	Name of a cluster (when using a multiple cluster implementation).
`CONFIGURATION_DIRECTORY_SERVER`	Host name where the configuration directory resides
`CONFIGURATION_DIRECTORY_PORT`	Port number for the configuration directory
`CONFIGURATION_DIRECTORY_BIND_ DN`	Bind DN of the configuration directory
`CONFIGURATION_DIRECTORY_BIND_ DN_ PASSWORD`	Bind DN password of the configuration directory

Table 4-2 Corporate Directory Properties

Property	Description
`CORPORATE_DIRECTORY_SERVER`	Host name where the corporate directory resides
`CORPORATE_DIRECTORY_PORT`	Corporate directory port number
`CORPORATE_DIRECTORY_BASE`	Corporate directory base, for example: `ou=People, o=iplanet.com`
`CORPORATE_DIRECTORY_BIND_DN`	Corporate directory BIND DN, for example: `cn=Directory Manager` This property is optional when creating a cluster.

Table 4-2 Corporate Directory Properties *(Continued)*

Property	Description
CORPORATE_DIRECTORY_BIND_DN_PASSWORD	Corporate directory BIND DN Password. This property is optional when creating a cluster.

Table 4-3 Database Properties

Property	Description
DATABASE_DRIVER_IDENTIFIER	Driver name used to register a third-party jdbc driver
DATABASE_IDENTIFIER	Database identifier when a native database driver is used
DATABASE_NAME	Database name (optional if the database type is ORACLE)
DATABASE_PASSWORD	Database user password
DATABASE_TYPE	The Database Type. The value can be: IPMClusterProperty.INFORMIX IPMClusterProperty.ORACLE IPMClusterProperty.SYBASE
DATABASE_URL	URL that specifies the subprotocol (the database connectivity mechanism), the database and list of properties
DATABASE_USER_NAME	the database user name
DATASOURCE_NAME	JNDI Name used for the database when you registered your datasource with the iPlanet Application Server

Table 4-4 General Properties

Property	Description
ADMIN_USERS	Comma-separated list of LDAP user IDs of people who have administrative permission to access Process Administrator and Process Business Manager interfaces for this particular cluster, and to deploy applications to this cluster. You must provide at least one LDAP user ID in this field. This property is needed only when using a multiple cluster implementation.
DESCRIPTION	Description of the cluster
PRETTY_NAME	Pretty name of the cluster that appears in the Builder. This property is optional when creating a cluster.
SMTP_SERVER	SMTP server host name. This property is optional when creating a cluster.
SMTP_PORT	SMTP port number . This property is optional when creating a cluster.
SMTP_REPLY_TO	SMTP reply to. This property is optional when creating a cluster.

Other Cluster Properties

This section lists cluster properties that specify URLs for accessing various Process Manager components and cluster properties that specify information about events. It also lists other pre-defined variables.

NOTE All properties listed here are `public static int`

Table 4-5 URL Properties

Property	Description
DEPLOY_URL	URL where Process Manager applications are deployed.
EXPRESS_URL	URL for the Process Manager Express
ADMINISTRATOR_URL	URL for the Process Administration interface

Table 4-5 URL Properties *(Continued)*

Property	Description
BUSINESS_URL	URL for the Process Business Manager's interface
APPLICATION_URL	URL for application

Table 4-6 Event Properties

Property	Description
EVENT_USER	The cluster uses the EVENT_USER ID when it makes asynchronous requests into the Process Manager Engine, such as when the timer agent checks for expired work items. This user and password combination should be a valid combination inside the corporate directory.
EVENT_USER_PASSWORD	Password for the EVENT_USER.

Table 4-7 Other Pre-defined Variables

Property	Description
INFORMIX ORACLE SYBASE	These are used as the values of the DATABASE_TYPE variable.
LOG_INFORMATION LOG_SECURITY LOG_ERROR	These are used as an argument to the getLog() method on IPMCluster to specify what kind of log to retrieve.

PMClusterPropertyFactory

The purpose of this class is to create new IPMClusterProperty objects, which contain the properties for a cluster.

```
// Create a PMClusterProperty object
IPMClusterProperty prop1 = PMClusterPropertyFactory.create ( ) ;
```

```
 IPMClusterProperty prop = PMClusterPropertyFactory.create ( );
// Set the various properties of the interface
....
....
...
// look up in JNDI name space to get a handle on the
// IPMClusterManager bean interface
...
....
// maybe then create a cluster
IPMClusterManager.createCluster ( prop ) ;
```

Code Samples

This section presents the following code samples:

- "Mount the Cluster Manager and Get the Default Cluster"
- "Create a Cluster"
- "Get and Set Cluster Properties"

Mount the Cluster Manager and Get the Default Cluster

This code sample gets the cluster manager and the default cluster.

```
// Get the cluster manager
IPMClusterManager myClusterManager = null;

try {
   javax.naming.Context cxt = new javax.naming.InitialContext();
   IPMClusterManagerHome clManagerHome = (IPMClusterManagerHome)
      cxt.lookup(IPMClusterManager.JNDI_ROOT);
   myClusterManager = clManagerHome.create();
   }

catch( Exception e )
   {
   System.out.println("Cluster manager creation failed" + e);
   }

// Get the default cluster in a single cluster implementation
IPMCluster myCluster = myClusterManager.getCluster(
   IPMCluster.DEFAULT);

// Get a specific cluster in a multiple cluster implementation
IPMCluster myCluster = myClusterManager.getCluster(
   IPMCluster.NAME_OF_CLUSTER);
```

Create a Cluster

This code sample creates a new cluster.

```
// Create a PMClusterProperty object
IPMClusterProperty prop1 = PMClusterPropertyFactory.create ( ) ;

// Populate the properties in the PMClusterProperty object
prop1.setValue(IPMClusterProperty.CONFIGURATION_DIRECTORY_SERVER
, "westminster");
prop1.setValue(IPMClusterProperty.CONFIGURATION_DIRECTORY_PORT,
"4141");

// Continue setting properties
...

// Now create the cluster
IPMCluster cluster1 = myClusterManager.createCluster( prop1 );
```

Get and Set Cluster Properties

This code sample gets and sets cluster properties.

```
// myCluster is the default cluster

// Get some properties
IPMClusterProperty myProp = myCluster.getClusterProperty();
String description =
myProp.getValue(IPMClusterProperty.DESCRIPTION);

// Change some property values
IPMClusterProperty myProp = myCluster.getClusterProperty();
myProp.setValue(IPMClusterProperty.DESCRIPTION,
   "New cluster description");
myProp.setValue(IPMClusterProperty.SMTP_REPLY_TO ,
   "pmadministrator@iplanet.com");

// Save the changes to the cluster
myProp.changeCluster(myprop);
```

Chapter 5

Deployment Manager

The deployment manager is responsible for installing and removing applications from the cluster. The deployment manager is also responsible for changing the deployment state of a deployed application by changing its deployment descriptor when applicable.

Process Manager clients can access the deployment manager by calling the getDeploymentManager() method on the cluster bean. Given the deployment manager, you can access the deployment descriptor for individual applications. Given a deployment descriptor, you can change the application's stage, status, mode and testing state. For explanations of these states, see the section "Deployment States." (Note however that you cannot programmatically install applications, they must be deployed from the Process Builder.)

This chapter has the following sections:

NOTE You can find all the necessary classes in the `pm65classes.jar` file in the `api` directory on the Process Manager CD.

Deployment States

The deployment descriptor for a deployed applications specifies the following states for the application:

- STAGE
- MODE
- STATUS
- TESTING

STAGE

The application can be either in DEVELOPMENT stage or in PRODUCTION stage.

- DEVELOPMENT stage -- the application can be completely rewritten from the Process Builder.
- PRODUCTION stage -- only limited changes can be made from the Process Builder.

The stage can be changed programmatically. In the interface, the change from DEVELOPMENT to PRODUCTION can only done from the Process Builder and NOT from the Administrator UI.

NOTE An application in the production stage can be in the testing state which means that even though the changes that can be made from the Process Builder are restrictive all the work items can still be assigned to the creator of the process instance. This kind of a scenario is useful when an application has been deployed to production but a final pass is being made to make sure that everything is working.

MODE

The mode of the application can be either OPEN or CLOSED

- OPEN mode -- new process instances can be created.
- CLOSED mode -- no new process instances can be created but old process instances can continue through the system to completion.

The mode can be changed programmatically. In the interface, administrators can change the MODE from OPEN to CLOSED and back to OPEN from the Administrator UI.

STATUS

The application STATUS can be either STARTED or STOPPED.

- STARTED status -- the application can be accessed from the Express UI.
- STOPPED status -- the application cannot be accessed through the Express UI.

The stage can be changed programmatically. In the interface, administrators can change the STATUS from STARTED to STOPPED and back to STARTED from the Administrator UI.

TESTING

The TESTING state can be either TRUE or FALSE.

- TRUE -- all work items are automatically assigned to the creator of the process instance.
- FALSE -- the work items are assigned to the real user.

The stage can be changed programmatically. In the interface, administrators can change the TESTING from TRUE to FALSE and back to FALSE from the Administrator UI.

NOTE An application in the production stage can be in the testing state which means that even though the changes that can be made from the Process Builder are restrictive all the wartimes can still be assigned to the creator of the process instance. This kind of a scenario is useful when an application has been deployed to production but a final pass is being made to make sure that everything is working.

IDeploymentManager Interface

Use the IDeploymentManager interface to get access to the deployment descriptors for installed applications and to remove an application. Given the deployment descriptor for an application, you can change deployment details, such as changing its stage, status, mode, and testing state.

To get the IDeploymentManager for a cluster, call the `getDeploymentManager()` method on the relevant IPMCluster object. For an example of accessing the deployment manager, see the code sample in the section "IDeploymentDescriptor Interface."

Where are the Classes and Interfaces?

The cluster manager and all classes related to the cluster are in the `com.netscape.pm.model` package.

The deployment manager and deployment descriptor classes are in the `com.netscape.pm.dm` package.

All the classes are in the `pm65classes.jar` file. If you have installed the Process Manager Builder, you can find this jar file in the directory *builder-root*`\support\sdk`. You may also be able to find it on the CD.

IDeploymentDescriptor Interface

Given the deployment manager, clients can call `getInstalledApplications()` to get a hashtable of deployment descriptors for all installed applications. The hashtable is keyed by application name. You can then get access the hashtable to retrieve the deployment descriptor for an individual application.

Given the deployment descriptor, you can call methods to find an application's stage, mode and status.

Code Example

The following code example illustrates how to get the deployment descriptor for an application named `myApp`.

```
// Get the cluster manager
IPMClusterManager myClusterManager = null;
try {
     javax.naming.Context cxt = new
javax.naming.InitialContext();
   IPMClusterManagerHome clManagerHome = (IPMClusterManagerHome)
      cxt.lookup(IPMClusterManager.JNDI_ROOT);
   myClusterManager = clManagerHome.create();
  }
catch( Exception e )
  {
   System.out.println("Cluster manager creation failed" + e);
  }

// Get the default cluster
IPMCluster myCluster = myClusterManager.getCluster(
  IPMCluster.DEFAULT);

// Get the deployment manager
IDeploymentManager myDepManager =
myCluster.getDeploymentManager();

// Get a hastable of all installed apps
Hashtable appList = myDepManager.getInstalledApplications ( );

// Get the deployment descriptor for myapp
IDeploymentDescriptor myAppDD = appList.get ("myapp");
```

Each application has a deployment descriptor that contains the current settings for the mode, stage, status and testing state of the application. Clients can access the deployment descriptor of a particular application to manipulate these individual parameters.

For more details, see the javadocs, which can be found in the `support\sdk\docs` directory of the Process Manager Builder installation.

Chapter 6

Working with Applications, Process Instances and Work Items

The Process Manager Application API provides classes and methods that let you find and work with applications, process instances and work items. You would use this API to build Java applications that embed the functionality of the Process Manager engine.

For example, you can create back-end systems that create process instances programmatically rather than through the UI.. After creating a process instance, the back-end system can check its status and interact with work items.

You can also use the Process Manager Application API to write your own front-end user-interface to the Process Manager engine so that your users use the new interface rather than using the Process Express. For example, suppose you want to have a batch delegate UI. The inbuilt delegate feature only allows you to delegate one work item. You could write a web-based UI that allows the user to select multiple work items. Then in the back-end, you would call `wi.delegate()` repeatedly for each work item.

In every Process Manager installation, a cluster manager bean manages the Process Manager clusters. Given the cluster manager bean, you can get access to individual cluster beans. Given an individual cluster bean, you can access the application bean for the cluster. Given the application bean, you can find and work with individual process instances and work items in the application.

See Chapter 4, "Cluster Management," for information about accessing the cluster manager and individual cluster beans.

The Application API consists of the following interfaces:

- IPMApplication -- has methods for finding work items and process instances, and for testing the state of an application.
- IProcessInstance -- has methods for working with individual process instances, such as getting and setting data field values, getting information such as the creation date and creator, changing the state, and suspending, terminating or resuming the process instance.
- IWorkItem -- has methods for working with individual work items, such as changing assignees, expiring or extending the expiration date, finding which node (activity) it is at,and suspending or resuming it.
- IFinder -- has methods for finding process instances and work items in the application.

You can find all the necessary classes in the `pm65classes.jar` file. If you have installed the Process Manager Builder, you can find this jar file in the directory *builder-root*`\support\sdk`. You may also be able to find it on the CD.

IPMApplication

The application bean is the main access point for process instances. It has methods that allow the user to initiate and edit process instances and work items associated with the application. The IPMApplication bean is best thought of as a factory for PIs and WIs.

It also has methods that let you get the application's stage, mode, status and testing state. See the section "Deployment States" on page 148 for more information about these settings.

To remove an application or to change its state, use the IDeploymentDescriptor interface, as discussed in Chapter 5, "Deployment Manager."

To get a handle to a specific application, mount the application bean. For example:

```
String jndiName = IPMApplication.DEFAULT_JNDI_ROOT + "/" +
appName;

try
{
  javax.naming.Context cxt = javax.naming.InitialContext ( ) ;
  IPMApplicationHome home = (IPMApplicationHome)
      cxt.lookup( jndiName );
  IPMApplication myApp = home.create();
}
catch( Exception e )
{
}
```

All the methods on IPMApplication throw a `PMException` in case of error. The methods `findPI` and `findWI` also throw `XProcessInstanceNotFound` and `XWorkItemNotFound` respectively in the event the object cannot be located.

For details of the methods on IPMApplication, consult the javadocs, which are in the `support\sdk\docs` directory of the Process Manager Builder installation directory.

IProcessInstance

The IProcessInstance Interface has methods for getting information about process instances and for performing operations on them such as suspending and resuming them or setting their data field values.

Given an IPMCluster bean (see Chapter 4, “Cluster Management”) you can get the Finder bean for the cluster. Given the Finder bean you can call the `findMyInstances()` method to find all process instances in the cluster.

Given an application bean, you can find a specific process instance by calling its `findProcessInstance()` method and specifying the process instance’s key.

For example:

```
// myApp is a mounted application bean
IProcessInstancePK pk = ProcessInstancePKFactory.create( PID);
pk.setEditable( true );    // for update
IProcessInstance pi = myApp.findProcessInstance( pk );
```

For details of the methods on IProcessInstance, consult the javadocs, which are in the `support\sdk\docs` directory of the Process Manager Builder installation directory.

IWorkItem

The IWorkItem Interface has methods for getting information about work items and for performing operations on them such as changing their assignees, expiring them or extending the expiration date, suspendig them or resuming them and so on.

Given an IPMCluster bean (see Chapter 4, "Cluster Management") you can get the Finder bean for the cluster. Given the Finder bean you can call the `findWorkItems()` method to find all work items in the cluster. If you know the key for a specific process instance, you can use the Finder bean to find all the work items in a specific process instance.

Given a cluster bean, you can also get an application bean for a specific application. Given the application bean, you can get a specific work item by calling its `findWorkItem()` method and specifying the work item's key.

For details of the methods on IWorkItem, consult the javadocs, which are in the `support\sdk\docs` directory of the Process Manager Builder installation directory.

IFinder

The Ifinder bean is the access point for getting lists of work items and process instances.

In every Process Manager installation, a cluster manager bean manages the Process Manager clusters. Given the cluster manager bean, you can get access to individual cluster beans. Given an individual cluster bean, you can get the finder bean for the cluster by calling the `getFinder()` method.

The finder has methods for getting worklists, which are lists of work items. The worklist queries only return IWorkItems currently assigned to the principal in read-only mode; the principal cannot retrieve an IWorkItem that is not assigned to them via this interface.

The finder has methods for getting process instances. The process instance search methods only return IProcessInstances that were created by the principal in read-only mode; the principal cannot gain access to a process instance that they did not initiate.

For details of the IFinder methods, see the javadocs, which can be found in the `support\sdk\docs` directory of the Process Manager Builder installation.

Chapter 7

Web Services

This chapter describes how you can create client applications that use web services to communicate with iPlanet Process Manager applications. It provides an overview of the Simple Object Access Protocol (SOAP), describes the type of web services functionality provided by Process Manager, and shows how to create SOAP client applications.

Overview of SOAP and WSDL

iPlanet Process Manager provides web services using the Apache implementation of the Simple Object Access Protocol (SOAP) specification. SOAP is an XML-based protocol for the exchange of information in a decentralized, distributed environment. Refer to the *iPlanet Process Manager Release Notes* for the version of Apache SOAP that is supported for this release.

SOAP messages are XML documents that are typically exchanged over the HTTP transport. SOAP messages consist of a SOAP envelope (which defines the framework for the contents of a message), SOAP encoding rules (which define how instances of objects are serialized and marshalled between applications), and the SOAP RPC representation (which defines the conventions for remote procedure calls and responses). Process Manager handles the details of creating SOAP messages for you, including specifying the SOAP encoding rules and building the SOAP remote procedure calls and responses.

The Web Services Description Language (WSDL) is a specification that uses XML documents to describe web services. A WSDL file encodes network service information such as ports, messages, operations, and binding. You typically use a WSDL file as input to third-party utilities to generate stub code, which can then be used to create SOAP client applications.

For more information about the SOAP protocol, refer to http://www.w3.org/TR/SOAP. For information on the version of SOAP implemented by iPlanet Process Manager, refer to the iAS EPE Release Notes, available at http://docs.sun.com/prod/s1.ipaspro/. For more information on the WSDL specification, refer to http://www.w3.org/TR/wsdl.

Terminology

Compound Data Type In SOAP encoding style, a compound type is a composite type, with each part being either a SOAP basic type or another compound type ultimately derived from SOAP basic types. SOAP uses compound data types to serialize objects passed in SOAP messages.

Simple Data Type In SOAP encoding style, a simple type is scalar data type.

SOAP The Simple Object Access Protocol specifies a way to exchange information in a decentralized, distributed environment. SOAP encodes messages as XML documents containing three parts: an envelope providing information about the message and how to process it, a set of encoding rules that defines objects included in the message, and a convention for representing remote procedure calls and responses. SOAP messages are typically sent using the HTTP/HTTPS protocols. iPlanet Process Manager handles the creation, sending, and decoding of SOAP messages for iPM applications. For more information on SOAP, refer to http://www.w3.org/TR/SOAP.

Skeleton Code generated when compiling a WSDL file. Skeleton code can be used to implement a server application that exports web services. WSDL compilers are available from various vendors and for various programming languages.

SOAP Fault A SOAP Fault is an element of a SOAP message that contains error and/or status information. If present, the SOAP Fault appears within the body of a SOAP message.

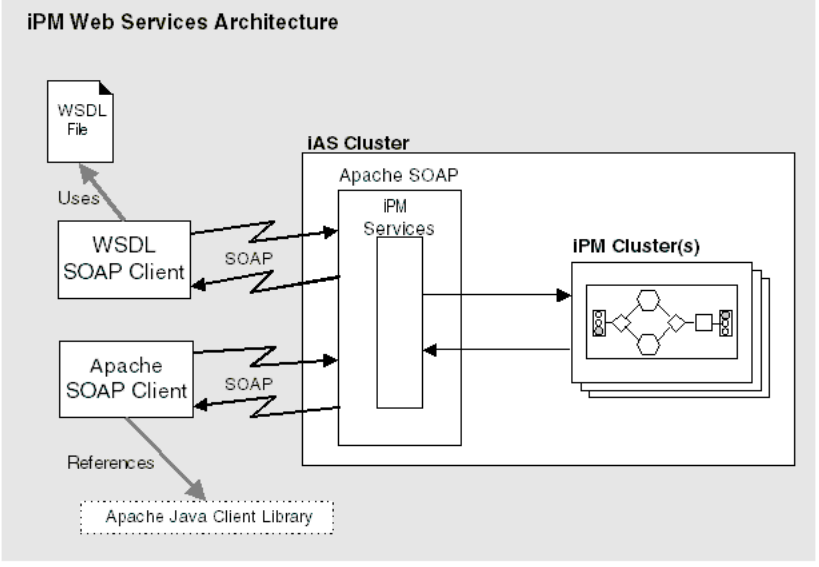
Stub Code generated from compiling a WSDL file. Stub code can be used to implement a client application that accesses web services made available from a server application. WSDL compilers are available from various vendors and for various programming languages.

WSDL The Web Services Description Language is a specification that uses XML documents to describe Web services. A WSDL file typically encodes network service information such as ports, messages, operations, and binding. iPlanet Process Manager provides a WSDL file that you can use as a starting point for creating your own SOAP clients to Process Manager. For more information on the WSDL specification, refer to http://www.w3.org/TR/wsdl.

iPM Web Services Architecture

Figure 7-1 illustrates how SOAP client applications access a Process Manager application.

Figure 7-1 iPM Web Services Architecture



Process Manager exposes web services that allows SOAP clients to perform operations on a process in an iPM cluster. SOAP clients call into the web services using Apache SOAP.

An *Apache SOAP client* is a Java application that uses Apache SOAP services to call into the Web Services API. A *WSDL SOAP client* is a client application created from stubs generated when compiling the iPM WSDL file provided with your Process Manager distribution. The following section, "Web Services Environment," describes the configuration files provided with your distribution to set up Process Manager for web services.

Web Services Environment

This section describes how to set up Process Manager to implement web services. It first discusses the configuration files provided with your distribution. It then describes how to configure your environment to use iPM Web Services.

Web Services Configuration Files

Table 7-1 lists configuration files that are provided with your Process Manager distribution. The following section, "Environment Setup," describes how these files are used when setting up your web services environment. In this table, `<IPM_ROOT>` indicates the root of your Process Manager installation.

Table 7-1 iPM Web Services Configuration Files

File	Location (relative to <IPM_ROOT>)	Description
`DeploymentDescriptor.xml`	`bpm/soap/apache-2_2`	Used by Process Manager to export web services for iPM applications.
`PMService-service.wsdl`	`bpm/soap/wsdl/`	Used to generate stubs for WSDL clients. You must customize this file to indicate the host name and port for your installation.
`PMService.wsdl`	`bpm/soap/wsdl/`	Used when generating stubs for WSDL clients. This file specifies the iPM operations available as web services.
`PMService.xsd`	`bpm/soap/wsdl/`	Used when generating stubs for WSDL clients. This file specifies the iPM data types available for web services.
`ipmsoap-2_2.jar`	`bpm/soap/`	Process Manager supplied patch to Apache SOAP.

Table 7-1 iPM Web Services Configuration Files *(Continued)*

File	Location (relative to <IPM_ROOT>)	Description
pm65soap.jar	bpm/soap/	Used by Apache SOAP clients to access web services classes exported by Process Manager.

Environment Setup

Configuring your environment to use Process Manager Web Services consists of the following steps:

- Setting up Process Manager to use web services
- Setting up the development environment for Apache SOAP client applications
- Setting up the runtime environment for Apache SOAP client applications
- Editing the provided WSDL file for WSDL client applications

Setting Up iPM to use Web Services

The following procedure describes how to set up Process Manager to use web services.

➤ **To Set Up Process Manager to Use Web Services:**

1. Create a default iPM cluster.
2. Unjar `ipmsoap-22.jar` into the directory listed below:

   `<IAS_ROOT>/ias/APPS/modules/soap/WEB-INF/classes`

   `IAS_ROOT` is the root of your application server installation.
3. Edit the `deploy` script in your Apache SOAP installation as follows:

   On Windows platforms, edit the `deploy.bat` file. On Solaris, edit the `deploy` script. In each case, change IAS_HOME and CLASSPATH to point to the correct path in your environment.
4. Deploy iPM Web Services by doing the following:

   ```
   cd <IPM_ROOT>/bpm/soap/apache-2_2/
   deploy DeploymentDescriptor.xml
   ```

   `IPM_ROOT` is typically located at `<IAS_ROOT>/iplanet/ipm65`.

5. Verify that iPM Web Services is installed by launching the Apache SOAP Administrator, available at the following URL:

   ```
   http://<host>/NASApp/soap/admin/
   ```

   The service `iPM65WS` should be listed in this screen.

Development Environment for Apache SOAP Clients

For developing (compiling) Apache SOAP clients, environment considerations include the following:

- Java Development Kit (JDK) version
- SOAP implementation version
- XML parser
- CLASSPATH properly configured to access SOAP and XML parser libraries
- `ipmsoap-2_2.jar` must appear before `pm65soap.jar` in your CLASSPATH

For information on specific versions supported, refer to the *Process Manager Release Notes* available at the following location:

```
http://docs.sun.com/prod/s1.ipaspro/
```

Runtime Environment for Apache SOAP Clients

The runtime environment considerations for Apache SOAP clients include the following:

- Java Runtime Environment (JRE) version
- CLASSPATH properly configured to access `pm65soap.jar`

For information on specific versions supported, refer to the *Process Manager Release Notes* available at the following location:

```
http://docs.sun.com/prod/s1.ipaspro/
```

Environment for WSDL Clients

You must modify the `PMService-service.wsdl` file provided with your distribution to specify the host and port for web services. In this file, edit the following line so *host:port* reflect the URL of the Apache SOAP service running on an iAS cluster.

```
<soap:address location="http:/host:port/NASApp/soap/rpcrouter"/>
```

Limitations Using SOAP with iPlanet Process Manager

When using SOAP services to access the process manager, keep in mind the following limitations:

- iPM expects SOAP messages in the SOAPRPC style, not the document style
- SOAP header and SOAP message chaining are not supported
- Functionality is limited to the web services exposed by Process Manager
- Data type conversion is limited to corresponding Process Manager data types
- Digital signature fields and file attachments for data elements are not supported
- Serialization and deserialization of iPM custom fields is not supported

  This means custom fields cannot be used in a SOAP call involving saveActivity or completeActivity
- There is no authentication associated with calls to iPM Web Services

  However, iPM Web Services does performs role resolution and allows (or disallows) certain tasks based on the AccessControl mechanism built into the Form ACL in the Process Builder. For example, a call to completeActivity succeeds only if the remoteUser passed in is actually an assignee or belongs to a Role that is an assignee to the Activity being completed.
- The completeActivity method can only be used with activities that have simple assignments (that is, a single user assigned to the activity)

  WorkItems that have group assignments (use toGroup) or dynamic assignments that have resulted through the use of addAssignee calls cannot be handled by web services clients.

Creating Web Services Clients

You can create web services clients for iPlanet Process Manager either by creating an Apache SOAP client or by using code generated when compiling the iPM WSDL file provided with your Process Manager distribution. The following section, "Creating a Client Using the WSDL File" on page 167, describes the procedure for creating a WSDL client. The section, "Creating an Apache SOAP Client" on page 167 describes how to create an Apache SOAP client.

Typical Web Services Client Application

Typically, an iPM web services client creates a process instance for a specific entry point in a process. The client then performs an available activity for that process. The following steps list key iPM Web Services calls you make in the client to complete an activity for a process instance.

1. Get the entry point into your Process Manager application.

   Here are a series of method calls that are useful in finding an entry point into a Process Manager application, creating a process instance, and completing the activity.

   - `getClusterNames`

     To get a list of the cluster names running in an iAS process instance.

   - `getApplicationNames`

     From a specific cluster, get a list of the applications deployed on that cluster.

   - `getEntryPointNames`

     From a specific application, get a list of the entry points into the application.

   - `getEntryPointData`

     From a specific entry point, gets a list of the editable data elements.

2. Create a process instance.

   Using information from the previous step, call `createProcessInstance` to create a process instance.

3. For the given process, complete an activity.

   Here are a series of calls that are useful in finding and completing an activity for a process instance.

   - `getActivityInfoList`

     Returns the worklist on a given cluster for a user. You can use this list to find specific activities.

   - `getActivityData`

     Returns a list of data elements associated with an activity. You can use this list to perform work on the given activity.

- `completeActivity`

  Performs the specified action for a user activity.

Creating a Client Using the WSDL File

This section describes how to create a web services client using the following WSDL files provided with your distribution at `iASROOT/soap/wsdl/`.

- `PMService.wsdl`

  Describes the web services methods available to access any iPM application.

- `PMService.xsd`

  Describes the data types available to web services clients.

- `PMService-service.wsdl`

  Used by third party applications to compile the WSDL files into stubs that can be used to create web services client applications. This file must be edited to reflect your iPM application's host name and port number, as described in "Environment for WSDL Clients" on page 164.

To create a web services client, compile `PMService-service.wsdl` using a WSDL compiler compliant with the version of SOAP certified for this release. Using the stub files generated by the WSDL compiler, create your web services client application.

Creating an Apache SOAP Client

This section describes how to create an Apache SOAP client application. It lists the Java methods and classes available to clients from the iPM Web Services API. It also contains a running code example that shows how an Apache web services client can call into an iPM application. This example assumes you have set up your environment to use Apache SOAP, as described in "Development Environment for Apache SOAP Clients" on page 164, and that you have deployed an iPM application to be invoked by a web services client. Finally, this section contains a description of the sample Apache SOAP client application provided with your Process Manager distribution.

iPM Web Services API for Apache SOAP Clients

This section lists the iPM Web Services methods available to Apache SOAP client applications. Table 7-2 lists the methods available to your client applications.

Table 7-2 iPM Web Services Methods available to Apache SOAP Clients

completeActivity	getClusterInfo
createProcessInstance	getClusterNames
getActivityData	getEntryPointData
getActivityInfo	getEntryPointNames
getActivityInfoList	getProcessInfo
getApplicationNames	saveActivity

completeActivity

Completes the specified action associated with a user activity, saving the data associated with the activity. This action corresponds to a remote user completing and submitting a form, and then moving the process to the next stage.

Typically, you first call getActivityData to get a list of data elements for the activity, set the data for each element in the returned data list, and then call completeActivity.

```
void completeActivity(IWFActivityID activityID, String remoteUser,
                      IWFDataList datalist, String action)
```

`activityID`	ID of the activity.
`remoteUser`	Authorized user for the application.
`datalist`	List of IWFData elements associated with the activity. This list of data elements should be obtained by a prior call to getActivityData. You should first call IWFData.setValue on each data element in the list before completing the activity. For more information on setValue, refer to "IWFData" on page 177.
`action`	Name of the action being performed.
`Exception`	Generates a SOAP fault if the operation does not succeed.

createProcessInstance

Creates a process instance at a specific entry point in the process.

Typically, you first call getEntryPointData to get a list of data elements for the specified action in the process, set the data for each element in the returned data list, and then call createProcessInstance.

IWFProcessInfo createProcessInstance (String clusterName, String appName, String entryNodeName, String remoteUser, String actionName, IWFDataList datalist)

`clusterName`	Name of the cluster containing the deployed applications. Specify "PMCluster" if you are working with process applications deployed to a default cluster, otherwise specify the name of the named cluster.
`appName`	Name of the application deployed to the cluster.
`entryNodeName`	Entry point name of the application (previously returned by a call to getEntryPointNames).
`remoteUser`	Authorized user for the application.
`actionName`	Name of the action being performed at the entry point. This represents one possible transition emanating from the entry point node.
`datalist`	List of IWFData elements associated with the action. This list of data elements should be obtained by a prior call to getEntryPointData. You should first call IWFData.setValue for each element in the list before creating the process instance. For more information on setValue, refer to "IWFData" on page 177.
`Exception`	Generates a SOAP fault if the operation does not succeed.

getActivityData

Returns the list of editable data elements associated with a user activity. You can change the value of these data elements and then pass the list into a call to saveActivity or completeActivity. Before calling saveActivity or completeActivity, you should first call IWFData.setValue for each item in the returned list. For more information on setValue, refer to "IWFData" on page 177.

IWFDataList getActivityData(IWFActivityID activityID, String remoteUser)

`activityID`	ID of the activity.
`remoteUser`	Authorized user for the application.
`Exception`	Generates a SOAP fault if the operation does not succeed.

getActivityInfo

Returns a user activity based on a primary key and a remote user (for role resolution). For more information on the IWFActivityInfo class, refer to "IWFActivityInfo" on page 174

IWFActivityInfo getActivityInfo(IWFActivityID activityID, String remoteUser)

`activityID`	ID of the activity.
`remoteUser`	Authorized user for the application.
`Exception`	Generates a SOAP fault if the operation does not succeed.

getActivityInfoList

Returns the worklist for a remote user within the context of a given iPM cluster.

IWFActivityInfoList getActivityInfoList(String clusterName, String remoteUser)

`clusterName`	Name of the cluster containing the deployed applications. Specify "PMCluster" if you are working with process applications deployed to a default cluster, otherwise specify the name of the named cluster.
`remoteUser`	Authorized user for the application.
`Exception`	Generates a SOAP fault if the operation does not succeed.

getApplicationNames

Returns the list of strings representing the names of deployed applications available on an iPM cluster.

IWFAppNamesList getApplicationNames(String clusterName)

`clusterName`	Name of the cluster containing the deployed applications. Specify "PMCluster" if you are working with process applications deployed to a default cluster, otherwise specify the name of the named cluster.
`Exception`	Generates a SOAP fault if the operation does not succeed.

getClusterInfo

Returns the properties of a given cluster. For more information on cluster properties, refer to "IPMClusterProperty" on page 139. This method returns a limited list of cluster properties

For more information on the IWFClusterInfo class, refer to "IWFClusterInfo" on page 176.

IWFClusterInfo getClusterInfo(String clusterName)

`clusterName`	Name of the cluster containing the deployed applications. Specify "PMCluster" if you are working with process applications deployed to a default cluster, otherwise specify the name of the named cluster.
`Exception`	Generates a SOAP fault if the operation does not succeed.

getClusterNames

Returns the names of iPM clusters available on an iAS cluster.

IWFClusterNamesList getClusterNames()

`Exception`	Generates a SOAP fault if the operation does not succeed.

getEntryPointData

Returns a list of editable data elements associated with a specific entry point node of a process application. Role resolution is performed to check if the remote user has access at the entry point. From the returned IWFDataList, you should change only the value of elements in the IWFDataList.

You typically use the returned data list when creating a process instance (by calling createProcessInstance). Before creating the process instance, you should first call IWFData.setValue on each element of the returned list. For more information on setValue, refer to "IWFData" on page 177.

IWFDataList getEntryPointData(String clusterName, String applicationName,
String entryPointName, String remoteUser)

`clusterName`	Name of the cluster containing the deployed applications. Specify "PMCluster" if you are working with process applications deployed to a default cluster, otherwise specify the name of the named cluster.

IWFDataList getEntryPointData(String clusterName, String applicationName, String entryPointName, String remoteUser) *(Continued)*

`applicationName`	Name of the application deployed to the cluster.
`entryPointName`	Entry point for the application.
`remoteUser`	Authorized user for the application.
`Exception`	Generates a SOAP fault if the operation does not succeed.

getEntryPointNames

Returns the list of entry point names for the given application, which is deployed on the given cluster for the given user

IWFEntryPointNamesList getEntryPointNames(String clusterName, String applicationName, String remoteUser)

`clusterName`	Name of the cluster containing the deployed applications. Specify "PMCluster" if you are working with process applications deployed to a default cluster, otherwise specify the name of the named cluster.
`applicationName`	Name of the application deployed to the cluster.
`remoteUser`	Authorized user for the application.
`Exception`	Generates a SOAP fault if the operation does not succeed.

getProcessInfo

Returns the information in a Process Instance, based on the Process Instance primary key. For more information on the IWFProcessInfo class, refer to "IWFProcessInfo" on page 181.

IWFProcessInfo getProcessInfo(IWFProcessID processID)

`processID`	ID of the process instance.
`Exception`	Generates a SOAP fault if the operation does not succeed.

saveActivity

Saves the data associated with a user activity. saveActivity does not complete the activity (move it to the next step).

Typically, you first call getActivityData to get a list of data elements for the activity, set the data for each element in the returned data list, and then call saveActivity.

void saveActivity(IWFActivityID activityID, IWFDataList datalist, String remoteUser)

`activityID`	ID of the activity.
`data`	List of data elements associated with the activity. This list of data elements should be obtained by a prior call to getActivityData. You should first call IWFData.setValue on each data element in the list before saving the activity. For more information on setValue, refer to "IWFData" on page 177.
`remoteUser`	Authorized user for the application.
`Exception`	Generates a SOAP fault if the operation does not succeed.

iPM Web Services Classes for Apache SOAP Clients

This section describes the classes used by Apache SOAP clients that access the iPM Web Services API. It provides a brief description of the class and lists the available accessor and mutator methods.

Table 7-3 iPM Web Services Classes for Apache SOAP Clients

IWFActivityID	IWFClusterNamesList
IWFActivityInfo	IWFData
IWFActivityInfoList	IWFDataList
IWFAppNamesList	IWFEntryPointNamesList
IWFAssignee	IWFProcessID
IWFAssigneeList	IWFProcessInfo
IWFClusterInfo	

IWFActivityID

Represents the unique identifier of a process activity in a running cluster.

Methods	Description
`String getActivityName()`	Returns the name of the node the activity belongs to.
`String getApplicationName()`	Returns the name of the application that this activity belongs to.
`String getClusterName()`	Returns the name of the cluster that the activity belongs to.
`String getForkId()`	Returns the fork Id of the activity.
`long getInstanceId()`	Returns the ID of the Process Instance that this activity belongs to.

IWFActivityInfo

Represents a process activity inside a cluster.

Methods	Description
`IWFActivityID getActivityId()`	Returns the unique identifier of the activity.
`String getApplicationName()`	Returns the name of the application that the activity belongs to.
`IWFAssigneeList getAssignees()`	Returns a list of current assignees of the activity.
`java.util.Date getCreationDate()`	Returns the creation date of the activity.
`java.util.Date getExpirationDate()`	Returns the expiration date of the activity (if one has been set).
`IWFProcessInfo getProcessInfo()`	Returns the process instance that this activity belongs to.
`int getState()`	Returns the running state of the activity.
`boolean hasExpired()`	Tests whether the activity has expired.
`boolean isAutomated()`	Tests whether the activity is automated.
`boolean isInExceptionState()`	Tests whether the activity is in exception state.

IWFActivityInfoList

List of IWFActivityInfo elements.

Methods	Description
`void addElement(IWFActivityInfo data)`	Adds an IWFActivityInfo element to the list.
`Iterator elements()`	Returns an iterator for the list.
`boolean exists(IWFActivityInfo data)`	Tests whether the specified IWFActivityInfo element exists in the list.
`IWFActivityInfo getElement(IWFActivityID id)`	Returns the IWFActivityInfo element from the list specified by the given ID.
`boolean isEmpty()`	Tests if the list is empty.
`void removeElement(IWFActivityInfo data)`	Removes an IWFActivityInfo element from the list.

IWFAppNamesList

The list of application names on an iPM cluster. Typically, you do not make any modifications to this list.

Methods	Description
`void addElement(String data)`	Adds an application name to the list.
`Iterator elements()`	Returns an iterator for the list.
`boolean exists(String name)`	Tests whether the named application exists in the list.
`boolean isEmpty()`	Tests if the list is empty.
`void removeElement(String data)`	Removes an application name from the list.

IWFAssignee

An abstraction for a Process Manager role or user.

Methods	Description
`String getType()`	Returns the role or user.
`String getValue()`	Returns the name of the role or user.

IWFAssigneeList

List of IWFAssignee elements.

Methods	Description
`void addElement(IWFAssignee data)`	Adds an IWFAssignee element to the list.
`Iterator elements()`	Returns an iterator for the list.
`boolean exists(IWFAssignee data)`	Tests whether the specified IWFAssignee element exists in the list.
`IWFAssignee getElement(String name, String type)`	Returns the IWFAssignee element from the list that is specified by both name and role/user.
`boolean isEmpty()`	Tests if the list is empty.
`void removeElement(IWFAssignee data)`	Removes an IWFAssignee element from the list.

IWFClusterInfo

The properties of a cluster.

Methods	Description
`String getProperty(String key)`	Returns the value of a cluster property specified by the given key. Valid values for key are listed in Table 7-4.
`Enumeration properties()`	Returns an enumeration of the cluster properties.

Table 7-4 Key List of Cluster Properties

CLUSTER_DN	DATABASE_IDENTIFIER
CLUSTER_NAME	DATABASE_NAME
CONFIGURATION_DIRECTORY_SERVER	DATABASE_USER_NAME
CONFIGURATION_DIRECTORY_PORT	DATASOURCE_NAME
CONFIGURATION_DIRECTORY_BIND_DN	DESCRIPTION
CORPORATE_DIRECTORY_SERVER	PRETTY_NAME
CORPORATE_DIRECTORY_PORT	SMTP_SERVER
CORPORATE_DIRECTORY_BASE	SMTP_PORT
CORPORATE_DIRECTORY_BIND_DN	SMTP_REPLY_TO
DATABASE_TYPE	EVENT_USER

IWFClusterNamesList

A list of cluster names on an iPM installation. Typically, you do not make any modifications to this list.

Methods	Description
`void addElement(String data)`	Adds the specified element to the list.
`Iterator elements()`	Returns an iterator for the list.
`boolean exists(String name)`	Tests whether the named cluster exists in the list.
`boolean isEmpty()`	Tests if the list is empty.
`void removeElement(String data)`	Removes an cluster name from the list.

IWFData

Represents a data dictionary element of the process application. The getActivityData and getEntryPointData methods of the iPM Web Service API returns a list containing these data elements (an IWFDataList). The IWFDataList represents fields in an HTML form or Process Manager activity.

Before using an IWFDataList in subsequent calls, you must set the data for each element in the list using IWFData.setValue.

NOTE When calling setValue, make sure the value passed in is a valid value for the data field. Process Manager throws an exception only if the value does not match the allowed type, as described in Table 7-5 on page 179.

For example, if you specify text for the value of a CheckBox, an exception is thrown. However if you specify the value 2 for the value of a CheckBox, the error will not be caught or reported, and may cause the process to misbehave.

`setValue` must be called before you do the following:

- Create a process instance (using the iPM Web Services API createProcessInstance), to initialize the values of data fields for work items in the process
- Complete or save an activity (using the iPM Web Services APIs completeActivity or saveActivity), to first load the values of data fields for work items before completing or saving an activity.

For more information on the iPM Web Services API, refer to "iPM Web Services API for Apache SOAP Clients" on page 167.

Methods	Description
`String getName()`	Returns the name of the data element.
`String getType()`	Returns the type of the data element, for example TYPE_TEXT. The type is typically built dynamically by the server and returned by calls to iPM Web Service APIs such as getActivityData.
`Object getValue()`	Returns a String representation of the data element in the process dictionary. Complex types represented by custom fields are not supported.

Methods	Description
`void setValue(String name, Object value)`	Initializes or sets the data element in the process dictionary. `name` is the name of the field for the data element, `value` is the value associated with the field. `value` must correspond to allowed values for the field represented by the data element, otherwise an exception is thrown. Refer to the following table for expected values for different data fields.

Table 7-5 Valid Values for Data Fields

iPM Field Type	Significance	Allowed Types	Valid Values
CheckBox	checked/unchecked	Integer	`0` - not checked `1` - checked
ComputedField	value	Text LongText Date DateTime Integer Float	Valid value for the specified type
Date	date	Date	a date in the format: DDMMYY or MMDDYY
DateTime	date time	Date	date in the format: DDMMYY:HHMMSS or MMDDYY:HHMMSS
Password	password	Text	Clear text password
Radio Buttons	valid option from the list	Text LongText Date DateTime Integer Float	Valid value for the specified type

Table 7-5 Valid Values for Data Fields *(Continued)*

iPM Field Type	Significance	Allowed Types	Valid Values
Select List	valid selection from the list	Text LongText Date DateTime Integer Float	Valid value for the specified type
TextArea	value from text area	Text	Text data. For size limitations on this field, refer to the *Process Builder's Guide*.
TextField	value from the text field	Text	Text data. For size limitations on this field, refer to the *Process Builder's Guide*.
URL	value from the URL field	Text	A valid URL, for example: *http://www.sun.com*
UserPickerWidget	user ID	Text	A valid corporate directory user ID, for example: *admin/admin21*

IWFDataList

Represents a list of IWFData elements. Refer to "IWFData" on page 177 for more information on using an IWFDataList.

Methods	Description
`void addElement(IWFData data)`	Adds an IWFData element to the list.
`Iterator elements()`	Returns an iterator for the list.
`boolean exists(IWFData data)`	Checks if a specified IWFData element is in the list.
`IWFData getElement(String name)`	Returns a specified IWFData element from the list.
`boolean isEmpty()`	Test if the list is empty.
`void removeElement(IWFData data)`	Removes a IWFData element from the list.

IWFEntryPointNamesList

List of entry point names on an iPM Application. Typically, you do not make any modifications to this list.

Methods	Description
`void addElement(String data)`	Adds the specified element to the list.
`Iterator elements()`	Returns an iterator for the list.
`boolean exists(String name)`	Tests whether the named entry point exists in the list.
`boolean isEmpty()`	Tests if the list is empty.
`void removeElement(String data)`	Removes an entry point name from the list.

IWFProcessID

The unique identifier of a process instance in a running cluster.

Methods	Description
`String getApplicationName()`	(Optional) Returns the name of the process application this instance belongs to.
`String getClusterName()`	Returns the name of the cluster this instance was created in.
`long getInstanceID()`	Returns the ID of the process instance.

IWFProcessInfo

Represents an instance of a process.

Methods	Description
`java.util.Date getCreationDate()`	Returns the date the process instance was created.
`String getCreator()`	Returns the ID of the creator of the process.
`String getApplicationName()`	Returns the name of the application that the process belongs.

Methods	Description
`String getEntryPointName()`	Returns the entry point at which the process instance was initiated.
`String getExitPointName()`	Returns the exit point at which the process completed.
`java.util.Date getLastModifiedDate()`	Returns the date of the last modification to data of the process instance.
`IWFActivityID getParentProcessId()`	Returns a unique identifier of the parent process (relevant only for subprocesses)
`IWFProcessID getProcessId()`	Returns a unique identifier of the process instance.
`int getState()`	Returns the running state of the process instance (for more information, refer to the Javadoc API documentation for com.netscape.pm.model.IProcessInstance)

Creating an Apache SOAP Client Application

This section contains code examples that illustrate the basic steps in creating an Apache SOAP client.

Set Up the Web Services Class

The first step is to set up import statements and declare your class, as you would for any Java application. Code Example 7-1 shows one way to do this.

Code Example 7-1 Building a Web Services Call—Set up the Web Services Class

```
package samples.wstest;

import java.io.*;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class wstest {
  public static void main (String[] args) throws Exception {
    if (args.length != 1 ) {
      System.err.println ("Usage: java " +
                          wstest.class.getName() +
                          "SOAP-router-URL");
```

Code Example 7-1 Building a Web Services Call—Set up the Web Services Class

```
      System.exit (1);
    }

    // Process the arguments.
    URL url = new URL (args[0]);

. . .
```

Create a Call Object

Next, create a Call object that you later use to invoke iPM Web Services API methods.

Code Example 7-2 Building a Web Services Call—Create a Call Object

```
. . .

    // Set up the call
    Call call = new Call ();

    SOAPMappingRegistry smr = call.getSOAPMappingRegistry() ;
    String pmNsURI =
                 "http://www.iplanet.com/ipm6/serviceapi/wsdl";
    String encodingStyle =
                 "http://schemas.xmlsoap.org/soap/encoding/";

. . .
```

Map the Data Type for the Call

You must specify each iPM Web Services data type that you will be using in your call. For more information, refer to the section "IWFData" on page 177.

Code Example 7-3 Building a Web Services Call—Map Data Types

```
. . .

    // Base Serializer for all iPM complex types .
    BaseSerializer ser_0 = new BaseSerializer() ;

    // WFDataList Serializer
    smr.mapTypes(encodingStyle,
          new QName( pmNsURI, "IWFDataList"),
```

Code Example 7-3 Building a Web Services Call—Map Data Types *(Continued)*

```
          com.sun.processmgr.serviceapi.types.IWFDataList.class,
          ser_0, ser_0);

    // . . . additional types

. . .
```

Set the Method and Parameters to the Call

Code Example 7-4 sets the method name and builds the parameter list to the method. After building the parameter list, it sets the parameters for the call. In this example, the value for the dataList parameter for the method createProcessInstance should be obtained by a prior call to either getActivityData or getEntryPointData.

Refer to "iPM Web Services API for Apache SOAP Clients" on page 167 for more information on web services methods exposed by Process Manager. Refer to "iPM Web Services Classes for Apache SOAP Clients" on page 173 for more information on using iPM Web Services data types.

Code Example 7-4 Building a Web Services Call—Set Method

```
. . .

    call.setTargetObjectURI ( service );
    call.setMethodName ( "createProcessInstance" );

    String encodingStyleURI = Constants.NS_URI_SOAP_ENC;
    call.setEncodingStyleURI(encodingStyleURI);

    // Build the data list for the entry point,
    // which is needed as a parameter to the method
    IWFDataList piData;
    piData = new WFDataList ();
    piData.addElement(new WFData("title", "TITLE_TEXT",
                      "My Credit Check"));
    piData.addElement(new WFData("person", "TITLE_TEXT",
                      "admin21"));

    // PARAMETERS TO THE CALL
    Vector params = new Vector ();
    params.addElement (new Parameter("defaultCluster",
                       String.class, "PMCluster", null));

    params.addElement (new Parameter("iPMapp",
                       String.class, "CreditHistory", null));

    params.addElement (new Parameter("entryNode",
```

Code Example 7-4 Building a Web Services Call—Set Method *(Continued)*

```
                      String.class, "", null));


    params.addElement (new Parameter("remoteUser",
                      String.class, "", null));


    params.addElement (new Parameter("action",
                      String.class, "PMCluster", null));


    params.addElement (new Parameter("dataList",
                      IWFDataList.class, piData, null));

    call.setParams(params);

. . .
```

Invoke the Method and Capture the Return Value

Now invoke the method on the iPM, check for errors, and capture information returned by the method call.

Code Example 7-5 Building a Web Services Call—Invoke Method

```
. . .

    Response resp = call.invoke (url,"" );
    Parameter ack = null ;
    // Check the response.
    if (resp.generatedFault ()) {
      Fault fault = resp.getFault ();
      System.out.println("Soap Call  failure: ");
      System.out.println("Fault Code = " +
                         fault.getFaultCode ());
      System.out.println("Fault String = " +
                         fault.getFaultString ());
      throw new Exception (" SOAP Service Exception :" +
                         fault.getFaultCode () + ":" +
                         fault.getFaultString () ) ;
    }
    else {
      ack = resp.getReturnValue ();
    }

    // process the return data
    IWFProcessInfo pi =  ack.getValue( ) ;
    System.out.println ("Process Instance Id:" +
```

Code Example 7-5 Building a Web Services Call—Invoke Method *(Continued)*

```
                              pi.getProcessId().getInstanceId());
    System.out.println ("Process Instance Creation Date :" +
                              pi.getCreationDate()) ;
    System.out.println ("Process Instance Creator :" +
                              pi.getCreator()) ;

. . .
```

Apache SOAP Client Sample

Your Process Manager distribution includes a sample Apache SOAP client application that instantiates the Credit History application, allowing the user to complete the Check Authorization activity. The sample can be found at the following location in your Process Manager installation:

```
<IPM_ROOT>/bpm/soap/sample/
```

This SOAP client provides the same functionality as Process Express for creating and completing the Credit History application. The initial data for process instantiation is pre-configured in a properties file. The sample application then uses the data generated by the Process Engine to complete the activity.

For more information on the Credit History application, refer to the *Process Builder's Guide*. For more information on calling iPM Web Services APIs in Apache SOAP clients, refer to "iPM Web Services API for Apache SOAP Clients" on page 167.

Setting Up the Apache SOAP Client Sample

The Apache SOAP client sample assumes the Credit History process definition has been deployed successfully using the Process Builder. It also assumes that the SOAP service related parameters, such as SOAP service location URL and name are specified in the `SOAPProperties.conf` file. Also, the initial parameters for process instantiation are specified in the `CALLParameters.conf` file.

The CLASSPATH for the sample should contain `pm65soap.jar`, to specify SOAP data types and (de)serializers for these types. The CLASSPATH should also include `xerces.jar` and `soap.jar`.

iPM Web Services API

The Apache SOAP Client sample creates a process instance and completes the intermediate activities by calling the following iPM Web Services APIs (in the following order):

- getEntryPointData

  Specifies the following parameters:

  - remoteUser
  - defaultCluster
  - appName
  - entryNodeName

  Returns IWFDataList containing the default entry point data, Person, and Title. To override the default entry point data, specify the desired Person name and Title in the `CALLParameters.conf` file.

- createProcessInstance

  Creates a process instance specifying the following:

  - configuration data: defaultCluster, appName, entryNodeName
  - user data: actionName (for example, Submit) and the entry point data returned from the call to getEntryPointData.

After creating the process instance, the Process Instance Id is displayed. The assignee of the activity can log in through Process Express and verify that a work item has been created corresponding to the user activity node “Check Authorization.”

The sample then prompts whether you want to complete the Check Authorization activity (that is, take action on the work item generated). If you specify to not take action, the sample application exits. If you specify to take action, the following choices are displayed:

- Create Report (approval to check authorization)
- Cancel Report (deny authorization request)

When completing the activity, the sample calls the following iPM Web Services APIs (in the following order):

- getActivityData

  Returns the activity data corresponding to the user activity “Check Authorization” for this process instance.

- completeActivity

  Using the data returned by getActivityData and the actionName (create/cancel report), completes the user activity.

- getProcessInfo

  Returns the final state of the process instance.

Appendix A

Scripts

This appendix contains the following code examples that show how to use custom data fields and custom activities for Process Manager applications.

- myObject.java
- EmployeeTrainingPerformer.java
- UpdatableList.java
- UpdatableList.jsb
- EmployeeTrainingPerformer.xml
- MenuOptions.xml
- TrainingDays.xml

These code examples are referenced from Chapter 3, "Advanced Office Setup Application," which describes how to customize the Office Setup sample application. The Office Setup sample application is described in the *Process Builder's Guide*. These scripts are also available in your Process Manager installation at:

```
<iPM_ROOT>/builder/manual/pg/scripts/
```

Code Example A-1 myObject.java

```
package customer.fields;

import java.io.*;

public class myObject implements Serializable {
String myvalue;
//String var2;
//String var3;
```

Code Example A-1 myObject.java

```
public myObject () {
}

}
```

Code Example A-2 EmployeeTrainingPerformer.java

```
/* EmployeeTrainingPerformer.java
 * Authors: Jocelyn Becker and Sagar
 * Last modified Sep 5 2000
 */

/* This custom activity schedules a new employee orientation training.
 * Employees in different departments attend trainings on different days.
 *
 * An xml file specifies which departments attend training
 * on which day, for example:
 *
 *  <DEPT>Engineering</DEPT>
 *  <DAY>monday</DAY>
 *  <DEPT>Marketing</DEPT>
 *  <DAY>tuesday</DAY>
 *  <DEPT>Human Resource</DEPT>
 *  <DAY>wednesday</DAY>
 *  <DEPT>Sales</DEPT>
 *  <DAY>thursday</DAY>
 *
 * The custom activity writes an HTML file that welcomes
 * the new employee and tells them which day to attend training.
 * Trainings are held at 2 o'clock in the afternoon.
 *
 * The employee will attend training on the first
 * appropriate day after they start work.
 * For example, if their training day is tuesday
 * and they start work on monday, they will be scheduled
 * for training the following day on tuesday
 * but if they start on thursday, they will be scheduled
 * for training on tuesday the following week.
 *
 * The custom activity gets
 * the employee's name from the Emp_name input parameter
 * the start date from the Start_Date input paramater
 * and the department name from the Dept input parameter
 *
 * These input parameters are mapped to actual data field names
 * in the TrainingPerformer.xml file:
 *
 * <INPUT>
```

Code Example A-2 EmployeeTrainingPerformer.java *(Continued)*

```
 *    <PARAMETER NAME="Emp_Name" DESCRIPTION="Employee Name">
 *    getData("dfEmpName") </PARAMETER>
 *    <PARAMETER NAME="Dept" DESCRIPTION="Dept. Name">
 *    getData("dfDeptName") </PARAMETER>
 *    <PARAMETER NAME="Start_Date" DESCRIPTION="Employee's Joining Date">
 *    getData("dfStartDate")</PARAMETER>
 *   ... </INPUT>
 *
 * Additonal input parameters include the path to the applications directory
 * and the process instance id:
 * <PARAMETER NAME="path" DESCRIPTION="Applications path">
 *   getApplicationPath()</PARAMETER>
 *   <PARAMETER NAME="id" DESCRIPTION="Process Instance ID">
 *   getProcessInstance().getInstanceId()</PARAMETER>
 *
 */

package customer.activities;

import java.util.*;
import java.io.*;
import java.net.*;

//import java.sql.*;

// This class schedules the training day for a new employee
// based on which department the employee is in.
public class EmployeeTrainingPerformer implements
       com.netscape.pm.model.ISimpleWorkPerformer {
    public final int SUN = 0;
    public final int MON = 1;
    public final int TUE = 2;
    public final int WED = 3;
    public final int THU = 4;
    public final int FRI = 5;
    public final int SAT = 6;



    public EmployeeTrainingPerformer (){
    }

    // The init() method does initialization stuff when the application starts.
    // It is not called for each new process instance.

    public void init(Hashtable env){
        // nothing to initialize here
    }

    // The perform() method defines what the custom activity does.
    // In this case, it reads the TrainingDays.xml file
    // to populate the trainingDays hash table
    // then it schedules the training.
```

Code Example A-2 EmployeeTrainingPerformer.java *(Continued)*

```
public void perform(Hashtable input, Hashtable output){
    System.out.println("Entering perform");

    // Get the employee's start date from the input hashtable
    Date startDate = (Date)input.get("Start_Date");
    //String startDate = sqlDate.toString();
   // startDate = startDate.replace('-', '/');

    // Get the new employee's name and their department
    // from the input hashtable.
    String dept = (String) input.get( "Dept" );
    String empName = (String) input.get( "Emp_Name" );

    // Get the application path from the input hashtable
    String appPath = (String)input.get("path");

    // Get the process instance id from the input hashtable
    int thisID = ((Double)input.get("id")).intValue();

    // Read the schedule from the TrainingDays.xml file
    Hashtable trainingDays = readSchedule(appPath);

    // Figure out what day of the week the employee will go to training
    Date trainingDate = scheduleTraining(startDate, dept, trainingDays);

    // Write a welcome page containing the training info
    String filename = writeWelcomePage(empName, thisID, appPath,
trainingDate);

     // Put the file name for the HTML page in the output hashtable
    output.put("welcomePage", filename);
    System.out.println("Exiting perform");
    }

    // The readSchedule function reads and parses the TrainingDays.xml file
    // to find out which departments go to training on which days.
    // The results are stored in the trainingDays hashtable, keyed by dept.
    // This happens in perform, not init, so that any changes to the
    // schedule are reflected in the next process instance that is created.
    private Hashtable readSchedule(String appPath){
        System.out.println("Entering readSchedule");
        Hashtable trainingDays = new Hashtable();
        //reading from file..
        try{
            int Pfound = 0;
            int MAX_LENGTH = 500;
            char xml[] = new char[MAX_LENGTH];
            String Path = appPath + "\\" + "TrainingDays.xml";

            java.io.File f = new java.io.File(Path);

            FileReader fr = new FileReader(f);

            BufferedReader in = new BufferedReader(fr);
```

Code Example A-2 EmployeeTrainingPerformer.java *(Continued)*

```
            int count = 0;
            count = in.read(xml, count, MAX_LENGTH);
            String charSet = new String(xml);

            int charSetLength = charSet.length();
            String temp = new String();
            String dept = new String();
            String day = new String();
            count = 0;
            // parse the file. Look for <DEPT> or <DAY>
            for(; count < charSetLength; count++)
            {
                if(charSet.charAt(count) == '<' )
                {
                    System.out.println("at <");
                    temp = "";
                    for(; charSet.charAt(count) != '>'; count++)
                    {
                        temp = temp + charSet.charAt(count);

                    }
                    temp = temp + charSet.charAt(count);

                    count++;
                    if(temp.equalsIgnoreCase("<DEPT>"))
                    {
                        for(dept = ""; charSet.charAt(count) != '<' ; dept =
                                                dept + charSet.charAt(count++));
                        trainingDays.put(dept, " ");
                    }

                    if(temp.equalsIgnoreCase("<DAY>"))
                    {
                        for(day = ""; charSet.charAt(count) != '<' ; day =
                                               day + charSet.charAt(count++));
                        trainingDays.put(dept, day);
                    }
                }
            }
        }
        catch(Exception ignore)
        {
            System.out.println("MY EXCEPTION IS: " + ignore.toString());
        }
        System.out.println("Exiting readSchedule");
        return trainingDays;
    }


// Schedule the training
public Date  scheduleTraining (Date startDate, String dept, Hashtable
trainingDays)
{
```

Code Example A-2 EmployeeTrainingPerformer.java *(Continued)*

```
    System.out.println("Entering schedule training");
    // Get info about the start date

    int thisDay = startDate.getDay();
    int dayOfMonth = startDate.getDate();
    int month = startDate.getMonth();
    int year = startDate.getYear();

    System.out.println("The day is " + (String)trainingDays.get(dept));
    // Using the dept as the key, get the value of the
    // training day from the the trainingDays hashtable
    if(((String)trainingDays.get(dept)).equals("monday") ){
                dayOfMonth=IncrementForMonday(thisDay, dayOfMonth);
        }
        else if(((String)trainingDays.get(dept)).equals("tuesday") ){
                dayOfMonth=IncrementForTuesday(thisDay, dayOfMonth);
        }
        else if(((String)trainingDays.get(dept)).equals("wednesday") ){
                dayOfMonth=IncrementForWednesday(thisDay, dayOfMonth);
        }
        else if(((String)trainingDays.get(dept)).equals("thursday") ){
                dayOfMonth=IncrementForThursday(thisDay, dayOfMonth);
        }
        else if(((String)trainingDays.get(dept)).equals("friday") ){
                dayOfMonth=IncrementForFriday(thisDay, dayOfMonth);
        }
        if(((String)trainingDays.get(dept)).equals("saturday") ){
                dayOfMonth=IncrementForSaturday(thisDay, dayOfMonth);
        }
        else if(((String)trainingDays.get(dept)).equals("sunday") ){
                dayOfMonth=IncrementForSunday(thisDay, dayOfMonth);
        }
        Date trainingDate = new Date(year,month,dayOfMonth);
        System.out.println("Exiting schedule training");
        return trainingDate;
}

// Create an HTML page that welcomes the
// new employee and tells them when to attend training.
// Trainings are held at 2 pm in the afternoon, so that
// if the employee needs to go to training on their start date
// they have time to find where to go and how to get there.

public String writeWelcomePage(String employeeName, int thisID, String
appPath, Date trainingDate) {
    // Format the date string
    System.out.println("Entering writeWelcomePage");
    String finalDate = formatDateString(trainingDate);

    // File name is Employee name + ProcessInstance
    String fileName = employeeName+thisID+".html";

    // Remove all white spaces from the filename
   fileName = fileName.replace(' ', '_');
```

Code Example A-2 EmployeeTrainingPerformer.java *(Continued)*

```
    // Path for this Application’s folder
    String thisPath = appPath +fileName;


    // Make a file in this Application’s folder
    try
    {
        RandomAccessFile HTMLfile = new RandomAccessFile(thisPath, "rw");

        HTMLfile.writeUTF("<HTML>");
        HTMLfile.writeUTF("<HEAD>");
        HTMLfile.writeUTF("<TITLE>New Employee Training</TITLE>");
        HTMLfile.writeUTF("</HEAD>");
        HTMLfile.writeUTF("<BODY>");
        HTMLfile.writeUTF("<CENTER><H1><FONT COLOR=MAGENTA>Hello <I>" +
                                  employeeName + "</I></FONT></H1></CENTER>");
        HTMLfile.writeUTF("<H3>Welcome to our company. </H3>");
        HTMLfile.writeUTF(
                  "<P> Please attend new employee orientation training on ");
        HTMLfile.writeUTF("<I>" + finalDate +
             " at 2 pm.</I> in Room B3, which is above the cafeteria.</P>");
        HTMLfile.writeUTF(
  "<P>We’ll have a tee-shirt, cap and other goodies for you at training!</P>");
        HTMLfile.writeUTF("</BODY>");
        HTMLfile.writeUTF("</HTML>");
        HTMLfile.close();
    }
    catch (Exception e)
    {
        System.out.println ("Trouble with writing welcome page: " + e);
    }
    System.out.println("Exiting writeWelcomePage");
    return fileName;

    }


private String formatDateString (Date trainingDate){
    System.out.println("Entering formatDateString");
    String finalDay ="?";
    try{
            // Logic to rearrange date String
            // eg Mon Oct 16 2000
            String dateStr = trainingDate.toString();
            int strIndx =  dateStr.indexOf("00:00");

            String tmpDay1 = dateStr.substring(0, strIndx);

            int endIndx = dateStr.indexOf("2000");
            String tmpDay2 = dateStr.substring(endIndx);
            finalDay = tmpDay1+ " " + tmpDay2;

            // 09:00:00 PDT/PST Remove PDT/PST
```

Code Example A-2 EmployeeTrainingPerformer.java *(Continued)*

```
            String time = dateStr.substring(strIndx, endIndx);

            int i = time.indexOf("P");
            time = time.substring(0, i);
            // end of string rearrangement logic
        }

        catch(Exception e){
            System.out.println("Error : "+e);
        }
        System.out.println("Exiting formatDateString");
        return finalDay;
}


    public void destroy(){
    }
    // helper functions to find training date
    private int IncrementForMonday(int thisDay, int dayOfMonth ){
        if(thisDay == SUN)
            // for Monday just Increment Once from Sunday
            dayOfMonth = dayOfMonth+1;
        if(thisDay == TUE)
            // for Monday just Increment 6 from Tue
            dayOfMonth = dayOfMonth+6;
        if(thisDay == WED)
             // for Monday just Increment 5 from Wed
            dayOfMonth = dayOfMonth+5;
        if(thisDay == THU)
            // for Monday just Increment 4 from Thursday
            dayOfMonth = dayOfMonth+4;
        if(thisDay == FRI)
            // for Monday just Increment 3 from Friday
            dayOfMonth = dayOfMonth+3;
        if(thisDay == SAT)
            // for Monday just Increment 2 from Saturday
            dayOfMonth = dayOfMonth+2;

        return dayOfMonth;
    }

    private int IncrementForTuesday(int thisDay, int dayOfMonth ){
        if(thisDay == SUN)
            dayOfMonth = dayOfMonth+2;
        if(thisDay == MON)
            dayOfMonth = dayOfMonth+1;
        if(thisDay == WED)
            dayOfMonth = dayOfMonth+6;
        if(thisDay == THU)
            dayOfMonth = dayOfMonth+5;
        if(thisDay == FRI)
            dayOfMonth = dayOfMonth+4;
        if(thisDay == SAT)
            dayOfMonth = dayOfMonth+3;
```

Code Example A-2 EmployeeTrainingPerformer.java *(Continued)*

```
        return dayOfMonth;
    }

    private int IncrementForWednesday(int thisDay, int dayOfMonth ){
        if(thisDay == SUN)
            dayOfMonth = dayOfMonth+3;
        if(thisDay == MON)
            dayOfMonth = dayOfMonth+2;
        if(thisDay == TUE)
            dayOfMonth = dayOfMonth+1;
        if(thisDay == THU)
            dayOfMonth = dayOfMonth+6;
        if(thisDay == FRI)
            dayOfMonth = dayOfMonth+5;
        if(thisDay == SAT)
            dayOfMonth = dayOfMonth+4;
        return dayOfMonth;
    }

    private int IncrementForThursday(int thisDay, int dayOfMonth ){
        if(thisDay == SUN)
            dayOfMonth = dayOfMonth+4;
        if(thisDay == MON)
            dayOfMonth = dayOfMonth+3;
        if(thisDay == TUE)
            dayOfMonth = dayOfMonth+2;
        if(thisDay == WED)
            dayOfMonth = dayOfMonth+1;
        if(thisDay == FRI)
            dayOfMonth = dayOfMonth+6;
        if(thisDay == SAT)
            dayOfMonth = dayOfMonth+5;
        return dayOfMonth;
    }

    private int IncrementForFriday(int thisDay, int dayOfMonth ){
        if(thisDay == SUN)
            dayOfMonth = dayOfMonth+5;
        if(thisDay == MON)
            dayOfMonth = dayOfMonth+4;
        if(thisDay == TUE)
            dayOfMonth = dayOfMonth+3;
        if(thisDay == WED)
            dayOfMonth = dayOfMonth+2;
        if(thisDay == THU)
            dayOfMonth = dayOfMonth+1;
        if(thisDay == SAT)
            dayOfMonth = dayOfMonth+6;
        return dayOfMonth;
    }

    private int IncrementForSaturday(int thisDay, int dayOfMonth ){
        if(thisDay == SUN)
            dayOfMonth = dayOfMonth+6;
```

Code Example A-2 EmployeeTrainingPerformer.java *(Continued)*

```
        if(thisDay == MON)
            dayOfMonth = dayOfMonth+5;
        if(thisDay == TUE)
            dayOfMonth = dayOfMonth+4;
        if(thisDay == WED)
            dayOfMonth = dayOfMonth+3;
        if(thisDay == THU)
            dayOfMonth = dayOfMonth+2;
        if(thisDay == FRI)
            dayOfMonth = dayOfMonth+1;
        return dayOfMonth;
    }

    private int IncrementForSunday(int thisDay, int dayOfMonth ){
        if(thisDay == MON)
            dayOfMonth = dayOfMonth+6;
        if(thisDay == TUE)
            dayOfMonth = dayOfMonth+5;
        if(thisDay == WED)
            dayOfMonth = dayOfMonth+4;
        if(thisDay == THU)
            dayOfMonth = dayOfMonth+3;
        if(thisDay == FRI)
            dayOfMonth = dayOfMonth+2;
        if(thisDay == SAT)
            dayOfMonth = dayOfMonth+1;
        return dayOfMonth;
    }

// end of class
}
```

Code Example A-3 UpdatableList.java

```
/* UpdatableList.java
 *
 * This data field displays itself as a SELECT list.
 * It reads the menu options from an XML file.
 * The data field creates a new instance of myObject to hold
 * the selected value. Then it saves the object to a file
 * in the application directory.
 * The selected option is read back from the object
 * when the data field is displayed again.
 */

/* The class definition of myObject is:
import java.io.*;

public class myObject implements Serializable {
```

Code Example A-3 UpdatableList.java *(Continued)*

```
String value;
//String var2;
//String var3;


public myObject () {
}

}
*/

package customer.fields;

import com.netscape.pm.model.*;
import com.netscape.pm.fields.*;
import com.netscape.pm.htmlFE.*;

import java.io.*;
import java.util.*;


public class UpdatableList extends BasicCustomField implements
IPresentationElement ,IDataElement{

    // The name of the file that stores the menu options
    // This is global across all process instances
    public String myFileName;


    public UpdatableList(){
        super();
    }

    // Method from BasicCustomeField that loads
    // properties that were set in the Builder
    protected void loadDataElementProperties(Hashtable entry )
                                              throws Exception {

    // Get the XML File name from the Builder properties
    myFileName = (String)entry.get("xmlfile");
    }

    /* The display() method for a work item
     * displays the data field in a form in the
     * work item as a SELECT menu whose name is the same as the datafield name.
     * If there is no initial selection, the default selection is "Choose now"
     * If a value has previously been selected, this value
     * is retrieved by calling getData() which in turn invokes load()
     * to load the value from the external store.
     */

    public void display(IProcessInstance pi, IHTMLPage html, int displayMode,
String displayFormat ) throws Exception {
```

Code Example A-3 UpdatableList.java *(Continued)*

```
        System.out.println("Entering display in work item");

        StringBuffer buffer = new StringBuffer();
        String selectedOption = null;
        // Get the value of the data field
        // If the value is not loaded, getData invokes load()
        myObject myobj = (myObject) pi.getData(getName());

        // If an object is found, set the selected option
        // to the value of the object's myvalue variable.
        if (myobj != null) {
            selectedOption = myobj.myvalue;
        }
        switch(displayMode){
            // In edit mode, display the data field as a SELECT menu
            // The menu options are stored in an xml file
            case MODE_EDIT:
                // Get the option names from the xml file and store
                // them in the vector optionNames.
                Vector optionNames =
fetchDataFromXML();

                buffer.append("<select size=\"1\" name=\"" +
                                                    getName() + "\" >");

                // If the option was not selected previously show
                // the default choice
                String optName = "";
                if(selectedOption==null){

                    buffer.append("<option selected>Choose now</option>");
                    // For each option in the vector optionNames
                    // write <OPTION> value="optionName"</OPTION>

                    for(int i=0; i<optionNames.size(); i++){
                        optName = (String)optionNames.elementAt(i);
                        buffer.append("<option value=\""+ optName + "\">");
                        buffer.append(optName);
                        buffer.append("</option>");
                        System.out.println("display(): Newest option: " +
                                           (String)optionNames.elementAt(i));
                    }
                 System.out.println("In display method, the buffer string is "
                                    + buffer);

                }

                // Else write <OPTION SELECTED> value=selectedOption</OPTION>
                // and the rest of the options below that
                else
{
                    buffer.append("<option selected>" + selectedOption +
                                  "</option>");
```

Code Example A-3 UpdatableList.java *(Continued)*

```
                    for(int i=0; i<optionNames.size(); i++){

                        // For each option in the vector optionNames
                        // check if this option is the selected one
                        // If it is, ignore it since we already wrote the HTML
                        // code for the selected option.
                        // If it is not the selected one,
                        // write <OPTION> value="optionName"</OPTION>
                        optName = (String)optionNames.elementAt(i);
                        if(!optName.equals(selectedOption)){
                            buffer.append("<option value=\""+ optName + "\">");
                            buffer.append(optName);
                            buffer.append("</option>");
                        }
                    }
                }
                // End the Select list
                buffer.append("</select>");
                break;

            case MODE_VIEW:
             // In View  mode, display the selected option as a string
             // The user cannot change the value in View mode
                buffer.append(" "+ selectedOption);
                break;
        }
        // Write the contents to the HTML page
        html.write(buffer.toString());
    }


    /* The display method for an entry point
     * displays the data field in an entrypoint form
     * as a SELECT menu whose name is the same as the data field name.
     * In this case, there will be no previously selected value in edit mode.
     * The data field should not be displayed in view mode.
     */

    public void display(IHTMLPage html, int displayMode,
                        String displayFormat ) throws Exception {

        StringBuffer buffer = new StringBuffer();
        switch(displayMode){
            case MODE_EDIT:

                // Get the option names from the xml file and store
                // them in the vector optionNames.
                Vector optionNames = fetchDataFromXML();

                //The selected option is "Choose now"
                buffer.append("<select size=1 name=" + getName() + " >");
                buffer.append("<option selected>Choose now</option>");

                // Get the option names from the xml file
```

Code Example A-3 UpdatableList.java *(Continued)*

```
                // and store them in the vector optionNames.
                // For each option, write <option> value=optionName</option>

                for(int i=0; i<optionNames.size(); i++){

                    buffer.append("<option value=\"" +
                                  (String)optionNames.elementAt(i)+"\">");
                    buffer.append((String)optionNames.elementAt(i));
                    buffer.append("</option>");
                }

                // Close the select tag
                buffer.append("</select>");
                break;

            // Display a warning in view mode.
            // We hope that the developer will see this during testing.

            case MODE_VIEW:
                buffer.append("<B>Incorrect use of this datafield,");
                buffer.append(
          " <BLINK>it must be in EDIT Mode </BLINK>in an Entry Point!!!</B>");
                break;
        }
        // Write the contents to the HTML page
        html.write(buffer.toString());

    }

    // The update() method parses the form parameters when
    // the HTML form is submitted.
    // In this case, it creates a new instance of myObject
    // whose "myvalue" variable is set to the value of the form element
    // that has the same name as this data field.
    // The new object is put into the process instance
    public void update(IProcessInstance pi, IPMRequest rq ) throws Exception {

        // Create a new myObject to hold the results
        myObject obj1 = new myObject();
        try {
            obj1.myvalue = (String) rq.getParameter(getName());
            // put the object into the pi
            pi.setData(getName(), obj1);
        }
        catch (Exception e) {
            System.out.println("Problem translating the form values: " + e);
        }
    }


    // The load() method generates and sets the value of the data field.
    // This method is called if getData() is invoked when the
    // data field value is not already loaded.
    // In this case, read the object from the file
```

Code Example A-3 UpdatableList.java *(Continued)*

```
    // and get its "myvalue" variable
    public void load(IProcessInstance pi) throws Exception
    {
        // Get the name of the file. It is stored as the entity key.
        // An example is thisfield123.txt
        String thisFileName = (String)
pi.getEntityKey(getName());
        if (thisFileName != null)
            {
             try {
            // Get a handle to the file at the location
            // in the application directory
            // eg rootdir/Applications/myApplication/thisfield123.html
            String myPath = getMyApplicationsPath();
            thisFileName = myPath + "\\" + thisFileName;

            // Get a file reader and read in the object
            FileInputStream fis = new FileInputStream(thisFileName);
            ObjectInputStream ois = new ObjectInputStream(fis);
            myObject newobj = (myObject) ois.readObject();

            // Put the object in the data field in the process instance
            pi.setData(getName(), newobj);
            }
            catch (Exception e)
            {
                System.out.println("Error while reading value from file: " + e);
            }
            }
            else {
                pi.setData(getName(), null);
            }
    }

    // The store method stores the value of the data field in
    // external storage to make it persistent.
    // In this case, we store the value as an object in a file whose
    // name is dataFieldNamePID.html, for example thisDataField123.txt
    public void store(IProcessInstance pi) throws Exception{

        // Get the data field name
        String thisID = getName();
        // Get the process instance ID
        long procID = pi.getInstanceId();

        // Concatenate the data field name with the PID
        // to keep the name unique across all process instances
        thisID = thisID + procID;
        String thisFileName = thisID + ".txt";

        // Store the file name as the entity key
        pi.setEntityKey(getName(), thisFileName);

        // Get the application directory
```

Code Example A-3 UpdatableList.java *(Continued)*

```
        String appdir = getMyApplicationsPath();
        // Generate the file name where the value will be stored
        thisFileName = appdir + "\\" + thisFileName;

        // Get the value of the data field from the pi,
        // which should be an instance of myObject
        myObject myobj = (myObject) pi.getData(getName());

        // Write the object to a file
        try {
            FileOutputStream fos = new FileOutputStream(thisFileName);
            ObjectOutputStream oos = new ObjectOutputStream(fos);

           oos.writeObject(myobj);
        }
        catch (Exception e)
            {System.out.println("Error while storing data field value to file:"
             + e);
        }

        // end store method
    }

    public void create(IProcessInstance pi) throws Exception {
            // no default value
            System.out.println("entering create method but doing nothing here");
    }

    // Fetch the set of menu options from the XML file
    public Vector fetchDataFromXML(){
        Vector optionNames = new Vector();

        try
        {
            int MAX_LENGTH = 2000;
            char xml[] = new char[MAX_LENGTH];

            // Get the path for the xml file
            // myFileName is a global variable --
            // it is the same for all process instances
            String Path = getMyApplicationsPath();
            Path = Path + "\\" + myFileName;

            // Get a file reader
            java.io.File f = new java.io.File(Path);
            FileReader fr = new FileReader(f);
            BufferedReader in = new BufferedReader(fr);


            int count =0;
            // Read the entire xml file into the array xml
            count = in.read(xml, count, MAX_LENGTH);

            String charSet = new String(xml);
```

Code Example A-3 UpdatableList.java *(Continued)*

```
            int charSetLength = charSet.length();

            count = 0;
            for(; count < charSetLength; count++)
            {
                parseForItemTag(count, charSetLength, charSet, optionNames);
            }
        }
        catch(Exception e)
        {
            System.out.println("Error while fetching data from xml file : "
                               + e);
        }

        // return the vector of option names
        return optionNames;
    }


    // This method parses an array of characters
    // to extract the items embedded ini <ITEM>...</ITEM> tags
    public void parseForItemTag (int count, int charSetLength,
                                 String charSet, Vector optionNames)
    {
            String temp;
            Object tempobj;
            if(charSet.charAt(count) == '<' )
             {
                 temp = "";
                 for(; charSet.charAt(count) != '>'; count++)
                 {
                     temp = temp + charSet.charAt(count);
                  }
                 temp = temp + charSet.charAt(count);
                 count++;
                 if(temp.equalsIgnoreCase("<ITEM>"))
                    {
                        for(temp = ""; charSet.charAt(count) != '<' ;
                            temp = temp + charSet.charAt(count++))

                        temp = temp;
                        // Convert the string to an object and
                        // add the object to the vector of options
                        tempobj = (Object) temp;
                        optionNames.addElement(tempobj);
                    }
             }
}


// Returns the path to the folder where the application is saved
String getMyApplicationsPath ()
{
    String path = "";
```

Code Example A-3 UpdatableList.java *(Continued)*

```
    try {
        path = getPMApplication().getHomePath();
    }
    catch (Exception e) {
        System.out.println("Exception while getting application path" + e);
    }
    return path;
}


// Use this function for debugging if necessary
// to read the object from the file after storing it.
public void  readMyObject (String filename) {
System.out.println("Entering readMyObject");
    // read the object from the file
try {
    FileInputStream fis = new FileInputStream(filename);
    ObjectInputStream ois = new ObjectInputStream(fis);
    myObject mo = (myObject) ois.readObject();

    System.out.println ("value: " + (String) mo.myvalue);
    }
    catch (Exception e)
    {System.out.println(e);
    }
    System.out.println("Exiting readMyObject");
}

// end of class
}
```

Code Example A-4 UpdatableList.jsb

```
/**
 * CONFIDENTIAL AND PROPRIETARY SOURCE CODE OF
 * NETSCAPE COMMUNICATIONS CORPORATION
 * Copyright (c) 1997, 1998 Netscape Communications Corporation.
 * All Rights Reserved.
 *
 * Use of this Source Code is subject to the terms of the applicable license
 * agreement from Netscape Communications Corporation.
 *
 * The copyright notice(s) in this Source Code does not indicate actual or
 * intended publication of this Source Code.
 *
 */
```

Code Example A-4 UpdatableList.jsb *(Continued)*

```
<JSB>
<JSB_DESCRIPTOR     NAME="customer.fields.UpdatableList"
                    DISPLAYNAME="Updatable Options List"
                    SHORTDESCRIPTION="Enhanced HTML Select" >

/***
 * Properties of the Core WFObject
 **/
    <JSB_PROPERTY   NAME="cn"
                    DISPLAYNAME="Name of this field"
                    SHORTDESCRIPTION="Unique identifier of this field"
                    ISDESIGNTIMEREADONLY>

    <JSB_PROPERTY   NAME="description"
                    DISPLAYNAME="Short Description"
                    SHORTDESCRIPTION="Short Description of this field">

    <JSB_PROPERTY   NAME="prettyname" TYPE="string"
                    DISPLAYNAME="Display Name"
                    SHORTDESCRIPTION="Display Name of this field">

    <JSB_PROPERTY   NAME="help" TYPE="string"
                    DISPLAYNAME="Help Message"
                    SHORTDESCRIPTION="Help message associated with this
field">

/***
 * Properties of the Field Object
 **/
    <JSB_PROPERTY   NAME="fieldclassid"
                    DISPLAYNAME="Field Class ID"
                    SHORTDESCRIPTION="Defines how the field is displayed to
the user"
                    DEFAULTVALUE="customer.fields.UpdatableList"
                    ISDESIGNTIMEREADONLY>

    <JSB_PROPERTY   NAME="fieldtype"
                    DISPLAYNAME="Data Type"
                    SHORTDESCRIPTION="Defines how the field is stored by the
application"
                    DEFAULTVALUE="TEXT"
                    VALUESET="TEXT,LONGTEXT,INT,FLOAT,DATE,DATETIME"
                    ISEXPERT>

    <JSB_PROPERTY   NAME="allow_search"
                    TYPE="boolean"
                    DISPLAYNAME="Allow Search"
                    DEFAULTVALUE="false"
                VALUESET="true,false"
                    SHORTDESCRIPTION="Can participants use this field to
search for instances?">
```

Code Example A-4 UpdatableList.jsb *(Continued)*

```
/***
 * Properties related to this specific format of field
 **/
 <JSB_PROPERTY  NAME="length"
                TYPE="int"
                DISPLAYNAME="Length"
                DEFAULTVALUE=20
                SHORTDESCRIPTION="Length of the Select"
                ISEXPERT>
 <JSB_PROPERTY  NAME="xmlfile"
                TYPE="string"
                DISPLAYNAME="XML file name"
                DEFAULTVALUE="MenuOptions.xml"
                SHORTDESCRIPTION="XML file for populating the options for
Select">
 <JSB_PROPERTY  NAME="default_value"
                TYPE="string"
                DISPLAYNAME="Default Value"
                SHORTDESCRIPTION="Default Value of this field">
 <JSB_PROPERTY  NAME="on_focus"
                TYPE="JS Expression"
                DISPLAYNAME="on Focus"
                SHORTDESCRIPTION="Focus Handler">
 <JSB_PROPERTY  NAME="on_change"
                TYPE="JS Expression"
                DISPLAYNAME="on Value Change"
                SHORTDESCRIPTION="Value Change Event Handler">
 <JSB_PROPERTY  NAME="on_blur"
                TYPE="JS Expression"
                DISPLAYNAME="on Blur"
                SHORTDESCRIPTION="Blur Handler">


</JSB>

/*** Template Property Squeleton: Cut, paste and fill it up.
 <JSB_PROPERTY  NAME="[[]]"
                TYPE="[[]]"
                DISPLAYNAME="[[]]"
                SHORTDESCRIPTION="[[]]"
                DEFAULTVALUE="[[]]"
                VALUESET="[[]]"
                ?ISDESIGNTIMEREADONLY?>
 ***/
```

Code Example A-5 EmployeeTrainingPerformer.xml

```
<?xml version = "1.0" ?>
<WORKPERFORMER TYPE="com.netscape.pm.model.ISimpleWorkPerformer"
                                    NAME="EmployeeTrainingPerformer"
                                    CLASS_ID="customer.activities.EmployeeTraini
ngPerformer"
                                    VERSION="1.1">
<ENVIRONMENT>
</ENVIRONMENT>
<INPUT>
    <PARAMETER NAME="Emp_Name" DESCRIPTION="Employee Name">
    getData("dfEmpName")
    </PARAMETER>
    <PARAMETER NAME="Dept" DESCRIPTION="Dept. Name">
    getData("dfDeptName")
    </PARAMETER>
    <PARAMETER NAME="Start_Date" DESCRIPTION="Employee's Joining Date">
    getData("dfStartDate")
    </PARAMETER>
    <PARAMETER NAME="path" DESCRIPTION="Applications path">
    getApplicationPath()
    </PARAMETER>
    <PARAMETER NAME="id" DESCRIPTION="Process Instance ID">
    getProcessInstance().getInstanceId()
    </PARAMETER>
</INPUT>
<OUTPUT>
    <PARAMETER NAME="welcomePage" DESCRIPTION="Greeting for New Employeer">
    mapTo("dfWelcomeURL")
    </PARAMETER>
</OUTPUT>
<EXCEPTIONS>
</EXCEPTIONS>
<DESIGN>
</DESIGN>
</WORKPERFORMER>
```

Code Example A-6 MenuOptions.xml

```
<xml version="1.0" encoding="us-ascii">
     <ITEMSET>
        <ITEM>Apple Imac</ITEM>
  <ITEM>HP-4150 laptop</ITEM>
  <ITEM>HP-9150 laptop</ITEM>
  <ITEM>Sun Solaris workstation</ITEM>
  <ITEM>Windows NT/98</ITEM>
  <ITEM>Windows 2000</ITEM>
     </ITEMSET>
```

Code Example A-7 TrainingDays.xml

```
<xml version="1.0" encoding="us-ascii">
  <DEPT>Engineering</DEPT>
  <DAY>monday</DAY>
  <DEPT>Marketing</DEPT>
  <DAY>tuesday</DAY>
  <DEPT>Human Resource</DEPT>
  <DAY>wednesday</DAY>
  <DEPT>Sales</DEPT>
  <DAY>thursday</DAY>
```

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X