AS/400 Toolbox for Java: Graphical Toolbox and PCML

AS/400 Toolbox for Java: Graphical Toolbox and PCML

Contents

Chapter 1. Graphical Toolbox

Overview

The Graphical Toolbox contains tools to help you create custom user interface panels in Java. You can incorporate the panels into your Java applications, applets, or Operations Navigator plugins. The panels may contain data obtained from the AS/400, or data obtained from another source such as a file in the local file system or a program on the network.

The **GUI Builder** is a WYSIWYG visual editor for creating Java dialogs, property sheets and wizards. With the GUI Builder you can add, arrange, or edit user interface controls on a panel, and then preview the panel to verify that the layout behaves the way you expected. You can use the panel definition in a dialog, insert panels into property sheets and wizards, or arrange the panels in splitter panes, deck panes and tabbed panes.

The **Resource Script Converter** converts Windows user interface elements into a form usable by Java programs. With the Resource Script Converter you can process Windows resource scripts (RC files) from your existing Windows applications and produce definitions of dialogs, property sheets and wizards that can be displayed in Java.

Underlying these two tools is a new technology called the **Panel Definition Markup Language**, or **PDML**. PDML is based on the Extensible Markup Language (XML) and defines a platform-independent language for describing the layout of user interface elements. Once your panels are defined in PDML, you can use the runtime API provided by the Graphical Toolbox to display them. The API displays your panels by interpreting the PDML and rendering your user interface using the Java Foundation Classes.

Benefits of the Graphical Toolbox

With the Graphical Toolbox you have the ability to create Java-based user interfaces quickly and easily. The GUI Builder lets you have precise control over the layout of UI elements on your panels. Because the layout is described in PDML, you are not required to develop any Java code for the user interface, and you do not need to recompile code in order to make changes. As a result, significantly less time is required to create and maintain your Java applications. The Resource Script Converter lets you migrate large numbers of Windows panels to Java quickly and easily.

Defining user interfaces in PDML creates some additional benefits. Because all of a panel's information is consolidated in a formal markup language, the tools can be enhanced to perform additional services on behalf of the developer. For example, both the GUI Builder and the Resource Script Converter are capable of generating HTML skeletons for the panel's online help. The tools scan the PDML markup for a panel and automatically determine what help topics are required. Anchor tags for the help topics are built right into the help skeleton, which frees the help writer to focus on developing appropriate content. The Graphical Toolbox runtime environment automatically displays the correct help topic in response to a user's request.

In addition, PDML provides tags that associate each control on a panel with a property on a Java bean. Once you have identified the bean classes that will supply data to the panel and have associated a property name with each of the appropriate controls, you can request that the tools generate Java source code skeletons for the bean objects. At runtime, the Graphical Toolbox automatically transfers data between the beans and the controls on the panel that you identified.

The Graphical Toolbox runtime environment provides support for event handling, user data validation, and common types of interaction among the elements of a panel. The correct platform look and feel for your user interface is automatically set based on the underlying operating system, and the GUI Builder lets you toggle the look and feel so that you can evaluate how your panels will look on different platforms.

Graphical Toolbox tools

The Graphical Toolbox provides you with two tools and, therefore, two ways of automating the creation of your user interfaces. You can use the GUI Builder to quickly and easily create new panels from scratch, or you can use the Resource Script Converter to convert existing Windows-based panels to Java. Both tools support internationalization for different locales.

GUI Builder. Three windows are displayed when you invoke the GUI Builder for the first time, as shown below.

Use the main GUI Builder window, shown in Figure 1, to create, edit and manage your PDML files, and to manage the elements of each panel.

Figure 1. GUI Builder window.

Use the Properties window, shown in Figure 2, to view or change the properties of the currently selected control.

Figure 2. Properties window.

Use the Toolbox window to select the next user interface control to be added to a panel. The Toolbox window also provides facilities for aligning groups of controls, for previewing the panel, and for requesting online help for a GUI Builder function. See Explanation of the Toolbox Widgets for a description of what each icon does.

Figure 3. Toolbox window.

The panel being edited is typically displayed in the lower right portion of the GUI Builder workspace.

Resource Script Converter. The Resource Script Converter consists of a two-paned tabbed dialog as shown in Figure 4. On the **Convert** pane you specify the name of the Microsoft or VisualAge for Windows RC file that is to be converted to PDML. You can specify the name of the target PDML file and associated Java resource bundle that will contain the translated strings for the panels. In addition, you can request that online help skeletons be generated for the panels, generate Java source code skeletons for the objects that supply data to the panels, and serialize the panel definitions for improved performance at runtime. The Converter's online help provides a detailed description of each input field on the Convert pane.

Figure 4. Resource Script Converter.

After the conversion has run successfully, you can use the **View** pane to view the contents of your newly-created PDML file, and preview your new Java panels. You can use the GUI Builder to make minor adjustments to a panel if needed. The Converter always checks for an existing PDML file before performing a conversion, and attempts to preserve any changes in case you need to run the conversion again later.

Getting started with the Graphical Toolbox

Use the following topics to to learn more about the Graphical Toolbox:

- Setting up the Graphical Toolbox
- Creating your user interface
- Displaying your panels at runtime
- Graphical Toolbox example
- Using the Graphical Toolbox in a browser
- PDML tags and syntax

[Legal | AS/400 Glossary]

Explanation of the Toolbox Widgets

Below is the Java GUI Editor's toolbox and an explanation of what each icon does when selected:

The pointer button allows you to move and resize a component on a panel.

The label widget allows you to insert a static label on a panel.

The text widget allows you to insert a text box on a panel.

The button widget allows you to insert a button on a panel.

The combo box widget allows you to insert a drop down list box on a panel.

The list box widget allows you to insert a list box on a panel.

The radio button widget allows you to insert a radio button on a panel.

- The check box widget allows you to insert a check box on a panel.
- The image widget allows you to insert an image on a panel.
- The group box widget allows you to insert a labeled group box on a panel.
- The tree widget allows you to insert an hierarchical tree on a panel.
- The table widget allows you to insert a table on a panel.
- The slider widget allows you to insert an adjustable slider on a panel.
- The progress bar widget allows you to insert a progress bar on a panel.
- The deck pane widget allows you to insert a deck pane on a panel. A deck pane contains a stack of panels and only one is visually displayed at a time.
- The split pane widget allows you to insert a split pane on a panel. A split pane is one pane divided into two horizontal or vertical panes.
- The tabbed pane widget allows you to insert a tabbed pane on a panel. A tabbed pane contains a collection of panels and only one is visually displayed at a time. The tabbed pane displays the collection of panels as a series of tabs and the user selects a tab to display a panel. The panel's title is used as the text for a tab.
- The custom widget allows you to insert a custom-defined user interface component on a panel.
- The grid widget allows you to enable a grid on a panel.
- The align top button allows you to align multiple components on a panel with the top edge of a specific, or primary, component.
- The align bottom button allows you to align multiple components on a panel with the bottom edge of a specific, or primary, component.
- The equalize height button allows you to equalize the height of multiple components with the height of a specific, or primary, component.
- The center vertically button allows you to center a selected component vertically relative to the panel.
- The margin button allows you to view the panel's margins.
- The align left button allows you to align multiple components on a panel with the left edge of a specific, or primary, component.
- The align right button allows you to align multiple components on a panel with the left edge of a specific, or primary, component.
- The equalize width button allows you to equalize the width of multiple components with the width of a specific, or primary, component.
- The center horizontally button allows you to center a selected component horizontally relative to the panel.
- The preview button allows you to preview what a panel will look like.
- The help button allows you to get more specific information on the Graphical Toolbox.

[Legal | AS/400 Glossary]

Setting up the Graphical Toolbox

Installing the Graphical Toolbox on your workstation

To develop Java programs using the Graphical Toolbox, you should install the Graphical Toolbox JAR files on your workstation. There are two ways to do this.

If you have already installed the AS/400 Toolbox for Java licensed program on an AS/400, you can copy the JAR files from the directory **/QIBM/ProdData/HTTP/Public/jt400/lib.** You can use FTP to do this (ensure that you transfer the files in binary mode), or use Client Access/400 to map a network drive. The AS400ToolboxInstaller class that comes with the AS/400 Toolbox for Java can also be used to install the Graphical Toolbox JAR files - specify the package name “OPNAV”. For more information, see Client installation and update classes.

You can also install the Graphical Toolbox when you install Client Access Express. The AS/400 Toolbox for Java is now shipped as part of Client Access Express. If you are installing Client Access Express for the first time, choose Custom Install and select the **AS/400 Toolbox for Java** component on the install menu. If you have already installed Client Access Express, you can use the Selective Setup program to install this component if it is not already present.

Setting your classpath

The Graphical Toolbox is delivered as a set of JAR files:

- **uitools.jar** Contains the GUI Builder and Resource Script Converter tools.
- **jui400.jar** Contains the runtime API for the Graphical Toolbox. Java programs use this API to display the panels constructed using the tools. These classes may be redistributed with applications.
- **data400.jar** Contains the runtime API for the Program Call Markup Language (PCML). Java programs use this API to call AS/400 programs whose parameters and return values are identified using PCML. These classes may be redistributed with applications.
- **util400.jar** Contains utility classes for formatting AS/400 data and handling AS/400 messages. These classes may be redistributed with applications.
- **x4j400.jar** Contains the XML parser used by the API classes to interpret PDML and PCML documents.

To use the Graphical Toolbox, you must add these JAR files to your CLASSPATH environment variable (or specify them on the `classpath` option on the command line). For example, if you have copied the files to the directory **C:\jt400\lib** on your workstation, you must add the following path names to your classpath:

```
C:\jt400\lib\uitools.jar;
C:\jt400\lib\jui400.jar;
C:\jt400\lib\data400.jar;
C:\jt400\lib\util400.jar;
C:\jt400\lib\x4j400.jar;
```

If you have installed the Graphical Toolbox using Client Access Express, the JAR files will all reside in the directory **\Program Files\Ibm\Client Access\jt400\lib** on the drive where you have installed Client Access Express. The path names in your classpath should reflect this.

Note: Internationalized versions of the GUI Builder and Resource Script Converter tools are available. To run a non-U.S. English version you must add the correct version of **uitools.jar** for your language and country to your Graphical Toolbox installation. These JAR files are available on the AS/400 in **/QIBM/ProdData/HTTP/Public/jt400/Mri29xx**, where 29*xx* is the 4-digit OS/400 NLV code corresponding to your language and country. (The names of the JAR files in the various MRI29xx directories include the correct 2-character Java language and country code suffixes.) This additional JAR file should be added to your classpath ahead of **uitools.jar** in the search order.

Using the Graphical Toolbox

Once you have installed the Graphical Toolbox, follow these links to learn how to use the tools:

- "Running the GUI Builder"
- "Running the Resource Script Converter" on page 8

[Legal | AS/400 Glossary]

Creating your user interface

Running the GUI Builder

To start the GUI Builder, invoke the Java interpreter as follows:

```
java com.ibm.as400.ui.tools.GUIBuilder [-plaf look and feel | -translate]
```

If you did not set your CLASSPATH environment variable to contain the Graphical Toolbox JAR files, then you will need to specify them on the command line using the `classpath` option. See Setting Up the Graphical Toolbox.

Options

`-plaf look and feel`

> The desired platform look and feel. This option lets you override the default look and feel that is set based on the platform you are developing on, so you can preview your panels to see how they will look on different operating system platforms. See the Swing 1.0.3 documentation for the various look and feel names.

`-translate`

> This option is used to restrict the actions allowable in the GUI Builder to those which only make sense when producing a version of the panel for another locale. Users will be able to move and resize panel elements, but they will not be allowed to change any properties.

Types of user interface resources

When you start the GUI Builder for the first time you will create a new PDML file by clicking **New File** on the **File** pulldown. Once you have created your new PDML file, you can define any of the following types of UI resources to be contained within it.

Panel The fundamental resource type. It describes a rectangular area within which UI elements are arranged. The UI elements may consist of simple controls, such as radio buttons or text fields, images, animations, custom controls, or more sophisticated subpanels (see Split Pane, Deck Pane and Tabbed Pane below). A panel may define the layout for a stand-alone window or dialog, or it may define one of the subpanels that is contained in another UI resource.

Property Sheet
A stand-alone window or dialog consisting of a tabbed pane and OK, Cancel, and Help buttons. Panel resources define the layout of each tabbed window.

Wizard
A stand-alone window or dialog consisting of a series of panels that are displayed to the user in a predefined sequence, with Back, Next, Cancel, Finish, and Help buttons. The wizard window may also display a list of tasks to the left of the panels which track the user's progress through the wizard.

Split Pane
A subpane consisting of two panels separated by a splitter bar. The panels may be arranged horizontally or vertically.

Tabbed Pane
A subpane consisting of the tabbed control that forms part of a property sheet.

Deck Pane
A subpane consisting of the collection of panels which forms part of a wizard. At runtime any of the panels may be displayed in response to a given user action.

String Table
A collection of string resources and their associated resource identifiers.

Generated files

The translatable strings for a panel are not stored in the PDML file itself, but in a separate Java resource bundle. The tools let you specify how the resource bundle is defined, either as a Java PROPERTIES file or as a ListResourceBundle subclass. A ListResourceBundle subclass is a compiled version of the translatable resources, which enhances the performance of your Java application. However, it will slow down the GUI Builder's preview feature, since the resources must be recompiled each time you preview a panel. Therefore it's best to start with a PROPERTIES file (the default setting) until you're satisfied with the design of your user interface.

You can use the tools to generate HTML skeletons for each panel in the PDML file. At runtime, the correct help topic is displayed when the user clicks on the panel's Help button or presses F1 while the focus is on one of the panel's

controls. You should insert your help content at the appropriate points in the HTML, within the scope of the `<helpcontent>` and `</helpcontent>` tags. Text within these tags is guaranteed to be preserved should you discover that you need to regenerate the skeleton after making a change to the panel's layout.

You can generate source code skeletons for the JavaBeans that will supply the data for a panel. Use the Properties window of the GUI Builder to fill in the DATACLASS and ATTRIBUTE properties for the controls which will contain data. The DATACLASS property identifies the class name of the bean, and the ATTRIBUTE property specifies the name of the gettor/settor methods that the bean class implements. Once you've added this information to the PDML file, you can use the GUI Builder to generate Java source code skeletons and compile them. At runtime, the appropriate gettor/settor methods will be called to fill in the data for the panel.

Note: The number and type of gettor/settor methods is dependent on the type of UI control with which the methods are associated. The method protocols for each control are documented in PDML Tags and Syntax.

Finally, you can serialize the contents of your PDML file. Serialization produces a compact binary representation of all of the UI resources in the file. This greatly improves the performance of your user interface, because the PDML file does not have to be interpreted in order to display your panels.

To summarize: If you have created a PDML file named **MyPanels.pdml**, the following files will also be produced based on the options you have selected on the tools:

- **MyPanels.properties** if you have defined the resource bundle as a PROPERTIES file
- **MyPanels.java** and **MyPanels.class** if you have defined the resource bundle as a ListResourceBundle subclass
- **<panel name>.html** for each panel in the PDML file, if you have elected to generate online help skeletons
- **<dataclass name>.java** and **<dataclass name>.class** for each unique bean class that you have specified on your DATACLASS properties, if you have elected to generate source code skeletons for your JavaBeans
- **<resource name>.pdml.ser** for each UI resource defined in the PDML file, if you've elected to serialize its contents.

Note: As you can see from the above, you will have problems if you use the same name for both a PDML file and a JavaBean class because two like-named class files will be produced, and at runtime you will see some very cryptic error messages. Be aware of this problem when choosing names for your PDML files.

Running the Resource Script Converter

To start the Resource Script Converter, invoke the Java interpreter as follows:

```
java com.ibm.as400.ui.tools.PDMLViewer
```

If you did not set your CLASSPATH environment variable to contain the Graphical Toolbox JAR files, then you will need to specify them on the command line using the `classpath` option. See Setting Up the Graphical Toolbox.

You can also run the Resource Script Converter in batch mode using the following command:

```
java com.ibm.as400.ui.tools.RC2XML file [options]
```

Where *file* is the name of the resource script (RC file) to be processed.

Options

-x *name*
: The name of the generated PDML file. Defaults to the name of the RC file to be processed.

-p *name*
: The name of the generated PROPERTIES file. Defaults to the name of the PDML file.

-r *name*
: The name of the generated ListResourceBundle subclass. Defaults to the name of the PDML file.

-package *name*
: The name of the package to which the generated resources will be assigned. If not specified, no package statements will be generated.

-l *locale*
: The locale in which to produce the generated resources. If a locale is specified, the appropriate 2-character ISO language and and country codes will be suffixed to the name of the generated resource bundle.

-h
: Generate HTML skeletons for online help.

-d
: Generate source code skeletons for JavaBeans.

-s
: Serialize all resources.

Mapping Windows Resources to PDML

All dialogs, property sheets, wizards and string tables found in the RC file will be converted to the corresponding Graphical Toolbox resources in the generated PDML file. You can also define DATACLASS and ATTRIBUTE properties for Windows controls that will be propagated to the new PDML file by following a simple naming convention when you create the identifiers for your Windows resources. These properties will be used to generate source code skeletons for your JavaBeans when you run the conversion.

The naming convention for Windows resource identifiers is:

```
IDCB_<class name>_<attribute>
```

where `<class name>` is the fully-qualified name of the bean class that you wish to designate as the DATACLASS property of the control, and `<attribute>` is the name of the bean property that you wish to designate as the ATTRIBUTE property of the control.

For example, a Windows text field with the resource ID `IDCB_com_MyCompany_MyPackage_MyBean_SampleAttribute` would produce a DATACLASS property of **com.MyCompany.MyPackage.MyBean** and an ATTRIBUTE property of **SampleAttribute**. If you elect to generate JavaBeans when you run the conversion, the Java source file **MyBean.java** would be produced, containing the package statement **package**

com.MyCompany.MyPackage, and gettor and settor methods for the **SampleAttribute** property.

[Legal | AS/400 Glossary]

Displaying your panels at runtime

The Graphical Toolbox provides a redistributable API that your Java programs can use to display user interface panels defined using PDML. The API displays your panels by interpreting the PDML and rendering your user interface using the Java Foundation Classes.

The Graphical Toolbox runtime environment provides the following services:

- Handles all data exchanges between user interface controls and the JavaBeans that you identified in the PDML.
- Performs validation of user data for common integer and character data types, and defines an interface that allows you to implement custom validation. If data is found to be invalid, an error message is displayed to the user.
- Defines standardized processing for Commit, Cancel and Help events, and provides a framework for handling custom events.
- Manages interactions between user interface controls based on state information defined in the PDML. (For example, you may want to disable a group of controls whenever the user selects a particular radio button.)

The package com.ibm.as400.ui.framework.java contains the Graphical Toolbox runtime API.

The elements of the Graphical Toolbox runtime environment are shown in Figure 1. Your Java program is a client of one or more of the objects in the **Runtime Managers** box.

Figure 1. Graphical Toolbox Runtime Environment

Examples

Assume that the panel **MyPanel** is defined in the file **TestPanels.pdml**, and that a properties file **TestPanels.properties** is associated with the panel definition. Both files reside in the directory **com/ourCompany/ourPackage**, which is accessible either from a directory defined in the classpath or from a ZIP or JAR file defined in the classpath. The following code creates the panel and displays it:

```
import com.ibm.as400.ui.framework.java.*;

// Create the panel manager. Parameters:
// 1. Resource name of the panel definition
// 2. Name of panel
// 3. List of DataBeans omitted

try {
PanelManager pm = new PanelManager("com.ourCompany.ourPackage.TestPanels",
                                   "MyPanel",
                                   null);
```

```
}
catch (DisplayManagerException e) {
e.displayUserMessage(null);
System.exit(-1);
}

// Display the panel
pm.setVisible(true);
```

Once the DataBeans that supply data to the panel have been implemented and the attributes have been identified in the PDML, the following code may be used to construct a fully-functioning dialog:

```
import com.ibm.as400.ui.framework.java.*;
import java.awt.Frame;

// Instantiate the objects which supply data to the panel
TestDataBean1 db1 = new TestDataBean1();
TestDataBean2 db2 = new TestDataBean2();

// Initialize the objects
db1.load();
db2.load();

// Set up to pass the objects to the UI framework
DataBean[] dataBeans = { db1, db2 };

// Create the panel manager. Parameters:
// 1. Resource name of the panel definition
// 2. Name of panel
// 3. List of DataBeans
// 4. Owner frame window

Frame owner;
...
try {
PanelManager pm = new PanelManager("com.ourCompany.ourPackage.TestPanels",
                                   "MyPanel",
                                   dataBeans,
                                   owner);
}
catch (DisplayManagerException e) {
e.displayUserMessage(null);
System.exit(-1);
}

// Display the panel
pm.setVisible(true);
```

[Legal | AS/400 Glossary]

Graphical Toolbox Example

To demonstrate how to use the Graphical Toolbox to build your user interface, this example constructs a simple panel that illustrates the basic features and operation of the Graphical Toolbox environment. Then the example shows to build a small Java application that displays the panel. When the user enters data in the text field and clicks on the **Close** button, the application will echo the data to the Java console.

Constructing the panel

When we invoke the GUI Builder, we create a new PDML file called **MyGUI.pdml**. We insert a new panel PANEL_1 into this file, which has a title of “Simple Example”. We have already added three elements to this panel using the buttons in the Toolbox window: a label, a text field, and a pushbutton.

We entered the text for the label in the correct field in the Properties window, which displays when the label is selected.

Next we consider the text field. Because the text field will contain data, we can set several properties that will allow the GUI Builder to perform some additional work on our behalf. We set the Data Class property to the name of a bean class named **SampleBean** that will supply the data for this text field.

We set the Attribute property to the name of the bean property, **UserData**, that will contain the data.

In effect, we are binding the **UserData** property to this text field. At run-time, the Graphical Toolbox obtains the initial value for this field by calling **`SampleBean.getUserData`**, and the modified value is sent back to the application when the panel closes by calling **`SampleBean.setUserData`**.

Now we want to set some data validation on the text field. We specified that the user is required to supply some data, and that it must be a string with a maximum length of 15 characters.

Finally, we indicated that the context-sensitive help for the text field should be the help topic associated with the label “Enter some data”.

For the pushbutton, we modified the style property to give it default emphasis.

We have also set the Action property to COMMIT. This causes the `setUserData` method on the bean to be called when the button is pressed.

Before saving the panel, we set properties at the level of the PDML file to generate both the online help skeleton and the JavaBean. Then we saved the file by clicking on the icon in the main GUI Builder window.

Generated files

Now that we have saved the panel definition, let us look at the files produced by the GUI Builder. First, we show the contents of **MyGUI.pdml** to give you a flavor of how the Panel Definition Markup Language works. Because all of your dealings with PDML are through the tools provided by the Graphical Toolbox, it is not necessary to understand the format of this file in detail.

```
<!– Generated by GUIBUILDER –>
<PDML version="1.0" source="JAVA" basescreensize="1280x1024">

 <PANEL name="PANEL_1">
  <TITLE>PANEL_1</TITLE>
  <SIZE>351,162</SIZE>
  <LABEL name="LABEL_1" disabled="no">
   <TITLE>LABEL_1</TITLE>
   <LOCATION>18,36</LOCATION>
   <SIZE>94,18</SIZE>
  </LABEL>
  <TEXTFIELD name="TEXT_1" masked="no" editable="yes" disabled="no">
   <TITLE>TEXT_1</TITLE>
   <LOCATION>125,31</LOCATION>
   <SIZE>191,26</SIZE>
   <DATACLASS>SampleBean</DATACLASS>
   <ATTRIBUTE>UserData</ATTRIBUTE>
   <STRING minlength="0" maxlength="15" required="yes"/>
   <HELPALIAS>LABEL_1</HELPALIAS>
  </TEXTFIELD>
  <BUTTON name="BUTTON_1" disabled="no">
   <TITLE>BUTTON_1</TITLE>
   <LOCATION>125,100</LOCATION>
   <SIZE>100,26</SIZE>
   <STYLE>DEFAULT</STYLE>
   <ACTION>COMMIT</ACTION>
  </BUTTON>
 </PANEL>

</PDML>
```

Associated with every PDML file is a resource bundle. In this case we chose to save the translatable resources in a PROPERTIES file, which is called **MyGUI.properties**. You will notice that the PROPERTIES file also contains customization data for the GUI Builder.

```
##Generated by GUIBUILDER
BUTTON_1=Close
TEXT_1=
@GenerateHelp=1
```

```
@Serialize=0
@GenerateBeans=1
LABEL_1=Enter some data:
PANEL_1.Margins=18,18,18,18,18,18
PANEL_1=Simple Example
```

Recall that we generated a Java source code skeleton for the JavaBean object. Here are the contents of **SampleBean.java**.

```
import com.ibm.as400.ui.framework.java.*;

public class SampleBean extends Object
    implements DataBean
{
    private String m_sUserData;

    public String getUserData()
    {
        return m_sUserData;
    }

    public void setUserData(String s)
    {
        m_sUserData = s;
    }

    public Capabilities getCapabilities()
    {
        return null;
    }

    public void verifyChanges()
    {
    }

    public void save()
    {
    }

    public void load()
    {
        m_sUserData = "";
    }
}
```

Note that the skeleton already contains an implementation of the gettor and settor methods for the `UserData` property. The other methods are defined by the `DataBean` interface and are, therefore, required.

The GUI Builder has already invoked the Java compiler for the skeleton and produced the corresponding class file. For the purposes of our simple example, we do not need to modify the bean implementation. In a real Java application you would typically modify the `load` and `save` methods to transfer data from an external data source. The default implementation of the other two methods is often sufficient. For more information, see the documentation on the **DataBean** interface in the package com.ibm.as400.ui.framework.java.

Constructing the application

Now that we have saved the panel definition and the generated files, we are ready to construct our application. All we really need is a new Java source file that will contain our main entry point for the application. This file will be called **SampleApplication.java**. It contains the following code:

```
import com.ibm.as400.ui.framework.java.*;
import java.awt.Frame;

public class SampleApplication
{
    public static void main(String[] args)
    {
        // Instantiate the bean object that supplies data to the panel
        SampleBean bean = new SampleBean();

        // Initialize the object
        bean.load();

        // Set up to pass the bean to the panel manager
        DataBean[] beans = { bean };

        // Create the panel manager. Parameters:
        // 1. PDML file as a resource name
        // 2. Name of panel to display
        // 3. List of data objects that supply panel data
        // 4. An AWT Frame to make the panel modal

        PanelManager pm = null;
        try { pm = new PanelManager("MyGUI", "PANEL_1", beans, new
Frame()); }
        catch (DisplayManagerException e)
        {
            // Something didn't work, so display a message and exit
            e.displayUserMessage(null);
            System.exit(1);
        }

        // Display the panel - we give up control here
        pm.setVisible(true);

        // Echo the saved user data
        System.out.println("SAVED USER DATA: '" + bean.getUserData() + "'");

        // Exit the application
        System.exit(0);
    }
}
```

It is the responsibility of the calling program to initialize the bean object or objects by calling **load**. If the data for a panel is supplied by multiple bean objects, then each of the objects must be initialized before passing them to the Graphical Toolbox environment.

The class **com.ibm.as400.ui.framework.java.PanelManager** supplies the API for displaying standalone windows and dialogs. The name of the PDML file as supplied on the constructor is treated as a resource name by the Graphical Toolbox - the directory, ZIP file, or JAR file containing the PDML must be identified in the classpath.

Because a **Frame** object is supplied on the constructor, the window will behave as a modal dialog. In a real Java application, this object might be obtained from a suitable parent window for the dialog. Because the window is modal, control does not return to the application until the user closes the window. At that point, the application simply echoes the modified user data and exits.

Running the application

Here is what the window looks like when the application is compiled and run:

If the user presses F1 while focus is on the text field, the Graphical Toolbox will display a help browser containing the online help skeleton that the GUI Builder generated.

We could edit the HTML if we chose, and add actual help content for the help topics shown.

If the data in the text field is not valid (for example, if the user clicked on the **Close** button without supplying a value), the Graphical Toolbox will display an error message and return focus to the field so that data can be entered.

Summary

That is it for our initial example. To give you an overview of the how the Graphical Toolbox works, we have left out a great many features that you can build into a panel. And we have not even brought up the topic of property sheets or wizards. We encourage you to experiment, using the GUI Builder's online help as a guide.

For information on how to run this sample as an applet, see Using the Graphical Toolbox in a Browser.

Using the Graphical Toolbox in a browser

You can use the Graphical Toolbox to build panels for Java applets that run in a web browser. This section describes how to convert the simple panel from the Graphical Toolbox Example to run in a browser. The minimum browser levels supported are Netscape 4.05 and Internet Explorer 4.0. In order to avoid having to deal with the idiosyncrasies of individual browsers, we recommend that your applets run using Sun's Java Plug-in. Otherwise, you will need to construct signed JAR files for Netscape, and separate signed CAB files for Internet Explorer.

Constructing the applet: The code to display a panel in an applet is nearly identical to the code used in the Java application example, but, of course, the code must be repackaged in the **init** method of a **JApplet** subclass. Also, we must add

some code to ensure that the applet panel is sized to the dimensions specified in the panel's PDML definition. Here is the source code for our example applet, **SampleApplet.java**.

```
import com.ibm.as400.ui.framework.java.*;
import com.sun.java.swing.*;
import java.awt.*;
import java.applet.*;
import java.util.*;
public class SampleApplet extends JApplet
{
    // The following are needed to maintain the panel's size
    private PanelManager        m_pm;
    private Dimension           m_panelSize;
    // Define an exception to throw in case something goes wrong
    class SampleAppletException extends RuntimeException {}
    public void init()
    {
        System.out.println("In init!");
        // Trace applet parameters
        System.out.println("SampleApplet code base=" + getCodeBase());
        System.out.println("SampleApplet document base=" + getDocumentBase());
        // Do a check to make sure we're running a JVM that's compatible with Swing 1.0.3
        if (System.getProperty("java.version").compareTo("1.1.5") < 0)
            throw new IllegalStateException("SampleApplet cannot run on Java VM version " +
                                            System.getProperty("java.version") + " - requires
        // Instantiate the bean object that supplies data to the panel
        SampleBean bean = new SampleBean();
        // Initialize the object
        bean.load();
        // Set up to pass the bean to the panel manager
        DataBean[] beans = { bean };
        // Update the status bar
        showStatus("Loading the panel definition...");
        // Create the panel manager. Parameters:
        // 1. PDML file as a resource name
        // 2. Name of panel to display
        // 3. List of data objects that supply panel data
        // 4. The content pane of the applet
        try { m_pm = new PanelManager("MyGUI", "PANEL_1", beans, getContentPane()); }
        catch (DisplayManagerException e)
        {
            // Something didn't work, so display a message and exit
            e.displayUserMessage(null);
            throw new SampleAppletException();
        }
        // Identify the directory where the online help resides
        m_pm.setHelpPath("http://MyDomain/MyDirectory/");
        // Display the panel
        m_pm.setVisible(true);
    }
    public void start()
    {
        System.out.println("In start!");
        // Size the panel to its predefined size
        m_panelSize = m_pm.getPreferredSize();
        if (m_panelSize != null)
        {
            System.out.println("Resizing to " + m_panelSize);
            resize(m_panelSize);
        }
        else
            System.err.println("Error: getPreferredSize returned null");
    }
    public void stop()
    {
        System.out.println("In stop!");
```

```
    }
    public void destroy()
    {
        System.out.println("In destroy!");
    }
    public void paint(Graphics g)
    {
        // Call the parent first
        super.paint(g);
        // Preserve the panel's predefined size on a repaint
        if (m_panelSize != null)
            resize(m_panelSize);
    }
}
```

The applet's content pane is passed to the Graphical Toolbox as the container to be laid out. In the **start** method, we size the applet pane to its correct size, and we override the **paint** method in order to preserve the panel's size when the browser window is resized.

When running the Graphical Toolbox in a browser, the HTML files for your panel's online help cannot be accessed from a JAR file. They must reside as separate files in the directory where your applet resides. The call to **`PanelManager.setHelpPath`** identifies this directory to the Graphical Toolbox, so that your help files can be located.

HTML tags: Because we recommend the use of Sun's Java Plug-in to provide the correct level of the Java runtime environment, the HTML for identifying a Graphical Toolbox applet is not as straightforward as we would like. Fortunately, the same HTML template may be reused, with only slight changes, for other applets. The markup is designed to be interpreted in both Netscape Navigator and Internet Explorer, and it generates a prompt for downloading the Java Plug-in from Sun's web site if it's not already installed on the user's machine. For detailed information on the workings of the Java Plug-in see the Java Plug-in HTML Specification. 

Here is our HTML for the sample applet, in the file **MyGUI.html**:

```
<html>
<head>
<title>Graphical Toolbox Demo</title>
</head>
<body>
<h1>Graphical Toolbox Demo Using Java(tm) Plug-in</h1>
<p>
<!– BEGIN JAVA(TM) PLUG-IN APPLET TAGS –>
<!– The following tags use a special syntax which allows both Netscape and Internet Explorer to loa
<!– the Java Plug-in and run the applet in the Plug-in's JRE.  Do not modify this syntax.
<!– For more information see http://java.sun.com/products/jfc/tsc/swingdoc-current/java_plug_in.htm
<OBJECT classid="clsid:8AD9C840-044E-11D1-B3E9-00805F499D93"
        width="400"
        height="200"
        align="left"
        codebase="http://java.sun.com/products/plugin/1.1.1/jinstall-111-win32.cab#Version=1,1,1,0"
    <PARAM name="code"     value="SampleApplet">
    <PARAM name="codebase" value="http://w3.rchland.ibm.com/~dpetty/applets/">
    <PARAM name="archive"  value="MyGUI.jar,jui400.jar,util400.jar,x4j400.jar">
    <PARAM name="type"     value="application/x-java-applet;version=1.1">
    <COMMENT>
    <EMBED type="application/x-java-applet;version=1.1"
           width="400"
           height=200"
           align="left"
           code="SampleApplet"
```

```
          codebase="http://w3.rchland.ibm.com/~dpetty/applets/"
          archive="MyGUI.jar,jui400.jar,util400.jar,x4j400.jar"
          pluginspage="http://java.sun.com/products/plugin/1.1.1/plugin-install.html">
        <NOEMBED>
    </COMMENT>
        No support for JDK 1.1 applets found!
        </NOEMBED>
    </EMBED>
</OBJECT>
<!– END JAVA(TM) PLUG-IN APPLET TAGS –>
<p>
</body>
</html>
```

It is important that the version information be set for 1.1.1. The 1.2 version of the Java Plug-in will not work with the Graphical Toolbox, unless you choose to include the Swing 1.0.3 JAR file in the **archive** statement.

Note: In this example, we have chosen to store the XML parser JAR file, **x4j400.jar**, on the web server. This is required only when you include your PDML file as part of your applet's installation. For performance reasons, you would normally *serialize* your panel definitions so that the Graphical Toolbox does not have to interpret the PDML at runtime. This greatly improves the performance of your user interface by creating compact binary representations of your panels. For more information see the description of "Generated files" on page 7.

Installing and running the applet: Install the applet on your favorite web server by performing the following steps:

- Compile **SampleApplet.java**.
- Create a JAR file named **MyGUI.jar** to contain the applet binaries. These include the class files produced when you compiled **SampleApplet.java** and **SampleBean.java**, the PDML file **MyGUI.pdml**, and the resource bundle **MyGUI.properties**.
- Copy your new JAR file to a directory of your choice on your web server. Copy the HTML files containing your online help into the server directory.
- Copy the Graphical Toolbox JAR files into the server directory.
- Finally, copy the HTML file **MyGUI.html** containing the imbedded applet into the server directory.

Now you are ready to run the applet. Point your web browser to **MyGUI.html** on the server. If you do not already have the Java Plug-in installed, you will be asked if you want to install it. Once the Plug-in is installed and the applet is started, your browser display should look similar to the following:

Tip: When testing your applets, ensure that you have removed the Graphical Toolbox jars from the CLASSPATH environment variable on your workstation. Otherwise, you will see error messages saying that the resources for your applet cannot be located on the server.

PDML tags and syntax

Panel Definition Markup Language (PDML) tags are case-sensitive and must be coded by defining attribute tags, for example, **<TITLE></TITLE>** and **<SIZE></SIZE>**, in a predefined, consecutive order.

In addition to a set of required tags, PDML defines optional tags that allow you to add functionality and formatting to your panels.

Required PDML tags

Just as the Hyper-Text Markup (HTML) and Standard Generalized Markup (SGML) languages require that you include certain tags within each HTML and SGML coded page (for example, <html>/</html>, <body>/</body>, and the base document element), the PDML tag structure requires a starting and ending PDML tag and a starting and ending panel tag, with title and size attribute tags enclosed within the panel tag.

The following is the general layout of all **required** PDML tags:

```
<PDML version=version source=platform basescreensize=screensize>        <PANELname=name>
                <SIZE> (page 39)size/SIZE> (page 39)        </PANEL></PDML>
```

Optional PDML tags

Just as HTML provides you with tags that format your web pages and make them interactive (for example, tables, radio buttons, and text fields), you can use PDML tags to format and activate your panels.

Three sets of PDML tags perform formatting and user interaction:

- Panel component tags create elements on a panel for accepting a user's request for a specific action to be performed (for example, a radio button), navigation (for example, a slider), and text formatting (for example, a table).
- Property sheet tags add hyperlinked icons to a panel that serve as entry points to other panels when the icons are clicked.
- Wizard tags present a series of consecutively ordered windows so that when the windows are followed in order, they aid the user in performing a specific task.

The following is an example of the general layout of all PDML tags, including representatives of panel component, wizard, and property sheet tags:

```
<PDML>
        <PANEL>
                <PANEL COMPONENT>                </PANEL COMPONENT>        </PANEL>
        <PROPSHEET>                <PANEL>
                        <PANEL COMPONENT>                                <PANEL COMPONENT>
        </PROPSHEET>        <WIZARD>                <PANEL>
                        <PANEL COMPONENT>                                <PANEL COMPONENT>
        </WIZARD></PDML>
```

[Legal | AS/400 Glossary]

PDML tag

The PDML tag is the basic Panel Definition Markup Language tag and, therefore, must be both the first and the last tag within a PDML document. The PDML tag takes the following form:

<PDML version=version **source=**platform **basescreensize=**screensize **</PDML>**		
NAME	**VALUE**	**DESCRIPTION**
<PDML>	*version*	The version of PDML used in the document; currently, 1.0 is the only version.
	platform	While other platforms may be supported in the future, you need to indicate one of the following two platforms as the platform on which the panels were originally defined: **Windows:** The PDML is assumed to have been generated by the Resource Script Converter tool. The logical panel coordinates contained in the PDML are interpreted as Windows dialog logical units. **Java:** The PDML is assumed to have been generated by the Graphical Toolbox, or created manually. The logical panel coordinates contained in the PDML will be interpreted as pixels.
	basescreensize	The width and height dimensions in pixels of the device on which the panels were originally defined, for example, “1024x768”.

[Legal | AS/400 Glossary]

Panel components

The panel component tags are optional user interface components that are used to enhance your Panel Definition Markup Language (PDML) files. The names you assign to the components must be unique within the scope of a given panel definition. Below are the definitions of the panel component tags:

- Action tag defines events that occur when the user makes a selection. (See button, checkbox, choice, combobox, item, and tree tags.)
- Button tag creates a pushbutton on the panel.
- Buttongroup tag defines a group of buttons, radio buttons, or check boxes on a panel, only one of which may be selected at a time.
- Checkbox tag creates a check box on the panel.
- Choice tag allows the user to make more than one choice from a drop down list. (See combobox tag.)
- Column tag defines the columns within a table. (See table tag.)
- Combobox tag creates a drop down list box on the panel. You may define the combo box as editable .
- Custom tag adds a user-defined AS/400 user interface component to a panel.
- Deselected tag helps define the user's choice. (See checkbox, combobox, item, list, radiobutton, and table tags.)
- Groupbox tag creates a labeled box on a panel.
- Image tag enables you to display an image on a panel.
- Item tag allows the user to make more than one choice from a list (See list tag.)
- Label tag creates a static text component on a panel.
- List tag creates a list box on a panel.
- Progressbar tag creates a progress indicator on a panel.
- Radiobutton tag creates a radio button on a panel.
- Root and node tag defines the hierarchy for a tree.
- Selected tag helps define the user's choice. (See checkbox, combobox, item, list, radiobutton, and table tags.)
- Slider tag enables you to create a horizontal or vertical adjustable slider control.
- Subpanel tags are tags that must be created within the scope of the panel tag and can be nested within each other. Subpanel tags cannot be defined in a standalone window or dialog. There are three types of subpanel tags:
-
  - Deckpane tag stacks, or "decks," a series of windows that can be opened one at a time.
  - Splitpane tag defines two windows separated by a graphical divider.
  - Tabbedpane tag enables you to browse windows through tabbed panes.
- Table tag creates a multi column table on a panel.
- Textarea tag creates an editable multi line text entry field on a panel.
- Textfield tag creates an editable text entry field on a panel.
- Tree tag displays an hierarchical tree on a panel.

[Legal | AS/400 Glossary]

Button tag

The following is the basic syntax of the button tag :

<BUTTON name=name>

where *name* identifies the button programmatically.

The button tag can be expanded with the following elements:

<BUTTON name=name> **<TITLE>***title*</TITLE>** **<LOCATION>***location*</LOCATION>** **<SIZE>***size*</SIZE>** **<STYLE>***style*</STYLE>** **<ACTION>***action*</ACTION>** **<FLYOVER>***helptext*</FLYOVER>** **<HELPALIAS>***component_name*</HELPALIAS>**		
ATTRIBUTE	**VALUE**	**DESCRIPTION**
<TITLE>	*title*	Defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the Panel Definition Markup Language (PDML) source file. Use the “%” character to make the character that immediately follows the “%” the “mnemonic” or “keyboard shortcut key” for the using a button. Use “%%” to use “%” as a literal character. This tag is required.
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel’s location from the top left corner in logical units. This tag is required.
<SIZE>	*size*	Expresses the width and height, respectively, of the component in logical units. This tag is required.
<STYLE>	*style*	Sets the style for the button’s appearance. *DEFAULT* status creates a button with a predefined appearance, and whose action is performed upon pressing the Enter key.

<ACTION>	*action*	Initiates the desired action once the button is clicked. Possible *action* values are: • Commit makes changes to the data objects on the panel by calling the save method for each object. Performing this action closes the window. • Cancel closes the panel window. No changes are saved. • Help accesses the help browser window. However, you may also define the *action* as the name of a handler class, which will receive control when the button is clicked. The class will use the EventHandler class and the java.awt.event.ActionListener interface. The framework will automatically create an object of the desired class and call its actionPerformed method.
<FLYOVER>	*helptext*	Creates a help window that hovers over the label. *Helptext* defines the key for loading the text string from the resource bundle. If the string is not found, the *helptext* string will appear as it does in the PDML source code.
<HELPALIAS>	*component_name*	An optional tag where the help topic should be displayed when context help is requested for this component.

[Legal | AS/400 Glossary]

Buttongroup tag

The following is the basic syntax of the buttongroup tag :

<RADIOBUTTON name=name>

where *name* identifies the button group programmatically.

The buttongroup tag can be expanded with the following elements:

<BUTTONGROUP name=name> **<DATACLASS>***class_name*</DATACLASS> **<ATTRIBUTE>***attribute_name*</ATTRIBUTE> **</BUTTONGROUP>**		
ATTRIBUTE	**VALUE**	**DESCRIPTION**
<DATACLASS>	*class_name*	Expresses the state of the button group.
<ATTRIBUTE>	*attribute_name*	Specifies whether or not the button has been selected. **GETTOR PROTOCOLS:** **String get<attribute_name>()** **where the return value is the name of the button component that should be selected** **SETTOR PROTOCOLS:** **void set<attribute_name>(String selected)** **where** *selected* **identifies the name of the button component that is currently selected.**

The buttongroup tag supports the radio button and check box components within the dataclass and attribute tags. However, the button group does not support the deselected tag for buttons.

[Legal | AS/400 Glossary]

Choice tag

The following is the basic syntax of the choice tag :

<CHOICE name= name>

where *name* identifies the choice programmatically.

The choice tag can be expanded with the following elements:

<CHOICE name=name> **<TITLE>***title*</TITLE> **<ENABLE></ENABLE> <DISABLE></DISABLE> <SHOW></SHOW> <HIDE></HIDE> <REFRESH></REFRESH> <DISPLAY></DISPLAY> </CHOICE>**		
ATTRIBUTE	**VALUE**	**DESCRIPTION**

<TITLE>	*title*	Inserted into the drop down list box. *Title* defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the PDML source file. This is a required tag.
<ENABLE>		See the action tag
<DISABLE>		See the action tag
<SHOW>		See the action tag
<HIDE>		See the action tag
<REFRESH>		See the action tag
<DISPLAY>		See the action tag

If one or more choice tags are specified, the choices gettor (page 27) is not called.

[Legal | AS/400 Glossary]

Combobox tag

The following is the basic syntax of the combobox tag:

<COMBOBOX name= name **editable =**“yes”|“no”>

where *name* identifies the combo box programmatically.

The combobox tag can be expanded with the following elements:

```
<COMBOBOX name=name>
<LOCATION>location</LOCATION>
<SIZE>size</SIZE>
<DATACLASS>class_name</DATACLASS>
<ATTRIBUTE>attribute_name</ATTRIBUTE>
<CHOICE></CHOICE>
<SHORT></SHORT>
<INTEGER></INTEGER>
<LONG></LONG>
<FLOAT></FLOAT>
<STRING></STRING>
<FORMAT></FORMAT>
<FLYOVER>helptext</FLYOVER>
<HELPALIAS>component_name</HELPALIAS>
</COMBOBOX>
```

ATTRIBUTE	VALUE	DESCRIPTION
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel’s location from the top left corner in logical units. This tag is required.

<SIZE>	*size*	Expresses the width and height, respectively, of the component in logical units. This tag is required.
<DATACLASS>	*class_name*	Supplies the data for the combobox using the DataBean interface.
<ATTRIBUTE>	*attribute_name*	Contains the data for the combobox. If the dataclass tag is present, the attribute tag must be used. **GETTOR PROTOCOLS:** where the return value either identifies the name of the drop down list choice that should be selected or supplies the text string to be inserted if the combo box is editable. **ChoiceDescriptor[] get<attribute_name>Choices()** where the return value is an array of *ChoiceDescriptor*'s, each of which describes a choice in the combo box's drop down list. If choice tags statically define the combo box element, the gettor is not necessary. **SETTOR PROTOCOLS:** **void set<attribute_name>(String selected)** where *selected* either identifies the name of the currently selected choice or the text string entered by the user if the combo box is editable.
<CHOICE>		See the choice tag.
<SHORT>		See the data format tags.
<LONG>		See the data format tags.
<FLOAT>		See the data format tags.
<STRING>		See the data format tags.

<FORMAT>		See the data format tag.
<FLYOVER>	*helptext*	Should be displayed when context help is requested for this component.
<HELPALIAS>	*component_name*	Should be displayed when context help is requested for this component.

As the panel designer, you may want a particular action performed when a combo box choice is selected.

[Legal | AS/400 Glossary]

Column tag

The following is the basic syntax of the column tag :

<COLUMN editable = “yes”|“no”>

The column tag can be expanded with the following elements:

```
<COLUMN name=name>
<TITLE>title</TITLE>
<DEFAULTWIDTH>width</DEFAULTWIDTH>
<DATACLASS>class_name</DATACLASS>
<ATTRIBUTE>attribute_name</ATTRIBUTE>
<ITEM></ITEM>
</COLUMN>
```

ATTRIBUTE	VALUE	DESCRIPTION
<TITLE>	*title*	Displays the column header. The *title* defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the Panel Definition Markup Language (PDML) source file. This tag is required.
<DEFAULTWIDTH>	*width*	Expresses the width of the column in logical units. If the tag is omitted, columns will have equal widths.
<DATACLASS>	*class_name*	Supplies the data for the column by using the DataBean interface.

<ATTRIBUTE>	*attribute_name*	Contains the data for the column. If the dataclass tag is used, the attribute tag must be used. **GETTOR PROTOCOLS:** **ItemDescriptor[] get<attribute_name>List()** where the return value is an array of *ItemDescriptor*'s, each of which describes an element in the column. If the item tag statically defines the column elements, the gettor does not need to be used. **String[] get<attribute_name>()Selection** where the return value is an array of names that identify the column elements that should be selected. If not used, items will not be selected when the table is displayed. **SETTOR PROTOCOLS:** **void set<attribute_nme>List(ItemDescriptor[] values)** where *values* is an array of *ItemDescriptor*'s, each of which describes an element in the column. **void set<attribute_name>(String[] items)Selection** where *items* is an array of names that identifies the column elements selected by the user.
<ITEM>		See the item tag.

[Legal | AS/400 Glossary]

Custom tag

The following is the basic syntax of the custom tag:

<CUSTOM name=name>

where *name* identifies the component programmatically.

The custom tag can be expanded with the following elements:

<CUSTOM name=name> **<LOCATION>***location*<**/LOCATION>** **<SIZE>***size*<**/SIZE>** **<MANAGERCLASS>***class_name*<**/MANAGERCLASS>** **</CUSTOM>**		
ATTRIBUTE	**VALUE**	**DESCRIPTION**
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel's location from the top left corner in logical units. This tag is required.
<SIZE>	*size*	Expresses the width and height, respectively, of the component in logical units. This tag is required.
<MANAGERCLASS>	*class_name*	Serves as the class name of the manager class which will manage the component on behalf of the AS/400 user interface framework, using the ComponentManager interface. This tag is required.

[Legal | AS/400 Glossary]

Deckpane tag

The following is the basic syntax of the deckpane tag :

<DECKPANE name=name **orientation=** "horizontal"|"vertical">

where *name* identifies the split pane programmatically.

The deckpane tag can be expanded with the following elements:

```
<DECKPANE name=name orientation="horizontal"|"vertical">
<TITLE>title<TITLE>
<LOCATION>location<LOCATION>
<PANE resource=resource_name name=panel_name type="deck">
</DECKPANE>
```

ATTRIBUTE	VALUE	DESCRIPTION
<TITLE>	*title*	Displays the text string as the contained tab. Therefore, the title tag is only needed when this subpanel is nested within a tabbed pane. *Title* defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the Panel Definition Markup Language (PDML) source file.
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel's location from the top left corner of the panel in logical units. The tag is required when the deck pane tag is declared within the scope of the panel tag.
<PANE> **source=** **name=** **type=**	*resource_name* Resource bundle name associated with the panel *panel_name* Name of the panel specified in the PDML document *type* Provides the AS/400 user interface framework with an indication of the pane type (panel, split, deck, or tab)	Identifies the panel for each window

The tabbed pane's largest constituent part determines the tabbed pane's size.

[Legal | AS/400 Glossary]

Data formatter class package

To validate data using the formatter class, refer to the package contents below:

AS400CharFormatter
AS400CnameFormatter
AS400CnameIBMFormatter
AS400Formatter
AS400NameFormatter
AS400NameIBMFormatter
AS400SQLNameColumnFormatter
AS400SQLNameFormatter
AS400SnameFormatter
AS400SnameIBMFormatter
DetailButtonHandler
MessageViewer
MessagesBean
ResourceLoader

[Legal | AS/400 Glossary]

Groupbox tag

The following is the basic syntax of the groupbox tag :

<GROUPBOX name= name>

where *name* identifies the group box programmatically.

The groupbox tag can be expanded with the following elements:

<GROUPBOX name=name>
<TITLE>*title*</TITLE>**
<LOCATION>*location*</LOCATION>**
<SIZE>*size*</SIZE>**
<HELPALIAS>*component_name*</HELPALIAS>**
</GROUPBOX>

ATTRIBUTE	VALUE	DESCRIPTION
<TITLE>	*title*	Displays the text string as the group box label. *Title* defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the Panel Definition Markup Language (PDML) source file. This tag is required.
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel's location from the top left corner in logical units. This tag is required.

<SIZE>	*size*	Expresses the width and height, respectively, of the component in logical units. This tag is required.
<HELPALIAS>	*component_name*	Should be displayed when context help is requested for this component.

[Legal | AS/400 Glossary]

Image tag

The following is the basic syntax of the image tag:

<IMAGE name=name>

where *name* identifies the image programmatically.

The image tag can be expanded with the following elements:

<IMAGE name=name> **<TITLE>***title***</TITLE>** **<LOCATION>***location***</LOCATION>** **<SIZE>***size***</SIZE>** **<FLYOVER>***helptext***<FLYOVER>** **</IMAGE>**		
ATTRIBUTE	**VALUE**	**DESCRIPTION**
<TITLE>	*title*	Creates the resource key of the filename string in the panel's resource bundle. The resource bundle identifies the binary file containing the image to be displayed. This tag is required.
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel's location from the top left corner in logical units. This tag is required.
<SIZE>	*size*	Expresses the width and height, respectively, of the image in logical units. If a size is specified, the size defined in the image file will be overridden and the image will be scaled to the size specified.

<FLYOVER>	*helptext*	Creates a help window that hovers over the label. *Helptext* defines the key for loading the text string from the resource bundle. If the string is not found, the *helptext* string will appear as it does in the Panel Definition Markup Language (PDML) source file.

[Legal | AS/400 Glossary]

Item tag

The following is the **basic syntax** of the item tag :

<ITEM name= name>

where *name* identifies the item programmatically.

The item tag can be expanded with the following elements:

```
<ITEM name=name>
<TITLE>title</TITLE>
<ENABLE></ENABLE>
<DISABLE></DISABLE>
<SHOW></SHOW>
<HIDE></HIDE>
<REFRESH></REFRESH>
<DISPLAY></DISPLAY>
</ITEM>
```

ATTRIBUTE	VALUE	DESCRIPTION
<TITLE>	*title*	Displays the text string in the listbox. *Title* defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the Panel Definition Markup Language (PDML) source file. This tag is required.
<ENABLE>		See the action tag.
<DISABLE>		See the action tag.
<SHOW>		See the action tag.
<HIDE>		See the action tag.
<REFRESH>		See the action tag.
<DISPLAY>		See the action tag.

If one or more item tags are defined, the list gettor/settor will not be used. 

[Legal | AS/400 Glossary]

Label tag

The following is the basic syntax of the label tag:

<**LABEL name**=name>

where *name* identifies the label programmatically.

The label tag can be expanded with the following elements:

<LABEL name=name> **<TITLE>***title*</**TITLE>** **<LOCATION>***location*</**LOCATION>** **<SIZE>***size*</**SIZE>** **<FLYOVER>***helptext*</**FLYOVER>** **<HELPALIAS>***component_name*</**HELPALIAS>** </**LABEL>**		
ATTRIBUTE	**VALUE**	**DESCRIPTION**
<TITLE>	*title*	Displays the text string as static text. *Title* defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the Panel Definition Markup Language (PDML) source file. This tag is required.
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel's location from the top left corner of the panel in logical units. This tag is required.
<SIZE>	*size*	Expresses the width and height, respectively, of the label in logical units. This tag is required.
<DATACLASS>	*class_name*	Supplies the data for the label, using the DataBean interface.

<ATTRIBUTE>	*attribute_name*	Contains the data for the label. **GETTOR PROTOCOLS:** **String get<attribute_name>()** where the return value is the string to be inserted into the text field. **SETTOR PROTOCOLS:** **void set<attribute_name>(String text)** where *text* is the string that was extracted from the text field.
<FLYOVER>	*helptext*	Creates a help window that hovers over the label. *Helptext* defines the key for loading the text string from the resource bundle. If the string is not found, the *helptext* string will appear as it does in the PDML source file.
<HELPALIAS>	*component_name*	Should be displayed when context help is requested for this component.

[Legal | AS/400 Glossary]

List tag

The following is the basic syntax of the list tag :

<LIST name=name **selection=**“single” | “single interval” | “multi-interval”>

where *name* identifies the list box programmatically.

The list tag can be expanded with the following elements:

```
<LIST name=name>
<LOCATION>location</LOCATION>
<SIZE>size</SIZE>
<DATACLASS>class_name</DATACLASS>
<ATTRIBUTE>attribute_name</ATTRIBUTE>
<ITEM></ITEM>
<SELECTED>selected</SELECTED>
<DESELECTED>deselected</DESELECTED>
<DOUBLECLICK>class_name</DOUBLECLICK>
<FLYOVER>helptext</FLYOVER>
<HELPALIAS>component_name</HELPALIAS>
</LIST>
```

ATTRIBUTE	VALUE	DESCRIPTION
<LOCATION>	*location*	Expresses the x- and y-coordinates of the list box's location from the top left corner in logical units. This tag is required.
<SIZE>	*size*	Expresses the width and height, respectively, of the component in logical units. This tag is required.
<DATACLASS>	*class_name*	Supplies the data for the list box.

<ATTRIBUTE>	*attribute_name*	Supplies the data for the list box. If the dataclass tag is used, the attribute tag must be used. **GETTOR PROTOCOLS:** **ItemDescriptor[] get<attribute_name>List()** where the return value is an array of *ItemDescriptors*, each of which describes an item in the list. If the item tag defines the listbox elements statically, the gettor is not necessary. **String[] get<attribute_name>()Selection** where the return value is an array of names which identify the list items that should be selected. If not implemented, no list items will be selected when the list is displayed. **SETTOR PROTOCOLS:** **void set<attribute_name>List(ItemDescriptor[] values)** where *values* is an array of *ItemDescriptor*'s, each of which describes an item in the list. **void set<attribute_name>(String[] items)Selection** where *items* is an array of names which identify the list items selected by the user.
<ITEM>		See the item tag.
<SELECTED>		See the selected tag.
<DESELECTED>		See the deselected tag.
<DOUBLECLICK>	*class_name*	Notifies the DoubleClickListener class when the user double-clicks on an entry in the list.

<FLYOVER>	*helptext*	Should be displayed when context help is requested for this component. The *helptext* defines the key for loading the text string from the resource bundle. If the string is not found, the *helptext* string will appear as it does in the Panel Definition Markup Language (PDML) source file.
<HELPALIAS>	*component_name*	Should be displayed when context help is requested for this component.

[Legal | AS/400 Glossary]

Panel tag

The panel tag creates the actual panel within a Panel Definition Markup Language (PDML) document. More than one panel tag can be included within a document, depending on how many panels need to be created. For the form of the panel tag within the scope of an entire document, see the PDML syntax. The panel tag, itself, takes the following form:

<PANEL name=*name***>**

where *name* is the name of the panel. The name of the panel must be unique to the document it is found in because the name is used to identify the panel programmatically.

The panel tag can be expanded upon with the following elements:

<PANEL name=*name***>** **<TITLE>***title***</TITLE>** **<SIZE>***size***</SIZE>** **<ICON>***icon***</ICON>** **<ACTIVATE>***class_name***</ACTIVATE>** **</PANEL>**		
NAME	**VALUE**	**DESCRIPTION**
<TITLE>	*title*	Displays the text string in the title bar of the panel.
<SIZE>	*size*	Expresses the width and height, respectively, of the panel in logical units.
<ICON>	*icon*	Provides the width and height, respectively, in logical units of the panel.

<ACTIVATE>	*class_name*	Serves as the qualified name of a handler class that receives control when the panel is initially displayed. The class must extend the EventHandler class, and implement the interface java.awt.event.ActionListener. The framework automatically creates an object of the specified class, and calls its actionPerformed method.

[Legal | AS/400 Glossary]

Progressbar tag

The following is the basic syntax of the progressbar tag :

<PROGRESSBAR name= name>

where *name* identifies the progress bar programmatically.

The progressbar tag can be expanded with the following elements:

```
<PROGRESSBAR name=name>
<LOCATION>location</LOCATION>
<SIZE>size</SIZE>
<MINVALUE>value</MINVALUE>
<MAXVALUE>value</MAXVALUE>
<FLYOVER>helptext<FLYOVER>
<HELPALIAS>component_name</HELPALIAS>
```

ATTRIBUTE	VALUE	DESCRIPTION
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel's location from the top left corner in logical units. This tag is required.
<SIZE>	*size*	Expresses the width and height, respectively, of the component in logical units. This tag is required.
<MINVALUE>	*value*	Defines the lowest value on the progress bar. This tag is required.
<MAXVALUE>	*value*	Defines the highest value on the progress bar. This tag is required.

<FLYOVER>	*helptext*	Should be displayed when context help is requested for this component. *Helptext* defines the key for loading the text string from the resource bundle. If the string is not found, the *helptext* string will appear as it does in the Panel Definition Markup Language (PDML) source file.
<HELPALIAS>	*component_name*	Should be displayed when context help is requested for this component.

The progress bar component does not exchange data with its associated data object at predefined times. Instead, changes to the progress bar must be made by a handler object with another component on the panel, or with the activation handler for the panel. See AS/400 user interface framework.

[Legal | AS/400 Glossary]

Radiobutton tag

The following is the basic syntax of the radiobutton tag :

<RADIOBUTTON name= name>

where *name* identifies the radio button programmatically.

The radiobutton tag can be expanded with the following elements:

```
<RADIOBUTTON name=name>
<TITLE>title</TITLE>
<LOCATION>location</LOCATION>
<SIZE>size</SIZE>
<DATACLASS>class_name</DATACLASS>
<ATTRIBUTE>attribute_name</ATTRIBUTE>
<SELECTED>selected</SELECTED>
<DESELECTED>deselected</DESELECTED>
<FLYOVER>helptext</FLYOVER>
<HELPALIAS>component_name</HELPALIAS>
</RADIOBUTTON>
```

ATTRIBUTE	VALUE	DESCRIPTION

<TITLE>	*title*	Displays the text string as the radio button label The *title* defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the Panel Definition Markup Language (PDML) source code. Use the “%” character to make the character that immediately follows the “%7quot; the ”mnemonic“ or ”keyboard shortcut key“; for the using a button. Use ”%%“ to use ”%“ as a literal character. This tag is required.
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel’s location from the top left corner in logical units. This tag is required.
<SIZE>	*size*	Expresses the width and height, respectively, of the component in logical units. This tag is required.
<DATACLASS>	*class_name*	Expresses whether the radio button is selected or not.

<ATTRIBUTE>	*attribute_name*	Contains the state of the radio button **GETTOR PROTOCOLS** **The gettor does not have to be used if the choice tag statically defines the combo box elements.** **boolean is<attribute_name>()** where the return value indicates whether the radio button is selected or not. **SETTOR PROTOCOLS** **void set<attribute_name>(boolean selected)** where *selected* indicates whether the radio button is selected or not.
<SELECTED>		Please see the selected tag.
<DESELECTED>		Please see the deselected tag.
<FLYOVER>	*helptext*	Should be displayed when context help is requested for this component. The *helptext* defines the key for loading the text string from the resource bundle. If the string is not found, the *helptext* string will appear as it does in the PDML source code.
<HELPALIAS>	*component_name*	Should be displayed when context help is requested for this component.

[Legal | AS/400 Glossary]

Root and node tags

The following is the basic syntax of the root and node tags :

<ROOT|NODE name= name>

where *name* identifies the node programmatically.

The root and node tags can be expanded with the following elements:

```
<ROOT|NODE name=name>
<TITLE>title</TITLE>
<ICON>image</ICON>
<ENABLE></ENABLE>
<DISABLE></DISABLE>
<SHOW></SHOW>
<HIDE></HIDE>
<REFRESH></REFRESH>
<DISPLAY></DISPLAY>
<NODE></NODE>
</ROOT|NODE>
```

ATTRIBUTE	VALUE	DESCRIPTION
<TITLE>	*title*	Displays the text string in a tree. *Title* defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the Panel Definition Markup Language (PDML) source file. This tag is required.
<ICON>	*image*	Identifies the filename of the image file to be displayed to the left of the item's text string. This tag is required.
<ENABLE>		See the actions tag.
<DISABLE>		See the actions tag.
<SHOW>		See the actions tag.
<HIDE>		See the actions tag
<REFRESH>		See the actions tag.
<DISPLAY>		See the actions tag.
<NODE>		A child node that may be coded, and the nodes may be nested to any depth.

When using the root tag, do not use gettor/settor methods.

[Legal | AS/400 Glossary]

Slider tag

The following is the basic syntax of the slider tag :

<SLIDER name=name **orientation=**"horizontal"|"vertical">

where *name* identifies the text area programmatically.

The slider tag can be expanded with the following elements:

```
<SLIDER name=name orientation="horizontal"|"vertical">
<LOCATION>location</LOCATION>
<SIZE>size</SIZE>
<DATACLASS>class_name</DATACLASS>
<ATTRIBUTE>attribute_name</ATTRIBUTE>
<MINVALUE>value</MINVALUE>
<MAXVALUE>value</MAXVALUE>
<MAJORTICKS>interval</MAJORTICKS>
<MINORTICKS>interval</MINORTICKS>
<ADJUST>class_name</ADJUST>
<FLYOVER>helptext</FLYOVER>
<HELPALIAS>component_name</HELPALIAS>
</SLIDER>
```

ATTRIBUTE	VALUE	DESCRIPTION
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel's location from the top left corner in logical units. This tag is required.
<SIZE>	*size*	Expresses the width and height, respectively, of the component in logical units. This tag is required.
<DATACLASS>	*class_name*	Supplies data displayed on the slider, using the DataBean interface.
<ATTRIBUTE>	*attribute_name*	Contains the data for the slider. If the dataclass tag is present, the attribute tag must be present. **GETTOR PROTOCOLS:** **int get<attribute_name>()** where the return value is the integer setting for the slider. The value must be greater than or equal to minvalue and less than or equal to maxvalue. **SETTOR PROTOCOLS:** **void set<attribute_name>(boolean selected)** where *value* is the current integer setting for the slider.

<MINVALUE>	*value*	Defines the lowest value on the slider. This tag is required.
<MAXVALUE>	*value*	Defines the highest value on the slider. This tag is required.
<MAJORTICKS>	*interval*	Marks the slider in major increments. When the tag is included, snap-to-ticks is automatically on.
<MINORTICKS>	*interval*	Sets unlabelled minor-markers on the slider. When the tag is included, snap-to-ticks is automatically on.
<ADJUST>	*class_name*	Allows the user to adjust the slider. The class must extend the EventHandler class and use the interface com.sun.java.swing.event.ChangeListener. The framework will automatically create an object of the specified class and call its stateChanged method.
<FLYOVER>	*helptext*	Should be displayed when context help is requested for this component. *Helptext* defines the key for loading the text string from the resource bundle. If the string is not found, the *helptext* string will appear as it does in the Panel Definition Markup Language (PDML) source file.
<HELPALIAS>	*component_name*	Should be displayed when context help is requested for this component.

[Legal | AS/400 Glossary]

Splitpane tag

The following is the basic syntax of the splitpane tag :

<SPLITPANE name=name **orientation=** “horizontal”|“vertical”>

where *name* identifies the split pane programmatically.

The splitpane tag can be expanded with the following elements:

```
<SPLITPANE name=name orientation= "horizontal"|"vertical">
<TITLE>title</TITLE>
<LOCATION>location</LOCATION>
<PANE resource=resource_name name=panel_name type="split">
</SPLITPANE>
```

ATTRIBUTE	VALUE	DESCRIPTION
<TITLE>	*title*	Displays the text string as the containing tab. Therefore, this tag is used only when this subpanel is nested within a tabbed pane. *Title* defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the Panel Definition Markup Language (PDML) source file.
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel's location from the top left corner of the panel in logical units. This tag is required when you define a split pane within the scope of the panel tag.
<PANE> **resource=** **name=** **type=**	*resource_name* where is the resource bundle name associated with the panel *panel_name* where is the name of the panel specified in the PDML document *type* where provides the AS/400 user interface framework with an indication of the pane type (panel, split, deck, or tab)	Identifies the panel for each window.

The size of a split pane is set by the size of the panels that the pane is included in. When the split pane is displayed, the panels appear in the order declared within the splitpane tag.

[Legal | AS/400 Glossary]

Tabbedpane tag

The following is the basic syntax of the tabbedpane tag :

<TABBEDPANE name=name **orientation=** “horizontal”|“vertical”>

where *name* identifies the tabbed pane programmatically.

The tabbedpane tag can be expanded with the following elements:

```
<TABBEDPANE name=name placement="top"|"bottom"|"left"|"right">
<TITLE>title</TITLE>
<LOCATION>location</LOCATION>
<PANE resource=resource_name name=panel_name type="tab">
</TABBEDPANE>
```

ATTRIBUTE	VALUE	DESCRIPTION
<TITLE>	*title*	Displays the text string as the containing tab. Therefore, the tag is only necessary when you are nesting one tabbed pane within another tabbed pane. *Title* defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the Panel Definition Markup Language (PDML) source file.
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel’s location from the top left corner of the panel in logical units. This tag is only necessary when the tabbed pane is declared within the scope of the panel tag.

<PANE> **name=** **placement=** **type=**	*resource_name* where is the resource bundle name associated with the panel *panel_name* where is the name of the panel specified in the PDML document *type* where provides the AS/400 user interface framework with an indication of the pane type (panel, split, deck, or tab)	Identifies the panel for each window.

When the tabbed pane is displayed, the tabbed panels appear in the order declared within the tabbedpane tag.

[Legal | AS/400 Glossary]

Table tag

The following is the basic syntax of the table tag :

<TABLE name=name **selection=**“single” | “singleinterval” | “multi-interval”>

where *name* identifies the table programmatically.

The table tag can be expanded with the following elements:

```
<TABLE name=name selection="single" | "singleinterval" | "multi-interval">
<LOCATION>location</LOCATION>
<SIZE>size</SIZE>
<COLUMN></COLUMN>
<SELECTED></SELECTED>
<DESELECTED></DESELECTED>
<DOUBLECLICK>class_name</DOUBLECLICK>
<FLYOVER>helptext</FLYOVER>
<HELPALIAS>component_name</HELPALIAS>
</TABLE>
```

ATTRIBUTE	VALUE	DESCRIPTION
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel’s location from the top left corner in logical units. This tag is required.
<SIZE>	*size*	Expresses the width and height, respectively, of the component in logical units. This tag is required.

<COLUMN>		See the column tag.
<SELECTED>		See the selected tag.
<DESELECTED>		See the deselected tag.
<DOUBLECLICK>	*class_name*	Signals the DoubleClickListener interface class when the user double-clicks on an entry in the table.
<FLYOVER>	*helptext*	Should be displayed when context help is requested for this component. *Helptext* defines the key for loading the text string from the resource bundle. If the string is not found, the *helptext* string will appear as it does in th Panel Definition Markup Language (PDML) source file.
<HELPALIAS>	*component_name*	Should be displayed when context help is requested for this component.

[Legal | AS/400 Glossary]

Tree tag

The following is the **basic syntax** of the tree tag :

<TREE name= name>

where *name* identifies the tree programmatically.

The tree tag can be expanded with the following elements:

```
<TREE name=name>
<LOCATION>location</LOCATION>
<SIZE>size</SIZE>
<DATACLASS>class_name</DATACLASS>
<ATTRIBUTE>attribute_name</ATTRIBUTE>
<ROOT></ROOT>
<DOUBLCLICK>class_name</DOUBLECLICK>
<FLYOVER>helptext</FLYOVER>
<HELPALIAS>component_name</HELPALIAS>
</TREE>
```

ATTRIBUTE	VALUE	DESCRIPTION
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel's location from the top left corner in logical units. This tag is required.

<SIZE>	*size*	Expresses the width and height, respectively, of the component in logical units. This tag is required.
<DATACLASS>	*class_name*	Supplies the data for the tree, using the DataBean interface.
<ATTRIBUTE>	*attribute_name*	Supplies the data for the tree. The tag must be present if the dataclass tag is used. **SETTOR PROTOCOL: The settor method is always called first to identify which parent node child elements are to be supplied.** **void set<attribute_name>TreeParent(String parent)** where *parent* is the name of the parent node for which children should be supplied when the gettor is called. **GETTOR PROTOCOL:** **NodeDescriptor[] get<attribute_name>Children()** where the return value is an array of *NodeDescriptor*'s, each of which describes a child element in the tree. If the node tags statically define the tree elements, the gettor is not necessary.
<ROOT>		See the root tag.
<DOUBLECLICK>	*class_name*	Notifies the listener class when the user double-clicks on an item in the tree, using the DoubleClickListener interface.

<FLYOVER>	*helptext*	Should be displayed when context help is requested for this component. *Helptext* defines the key for loading the text string from the resource bundle. If the string is not found, the *helptext* string will appear as it does in the Panel Definition Markup Language (PDML) source file.
<HELPALIAS>	*component_name*	Should be displayed when context help is requested for this component.

At this time, user-modifiable trees are not supported, nor are selection gettors/settors.

[Legal | AS/400 Glossary]

Textarea tag

The following is the basic syntax of the textarea tag :

<TEXTAREA name=name>

where *name* identifies the text area programmatically.

The textarea tag can be expanded with the following elements:

```
<TEXTAREA name=name>
<TITLE>title</TITLE>
<LOCATION>location</LOCATION>
<SIZE>size</SIZE>
<DATACLASS>class_name</DATACLASS>
<ATTRIBUTE>attribute_name</ATTRIBUTE>
<SHORT></SHORT>
<INTEGER></INTEGER>
<LONG></LONG>
<FLOAT></FLOAT>
<STRING></STRING>
<FORMAT></FORMAT>
<FLYOVER>helptext</FLYOVER>
<HELPALIAS>component_name</HELPALIAS>
</TEXTAREA>
```

ATTRIBUTE	VALUE	DESCRIPTION
<TITLE>	*title*	Displays the text string in the text area label. *Title* defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the Panel Definition Markup Language (PDML) source file.

<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel's location from the top left corner in logical units. This tag is required.
<SIZE>	*size*	Expresses the width and height, respectively, of the component in logical units. This tag is required.
<DATACLASS>	*class_name*	Supplies the data for the text area, using the DataBean interface.
<ATTRIBUTE>	*attribute_name*	Contains the data for the text area. If the dataclass tag is used, the attribute tag must be used. **GETTOR PROTOCOLS:** **String get<attribute_name>()** where the return value is the string to be inserted into the text area. **SETTOR PROTOCOLS:** **void set<attribute_name>(String text)** where *text* is the string that was extracted from the text area.
<SHORT>		See the data format tags.
<INTEGER>		See the data format tags.
<LONG>		See the data format tags.
<FLOAT>		See the data format tags.
<STRING>		See the data format tags.
<FORMAT>		See the data format tags.
<FLYOVER>	*helptext*	Should be displayed when context help is requested for this component.
<HELPALIAS>	*component_name*	Should be displayed when context help is requested for this component.

[Legal | AS/400 Glossary]

Textfield tag

The following is the basic syntax of the textfield tag :

<TEXTFIELD name=name>

where *name* identifies the text field programmatically.

The textfield tag can be expanded with the following elements:

```
<TEXTFIELD name=name>
<TITLE>title</TITLE>
<LOCATION>location</LOCATION>
<SIZE>size</SIZE>
<DATACLASS>class_name</DATACLASS>
<ATTRIBUTE>attribute_name</ATTRIBUTE>
<SHORT></SHORT>
<INTEGER></INTEGER>
<LONG></LONG>
<FLOAT></FLOAT>
<STRING></STRING>
<FORMAT></FORMAT>
<FLYOVER>helptext</FLYOVER>
<HELPALIAS>component_name</HELPALIAS>
</TEXTFIELD>
```

ATTRIBUTE	VALUE	DESCRIPTION
<TITLE>	*title*	Displays the text string in the text field label. *Title* defines the key for loading the text string from the resource bundle. If the string is not found, the *title* string will appear as it does in the Panel Definition Markup Language (PDML) source file.
<LOCATION>	*location*	Expresses the x- and y-coordinates of the panel's location from the top left corner in logical units. This tag is required.
<SIZE>	*size*	Expresses the width and height, respectively, of the component in logical units. This tag is required.
<DATACLASS>	*class_name*	Supplies the data for the text field, using the DataBean interface.

<ATTRIBUTE>	*attribute_name*	Contains the data for the text field. **GETTOR PROTOCOLS:** **String get<attribute_name>()** where the return value is the string to be inserted into the text field. **SETTOR PROTOCOLS:** **void set<attribute_name>(String text)** where *text* is the string that was extracted from the text field.
<SHORT>		See the data format tags.
<INTEGER>		See the data format tags.
<LONG>		See the data format tags.
<FLOAT>		See the data format tags.
<STRING>		See the data format tags.
<FORMAT>		See the data format tags.
<FLYOVER>	*helptext*	Should be displayed when context help is requested for this component. *Helptext* defines the key for loading the text string from the resource bundle. If the string is not found, the *helptext* string will appear as it does in the PDML source file.
<HELPALIAS>	*component_name*	should be displayed when context help is requested for this component.

[Legal | AS/400 Glossary]

Property Sheets

The propsheet tag creates the window that is loaded when you click on a particular icon. The AS/400 user interface framework automatically sets the correct platform look and feel for the propsheets, just as it does for individual panels. The following is the basic syntax of the propsheet tag:

<PROPSHEET name=*name***>** **<TITLE>***title***</TITLE>** **<ICON>***icon***</ICON>** **<Page resource=***resource_name* **name=***panel_name***>** **</PROPSHEET>**		
ATTRIBUTE	**VALUE**	**DESCRIPTION**
<TITLE>	*title*	Supplies the title bar for the property sheet window. If the tag is omitted, the title should be supplied using the setTitle method of the PropertySheetManager class.
<ICON>	*icon*	Serves as the resource key that identifies the image to be used as the icon for the property sheet. If you omit this tag, the standard Java window icon will be used.
<PAGE> **resource=** **name=**	*resource_name* the resource bundle name associated with the panel *panel_name* the name of the panel specified in the Panel Definition Markup Language (PDML) document	The tag that identifies each panel within the property sheet page

[Legal | AS/400 Glossary]

Chapter 2. Program Call Markup Language

Overview

Program Call Markup Language (PCML) is a tag language that helps you call AS/400 programs, but with writing less Java code. PCML is based upon the Extensible Markup Language (XML), a tag syntax you write to describe the input and output parameters for AS/400 programs. PCML enables you to define tags that fully describe AS/400 programs that will be called by your Java application.

Benefits

Ordinarily, in the Java environment, you have to write additional lines of code in your Java applications to construct AS/400 Toolbox for Java class objects for connecting to and retrieving information from an AS/400 and for performing the appropriate data translation.

However, by using PCML, your calls to the AS/400 with the AS/400 Toolbox for Java classes are automatically handled by the PCML class objects. The PCML class objects are generated from the PCML tags, the PCML-coded description of AS/400 programs, helping minimize the amount of code you need to write in order to call AS/400 programs from your application.

You use the AS/400 distributed program call (DPC) server, an established generalized server, to support your remote requests to call programs on an AS/400.

Platform requirements

While PCML was designed to support distributed program calls to AS/400 program objects from a Java platform, you can also use PCML to make calls to an AS/400 program from within an AS/400 environment as well.

Topics for more information

Refer to the following topics on how to use PCML:

- Call programs with the help of PCML
- Build program calls with PCML tags
- A PCML example

[Legal | AS/400 Glossary]

Building AS/400 program calls with PCML

To build AS/400 program calls with PCML, you must start by creating the following:

- Java application
- PCML source file

Depending upon your design process, you must write one or more PCML source files where you describe the interfaces to the AS/400 programs that will be called by your Java application. Refer to PCML syntax for a detailed description of the language.

Then, your Java application, shown in yellow in Figure 1 below, interacts with the ProgramCallDocument class. The ProgramCallDocument class uses your PCML source file to pass information between your Java application and the AS/400 programs.

Figure 1. Making program calls to the AS/400 using PCML.

When your application constructs the ProgramCallDocument object, the IBM XML parser reads and parses the PCML source file.

After the ProgramCallDocument class has been created, the application program uses the ProgramCallDocument class's methods to retrieve the necessary information from the AS/400 through the AS/400 distributed program call (DPC) server.

To increase run-time performance, the ProgramCallDocument class can be serialized during your product build time. The ProgramCallDocument is then constructed using the serialized file. In this case, the IBM XML parser is not used at run-time. Refer to "Using serialized PCML files".

Using PCML source files

Your Java application uses PCML by constructing a ProgramCallDocument with a reference to the PCML source file. The ProgramCallDocument considers the PCML source file to be a Java resource. Consequently, the PCML source file is found using the Java CLASSPATH.

The following Java code constructs a ProgramCallDocument:

```
AS400 as400 = new AS400();
ProgramCallDocument pcmlDoc = new ProgramCallDocument(as400, "myPcmlDoc");
```

The ProgramCallDocument will look for your PCML source in a file called `myPcmlDoc.pcml`. Notice that the `.pcml` extension is not specified on the constructor.

If you are developing a Java application in a Java "package," you can package-qualify the name of the PCML resource:

```
AS400 as400 = new AS400();
ProgramCallDocument pcmlDoc = new ProgramCallDocument(as400, "com.company.package.myPcmlDoc");
```

Using serialized PCML files

To increase run-time performance, you can use a serialized PCML file. A serialized PCML file contains serialized Java objects representing the PCML. The objects that are serialized are the same objects that are created when you construct the ProgramCallDocument from a source file as described above.

Using serialized PCML files gives you better performance because the IBM XML parser is not needed at run-time to process the PCML tags.

The PCML can be serialized using either of the following methods:

- From the command line:

```
java com.ibm.as400.ProgramcallDocument -serialize mypcml
```

  This method is helpful for having batch processes to build your application.

- From within a Java program:

```
ProgramCallDocument pcmlDoc; // Initialized elsewhere
pcmlDoc.serialize();
```

If your PCML is in a source file named `myDoc.pcml`, the result of serialization is a file named `myDoc.pcml.ser`.

PCML source files vs. serialized PCML files

Consider the following code to construct a ProgramCallDocument:

```
AS400 as400 = new AS400();
ProgramCallDocument pcmlDoc = new ProgramCallDocument(as400, "com.mycompany.mypackage.myPcml
```

The ProgramCallDocument constructor will first try to find a serialized PCML file named **myPcmlDoc.pcml.ser** in the **com.mycompany.mypackage** package in the Java CLASSPATH. If a serialized PCML file does not exist, the constructor will then try to find a PCML source file named **myPcmlDoc.pcml** in the **com.mycompany.mypackage** package in the Java CLASSPATH. If a PCML source file does not exist, an exception is thrown.

Qualified names

Your Java application uses the **ProgramCallDocument.setValue()** method to set input values for the AS/400 program being called. Likewise, your application uses the **ProgramCallDocument.getValue()** method to retrieve output values from the AS/400 program.

When accessing values from the ProgramCallDocument, you must specify the fully qualified name of the document element or <**data**> tag. The qualified name is a concatenation of the names of all the containing tags with each name separated by a period.

For example, given the following PCML source, the qualified name for the **"nbrPolygons"** item is **"polytest.parm1.nbrPolygons"**. The qualified name for accessing the **"x"** value for one of the points in one of the polygons is **"polytest.parm1.polygon.point.x"**.

If any one of the elements needed to make the qualified name is unnamed, all descendents of that element do not have a qualified name. Any elements that do not have a qualified name cannot be accessed from your Java program.

```
<pcml version="1.0">
  <program name="polytest" path="/QSYS.lib/MYLIB.lib/POLYTEST.pgm">
    <!– Parameter 1 contains a count of polygoins along with an array of polygons –>
    <struct name="parm1" usage="inputoutput">
      <data name="nbrPolygons" type="int" length="4" init="5" />
      <!– Each polygon contains a count of the number of points along with an array of points –>
      <struct name="polygon" count="nbrPolygons">
        <data name="nbrPoints" type="int" length="4" init="3" />
        <struct name="point" count="nbrPoints" >
          <data name="x" type="int" length="4" init="100" />
          <data name="y" type="int" length="4" init="200" />
```

```
        </struct>
      </struct>
    </struct>
  </program>
</pcml>
```

Accessing data in arrays

Any <**data**> or <**struct**> element can be defined as an array using the **count** attribute. Or, a <**data**> or <**struct**> element can be contained within another <**struct**> element that is defined as an array.

Furthermore, a <**data**> or <**struct**> element can be in a multidimensional array if more than one containing element has a **count** attribute specified.

In order for your application to set or get values defined as an array or defined within an array, you must specify the array index for each dimension of the array. The array indices are passed as an array of **int** values. Given the source for the array of polygons shown above, the following Java code can be used to retrieve the information about the polygons:

```
ProgramCallDocument polytest; // Initialized elsewhere
Integer nbrPolygons, nbrPoints, pointX, pointY;
nbrPolygons = (Integer) polytest.getValue("polytest.parm1.nbrPolygons");
System.out.println("Number of polygons:" + nbrPolygons);
indices = new int[2];
for (int polygon = 0; polygon < nbrPolygons.intValue(); polygon++)
{
    indices[0] = polygon;
    nbrPoints = (Integer) polytest.getValue("polytest.parm1.polygon.nbrPoints", indices );
    System.out.println("  Number of points:" + nbrPoints);
    for (int point = 0; point < nbrPoints.intValue(); point++)
    {
        indices[1] = point;
        pointX = (Integer) polytest.getValue("polytest.parm1.polygon.point.x", indices );
        pointY = (Integer) polytest.getValue("polytest.parm1.polygon.point.y", indices );
        System.out.println("    X:" + pointX + " Y:" + pointY);
    }
}
```

Debugging

When you use PCML to call programs with complex data structures, it is easy to have errors in your PCML that result in exceptions from the ProgramCallDocument class. If the errors are related to incorrectly describing offsets and lengths of data, the exceptions can be difficult to debug.

The **com.ibm.as400.data.PcmlMessageLog** class allows you to turn on a tracing function that prints to the standard output stream information that can be helpful in problem determination. You can call the following method to turn the tracing function on:

```
com.ibm.as400.data.PcmlMessageLog.setTraceEnabled(true);
```

When the tracing function is turned on, the following types of information are printed to the standard output stream:

- A dump of the hexadecimal data being transferred between the Java application and the AS/400 program. This shows the program input parameters after

character data is converted to EBCDIC and integers are converted to big-endian. It also shows the output parameters before they are converted to the Java environment.

The data is shown in a typical hexadecimal dump format with hexadecimal digits on the left and a character interpretation on the right. The following is an example of this dump format:

```
qgyolobj[6]
Offset : 0....... 4....... 8....... C....... 0....... 4....... 8....... C.......   0...4...8..
     0 : 5CE4E2D9 D7D9C640 4040                                                    **USRPRF
```

In the above example, the dump shows the seventh parameter has 10 bytes of data set to "*USRPRF ".

- For output parameters, following the hexadecimal dump is a description of how the data has been interpreted for the document.

```
/QSYS.lib/QGY.lib/QGYOLOBJ.pgm[2]
Offset : 0....... 4....... 8....... C....... 0....... 4....... 8....... C.......   0...4...8..
     0 : 0000000A 0000000A 00000001 00000068 D7F0F9F9 F0F1F1F5 F1F4F2F6 F2F5F400  *...........
    20 : 00000410 00000001 00000000 00000000 00000000 00000000 00000000 00000000  *...........
    40 : 00000000 00000000 00000000 00000000                                      *...........
Reading data – Offset: 0 Length: 4 Name: "qgyolobj.listInfo.totalRcds"
Byte data: 0000000A
Reading data – Offset: 4 Length: 4 Name: "qgyolobj.listInfo.rcdsReturned"
Byte data: 0000000A
Reading data – Offset: 8 Length: 4 Name: "qgyolobj.listInfo.rqsHandle"
Byte data: 00000001
Reading data – Offset: c Length: 4 Name: "qgyolobj.listInfo.rcdLength"
Byte data: 00000068
Reading data – Offset: 10 Length: 1 Name: "qgyolobj.listInfo.infoComplete"
Byte data: D7
Reading data – Offset: 11 Length: 7 Name: "qgyolobj.listInfo.dateCreated"
Byte data: F0F9F9F0F1F1F5
Reading data – Offset: 18 Length: 6 Name: "qgyolobj.listInfo.timeCreated"
Byte data: F1F4F2F6F2F5
Reading data – Offset: 1e Length: 1 Name: "qgyolobj.listInfo.listStatus"
Byte data: F4
Reading data – Offset: 1f Length: 1 Name: "qgyolobj.listInfo.[8]" Byte data: 00
Reading data – Offset: 20 Length: 4 Name: "qgyolobj.listInfo.lengthOfInfo"
Byte data: 00000410
Reading data – Offset: 24 Length: 4 Name: "qgyolobj.listInfo.firstRecord"
Byte data: 00000001
Reading data – Offset: 28 Length: 40 Name: "qgyolobj.listInfo.[11]"
Byte data: 0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

The above messages can be very helpful in diagnosing cases where the output data coming from the AS/400 program does not match the PCML source. This can easily occur when you are using dynamic lengths and offsets.

[Legal | AS/400 Glossary]

PCML syntax

PCML consists of the following tags, each of which has its own attribute tags:

- The program tag begins and ends code that describes one program
- The struct tag defines a named structure which can be specified as an argument to a program or as a field within another named structure. A structure tag contains a data or a structure tag for each field in the structure.
- The data tag defines a field within a program or structure.

For example, below, the PCML syntax describes one program with one category of data and some isolated data.

```
<program>
<struct>
<data> </data>
</struct>
<data> </data>
</program>
```

[Legal | AS/400 Glossary]

The program tag

The program tag can be expanded with the following elements:

<program name="*name*" **[path=**"*path-name*" **]** **[parseorder=**"*name-list*" **] >** **</program>**		
Attribute	Value	Description
name=	*name*	Specifies the name of the program.
path=	*path-name*	Specifies the path to the program object. The default value is to assume the program is in the QSYS library.

parseorder=	*name-list*	Specifies the order in which output parameters will be processed. The value specified is a blank separated list of parameter names in the order in which the parameters are to be processed. The names in the list must be identical to the names specified on the **name** attribute of tags belonging to the **<program>**. The default value is to process output parameters in the order the tags appear in the document. Some programs return information in one parameter that describes information in a previous parameter. For example, assume a program returns an array of structures in the first parameter and the number of entries in the array in the second parameter. In this case, the second parameter must be processed in order for the ProgramCallDocument to determine the number of structures to process in the first parameter.

[Legal | AS/400 Glossary]

The struct tag

The structure tag can be expanded with the following elements:

```
<struct name="name"
[ count="{number | data-name }"]
[ maxvrm="version-string" ]
[ minvrm="version-string" ]
[ offset="{number | data-name }" ]
[ offsetfrom="{number | data-name | struct-name }" ]
[ outputsize="{number | data-name }" ]
[ usage="{ inherit | input | output | inputoutput }" ]>
</struct>
```

Attribute	Value	Description
name=	*name*	Specifies the name of the **<struct> element**

count=	*number* where *number* defines a fixed, never-changing sized array. *data-name* where *data-name* defines the name of a **<data>** element within the PCML document that will contain, at runtime, the number of elements in the array. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See "Resolving relative names" on page 69 for more information on how relative names are resolved.	Specifies that the element is an array and identifies the number of entries in the array. If this attribute is omitted, the element is not defined as an array, although it may be contained within another element that is defined as an array.
maxvrm=	*version-string*	Specifies the highest AS/400 version on which the element exists. If the AS/400 version is greater than the version specified on the attribute, the element and its children, if any exist, will not be processed during a call to a program. The **maxvrm** element is helpful for defining program interfaces which differ between releases of AS/400. The syntax of the version string must be "VvRrMm," where the capitals letters "V," "R," and "M" are literal characters and "v," "r," and "m" are one or more digits representing the version, release and modification level, respectively. The value for "v" must be from 1 to 255 inclusively. The value for "r" and "m" must be from 0 to 255, inclusively.

minvrm=	*version-string*	Specifies the lowest AS/400 version on which this element exists. If the AS/400 version is less than the version specified on this attribute, this element and its children, if any exist, will not be processed during a call to a program. This attribute is helpful for defining program interfaces which differ between releases of AS/400. The syntax of the version string must be “VvRrMm,” where the capitals letters “V,” “R,” and “M” are literal characters and “v,” “r,” and “m” are one or more digits representing the version, release and modification level, respectively. The value for “v” must be from 1 to 255, inclusively. The value for “r” and “m” must be from 0 to 255, inclusively.

offset=	*number* where *number* defines a fixed, never-changing offset. *data-name* where *data-name*defines the name of a **<data>** element within the PCML document that will contain, at runtime, the offset to the element. The **data-name** specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See "Resolving relative names" on page 69 for more information on how relative names are resolved.	Specifies the offset to the **<struct>** element within an output parameter. Some programs return information with a fixed structure followed by one or more variable length fields or structures. In this case, the location of a variable length element is usually specified as an offset or displacement within the parameter. The **offset** attribute is used to describe the offset to this **<struct>** element. **Offset** is used in conjunction with the **offsetfrom** attribute. If the **offsetfrom** attribute is not specified, the base location for the offset specified on the **offset** attribute is the parent of the element. See "Specifying offsets" on page 69 for more information on how to use the **offset** and **offsetfrom** attributes. The **offset** and **offsetfrom** attributes are only used to process output data from a program. These attributes do not control the offset or displacement of input data. If the attribute is omitted, the location of the data for the element is immediately following the preceding element in the parameter, if any.

offsetfrom=	*number* where *number* defines a fixed, never-changing base location. A *number* attribute is most typically used to specify **number=***"0"* indicating that the offset is an absolute offset from the beginning of the parameter. *data-name* where *data-name* defines the name of a **<data>** element to be used as a base location for the offset. The element name specified must be the parent or an ancestor of this element. The value from the **offset** attribute will be relative to the location of the element specified on this attribute. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference an ancestor of this element. See "Resolving relative names" on page 69 for more information on how relative names are resolved. *struct-name* where *struct-name* defines the name of a **<struct>** element to be used as a base location for the offset. The element name specified must be the parent or an ancestor of this element. The value from the **offset** attribute will be relative to the location of the element specified on this attribute. The *struct-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference an ancestor of this element. See "Resolving relative names" on page 69 for more information on how relative names are resolved.	Specifies the base location from which the **offset** attribute is relative. If the **offsetfrom** attribute is not specified, the base location for the offset specified on the **offset** attribute is the parent of this element. See "Specifying offsets" on page 69 for more information on how to use the **offset** and **offsetfrom** attributes. The **offset** and **offsetfrom** attributes are only used to process output data from a program. These attributes do not control the offset or displacement of input data.

outputsize=	*number* where *number* defines a fixed,never-changing number of bytes to reserve. *data-name* where *data-name* defines the name of a **<data>** element within the PCML document that will contain, at runtime, the number of bytes to reserve for output data. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See "Resolving relative names" on page 69 for more information on how relative names are resolved.	Specifies the number of bytes to reserve for output data for the element. For output parameters which are variable in length, the **outputsize** attribute is needed to specify how many bytes should be reserved for data to be returned from the AS/400 program. **Outputsize** can be specified on all variable length fields and variable sized arrays, or it can be specified for an entire parameter that contains one or more variable length fields. **Outputsize** is not necessary and should not be specified for fixed-size output parameters. The value specified on the attribute is used as the total size for the element including all children of the element. Therefore, the **outputsize** attribute is ignored on any children or descendants of the element. If the attribute is omitted, the number of bytes to reserve for output data is determined at runtime by adding the number of bytes to reserve for all of the children of the **<struct>** element.
usage=	*inherit*	Usage is inherited from the parent element. If the structure does not have a parent, usage is assumed to be **inputoutput**.
	input	The structure is an input value to the host program. For character and numeric types, the appropriate conversion is performed.
	output	The structure is an output value from the host program. For character and numeric types, the appropriate conversion is performed.
	inputoutput	The structure is both and input and an output value.

Resolving relative names

Several attributes allow you to specify the name of another element, or tag, within the document as the attribute value. The name specified can be a name that is relative to the current tag.

Names are resolved by seeing if the name can be resolved as a child or descendent of the tag containing the current tag. If the name cannot be resolved at this level, the search continues with the next highest containing tag. This resolution must eventually result in a match of a tag that is contained by the <**pcml**> tag, in which case the name is considered to be an absolute name, not a relative name.

```
<pcml version="1.0">
  <program name="polytest" path="/QSYS.lib/MYLIB.lib/POLYTEST.pgm">
    <!– Parameter 1 contains a count of polygoins along with an array of polygons –>
    <struct name="parm1" usage="inputoutput">
      <data name="nbrPolygons" type="int" length="4" init="5" />
      <!– Each polygon contains a count of the number of points along with an array of points –>
      <struct name="polygon" count="nbrPolygons">
        <data name="nbrPoints" type="int" length="4" init="3" />
        <struct name="point" count="nbrPoints" >
          <data name="x" type="int" length="4" init="100" />
          <data name="y" type="int" length="4" init="200" />
        </struct>
      </struct>>
    </struct>
  </program>
</pcml>
```

Specifying offsets

Some programs return information with a fixed structure followed by one or more variable length fields or structures. In this case, the location of a variable length element is usually specified as an offset or displacement within the parameter.

An offset is the distance in bytes from a the beginning of the parameters to the beginnning of a field or structure. A displacement is the distance in bytes from the beginning of one structure to the beginning of another structure.

For offsets, since the distance is from the beginning of the parameter, you should specify **offsetfrom=″0″**. The following is an example of an offset from the beginning of the parameter:

```
<pcml version="1.0">
  <program name="myprog" path="/QSYS.lib/MYLIB.lib/MYPROG.pgm">
    <!– receiver variable contains a path –>
    <struct name="reciever" usage="output" outputsize="2048">
      <data name="pathType"         type="int"  length="4" />
      <data name="offsetToPathName" type="int"  length="4" />
      <data name="lengthOfPathName" type="int"  length="4" />
      <data name="pathName"         type="char" length="lengthOfPathName"
                offset="offsetToPathName"  offsetfrom="0"/>
    </struct>
  </program>
</pcml>
```

For displacements, since the distance is from the beginning of another structure, you specify the name of the structure to which the offset is relative. The following is an example of an displacement from the beginning of a named structure:

```
<pcml version="1.0">
  <program name="myprog" path="/QSYS.lib/MYLIB.lib/MYPROG.pgm">
    <!– receiver variable contains an object –>
    <struct name="reciever" usage="output" >
```

```
      <data name="objectName"       type="char"  length="10" />
      <data name="libraryName"      type="char"  length="10" />
      <data name="objectType"       type="char"  length="10" />
      <struct name="pathInfo" usage="output" outputsize="2048" >
        <data name="pathType"         type="int"  length="4" />
        <data name="offsetToPathName" type="int"  length="4" />
        <data name="lengthOfPathName" type="int"  length="4" />
        <data name="pathName"         type="char" length="lengthOfPathName"
                  offset="offsetToPathName"  offsetfrom="pathInfo"/>
      </struct>
    </struct>
  </program>
</pcml>
```

[Legal | AS/400 Glossary]

The data tag

The data tag can have the following attributes. Attributes enclosed in brackets, [], indicate that the attribute is optional. If you specify an optional attribute, do not include the brackets in your source. Some attribute values are shown as a list of choices enclosed in braces, {}, with possible choices separated by vertical bars, |. When you specify one of these attributes, do not include the braces in your source and only specify one of the choices shown.

```
<data type="{ char | int | packed | zoned | float | byte | struct }"
[ ccsid="{ number | data-name }" ]
[ count="{ number | data-name }" ]
[ init="string" ]
[ length="{ number | data-name }" ]
[ maxvrm="version-string" ]
[ minvrm="version-string" ]
[ name="name" ]
[ offset="{ number | data-name }" ]
[ offsetfrom="{ number | data-name | struct-name }" ]
[ outputsize="{ number | data-name | struct-name }" ]
[ precision="number" ]
[ struct="struct-name" ]
[ usage="{ inherit | input | output | inputoutput }" ]>
</data>
```

Attribute	Value	Description

type=	*char* where *char* indicates a character value. The **length** attribute specifies the number of bytes of data which may be different than the number of characters. A *char* data value is returned as a *java.lang.String*. *int* where *int* is an integer value. The **length** attribute specifies the number of bytes, “2” or “4”. The **precision** attribute specifies the number of bits of precision. For example, **length=***“2”* **precision=***“15”* Specifies a 16-bit signed integer. An *int* data value with these specifications is returned as a *java.lang.Short*. **length=***“2”* **precision=***“16”* Specifies a 16-bit unsigned integer. An *int* data value with these specifications is returned as a *java.lang.Integer*. **length=***“4”* **precision=***“31”* Specifies a 32-bit signed integer. An *int* data value with these specifications is returned as a *java.lang.Integer*. **length=***“4”* **precision=***“32”* Specifies a 32-bit unsigned integer. An *int* data value is returned as a *java.lang.Long*. For **length=***“2”*, the default precision is “15”. For **length=***“4”*, the default precision is “31”. *packed* where *packed* is a packed decimal value. The **length** attribute specifies the number of digits. The **precision** attribute specifies the number of decimal positions. A *packed* data value is returned as a *java.math.BigDecimal*. *zoned* where *zoned* is a zoned decimal value. The **length** attribute specifies the number	Indicates the type of data being used (character, integer, packed, zoned, floating point, byte, or struct).

ccsid=	*number* where *number* defines a fixed, never-changing CCSID. *data-name* where *data-name* defines the name that will contain, at runtime, the CCSID of the character data. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See "Resolving relative names" on page 69 for more information on how relative names are resolved.	Specifies the host Coded Character Set ID (CCSID) for character data for the **<data>** element. The **ccsid** attribute can be specified only for **<data>** elements with **type=***"char"*. If this attribute is omitted, character data for this element is assumed to be in the default CCSID of the host environment.
count=	*number* where *number* defines a fixed, never-changing number of elements in a sized array. *data-name* where *data-name* defines the name of a **<data>** element within the PCML document that will contain, at runtime, the number of elements in the array. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See "Resolving relative names" on page 69 for more information on how relative names are resolved.	Specifies that the element is an array and identifies the number of entries in the array. If the *count* attribute is omitted, the element is not defined as an array, although it may be contained within another element that is defined as an array.

init=	*string*	Specifies an initial value for the **<data>** element. The *init* value is used if an initial value is not explicitly set by the application program when **<data>** elements with **usage=***"input"* or **usage=***"inputoutput"* are used. The initial value specified is used to initialize scalar values. If the element is defined as an array or is contained within a structure defined as an array, the initial value specified is used as an initial value for all entries in the array.
length=	*number* where *number* defines a fixed, never-changing length. *data-name* where *data-name* defines the name of a **<data>** element within the PCML document that will contain, at runtime, the length. A *data-name* can be specified only for **<data>** elements with **type=***"char"* or **type=***"byte"*. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See "Resolving relative names" on page 69 for more information on how relative names are resolved.	Specifies the length of the data element. Usage of this attribute varies depending on the data type.
Data Type	**Description**	
type=*"char"*	The **length** attribute specifies the number of bytes, of data for this element. Note that this is not necessarily the number of characters. A literal *number* or *data-name* must be specified.	

type=*"int"*	The **length** attribute specifies the number of bytes, "2" or "4", of data for this element. The **precision** attribute is used to specify the number of bits of precision and indicates whether the integer is signed or unsigned. A literal *number* must be specified.	
type=*"packed"*	The **length** attribute specifies the number of numeric digits of data for this element. The **precision** attribute is used to specify the number of decimal digits. A literal *number* must be specified.	
type=*"zoned"*	The **length** attribute specifies the number of numeric digits of data for this element. The precision attribute is used to specify the number of decimal digits. A literal *number* must be specified.	
type=*"float"*	The **length** attribute specifies the number of bytes, 4 or 8, of data for this element. A literal *number* must be specified.	
type=*"byte"*	The **length** attribute specifies the number of bytes of data for this element. A literal *number* or *data-name* must be specified.	
type=*"struct"*	The **length** attribute is not allowed.	

maxvrm=	*version-string*	Specifies the highest AS/400 version on which this element exists. If the AS/400 version is greater than the version specified on this attribute, this element and its children, if any exist, will not be processed during a call to a program. This attribute is helpful for defining program interfaces which differ between releases of AS/400. The syntax of the version string must be “VvRrMm”, where the capitals letters “V,” “R,” and “M” are literal characters and “v,” “r,” and “m” are one or more digits representing the version, release and modification level, respectively. The value for “v” must be from 1 to 255 inclusively. The value for “r” and “m” must be from 0 to 255, inclusively.
minvrm=	*version-string*	Specifies the lowest AS/400 version on which this element exists. If the AS/400 version is less than the version specified on this attribute, this element and its children, if any exist, will not be processed during a call to a program. This attribute is helpful for defining program interfaces which differ between releases of AS/400. The syntax of the version string must be “VvRrMm,” where the capitals letters “V,” “R,” and “M” are literal characters and “v,” “r,” and “m” are one or more digits representing the version, release and modification level, respectively. The value for “v” must be from 1 to 255 inclusively. The value for “r” and “m” must be from 0 to 255, inclusively.
name=	*name*	Specifies the name of the **<data>** element.

offset=	*number* where *number* defines a fixed, never-changing offset. *data-name* where *data-name* defines the name of a **<data>** element within the PCML document that will contain, at runtime, the offset to this element. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See "Resolving relative names" on page 69 for more information on how relative names are resolved.	Specifies the offset to the **<data>** element within an output parameter. Some programs return information with a fixed structure followed by one or more variable length fields or structures. In this case, the location of a variable length element is usually specified as an offset or displacement within the parameter. An **offset** attribute is used in conjunction with the **offsetfrom** attribute. If the **offsetfrom** attribute is not specified, the base location for the offset specified on the **offset** attribute is the parent of this element. See "Specifying offsets" on page 69 for more information on how to use the **offset** and **offsetfrom** attributes. The **offset** and **offsetfrom** attributes are only used to process output data from a program. These attributes do not control the offset or displacement of input data. If this attribute is omitted, the location of the data for this element is immediately following the preceding element in the parameter, if any.

offsetfrom=	*number* where *number* defines a fixed, never-changing base location. *Number* is most typically used to specify **number=***"0"* indicating that the offset is an absolute offset from the beginning of the parameter. *data-name* where *data-name* defines the name of a **<data>** element used as a base location for the offset. The element name specified must be the parent or an ancestor of this element. The value from the **offset** attribute will be relative to the location of the element specified on this attribute. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference an ancestor of this element. See "Resolving relative names" on page 69 for more information on how relative names are resolved. *struct-name* where *struct-name* defines the name of a **<struct>** element used as a base location for the offset. The element name specified must be the parent or an ancestor of this element. The value from the **offset** attribute will be relative to the location of the element specified on this attribute. The *struct-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference an ancestor of this element. See "Resolving relative names" on page 69 for more information on how relative names are resolved.	Specifies the base location from which the **offset** attribute is relative. If the **offsetfrom** attribute is not specified, the base location for the offset specified on the **offset** attribute is the parent of this element. See "Specifying offsets" on page 69 for more information on how to use the **offset** and **offsetfrom** attributes. The **offset** and **offsetfrom** attributes are only used to process output data from a program. These attributes do not control the offset or displacement of input data.

outputsize=	*number* where a *number* defines a fixed, never-changing number of bytes to reserve. *data-name* where *data-name* defines the name of a **<data>** element within the PCML document that will contain, at runtime, the number of bytes to reserve for output data. The *data-name* specified can be a fully qualified name or a name that is relative to the current element. In either case, the name must reference a **<data>** element that is defined with **type=***"int"*. See "Resolving relative names" on page 69 for more information on how relative names are resolved.	Specifies the number of bytes to reserve for output data for the element. For output parameters which are variable in length, the **outputsize** attribute is needed to specify how many bytes should be reserved for data to be returned from the AS/400 program. An **outputsize** attribute can be specified on all variable length fields and variable sized arrays, or it can be specified for an entire parameter that contains one or more variable length fields. **Outputsize** is not necessary and should not be specified for fixed-size output parameters. The value specified on this attribute is used as the total size for the element including all the children of the element. Therefore, the **outputsize** attribute is ignored on any children or descendants of the element. If **outputsize** is omitted, the number of bytes to reserve for output data is determined at runtime by adding the number of bytes to reserve for all of the children of the **<struct>** element.
precision=	*number*	Specifies the number of bytes of precision for some numeric data types.
Data Type	**Description**	
type=*"int"* **length=***"2"*	Use **precision=***"15"* for a signed 2-byte integer. Use **precision=***"16"* for an unsigned 2-byte integer. The default value is "15".	
type=*"int"* **length=***"4"*	Use *precision="31"* for a signed 4-byte integer. Use *precision="32"* for an unsigned 4-byte integer.	

type=*"zoned"*	The precision specifies the number of decimal digits. The number specified must be greater than or equal to zero and less than or equal to the total number of digits specified on the *length* attribute.	
type=*"zoned"*	The precision specifies the number of decimal digits. The number specified must be greater than or equal to zero and less than or equal to the total number of digits specified on the *length* attribute.	
struct=	*name*	Specifies the name of a **<struct>** element for the **<data>** element. A **struct** attribute can be specified only for **<data>** elements with **type=***"struct"*.
usage=	*inherit*	Usage is inherited from the parent element. If the structure does not have a parent, usage is assumed to be *inputoutput*.
	input	Defines an input value to the host program. For character and numeric types, the appropriate conversion is performed.
	output	Defines an output value from the host program. For character and numeric types, the appropriate conversion is performed.
	inputoutput	Defines both and input and an output value.

Resolving relative names

Several attributes allow you to specify the name of another element, or tag, within the document as the attribute value. The name specified can be a name that is relative to the current tag.

Names are resolved by seeing if the name can be resolved as a child or descendent of the tag containing the current tag. If the name cannot be resolved at this level, the search continues with the next highest containing tag. This resolution must eventually result in a match of a tag that is contained by the <**pcml**> tag, in which case the name is considered to be an absolute name, not a relative name.

```
<pcml version="1.0">
  <program name="polytest" path="/QSYS.lib/MYLIB.lib/POLYTEST.pgm">
    <!– Parameter 1 contains a count of polygoins along with an array of polygons –>
    <struct name="parm1" usage="inputoutput">
```

```
      <data name="nbrPolygons" type="int" length="4" init="5" />
      <!– Each polygon contains a count of the number of points along with an array of points –>
      <struct name="polygon" count="nbrPolygons">
        <data name="nbrPoints" type="int" length="4" init="3" />
        <struct name="point" count="nbrPoints" >
          <data name="x" type="int" length="4" init="100" />
          <data name="y" type="int" length="4" init="200" />
        </struct>
      </struct>
    </struct>
  </program>
</pcml>
```

Specifying offsets

Some programs return information with a fixed structure followed by one or more variable length fields or structures. In this case, the location of a variable length element is usually specified as an offset or displacement within the parameter.

An offset is the distance in bytes from the beginning of the parameters to the beginnings of a field or structure. A displacement is the distance in bytes from the beginning of one structure to the beginning of another structure.

For offsets, since the distance is from the beginning of the parameter, you should specify **offsetfrom="0"**. The following is an example of an offset from the beginning of the parameter:

```
<pcml version="1.0">
  <program name="myprog" path="/QSYS.lib/MYLIB.lib/MYPROG.pgm">
    <!– receiver variable contains a path –>
    <struct name="receiver" usage="output" outputsize="2048">
      <data name="pathType"         type="int"  length="4" />
      <data name="offsetToPathName" type="int"  length="4" />
      <data name="lengthOfPathName" type="int"  length="4" />
      <data name="pathName"         type="char" length="lengthOfPathName"
                offset="offsetToPathName"  offsetfrom="0"/>
    </struct>
  </program>
</pcml>
```

For displacements, since the distance is from the beginning of another structure, you specify the name of the structure to which the offset is relative. The following is an example of an displacement from the beginning of a named structure:

```
<pcml version="1.0">
  <program name="myprog" path="/QSYS.lib/MYLIB.lib/MYPROG.pgm">
    <!– receiver variable contains an object –>
    <struct name="receiver" usage="output" >
      <data name="objectName"       type="char"  length="10" />
      <data name="libraryName"      type="char"  length="10" />
      <data name="objectType"       type="char"  length="10" />
      <struct name="pathInfo" usage="output" outputsize="2048" >
        <data name="pathType"         type="int"  length="4" />
        <data name="offsetToPathName" type="int"  length="4" />
        <data name="lengthOfPathName" type="int"  length="4" />
        <data name="pathName"         type="char" length="lengthOfPathName"
                  offset="offsetToPathName"  offsetfrom="pathInfo"/>
      </struct>
    </struct>
  </program>
</pcml>
```

Example of using the Program Call Markup Language

Below are some examples of using Program Call Markup Language to call OS/400 APIs. Each example shows the PCML source followed by a Java program.

Note that in order to run these examples, the person running the program must sign on with a user profile that has proper authority to do the following:

- Call the OS/400 API in the example
- Access the information being requested

The proper authority for each example varies but may include specific object authorities and special authorities.

License information

IBM grants you a nonexclusive license to use this as an example from which you can generate similar function tailored to your own specific needs. This sample is provided in the form of source material which you may change and use.

DISCLAIMER

This sample code is provided by IBM for illustrative purposes only. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. All programs contained herein are provided to you “AS IS” without any warranties of any kind. ALL WARRANTIES, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE EXPRESSLY DISCLAIMED.

Your license to this sample code provides you no right or licenses to any IBM patents. IBM has no obligation to defend or indemnify against any claim of infringement, including but not limited to: patents, copyright, trade secret, or intellectual property rights of any kind.

COPYRIGHT

Simple example of retrieving data

This example shows the PCML source and Java program needed to retrieve information about a user profile on the AS/400. The API being called is the *Retrieve User Information* (**QSYRSURI**) API.

PCML source for calling QSYRUSRI

```
<pcml version="1.0">
<!– PCML source for calling "Retreive user Information" (QSYRUSRI) API –>
  <!– Format AUTU0150 - Other formats are available –>
  <struct name="usri0100">
    <data name="bytesReturned"              type="int"    length="4"  usage="output"/>
    <data name="bytesAvailable"             type="int"    length="4"  usage="output"/>
    <data name="userProfile"                type="char"   length="10" usage="output"/>
    <data name="previousSignonDate"         type="char"   length="7"  usage="output"/>
    <data name="previousSignonTime"         type="char"   length="6"  usage="output"/>
    <data                                   type="byte"   length="1"  usage="output"/>
    <data name="badSignonAttempts"          type="int"    length="4"  usage="output"/>
    <data name="status"                     type="char"   length="10" usage="output"/>
    <data name="passwordChangeDate"         type="byte"   length="8"  usage="output"/>
    <data name="noPassword"                 type="char"   length="1"  usage="output"/>
    <data                                   type="byte"   length="1"  usage="output"/>
    <data name="passwordExpirationInterval" type="int"    length="4"  usage="output"/>
    <data name="datePasswordExpires"        type="byte"   length="8"  usage="output"/>
    <data name="daysUntilPasswordExpires"   type="int"    length="4"  usage="output"/>
    <data name="setPasswordToExpire"        type="char"   length="1"  usage="output"/>
    <data name="displaySignonInfo"          type="char"   length="10" usage="output"/>
  </struct>
  <!– Program QSYRUSRI and its parameter list for retrieving USRI0100 format –>
  <program name="qsyrusri" path="/QSYS.lib/QSYRUSRI.pgm">
    <data name="receiver"                   type="struct"             usage="output"
          struct="usri0100"/>
    <data name="receiverLength"             type="int"    length="4"  usage="input" />
    <data name="format"                     type="char"   length="8"  usage="input"
          init="USRI0100"/>
    <data name="profileName"                type="char"   length="10" usage="input"
          init="*CURRENT"/>
    <data name="errorCode"                  type="int"    length="4"  usage="input"
          init="0"/>
  </program>
</pcml>
```

Java program source for calling QSYRUSRI

```
import com.ibm.as400.data.ProgramCallDocument;
import com.ibm.as400.data.PcmlException;
import com.ibm.as400.access.AS400;
import com.ibm.as400.access.AS400Message;
// Example program to call "Retrieve User Information" (QSYRUSRI) API
public class qsyrusri {
    public qsyrusri() {
    }
    public static void main(String[] argv)
    {
        AS400 as400System;         // com.ibm.as400.access.AS400
        ProgramCallDocument pcml;  // com.ibm.as400.data.ProgramCallDocument
        boolean rc = false;        // Return code from ProgramCallDocument.callProgram()
        String msgId, msgText;     // Messages returned from AS/400
        Object value;              // Return value from ProgramCallDocument.getValue()
        System.setErr(System.out);
        // Construct AS400 without parameters, user will be prompted
        as400System = new AS400();
    try
        {
            // Uncomment the following to get debugging information
            //com.ibm.as400.data.PcmlMessageLog.setTraceEnabled(true);
            System.out.println("Beginning PCML Example..");
            System.out.println("    Constructing ProgramCallDocument for QSYRUSRI API...");
            // Construct ProgramCallDocument
            // First parameter is system to connect to
            // Second parameter is pcml resource name. In this example,
            // serialized PCML file "qsyrusri.pcml.ser" or
            // PCML source file "qsyrusri.pcml" must be found in the classpath.
            pcml = new ProgramCallDocument(as400System, "qsyrusri");
            // Set input parameters. Several parameters have default values
            // specified in the PCML source. Do not need to set them using Java code.
            System.out.println("    Setting input parameters...");
            pcml.setValue("qsyrusri.receiverLength", new Integer((pcml.getOutputsize("qsyrusri.r
            // Request to call the API
            // User will be prompted to sign on to the system
            System.out.println("    Calling QSYRUSRI API requesting information for the sign-on
            rc = pcml.callProgram("qsyrusri");
            // If return code is false, we received messages from the AS/400
            if(rc == false)
            {
                // Retrieve list of AS/400 messages
                AS400Message[] msgs = pcml.getMessageList("qsyrusri");
                // Iterate through messages and write them to standard output
                for (int m = 0; m < msgs.length; m++)
                {
                    msgId = msgs[m].getID();
                    msgText = msgs[m].getText();
                    System.out.println("    " + msgId + " - " + msgText);
                }
                System.out.println("** Call to QSYRUSRI failed. See messages above **");
                System.exit(0);
            }
            // Return code was true, call to QSYRUSRI succeeded
            // Write some of the results to standard output
            else
            {
                value = pcml.getValue("qsyrusri.receiver.bytesReturned");
                System.out.println("        Bytes returned:      " + value);
                value = pcml.getValue("qsyrusri.receiver.bytesAvailable");
                System.out.println("        Bytes available:     " + value);
                value = pcml.getValue("qsyrusri.receiver.userProfile");
                System.out.println("        Profile name:        " + value);
                value = pcml.getValue("qsyrusri.receiver.previousSignonDate");
                System.out.println("        Previous signon date:" + value);
                value = pcml.getValue("qsyrusri.receiver.previousSignonTime");
                System.out.println("        Previous signon time:" + value);
            }
        }
        catch (PcmlException e)
```

}

Example of retrieving a list of information

This example shows the PCML source and Java program needed to retrieve a list of authorized users on an AS/400. The API being called is the *Open List of Authorized Users* (**QGYOLAUS**) API.

This example illustrates how to access an array of structures returned by an AS/400 program.

PCML source for calling QGYOLAUS

```
<pcml version="1.0">
<!– PCML source for calling "Open List of Authorized Users" (QGYOLAUS) API –>
  <!– Format AUTU0150 - Other formats are available –>
  <struct name="autu0150">
    <data name="name"         type="char" length="10" />
    <data name="userOrGroup"  type="char" length="1"  />
    <data name="groupMembers" type="char" length="1"  />
    <data name="description"  type="char" length="50" />
  </struct>
  <!– List information structure (common for "Open List" type APIs) –>
  <struct name="listInfo">
    <data name="totalRcds"    type="int"  length="4" />
    <data name="rcdsReturned" type="int"  length="4" />
    <data name="rqsHandle"    type="byte" length="4" />
    <data name="rcdLength"    type="int"  length="4" />
    <data name="infoComplete" type="char" length="1" />
    <data name="dateCreated"  type="char" length="7" />
    <data name="timeCreated"  type="char" length="6" />
    <data name="listStatus"   type="char" length="1" />
    <data                     type="byte" length="1" />
    <data name="lengthOfInfo" type="int"  length="4" />
    <data name="firstRecord"  type="int"  length="4" />
    <data                     type="byte" length="40" />
  </struct>

  <!– Program QGYOLAUS and its parameter list for retrieving AUTU0150 format –>
  <program name="qgyolaus" path="/QSYS.lib/QGY.lib/QGYOLAUS.pgm" parseorder="listInfo receiver">
    <data   name="receiver"       type="struct" struct="autu0150" usage="output"
          count="listInfo.rcdsReturned" outputsize="receiverLength" />
    <data   name="receiverLength" type="int"    length="4"  usage="input" init="16384" />
    <data   name="listInfo"       type="struct" struct="listInfo" usage="output" />
    <data   name="rcdsToReturn"   type="int"    length="4"  usage="input" init="264" />
    <data   name="format"         type="char"   length="10" usage="input" init="AUTU0150" />
    <data   name="selection"      type="char"   length="10" usage="input" init="*USER" />
    <data   name="member"         type="char"   length="10" usage="input" init="*NONE" />
    <data   name="errorCode"      type="int"    length="4"  usage="input" init="0" />
  </program>
  <!– Program QGYGTLE returned additional "records" from the list
       created by QGYOLAUS. –>
  <program name="qgygtle" path="/QSYS.lib/QGY.lib/QGYGTLE.pgm" parseorder="listInfo receiver">
    <data   name="receiver"       type="struct" struct="autu0150" usage="output"
            count="listInfo.rcdsReturned" outputsize="receiverLength" />
    <data   name="receiverLength" type="int"    length="4" usage="input" init="16384" />
    <data   name="requestHandle"  type="byte"   length="4" usage="input" />
    <data   name="listInfo"       type="struct" struct="listInfo" usage="output" />
    <data   name="rcdsToReturn"   type="int"    length="4" usage="input" init="264" />
    <data   name="startingRcd"    type="int"    length="4" usage="input" />
    <data   name="errorCode"      type="int"    length="4" usage="input" init="0" />
  </program>
  <!– Program QGYCLST closes the list, freeing resources on the AS/400 –>
  <program name="qgyclst" path="/QSYS.lib/QGY.lib/QGYCLST.pgm" >
    <data   name="requestHandle"  type="byte"   length="4" usage="input" />
    <data   name="errorCode"      type="int"    length="4" usage="input" init="0" />
  </program>
</pcml>
```

Java program source for calling QGYOLAUS

```
import com.ibm.as400.data.ProgramCallDocument;
import com.ibm.as400.data.PcmlException;
import com.ibm.as400.access.AS400;
import com.ibm.as400.access.AS400Message;
// Example program to call "Retrieve List of Authorized Users" (QGYOLAUS) API
public class qgyolaus
{
  public static void main(String[] argv)
  {
    AS400 as400System;         // com.ibm.as400.access.AS400
    ProgramCallDocument pcml;  // com.ibm.as400.data.ProgramCallDocument
    boolean rc = false;        // Return code from ProgramCallDocument.callProgram()
    String msgId, msgText;     // Messages returned from AS/400
    Object value;              // Return value from ProgramCallDocument.getValue()
    int[] indices = new int[1]; // Indices for access array value
    int nbrRcds,                // Number of records returned from QGYOLAUS and QGYGTLE
        nbrUsers;               // Total number of users retrieved
    String listStatus;          // Status of list on AS/400
    byte[] requestHandle = new byte[4];
    System.setErr(System.out);
    // Construct AS400 without parameters, user will be prompted
    as400System = new AS400();
    try
    {
      // Uncomment the following to get debugging information
      //com.ibm.as400.data.PcmlMessageLog.setTraceEnabled(true);
      System.out.println("Beginning PCML Example..");
      System.out.println("    Constructing ProgramCallDocument for QGYOLAUS API...");
      // Construct ProgramCallDocument
      // First parameter is system to connect to
      // Second parameter is pcml resource name. In this example,
      // serialized PCML file "qgyolaus.pcml.ser" or
      // PCML source file "qgyolaus.pcml" must be found in the classpath.
      pcml = new ProgramCallDocument(as400System, "qgyolaus");
      // All input parameters have default values specified in the PCML source.
      // Do not need to set them using Java code.
      // Request to call the API
      // User will be prompted to sign on to the system
      System.out.println("    Calling QGYOLAUS API requesting information for the sign-on user.");
      rc = pcml.callProgram("qgyolaus");
      // If return code is false, we received messages from the AS/400
      if(rc == false)
      {
        // Retrieve list of AS/400 messages
        AS400Message[] msgs = pcml.getMessageList("qgyolaus");
        // Iterate through messages and write them to standard output
        for (int m = 0; m < msgs.length; m++)
        {
            msgId = msgs[m].getID();
            msgText = msgs[m].getText();
            System.out.println("    " + msgId + " - " + msgText);
        }
        System.out.println("** Call to QGYOLAUS failed. See messages above **");
        System.exit(0);
      }
      // Return code was true, call to QGYOLAUS succeeded
      // Write some of the results to standard output
      else
      {
        boolean doneProcessingList = false;
        String programName = "qgyolaus";
        nbrUsers = 0;
        while (!doneProcessingList)
        {
          nbrRcds = pcml.getIntValue(programName + ".listInfo.rcdsReturned");
          requestHandle = (byte[]) pcml.getValue(programName + ".listInfo.rqsHandle");
          // Iterate through list of users
          for (indices[0] = 0; indices[0] < nbrRcds; indices[0]++)
          {
               value = pcml.getValue(programName + ".receiver.name", indices);
```

}

Example of retrieving multidimensional data

This example shows the PCML source and Java program needed to retrieve a list Network File System (NFS) exports from an AS/400. The API being called is the *Retrieve NFS Exports* (**QZNFRTVE**) API.

This example illustrates how to access arrays of structures within an array of structures.

PCML source for calling QZNFRTVE

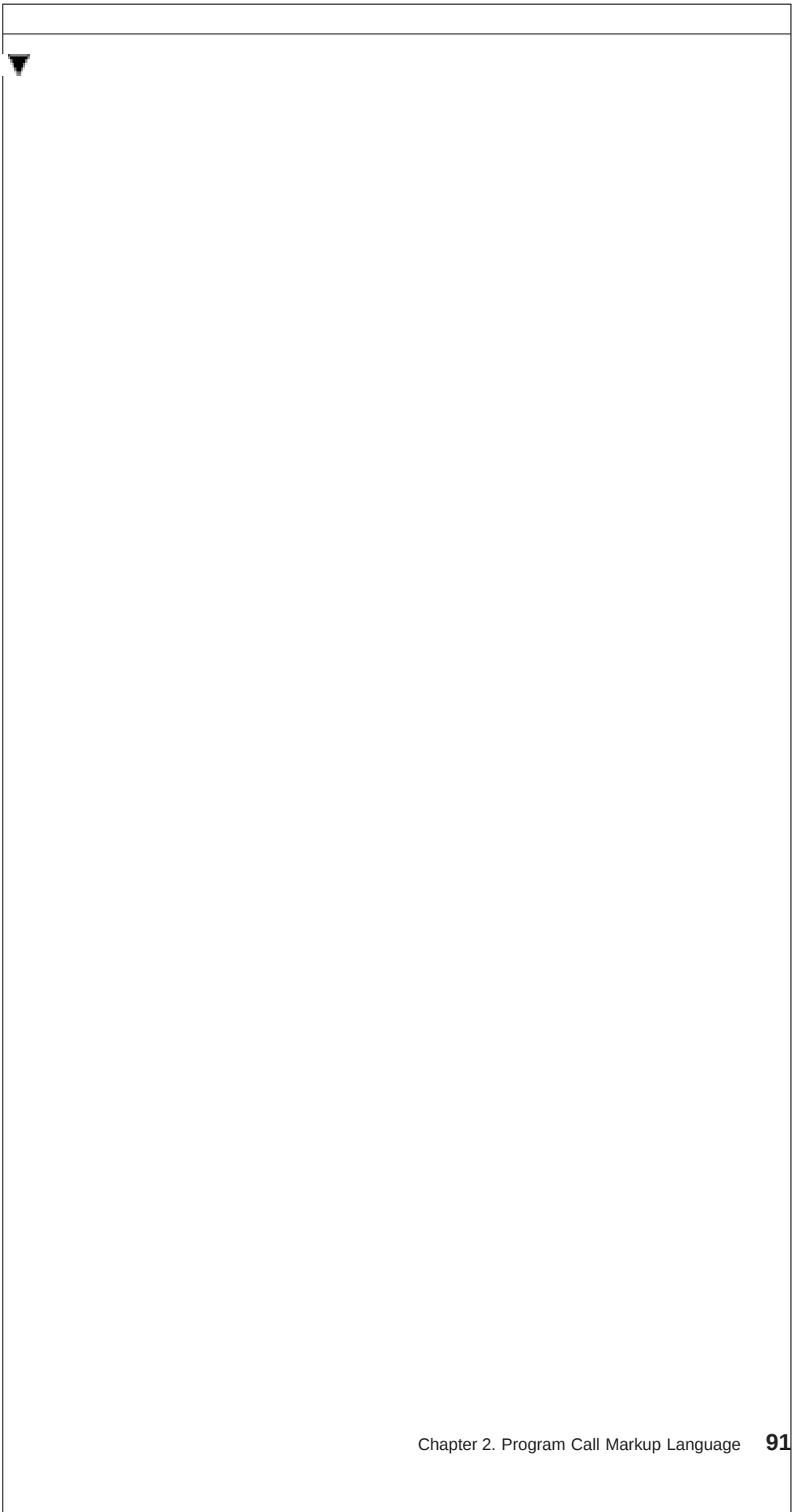
```
<pcml version="1.0">
  <struct name="receiver">
    <data name="lengthOfEntry"            type="int"  length="4" />
    <data name="dispToObjectPathName"     type="int"  length="4" />
    <data name="lengthOfObjectPathName"   type="int"  length="4" />
    <data name="ccsidOfObjectPathName"    type="int"  length="4" />
    <data name="readOnlyFlag"             type="int"  length="4" />
    <data name="nosuidFlag"               type="int"  length="4" />
    <data name="dispToReadWriteHostNames" type="int"  length="4" />
    <data name="nbrOfReadWriteHostNames"  type="int"  length="4" />
    <data name="dispToRootHostNames"      type="int"  length="4" />
    <data name="nbrOfRootHostNames"       type="int"  length="4" />
    <data name="dispToAccessHostNames"    type="int"  length="4" />
    <data name="nbrOfAccessHostNames"     type="int"  length="4" />
    <data name="dispToHostOptions"        type="int"  length="4" />
    <data name="nbrOfHostOptions"         type="int"  length="4" />
    <data name="anonUserID"               type="int"  length="4" />
    <data name="anonUsrPrf"               type="char" length="10" />
    <data name="pathName"                 type="char" length="lengthOfObjectPathName"
          offset="dispToObjectPathName" offsetfrom="receiver" />
    <struct name="rwAccessList" count="nbrOfReadWriteHostNames"
            offset="dispToReadWriteHostNames" offsetfrom="receiver">
      <data name="lengthOfEntry"          type="int"  length="4" />
      <data name="lengthOfHostName"       type="int"  length="4" />
      <data name="hostName"               type="char" length="lengthOfHostName" />
      <data                               type="byte" length="0"
            offset="lengthOfEntry" />
    </struct>
    <struct name="rootAccessList" count="nbrOfRootHostNames"
            offset="dispToRootHostNames" offsetfrom="receiver">
      <data name="lengthOfEntry"          type="int"  length="4" />
      <data name="lengthOfHostName"       type="int"  length="4" />
      <data name="hostName"               type="char" length="lengthOfHostName" />
      <data                               type="byte" length="0"
            offset="lengthOfEntry" />
    </struct>
    <struct name="accessHostNames" count="nbrOfAccessHostNames"
            offset="dispToAccessHostNames" offsetfrom="receiver" >
      <data name="lengthOfEntry"          type="int"  length="4" />
      <data name="lengthOfHostName"       type="int"  length="4" />
      <data name="hostName"               type="char" length="lengthOfHostName" />
      <data                               type="byte" length="0"
            offset="lengthOfEntry" />
    </struct>
    <struct name="hostOptions" offset="dispToHostOptions" offsetfrom="receiver" count="nbrOfHostOpt
      <data name="lengthOfEntry"          type="int"  length="4" />
      <data name="dataFileCodepage"       type="int"  length="4" />
      <data name="pathNameCodepage"       type="int"  length="4" />
      <data name="writeModeFlag"          type="int"  length="4" />
      <data name="lengthOfHostName"       type="int"  length="4" />
      <data name="hostName"               type="char" length="lengthOfHostName" />
      <data                               type="byte" length="0"
            offset="lengthOfEntry" />
    </struct>
    <data type="byte" length="0" offset="lengthOfEntry" />
  </struct>
  <struct name="returnedRcdsFdbkInfo">
    <data name="bytesReturned"            type="int" length="4" />
    <data name="bytesAvailable"           type="int" length="4" />
    <data name="nbrOfNFSExportEntries"    type="int" length="4" />
    <data name="handle"                   type="int" length="4" />
  </struct>

  <program name="qznfrtve" path="/QSYS.lib/QZNFRTVE.pgm" parseorder="returnedRcdsFdbkInfo receiver"
    <data name="receiver"             type="struct" struct="receiver" usage="output"
      count="returnedRcdsFdbkInfo.nbrOfNFSExportEntries" outputsize="receiverLength"/>
    <data name="receiverLength"       type="int"    length="4" usage="input" init="4096" />
    <data name="returnedRcdsFdbkInfo" type="struct" struct="returnedRcdsFdbkInfo" usage="output" />
    <data name="formatName"           type="char"   length="8" usage="input" init="EXPE0100" />
    <data name="objectPathName"       type="char"   length="lengthObjPathName" usage="input" init="
```

```
</pcml>
```

Java program source for calling QZNFRTVE

```
import com.ibm.as400.data.ProgramCallDocument;
import com.ibm.as400.data.PcmlException;
import com.ibm.as400.access.AS400;
import com.ibm.as400.access.AS400Message;
// Example program to call "Retrieve NFS Exports" (QZNFRTVE) API
public class qznfrtve
{
  public static void main(String[] argv)
  {
    AS400 as400System;         // com.ibm.as400.access.AS400
    ProgramCallDocument pcml;  // com.ibm.as400.data.ProgramCallDocument
    boolean rc = false;        // Return code from ProgramCallDocument.callProgram()
    String msgId, msgText;     // Messages returned from AS/400
    Object value;              // Return value from ProgramCallDocument.getValue()
    System.setErr(System.out);
    // Construct AS400 without parameters, user will be prompted
    as400System = new AS400();
    int[] indices = new int[2]; // Indices for access array value
    int nbrExports;             // Number of exports returned
    int nbrOfReadWriteHostNames, nbrOfRWHostNames,
        nbrOfRootHostNames,      nbrOfAccessHostnames, nbrOfHostOpts;
    try
    {
      // Uncomment the following to get debugging information
      // com.ibm.as400.data.PcmlMessageLog.setTraceEnabled(true);
      System.out.println("Beginning PCML Example..");
      System.out.println("    Constructing ProgramCallDocument for QZNFRTVE API...");
      // Construct ProgramCallDocument
      // First parameter is system to connect to
      // Second parameter is pcml resource name. In this example,
      // serialized PCML file "qznfrtve.pcml.ser" or
      // PCML source file "qznfrtve.pcml" must be found in the classpath.
      pcml = new ProgramCallDocument(as400System, "qznfrtve");
      // Set input parameters. Several parameters have default values
      // specified in the PCML source. Do not need to set them using Java code.
      System.out.println("    Setting input parameters...");
      pcml.setValue("qznfrtve.receiverLength", new Integer( ( pcml.getOutputsize("qznfrtve.receiver
      // Request to call the API
      // User will be prompted to sign on to the system
      System.out.println("    Calling QZNFRTVE API requesting NFS exports.");
      rc = pcml.callProgram("qznfrtve");
      if (rc == false)
      {
        // Retrieve list of AS/400 messages
        AS400Message[] msgs = pcml.getMessageList("qznfrtve");
        // Iterate through messages and write them to standard output
        for (int m = 0; m < msgs.length; m++)
        {
            msgId = msgs[m].getID();
            msgText = msgs[m].getText();
            System.out.println("    " + msgId + " - " + msgText);
        }
        System.out.println("** Call to QZNFRTVE failed. See messages above **");
        System.exit(0);
      }
      // Return code was true, call to QZNFRTVE succeeded
      // Write some of the results to standard output
      else
      {
        nbrExports = pcml.getIntValue("qznfrtve.returnedRcdsFdbkInfo.nbrOfNFSExportEntries");
        // Iterate through list of exports
        for (indices[0] = 0; indices[0] < nbrExports; indices[0]++)
        {
          value = pcml.getValue("qznfrtve.receiver.pathName", indices);
          System.out.println("Path name = " + value);
          // Iterate and write out Read Write Host Names for this export
          nbrOfReadWriteHostNames = pcml.getIntValue("qznfrtve.receiver.nbrOfReadWriteHostNames",
          for(indices[1] = 0; indices[1] < nbrOfReadWriteHostNames; indices[1]++)
          {
            value = pcml.getValue("qznfrtve.receiver.rwAccessList.hostName", indices);
```

}



Printed in U.S.A.