New Features in PowerBuilder 9

Note: The features described in this document are still under development. The information may therefore be incomplete and is subject to change. Additional topics may be added in later beta releases.

This document describes the following features:

- DataWindow XML support
- JSP targets
- Windows clients for Web services
- PowerBuilder Document Object Model
- EJB clients for third-party application servers
- Source control enhancements
- The OrcaScript language
- PowerBuilder Native Interface
- Oracle9i support
- Unicode database support
- XML and CSV import and save
- Saving as XSL-FO
- DBCS versions of string functions
- DataWindow enhancements
- Debugger enhancements
DataWindow XML support

Description

This feature allows DataWindow row data to be exported and imported in the Extensible Markup Language (XML). You can specify the XML logical structure of how the row data iterates inside the root element of the XML document. This is done with XML template objects that are encapsulated in DataWindow objects and graphically constructed in a new View in the DataWindow Painter.

Usage

In the Export Template view, you can predefine how iterative children of the root element can be composed of any level of nested elements, comments, character references, or processing instructions, using explicit markup. You can also reference DataWindow column, computed field, and text controls for each row in XML, wherever character data is allowed, to be replaced with text at runtime.

You can specify DataWindow expressions wherever character data is allowed, anywhere within the Root element, which includes the header section.

You can define multiple named versions of these export templates for each DataWindow object. Their storage and persistence coincide with the DataWindow object and can deployed with it in an SRD or PBL. The template used is specified with the Export.XML.UseTemplate DataWindow property.

You can export the data in a DataWindow or DataStore object to XML using any of the techniques used for exporting to other formats such as PSR or HTML:

- Using the SaveAs method:

  ```
  ds_1.SaveAs("C:\TEMP\Temp.xml", XML!, TRUE)
  ```

- Using PowerScript dot notation:

  ```
  ls_xmlstring = dw_1.Object.DataWindow.Data.XML
  ```

- Using the Save Rows As menu item in the DataWindow painter when the Preview view is open. You can use this at design time to preview the XML that will be generated at runtime.

You can import data from an XML document with or without an XML template using the Rows>Import menu item in the DataWindow painter or the ImportFile, ImportString, and ImportClipboard functions. The template used is specified with the Import.XML.UseTemplate DataWindow property.

For more information

"DataWindow support for XML” on page 17.
**JSP targets**

<table>
<thead>
<tr>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSP authoring in PowerBuilder 9 offers the same robust and rapid development that customers have come to expect from PowerBuilder. In particular, JSP authoring automate as many of the development tasks as possible and offer users an easy to use development environment for JSP authoring.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>In the beta 3 release of PowerBuilder 9, you can preview the following features of JSP authoring:</td>
</tr>
<tr>
<td>• JSP Web Target wizard</td>
</tr>
<tr>
<td>This wizard creates a JSP target that has build and source folders and a deployment configuration.</td>
</tr>
<tr>
<td>• JSP page authoring</td>
</tr>
<tr>
<td>You can add JSP pages to a JSP target and edit them. Menu options and dialog boxes are available for adding actions, directives, and scripting elements. Adding applets and JavaBeans to a JSP page inserts the appropriate JSP action.</td>
</tr>
<tr>
<td>• JSP Web Target object model</td>
</tr>
<tr>
<td>Several class and method changes were required for implementation of the Web Target object model in JSP targets. This includes a 4GL server-side event model that provides foundation classes to handle many of the details of coding server scripts for Web pages, including server control classes, the DataWindow class, server variables, and parameter classes.</td>
</tr>
<tr>
<td>• JSP deployment configuration</td>
</tr>
<tr>
<td>You can create a deployment configuration for deploying a JSP Web Application to either EAServer or the Apache Tomcat Server. Deployment works to either EAServer or Tomcat.</td>
</tr>
<tr>
<td>• Custom tag library for the Web DataWindow</td>
</tr>
<tr>
<td>A custom tag library for the Web DataWindow is installed with this release.</td>
</tr>
</tbody>
</table>

For more information

“JavaServer Pages targets” on page 57
Windows clients for Web services

**Description**
A PowerBuilder application can act as a client to Web services. The Web service can be integrated into a PowerBuilder application, hiding much of the complexity of WSDL, XML grammar, and SOAP development from the user. This feature has been tested with WSDL 1.1 and SOAP 1.1.

**For more information**
“Building a Web Service Client” on page 97

PowerBuilder Document Object Model

**Note**
This feature is new in this beta release and is still under development. It has not yet undergone full internal testing, and users are not expected to test the feature in this beta.

**Description**
PBDOM is the PowerBuilder implementation of the Document Object Model (DOM), a programming interface defining the means by which XML documents can be accessed and manipulated. Although PBDOM is not an implementation of the World Wide Web Consortium (W3C) DOM API, the PBDOM PowerBuilder API can be used for reading, writing, and manipulating standard-format XML from within PowerScript code. PBDOM portrays an XML document as a collection of interconnected objects and provides intuitive methods indicating the use and functionality of each object.

**For more information**
“PowerBuilder Document Object Model” on page 109

EJB clients for third-party application servers

**Description**
A PowerBuilder application can act as a client to an EJB component running on an application server that is J2EE compliant. Tested servers include Sybase EAServer 4.1, IBM WebSphere 4.0, and BEA WebLogic 6.1. For this beta release, clients have been tested with EJB components that conform to the EJB 1.1 specification.
Usage
To connect to the server and communicate with the EJB component, clients use a set of classes implemented in a DLL file, pbejbclient90.dll. To use this DLL, you must place it in the application’s path, and you must add the pbejbclient90.pbd file to the client application’s library search path. The PBD acts as a wrapper for the DLL, enabling the PowerBuilder client to use the classes in the DLL as though they were PowerBuilder custom class user objects.

The PowerBuilder client uses local proxy objects for the EJB component to delegate calls to the EJB component’s methods.

For more information
“Building an EJB Client” on page 207.

Source control enhancements
Description
PowerBuilder 9 includes enhancements to improve the speed and performance of source control integration.

Usage
The following enhancements have been added for the beta release:

• Performing source control operations on a PBL
• Check-list dialog box enhancements
• Source control history enhancements
• PBNative configuration enhancements
• Library painter list view enhancement
• Source control logging enhancement

For more information
“Source control enhancements” on page 241

The OrcaScript language
Description
OrcaScript allows you to write batch scripts to process PowerBuilder applications and files without using the PowerBuilder development environment. You can use OrcaScript to get the latest version of a PowerScript target from source control, build the target PBLs, deploy components to EAServer, and compile PowerBuilder executable files—all without operator intervention.
PowerBuilder Native Interface

Description
The PowerBuilder Native Interface (PNI) is an interface specification that enables you to develop independent modules in C++ and integrate them with PowerBuilder.

For more information
“Introduction to PNI” on page 257
“Building PowerBuilder Extensions” on page 263
“Working with PNI” on page 279
“PNI Reference” on page 295
“PNI Tool Reference” on page 383

Oracle9i support

Description
PowerBuilder 9 supports connection pooling, NCHAR or NVARCHAR2 datatypes, and LOB datatypes with Oracle9i.

Note
The Oracle9i support described in this section was introduced in the PowerBuilder 8.0.2 maintenance release.

Connection pooling
Connection pooling is the use of a group (the pool) of reusable physical connections by several sessions in order to balance loads. The management of the pool is done by OCI, not the application. Connection pooling is not used by default. On the Connection tab page of the Database Profile Setup dialog box for Oracle9i, select the Use Connection Pool check box to turn on connection pooling. You can also set the CnnPool DBParm to Yes.

You must provide the full net service name created by Net8 in the Server Name box in the Connection Profile, for example adcora8i.sybase.com. The value of Maximum Connection Pool is 10, which means that you can connect to 10 different Oracle servers distinguished by service name at the same time. The maximum number of connections in one pool is 20, which means that you can keep 20 connections in a single pool at the same time.
The connection pool is created by the Oracle9i client, and is able to connect to Oracle8, Oracle8i, and Oracle9i servers. When a connection pool has been created, PowerBuilder maintains a physical connection to the Oracle server until the PowerBuilder application closes or the OCI.DLL is unloaded.

You cannot use the CnnPool DBParm in a PowerBuilder component deployed to EAServer. External users, such as an NT operating system account, cannot participate in a connection pool.

**NCHAR and NVARCHAR2 support**

PowerBuilder supports the NCHAR and NVARCHAR2 datatypes when connecting to an Oracle9i server using the pbo9090.dll. This support is not available for Oracle8i because the DataWindow object is unable to generate a prefix required when sending an ANSI string to an NCHAR or NVARCHAR2 column in a table on an Oracle8i server. The prefix instructs the server to convert data to a Unicode string. Oracle9i servers do not need to convert ANSI strings to UNICODE strings.

For Oracle8i, a client statement might look like this:

```
Update table1 set C1 = N'NEW STRING' where C1 = N'OLD STRING';
```

where the datatype of C1 is NCHAR or NVARCHAR2, and the prefix N instructs the Oracle8i server to convert the data to a Unicode string.

**LOB output parameter in Oracle stored procedures**

Using the pbo9090.dll interface, you can define an LOB (Large OBject) as an output parameter for an Oracle stored procedure or function to retrieve blob data. There is no limit to the number of LOB output parameters that you can define for each stored procedure or function.

**Timestamp datatype**

Oracle9i provides an extension of the Date datatype called TimeStamp, which stores the year, month, and day of the Date value plus the hours, minutes, and seconds:

```
Timestamp[fractional_seconds_precision]
```

The `fractional_seconds_precision` value is optional and provides the number of digits for indicating seconds. The range of valid values for use with PowerBuilder is 0-6.
ConnectAs DBParm

Oracle9i provides an option to connect with SYSOPER or SYSDBA system privileges if those privileges have been granted to the user. When you connect with SYSDBA or SYSOPER privileges using a username and password, you connect with a default schema, not with the schema that is generally associated with your username. For SYSDBA this schema is SYS; for SYSOPER the schema is PUBLIC. Use the ConnectAs DBParm to specify a connection with SYSOPER or SYSDBA privileges. See the online Help for a description of the ConnectAs DBParm.

Note You cannot connect as SYSDBA or SYSOPER when CnnPool='YES', because Oracle connection pooling does not support this option.

Unicode database support

Description

For Adaptive Server Enterprise 12.5 and Oracle9i, PowerBuilder supports access to Unicode databases, including conversion of data between DBCS and Unicode. PowerBuilder also supports Unicode data types in these DBMSs.

For Adaptive Server Anywhere 7.x and later releases, PowerBuilder supports access to Unicode databases, including conversion of data between DBCS and Unicode.

For other DBMSs, PowerBuilder can connect, save, and retrieve data in a Unicode database, but does not convert data between DBCS and Unicode.

When data is saved to the database, PowerBuilder sends ANSI or DBCS data to the database. The database itself must guarantee that the data is saved as Unicode data correctly. When PowerBuilder retrieves data, the data is assumed to be ANSI or DBCS data.

Note The Unicode database and datatype support described in this section was introduced in the PowerBuilder 8.0.2 Maintenance Release.

Usage

A Unicode database is a database whose character set is set to a Unicode format (such as UTF-8, UTF-16, UCS-2, or UCS-4). All data must be in Unicode format, and any data saved to the database must be converted to Unicode data implicitly or explicitly.
A database that uses ANSI (or DBCS) as its character set may use special datatypes to store Unicode data. These datatypes are `NCHAR`, `NVARCHAR`, and `NVARCHAR2`. Columns with this data type can store only Unicode data. Any data saved into such a column must be converted to Unicode explicitly.

<table>
<thead>
<tr>
<th>DBMS</th>
<th>Interface</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adaptive Server Enterprise 12.5</td>
<td>Native</td>
<td>PowerBuilder supports Unicode databases. The conversion between DBCS and Unicode is done automatically. Note: the <code>lang</code> variable of Open Client must be set to a DBCS language such as <code>chs</code>, <code>cht</code>, and so forth. PowerBuilder can access Unicode data in columns with the datatypes <code>NCHAR</code> or <code>NVARCHAR</code>. The conversion is done automatically.</td>
</tr>
<tr>
<td>ODBC, OLE DB, JDBC</td>
<td></td>
<td>PowerBuilder supports Unicode databases. The conversion is done automatically.</td>
</tr>
<tr>
<td>Oracle9i</td>
<td>Native</td>
<td>PowerBuilder supports Unicode databases. The conversion between DBCS and Unicode is done automatically. PowerBuilder can access Unicode data in columns with the datatypes <code>NCHAR</code> or <code>NVARCHAR2</code>. The conversion is done automatically.</td>
</tr>
<tr>
<td>ODBC</td>
<td>Driver dependent. If the ODBC driver is Unicode-aware, PowerBuilder supports Unicode databases and <code>NCHAR</code> and <code>NVARCHAR2</code> access.</td>
<td></td>
</tr>
<tr>
<td>OLE DB, JDBC</td>
<td></td>
<td>PowerBuilder supports Unicode databases and <code>NCHAR</code> and <code>NVARCHAR2</code> access. The conversion is done automatically.</td>
</tr>
<tr>
<td>ASA 7.x, 8.x</td>
<td>Native</td>
<td>None (no native driver).</td>
</tr>
<tr>
<td>ODBC</td>
<td></td>
<td>PowerBuilder supports Unicode databases. The conversion is done automatically between Unicode and the local code page (your machine locale). Note: The <code>translator</code> option must be set so that the ODBC driver can perform the conversion.</td>
</tr>
<tr>
<td>OLE DB, JDBC</td>
<td></td>
<td>PowerBuilder supports Unicode databases. The conversion is done automatically.</td>
</tr>
</tbody>
</table>
XML and CSV import and save

Description

The ImportFile, ImportString, and ImportClipboard methods for DataWindows and Graph objects support XML and CSV as import formats, and the SaveAs methods support XML. You can also import and save using these formats in the DataWindow and Database painters.

Usage

The import methods have an optional first parameter that enables you to specify the type of data to be imported. For ImportFile, the valid types are Text!, CSV!, XML!, DBase2!, and DBase3!. If you specify this parameter, you do not need to include the extension in the import file name:

```
gr_1.ImportFile( XML!, "c:\data\customers")
```

If you do not specify this parameter, the extensions .TXT, .CSV, .XML, and .DBF are recognized. For ImportString and ImportClipboard, the valid datatypes are Text!, CSV!, and XML!.

You can create XML templates to control the import and export of most DataWindow objects. You cannot use templates with Graph objects or DataWindow objects that use the Graph, OLE, or Composite presentation styles.

For more information

“XML support in the DataWindow” on page 20
“XML support in import methods” on page 44

Saving as XSL-FO

Description

Building on the ability to save data as XML, PowerBuilder can also save the DataWindow’s data and presentation as a document or string using XSL Formatting Objects (XSL-FO).

**Note** This feature is new in this beta release and is still under development. It has not yet undergone full internal testing, and users are not expected to test the feature in this beta.

Usage

The XSL (Extensible Stylesheet Language) W3C Recommendation has two parts, XSLT and XSL-FO. XSLT provides the transformation typically used to present XML documents as HTML in a browser. XSL-FO provides extensive formatting capabilities that are not dependent on the output format.
Processors such as the Apache XML FOP Project can convert XSL-FO documents into several output formats including PDF, PCL, and AWT. You can save a DataWindow object as XSL-FO, then download and use the processor to convert the XSL-FO string to the format of your choice, applying your own customizations to the conversion. For a DataWindow named dwemp, the following command lines show the syntax for producing a PDF, a print preview rendered on screen (-awt), and printable output rendered and sent to a printer (-print):

```
Fop dwemp.fo dwemp.pdf
Fop dwemp.fo -awt
Fop dwemp.fo -print
```

You can use the SaveAs method or the File>Save Rows As DataWindow painter menu item to save as XSL-FO.

For more information, see Extensible Stylesheet Language (XSL) Version 1.0 at http://www.w3.org/TR/xsl/. You can download the Apache XML Formatting Objects processor from the Apache XML Project Download page at http://xml.apache.org/fop/download.html.

### DBCS versions of string functions

**Description**

Separate versions of character-based string functions are available for use in DBCS environments. Characters in DBCS environments can be single byte, double byte, or mixed, but the standard string functions return results for single-byte characters only. The DBCS functions take the same arguments as their corresponding SBCS functions.

**Usage**

In PowerBuilder 9, the following functions have been modified to support double byte or mixed character sets:

<table>
<thead>
<tr>
<th>SBCS function</th>
<th>DBCS function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fill</td>
<td>FillW</td>
</tr>
<tr>
<td>Left</td>
<td>LeftW</td>
</tr>
<tr>
<td>LeftTrim</td>
<td>LeftTrimW</td>
</tr>
<tr>
<td>Match</td>
<td>MatchW</td>
</tr>
<tr>
<td>Mid</td>
<td>MidW</td>
</tr>
<tr>
<td>Pos</td>
<td>PosW</td>
</tr>
<tr>
<td>Right</td>
<td>RightW</td>
</tr>
<tr>
<td>RightTrim</td>
<td>RightTrimW</td>
</tr>
</tbody>
</table>
DataWindow enhancements

<table>
<thead>
<tr>
<th>SBCS function</th>
<th>DBCS function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Trim</td>
<td>TrimW</td>
</tr>
</tbody>
</table>

On SBCS machines, the DBCS versions of the string functions return the same results as the SBCS versions. On DBCS machines, you can use the SBCS functions, but to make certain you obtain character-based results, you must use the DBCS functions.

The LenW function that returns the length of a string or a blob in DBCS environments was already added in PowerBuilder 8 and continues to be supported in PowerBuilder 9.

DataWindow enhancements

Description

The following DataWindow enhancements have been made in PowerBuilder 9:

- DataWindow print enhancements
- Child DataWindow retrieval
- Group report scrolling

DataWindow print enhancements

Several print enhancements have been added for DataWindows in PowerBuilder 9 applications:

- A new property allows clipping the text in a text field on a printed report
  The DataWindow.Print.ClipText property allows you to clip the text of a static field to the dimensions of a text field when the text field has no visible border setting. Text is automatically clipped for text fields with visible border settings even if this property is not set. Values for this property are:
    Yes — The printed text does not overrun the text field
    No — (Default) The entire text can overrun the text field dimensions on printed pages

- A new property allows overriding default specifications of a print job
  The DataWindow.Print.OverridePrintJob property allows you to override the print job print settings defined in the PrintOpen method with the print specifications of the DataWindow. Values are:
    Yes — Override the print job print settings
    No — (Default) Do not override the print job print settings
New Features in PowerBuilder 9

- A new property allows you to specify a printer for a DataWindow report

The DataWindow.Print.PrinterName property allows you to direct a report to a specific printer. You can specify the printer name programmatically or in the Properties view of the DataWindow painter. If a printer name is not specified, print output is directed to the default printer for the user’s machine.

- Printing multiple copies of a DataWindow report is enhanced

The DataWindow.Print.Copies property can now be used to print multiple copies of a report even when the printer driver does not support the multiple-copies specification of a print job. If the printer driver does not support multiple copies, then the DataWindow engine compensates for the driver by repeatedly printing the current page for the indicated number of copies.

- The Print.Collate property can be set in the DataWindow painter

The DataWindow property to allow collating printed copies of a DataWindow report can now be set in the PowerBuilder UI. Previously you had to set this property programmatically. The default value for this property has also been changed from No to Yes to maintain behavior in existing applications where this property was set.

You can set all the new DataWindow properties in PowerScript dot notation and with the Describe and Modify methods. You can also set most of the new properties on the Printer Specifications page of the DataWindow Properties view in the DataWindow painter.

<table>
<thead>
<tr>
<th>DataWindow print property</th>
<th>UI option</th>
</tr>
</thead>
<tbody>
<tr>
<td>DataWindow.Print.ClipText (new)</td>
<td>Clip Text check box</td>
</tr>
<tr>
<td>DataWindow.Print.Collate (enhanced)</td>
<td>Collate Copies check box</td>
</tr>
<tr>
<td>DataWindow.Print.Copies (enhanced)</td>
<td>Not available in UI</td>
</tr>
<tr>
<td>DataWindow.Print.PrinterName (new)</td>
<td>Printer Name text box</td>
</tr>
<tr>
<td>DataWindow.Print.OverridePrintJob (new)</td>
<td>Override Print Job check box</td>
</tr>
</tbody>
</table>

Child DataWindow retrieval

In PowerBuilder 9, you can prevent the automatic retrieval of a child DataWindow when you retrieve the contents of the parent DataWindow. You can do this by clearing the AutoRetrieve check box for a DropDownDataWindow control on the Edit page of the control’s property sheet or by programmatically setting the new Edit.AutoRetrieve property to No.
The AutoRetrieve property is turned on by default. If you turn off the automatic retrieval property of a child DataWindow, you can retrieve the child DataWindow by obtaining its handle and using the handle to trigger the retrieval. In this case, you can wait until retrieval of the parent DataWindow is complete before programmatically retrieving the child DataWindow.

When you turn the AutoRetrieve property off and retrieve a child DataWindow programmatically, subsequent Reset or Retrieve calls on the parent DataWindow do not reset the contents of the child DataWindow.

A check box for the DataWindow.HideGrayLine property has been added to the General page of the Properties view for a DataWindow in the DataWindow painter. The check box is enabled only for a DataWindow with group headers. The HideGrayLine property allows you to hide a gray line indicating that a fixed page has been crossed when scrolling in a DataWindow with group headers. By default this property is turned off. You must select this property or set it programmatically to Yes to prevent the appearance of a gray line during scrolling of a group DataWindow.

**Debugger enhancements**

**Note** This feature is new in this beta release and is still under development. It has not yet undergone full internal testing, and users are not expected to test the feature in this beta.

**Description**

TipWatch and QuickWatch features have been added to the debugger to enable you to get the current value of variables and expressions from the Source view when execution stops at a breakpoint. You can also copy strings from the Source view to other tools in the debugger, and use predefined shortcut key combinations for common actions such as continue and step in, out, and over.

**TipWatch**

When execution stops at a breakpoint, you can place the mouse pointer over a variable in the Source view to display a pop-up tip window that shows the current value of that variable. TipWatch shows the current value for variables of simple datatypes. For object types, it shows only an internal identifier. For array types, it shows {...} to indicate that more information is available. To show complete information for object type and array type variables, use QuickWatch.
You can also select a simple expression to display its current value. If the selected string cannot be parsed, the pop-up window does not display. TipWatch does not evaluate function, assignment, or variable value modification expressions.

When you are debugging a remote component, Tip Watch does not evaluate expressions or indirect variables.

QuickWatch

When execution stops at a breakpoint, you can move the edit cursor to a variable or select an expression in the Source view and select QuickWatch from the Debug or pop-up menu to open the QuickWatch dialog box. You can also use the Shift+F9 shortcut.

The QuickWatch dialog box displays the current value of simple variables and detailed information about object variables, including the values of all fields in the variable. QuickWatch can also evaluate function expressions.

**Exercise caution when evaluating expressions**

QuickWatch evaluates all kinds of expressions, including functions, in local debugging. If you select a function and activate QuickWatch, the function is executed. This may have unexpected results. For example, if you select `dw_1.print()` and activate QuickWatch, the DataWindow is printed.

You can also:

- Select an item in the tree view and click Change Value or double-click the tree view item to open the Modify Variable dialog box so that you can change the value of the variable.
**Debugger enhancements**

- Change the variable or expression in the Expression text box and click Reevaluate to display the new value of the string in the tree view immediately.

- Click Add Watch to add the current watch variable or expression to the Watch view in the debugger.

When you are debugging a remote component, expressions and indirect variables are not evaluated, and you cannot modify variable values.

**Copy menu item**

When text is selected in the Source view in the debugger, you can select Copy from the pop-up menu in the Source view to copy the string to the clipboard. You can then paste the string into another dialog box to search for the string, insert a watch, or add a conditional breakpoint.

**Shortcut keys**

Default shortcut keys have been added for most items on the Debug menu. If you prefer to use a different set of shortcuts, select Tools>Keyboard Shortcuts to replace the defaults with your own key combinations.
DataWindow support for XML

DataWindow row data can be exported and imported in the Extensible Markup Language (XML). You can specify the XML logical structure of how the row data iterates inside the root element of the XML document. This is done with the introduction of export template objects, encapsulated in DataWindow objects, which are graphically constructed in a new View in the DataWindow Painter.

This section includes the following topics:
- About XML
- XML support in the DataWindow
- Exporting to XML
- Importing XML
- PowerScript enhancements for XML support

About XML

Like Hypertext Markup Language (HTML), Extensible Markup Language (XML) is a subset of Standardized General Markup Language (SGML) and has been designed specifically for use on the Web.

XML, however, is more complete and disciplined than HTML, and it is also a framework for creating markup languages—it allows you to define your own application-oriented markup tags. XML provides a set of rules for structuring data. Like HTML, XML uses tags and attributes, but the tags are used to delimit pieces of data, allowing the application that receives the data to interpret the meaning of each tag. These properties make XML particularly suitable for data interchange.

An XML document is made up of declarations, elements, comments, character references, and processing instructions, indicated in the document by explicit markup.
The simple XML document that follows contains an XML declaration followed by the start tag of the root element, <d_dept_list>, nested row and column elements, and finally the end tag of the root element. The root element is the starting point for the XML processor.

```xml
<?xml version="1.0">
<d_dept_list>
  <d_dept_list_row>
    <dept_id>100</dept_id>
    <dept_name>R &amp; D</dept_name>
    <dept_head_id>501</dept_head_id>
  </d_dept_list_row>
  ...
</d_dept_list>
```

This section contains a brief overview of XML rules and syntax. For a good introduction to XML, see the summary at http://www.w3.org/XML/1999/XML-in-10-points. For more detailed information, see the XML 1.0 specification defined by the World Wide Web Consortium (W3C) at http://www.w3.org/TR/REC-xml, or one of the many books about XML.

### Valid and well-formed XML documents

**An XML document must be valid, well-formed, or both.**

**Valid documents**

To define a set of tags for use in a particular application, XML uses a separate document named a document type definition (DTD). A DTD states what tags are allowed in an XML document and defines rules for how those tags can be used in relation to each other. It defines the elements that are allowed in the language, the attributes each element can have, and the type of information each element can hold. Documents can be verified against a DTD to ensure that they follow all the rules of the language. A document that satisfies a DTD is said to be valid.

If a document uses a DTD, the DTD must immediately follow the declaration.

**Well-formed documents**

The second way to specify XML syntax is to assume that a document is using its language properly. XML provides a set of generic syntax rules that must be satisfied, and as long as a document satisfies these rules, it is said to be well-formed. All valid documents must be well-formed.
Processing well-formed documents is faster than processing valid documents because the parser does not have to verify against the DTD. When valid documents are transmitted, the DTD must also be transmitted if the receiver does not already possess it. On the other hand, well-formed documents can be sent without other information.

XML documents should conform to a DTD if they are going to be used by more than one application. If they are not valid, there is no way to guarantee that various applications will be able to understand each other.

**XML syntax**

There are a few more restrictions on XML than on HTML; they make parsing of XML simpler.

Unlike HTML, XML does not allow you to omit tags. This guarantees that parsers know where elements end.

The following example is acceptable HTML, but not XML:

```html
<table>
  <tr>
    <td>Dog</td>
    <td>Cat</td>
    <td>Mouse
  </tr>
</table>
```

To change this into well-formed XML, you need to add all the missing end tags:

```xml
<table>
  <tr>
    <td>Dog</td>
    <td>Cat</td>
    <td>Mouse</td>
  </tr>
</table>
```

Empty elements cannot be represented in XML in the same way they are in HTML. An empty element is one that is not used to mark up data, so in HTML, there will not be an end tag. There are two ways to handle empty elements.

The first is to place a dummy tag immediately after the start tag. For example:

```xml
<img href="picture.jpg"></img>
```
The second method is to use a slash character at the end of the initial tag:

```xml
<img href="picture.jpg"/>
```

This tells a parser that the element consists only of one tag.

XML is case sensitive

XML is case sensitive, which allows it to be used with non-Latin alphabets. You must ensure that letter case matches in start and end tags:

```xml
<MyTag>

and

</MyTag>

belong to two different elements.

White space

White space within tags in XML is unchanged by parsers.

All elements must be nested.

All XML elements must be properly nested. All child elements must be closed before their parent elements close.

**XML support in the DataWindow**

PowerBuilder supports both the export and import of XML in DataWindow and DataStore objects. The possible uses of this feature include the following:

- You can code events in data entry or data reporting applications to export selected data values, or the entire contents of a DataWindow object, to a structured XML document. The structure of the XML document can be customized for use by other internal or external applications, processes, or systems.

- You can add a method to a custom class user object that uses DataStore objects for server-side database processing or middle-tier management of a client-side DataWindow object. The method would export data to XML, which could then be processed by a different component or subsystem, such as an Enterprise JavaBeans component or a Web service.

- A similar method could be added to a Web application that uses a Web DataWindow. The method might be invoked by a user action, such as selecting checkout in a shopping cart application.
Exporting to XML

You can export the data in a DataWindow or DataStore object to XML using any of the techniques used for exporting to other formats such as PSR or HTML:

- Using the SaveAs method:
  ```powerbuilder
ds_1.SaveAs("C:\TEMP\Temp.xml", XML!, TRUE)
  ```
- Using PowerScript dot notation:
  ```powerbuilder
l_s_xmlstring = dw_1.Object.DataWindow.Data.XML
  ```
- Using the Save Rows As menu item in the DataWindow painter:

  With the Preview view open, select File>Save Rows As, select XML from the Files of Type drop-down list, provide a file name, and click Save. You can use this at design time to preview the XML that will be generated at runtime.

When you export data, PowerBuilder uses an export template to specify the content of the generated XML.

---

Default export format

If you have not created or assigned an export template, PowerBuilder uses a default export format. This is the same format used when you create a new default export template. See “Creating templates” on page 24.

Composite, Graph, and OLE DataWindow objects cannot be imported or exported using a template. You must use the default export format. Graph objects must also be imported and exported using the default format.

---

Export templates

An export template lets you customize the XML that is generated.

You can specify optional XML and document type declarations that precede the root element in the exported XML, as well as the logical structure and nesting level of iterative DataWindow row data inside the root element. The children of the root element can contain elements, character references, and processing instructions as well as the row data, using explicit markup. For more information, see “Header and Detail sections” on page 26.
If the exported XML is used by different applications or processes, you can define a separate export template for each use.

The names of all templates that you create and save for the current DataWindow object display in the Use Template drop-down list for export templates on the Data Export page in the Properties view.

The template you select from the list is used to conform the XML generated by any of the methods of saving as XML to the specifications defined in the named template. Selecting a template from the list box sets the DataWindow object’s Export.XML.UseTemplate property. You can also modify the value of the UseTemplate property dynamically in a script. For example, an XML publishing engine would change templates dynamically to create different presentations of the same data.

When you open a DataWindow, the Export Template view displays the template specified in the DataWindow object’s UseTemplate property. If the property has not been set, the first saved template displays or, if there are no saved templates, the default structured template displays as a basis for editing.

When the DataWindow object is saved as XML, PowerBuilder uses the template specified in the UseTemplate property. If the property has not been set, PowerBuilder uses the default template.

When you use the File>Save Rows As menu item to save to XML in the development environment, PowerBuilder uses the template specified in the UseTemplate property, which may not be the template currently displayed in the Export Template view.

You can open a different template or create a new template in the Export Template view, but if you want to see the results of your changes to the template in the exported XML, you must:

1. Save the template.
2. Change the UseTemplate property to the open template.
Exporting metadata

You can also specify that metadata in the form of a DTD or schema should be exported when you save the DataWindow, and whether the metadata should be saved with the XML or in a separate file. For more information about the properties you can set on the Data Export page, see Export.XML.MetaDataType on page 49, Export.XML.SaveMetaData on page 50, and Export.XML.UseTemplate on page 53.

The Export Template view

You define and edit export templates in the Export Template view in the DataWindow painter. The view uses a TreeView control to represent the template.

Import templates

You can use templates designed in this view as import templates, but only the mapping of column names to element attribute names is used for import. All other information in the template is ignored.

When you create a new DataWindow object, PowerBuilder displays a default template in the Export Template view. You can edit only one template at a time in the Export Template view, but you can create multiple templates and save them with the DataWindow object. Each template is uniquely associated with the DataWindow object open in the painter.

From the pop-up menu for the Export Template view (with nothing selected), you can create new templates, open an existing template, save the current template, or delete the current template. You can only open and edit templates that are associated with the current DataWindow object.
The Export Template view

Creating templates

Creating new base templates

The New menu item creates a template that is empty except for the XML declaration, the root element, and the first element of the row data section, referred to as the Detail Start element. The name of the root element is the same as the name of the DataWindow, and the default name for the Detail Start element is the name of the root element with _row appended. For example:

```xml
<?xml version="1.0"?>
<d_name>
 <d_name_row/>
</d_name_row>
</d_name>
```

Creating new default templates

The New Default menu item creates a template with the same contents as the New menu item, as well as a flat structure of child elements of the Detail Start element. A child element is created for each DataWindow column name, with the exception of blob and computed columns, and the default tag for the element is the column’s name.

If the names of the column and the control are the same, the content of the child element displays with a control reference icon. If there is no control name that matches the column name, the content of the child element displays using the DataWindow expression icon. For example, consider this DataWindow object in which the dept_id column is used as a retrieval argument and does not display. The SQL syntax is:

```sql
SELECT "employee"."dept_id",
    "employee"."emp_lname",
    "employee"."emp_fname",
    "employee"."salary"
FROM "employee"
WHERE employee.dept_id = :deptnum
ORDER BY "employee"."emp_lname" ASC
```
In the default template, dept_id uses the DataWindow expression icon. All the other columns used the control reference icon.

![DataWindow](image)

**Saving templates**

To save a new template, give it a name and optionally a comment that identifies its use.

The template is stored with the DataWindow object in the PBL or SRD. After saving a template with a DataWindow object, you can see its basic definition in the Source editor for the DataWindow object. For example, this DataWindow has two templates with required elements only:

```xml
<export.xml(usetemplate="t_address" 
    template=(comment="Employee Phone Book" 
        name="t_phone" xml="<d_emplist><d_emplist_row 
            __pbband=~"detail~"></d_emplist_row></d_emplist>")
```
Header and Detail sections

An XML export template has a Header section and a Detail section, separated graphically by a line across the tree view.

The items in the Header section are generated only once when the DataWindow is exported to XML, unless the DataWindow is a group DataWindow. For group DataWindow objects, you can choose to generate the contents of the header section iteratively for each group. For more information, see “Generating group headers” on page 27.

The Detail section contains the row data, and is generated iteratively for each row in the DataWindow object.

Header section

The Header section can contain the following items. Only the root element is required:

- XML declaration
  This must be the first item in the tree view if it exists.

- Document type declaration
  If there is an XML declaration, the document type declaration must appear after the XML declaration and any optional processing instructions and comments, and before the root element. Otherwise, this must be the first item in the tree view.

- Comments
DataWindow support for XML

- Processing instructions
- Root element (start tag)

The Header section can also contain child elements, but they cannot be iterative except in the case of group DataWindows.

**Note** The root element displays in the Header section, but the entire content of the Detail section is contained in the root element.

Generating group headers

You can generate the contents of the header section iteratively for each group in a group DataWindow by checking the Iterate Header for Groups check box on the Data Export page in the Properties view, or by setting the Export.XML.HeadGroups DataWindow property. This property is on by default.

For example, a group DataWindow object includes the columns `sales_order_id` and `sales_order_order_date`.

The root element in the Header section of the template, Orders, has a child element, Order. Order has an id attribute whose value is a control reference to the column `sales_order_id`. Order also has a child element, OrderDate, that contains a column reference to the `sales_order_order_date` column. These elements make up the header section that will be iterated for each group.

The Detail Start element, Item, has an id attribute whose value is a control reference to the column `sales_order_items_line_id`. It also has three child elements that contain column references to the line items for product ID, quantity, and ship date.

When the DataWindow is exported with the Export.XML.HeadGroups property on, the order ID and date iterate for each group. The following XML output shows the first three iterations of the group header:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<Orders>
```

Sybase Preliminary and Confidential Beta 3 Draft
The Export Template view

```xml
<Order id="2001">
  <OrderDate>1996-03-14</OrderDate>
  <Item id="1">
    <Product>300</Product>
    <Qty>12</Qty>
    <ShipDate>1996-09-15</ShipDate>
  </Item>
  <Item id="2">
    <Product>301</Product>
    <Qty>12</Qty>
    <ShipDate>1996-09-14</ShipDate>
  </Item>
  <Item id="3">
    <Product>302</Product>
    <Qty>12</Qty>
    <ShipDate>1996-09-14</ShipDate>
  </Item>
</Order>
<Order id="2002">
  <OrderDate>1996-03-18</OrderDate>
  <Item id="2">
    <Product>401</Product>
    <Qty>24</Qty>
    <ShipDate>1996-09-18</ShipDate>
  </Item>
  <Item id="1">
    <Product>400</Product>
    <Qty>24</Qty>
    <ShipDate>1996-09-18</ShipDate>
  </Item>
</Order>
<Order id="2003">
  <OrderDate>1996-03-21</OrderDate>
  <Item id="3">
    <Product>400</Product>
    <Qty>12</Qty>
    <ShipDate>1996-09-23</ShipDate>
  </Item>
  ... 
</Order>
```

Detail section

The Detail section, which holds the row data, can contain the following items. Only the Detail Start element is required:

- Detail Start element
• Child or sibling elements to the Detail Start element
• DataWindow control references in text format
• DataWindow expressions (including DataWindow column references)
• Literal text (text not corresponding to a DataWindow control)
• Attributes
• Comments
• Processing instructions
• CDATA sections

You can assign attributes to all element types.

The Detail Start element can be a nested (or multiply-nested) child of an element from the Header section, permitting a nested detail. This may be useful for DataStores being packaged for submission to external processes, such as B2B, that require company and/or document information, date, or other master data preceding the detail.

If no Detail Start element is specified (that is, if the Starts Detail option has been deselected) only one iteration of row data is generated.

Representing tree view items

Each declaration, element, and so forth displays as a single tree view item with an image and font color that denotes its type. The end tags of elements and the markup delimiters used in an XML document do not display but are assumed.

```
<xs:element name="d_mvdw">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="d_mvdw_row"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```
Moving the separator

A line across the Export Template view separates the Header section from the Detail section. You can change the location of this line by selecting the element that you want as the Detail Start element and selecting Starts Detail from its pop-up menu.

There can be only one Detail Start element, and it must be inside the document’s root element. By default, the first child of the root element is the Detail Start element. It usually wraps a whole row, separating columns across rows. When the DataWindow is exported to XML, only this element and all children and/or siblings after it are iteratively generated for each row. Any elements in the root element above the separator line are generated only once.

Editing the export template

Every item in the Export Template view has a pop-up menu from which you can perform actions appropriate to that item, such as editing or deleting the item, adding or editing attributes, adding child elements or other items, and inserting elements, processing instructions, CDATA sections, and so forth, before the current item.

If an element has no attributes, you can edit its tag in the Export Template view by selecting it and left-clicking the tag or pressing F2. Literal text nodes can be edited in the same way. You can delete items (and their children) by pressing the Delete key.

The examples in this section show the delimiters used in the XML document. When you edit the template in dialog boxes opened from the Export Template view, you do not need to type these delimiters in text boxes.
The rest of this section describes some of the items in the export template. For more information, see the XML 1.0 specification at http://www.w3.org/TR/REC-xml.

**Using templates for data import**

You can use templates created in the Export Template view for import, but any DataWindow expressions, text, CDATA sections, comments, and processing instructions are ignored when data is imported.

**XML declaration**

The XML declaration specifies the version of XML being used. You may need to change this value for a future version of XML. It can also contain an encoding declaration and a standalone document declaration. From the pop-up menu, you can edit the declaration, and, if the document is well-formed, delete it. If you have deleted the XML declaration, you can insert one from the Insert Before item on the pop-up menu for the next entity in the template.

**Encoding declaration**

The encoding declaration specifies the character-set encoding used in the document, such as UTF-16 or ISO-10646-UCS-4.

**Note** This feature is new in this beta release and is still under development. It has not yet undergone full internal testing, and users are not expected to test the feature in this beta.

If there is no encoding declaration, the value defaults to UTF-8 encoding in ASCII environments. In DBCS environments, the default is the default system encoding on the computer where the XML document is generated. This ensures that the document displays correctly as a plain text file. However, since the DBCS data is serialized to Unicode, XML documents that use UTF-8, UTF-16 Big Endian, or UTF-16 Little Endian can all be parsed or generated correctly on DBCS systems.

Several other encodings are available, including ASCII, UCS4 Big Endian, UCS4 Little Endian, EBCDIC code pages IBM037 and IBM1140, ISO Latin-1, and Latin 1 Windows (code page 1252). You can select these values from a drop-down list box in the XML Declaration dialog box.
The Export Template view

**Standalone document declaration**

The standalone document declaration specifies whether the document contains no external markup that needs to be processed and can therefore stand alone (Yes), or that there are, or might be, external markup declarations in the document (No). The value in the default template is No, and if there is no standalone document declaration, the value is assumed to be No.

**Example**

This is an XML declaration that specifies XML version, encoding, and whether the document can stand alone:

```xml
<?xml version="1.0" encoding="UTF-16" standalone="yes"?>
```

**Document type declaration**

The document type declaration constrains the sequence and nesting of tags, attribute values, names and formats of external references, and so forth. You can edit the document type declaration to change its name, but the name must always be the same as the name of the root element. Changing the name in the document type declaration or the root element automatically changes the name in the other.

**Adding DTDs**

You can also add an identifier pointing to an external Document Type Definition (DTD) subset, and you can add an internal DTD subset. If you supply both external and internal subsets, entity and attribute-list declarations in the internal subset take precedence over those in the external subset.

**Public identifiers**

An external identifier can include a public identifier that can be used by an XML processor to generate an alternative URI. If an alternative URI cannot be generated, the URI provided in the system identifier is used. External identifiers without a public identifier are preceded by the keyword SYSTEM. External identifiers with a public identifier are preceded by the keyword PUBLIC.

**Exporting metadata**

If you specify a system or public identifier and/or an internal subset in the Document Type Declaration dialog box, a DTD cannot be generated when the data is exported to XML. A MetaDataType of XMLDTD! is ignored. For more information about the properties that control the export of metadata, see Export.XML.MetaDataType and Export.XML.SaveMetaData on page 50.

**Examples**

These are examples of valid document type declarations.

An external system identifier:

```xml
<!DOCTYPE d_dept_listing SYSTEM "d_dept_listing.dtd">
```
An external system identifier with a public identifier:

```xml
<!DOCTYPE d_test PUBLIC "--/MyOrg/DTD Test//EN"
 "http://www.mysite.org/mypath/mytest.dtd">
```

An external system identifier with an internal DTD. The internal DTD is enclosed in square brackets:

```xml
<!DOCTYPE d_orders
 SYSTEM "http://www.acme.com/dtds/basic.dtd"[
 <!ELEMENT Order (Date, CustID, OrderID, Items*)>
 <!ELEMENT Date (#PCDATA)>
 <!ELEMENT CustID (#PCDATA)>
 <!ELEMENT OrderID (#PCDATA)>
 <!ELEMENT Items (ItemID, Quantity)>
 <!ELEMENT ItemID (#PCDATA)>
 <!ELEMENT Quantity (#PCDATA)>
 ]>
```

**Comments**

Comments can appear anywhere in a document outside other markup. They can also appear within the document type declaration in specific locations defined by the XML specification.

Comments begin with `<!--` and end with `-->`. You cannot use the string `--` (a double hyphen) in a comment, and parameter entity references are not recognized in comments.

**Example**

```
<!-- this is a comment -->
```

**Processing instructions**

Processing instructions (PIs) enable you to provide information to the application that uses the processed XML. Processing instructions are enclosed in `<?` and `?>` delimiters and must have a name, called the target, followed by optional data that is processed by the application that uses the XML. Each application that uses the XML must process the targets that it recognizes and ignore any other targets.

The XML declaration at the beginning of an XML document is an example of a processing instruction. You cannot use the string `xml` as the name of any other processing instruction target.

**Example**

In this example, `usething` is the name of the target, and `thing=this.thing` is the data to be processed by the receiving application:
<usething thing=this.thing/>

Root element

You can change the name of the root element, add attributes and children, and insert comments, instructions, and, if they do not already exist, XML and/or document type declarations before it.

Changing the name of the root element changes the name of its start and end tags. You can change the name using the Edit menu item, or in the Element Attributes dialog box. Changing the name of the document type declaration, if it exists, also changes the name of the root element. The root element name is always the same as the document type declaration name.

You can add the following kinds of children to the root element:

- Elements
- Text
- DataWindow control references
- DataWindow expressions (including DataWindow column references)
- CDATA sections
- Comments
- Processing instructions

DataWindow controls and expressions

Adding a DataWindow control reference opens a dialog box containing a list of the columns, computed fields, and text controls in the document.

Adding a DataWindow expression opens the Modify Expression dialog box. This enables references to columns from the data source of the DataWindow. It also enables the calling of global functions. One use of this feature is to return a fragment of XML to embed, providing another level of dynamic XML generation.

DataWindow controls or expressions can also be referenced for element attribute values. To do so, select Edit/Add Attribute from the pop-up menu for elements. For each attribute specified, you can select a control reference from the drop-down list box, or you can enter a literal text value. A literal text value takes precedence over a control reference. You can also use the expression button to the right of the Text box to enter an expression.
The expression button and entry operates similarly to DataWindow properties in the Properties View. The button shows a green picture if an expression has been entered, and a red one if not. A control reference or text value specified in addition to the expression is treated as a default value. In the template, this combination is stored with the control reference or text value, followed by a tab, preceding the expression. For example:

```
attribute_name="text_val--tdw_expression~"
```

Control references can also be added to empty attribute values or element contents using drag-and-drop from the Control List view. DataWindow column references (in the form of expressions) can also be added using drag-and-drop from the Column Specification view.

---

**Drag-and-drop cannot replace**

You cannot drag-and-drop an item on top of another item to replace it. For example, if you want to replace a DataWindow control reference with another DataWindow control reference, or with a DataWindow expression, you first need to delete the control reference you want to replace.

---

**Importing XML**

You can import data from an XML document or string using the `ImportFile`, `ImportString`, or `ImportClipboard` methods. These methods have an optional first parameter that enables you to specify the type of data to be imported. See “XML support in import methods” on page 44.

You can also select XML as a file type in the dialog box that displays when you select Rows>Import in the DataWindow painter.

You can import data from an XML document or string with or without a template. To import data without a template, the data must correspond to the DataWindow column definition. The text content of the XML elements must match the column order, column type, and validation requirements of the DataWindow columns.
**Importing with a template**

If the XML document or string from which you want to import data does not correspond to the DataWindow column definition, or if you want to import attribute values, you must use a template.

The XML import template can be defined in the Export Template view. If you are defining a template for use only as an import template, do not include DataWindow expressions, text, CDATA sections, comments, and processing instructions. These items are ignored when data is imported. Element and attribute contents in the header section are also ignored.

If an export template for a DataWindow object exists, it can be used as an import template. Only the mapping of column names to element attribute names is used for import. All other information in the template is ignored.

An XML import template must map the XML element and attribute names in the XML document to DataWindow column names, and it must reflect the nesting of elements and attributes in the XML.

The order of elements and attributes with column reference content does not have to match the order of columns within the DataWindow, because import values are located by name match and nesting depth within the XML. However, the order of elements and attributes in the template must match the order in which elements and attributes occur in the XML. In this beta release, each element or attribute that has column reference content in the template must occur in each row in the XML document or string. The required elements and attributes in the XML can be empty.

The data for the DataWindow is held in the columns of the data table. Some data columns, such as those used for computed fields, may not have an associated control. To import data into a column that has no control reference, add a child DataWindow expression that contains the column name.

**Note** When you select a column name in the DataWindow Expression dialog box, tab characters are added before and after the name. You should remove these characters before saving the expression.
Setting the import template

The names of all templates for the current DataWindow object display in the Use Template drop-down list box for export templates on the Data Import page in the Properties view.

The template you select in the list box is used to conform the XML imported to the specifications defined in the named template. Selecting a template from the list box sets the DataWindow object’s Import.XML.UseTemplate property. You can also modify the value of the Import.XML.UseTemplate property dynamically in a script.

Example

This example uses a DataWindow object that includes the columns `emp_id`, `emp_fname`, `emp_lname`, and `dept_id`. The template used in this example includes only these columns. Any other columns in the DataWindow remain empty when you import using this template.

To illustrate how template import works, create a new template that has one element in the header section, called `before_detail_marker`. This element contains a column reference to the `emp_id` column.

The Detail Start element, `employee`, has an attribute, `dept_id`, whose value is a control reference to the column `dept_id`. It also has three children:

- The `emp_id` element contains a column reference to the `emp_id` column.
- The `emp_fname` element contains static text.
• The name element has two children, emp_fname and emp_lname, that contain column references to those columns.

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<employee_list>
  <before_detail_marker>102</before_detail_marker>
  <employee dept_id="100">
    <emp_id>102</emp_id>
    <emp_fname>static text content</emp_fname>
    <name>
      <emp_fname>Fran</emp_fname>
      <emp_lname>Whitney</emp_lname>
    </name>
  </employee>
  <employee dept_id="100">
    <emp_id>105</emp_id>
    <emp_fname>static text content</emp_fname>
    <name>
      <emp_fname>Matthew</emp_fname>
      <emp_lname>Cobb</emp_lname>
    </name>
  </employee>
  ...
```

The template exports and imports the dept_id DataWindow column using the attribute of the employee element, and the emp_id, emp_fname, and emp_lname columns using the elements’ contents. The following shows the beginning of the XML exported using this template:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<employee_list>
  <before_detail_marker>102</before_detail_marker>
  <employee dept_id="100">
    <emp_id>102</emp_id>
    <emp_fname>static text content</emp_fname>
    <name>
      <emp_fname>Fran</emp_fname>
      <emp_lname>Whitney</emp_lname>
    </name>
  </employee>
  <employee dept_id="100">
    <emp_id>105</emp_id>
    <emp_fname>static text content</emp_fname>
    <name>
      <emp_fname>Matthew</emp_fname>
      <emp_lname>Cobb</emp_lname>
    </name>
  </employee>
  ...
```
The exported XML can be reimported into the DataWindow columns `dept_id`, `emp_id`, `emp_fname`, and `emp_lname`. Before importing, you must set the import template on the Data Import page in the Properties view, or in a script using the DataWindow object’s `Import.XML.UseTemplate` property.

The following items are exported, but ignored on import:

- The before_detail_marker element is ignored because it is in the header section.
- The first occurrence of the element tag name `emp_fname` is ignored because it does not contain a mapping to a DataWindow column name.

If you change the nesting of the `emp_fname` and `emp_lname` elements inside the name element, the import fails because the order of the elements and the nesting in the XML and the template must match.

**Default data import**

When there is no import template assigned to a DataWindow object with the `Import.XML.UseTemplate` DataWindow property, PowerBuilder attempts to import the data using a default mechanism as described below.

**Elements that contain text**

The text between the start and end tags for each element can be imported if the XML document data corresponds to the DataWindow column definition. For example, this would be the case if the XML was exported from PowerBuilder using the default XML export template.

The text content of the XML elements must match the column order, column type, and validation requirements of the DataWindow columns, as is the case when you import data from a text file with the `ImportFile` method.

All element text contents are imported in order of occurrence. Any possible nesting is disregarded. The import process ignores tag names of the elements, attributes, and any other content of the XML document.

**Empty elements**

Empty elements (elements that have no content between the start and end tags) are imported as empty values into the DataWindow column. If the element text contains only white space, carriage returns, and new line or tab characters, the element is treated as an empty element.

Any attributes of empty elements are ignored.

**Elements with non-text content**

If the element has no text content, but does contain comments, processing instructions, or any other content, it is not regarded as an empty element and is not imported.
Example with no empty elements

The three XML documents that follow all show the same result when ImportFile is called with or without default arguments for start and end column, start and end row, and DataWindow start column. The DataWindow object has five columns: emp_id, emp_fname, emp_lname, phone, and birth_date.

Example 1

```xml
<?xml version="1.0"?>
<d_emp_birth_listing>
  <d_emp_birth_row>
    <element_1>105</element_1>
    <element_2>Matthew</element_2>
    <element_3>Cobb</element_3>
    <element_4>6175553840</element_4>
    <element_5>04/12/1960</element_5>
  </d_emp_birth_row>
  <d_emp_birth_row>
    <element_1>148</element_1>
    <element_2>Julie</element_2>
    <element_3>Jordan</element_3>
    <element_4>6175557835</element_4>
    <element_5>11/12/1951</element_5>
  </d_emp_birth_row>
</d_emp_birth_listing>
```

Example 2

```xml
<?xml version="1.0"?>
<root_element>
  <element_1>105</element_1>
  <element_2>Matthew</element_2>
  <element_3>Cobb</element_3>
  <element_4>6175553840</element_4>
  <element_5>04/12/1960</element_5>
  <element_6>148</element_6>
</root_element>
```
<element_7>Julie</element_7>  
<element_8>Jordan</element_8>  
<element_9>6175557835</element_9>  
<element_10>11/12/1951</element_10>  
</root_element>

Example 3

```xml
<?xml version="1.0"?>
<root_element>
<!-- some comment -->
<row_element>
<!-- another comment -->
</row_element>
<row_element>
<name Title="Mr">
  <first>Matthew</first>
  <last>Cobb</last>
</name>
</row_element>
<row_element>
<name Title="Ms">
  <first>Julie</first>
  <last>Jordan</last>
</name>
</row_element>
</root_element>
```

Result

All three XML documents produce this result:

<table>
<thead>
<tr>
<th>emp_id</th>
<th>emp_f_name</th>
<th>emp_l_name</th>
<th>phone</th>
<th>birth_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>105</td>
<td>Matthew</td>
<td>Cobb</td>
<td>6175553840</td>
<td>04/12/1960</td>
</tr>
<tr>
<td>148</td>
<td>Julie</td>
<td>Jordan</td>
<td>6175557835</td>
<td>11/12/1951</td>
</tr>
</tbody>
</table>

Example with empty elements

This example uses the same DataWindow object, but there are empty elements in the XML document, as well as an element containing a comment. The element containing a comment is ignored.

Example 4

```xml
<?xml version="1.0"?>
<root_element>
<!-- some comment -->
</row_element>
```
PowerScript enhancements for XML support

The XML document produces this result:

```xml
<?process me="no"?>
105
<not_empty><!-- comment --></not_empty>

<name Title="Mr">
  <first>Matthew</first>
  <!-- another comment -->
  <last>Cobb</last>
</name>

<birthdate>04/01/1962</birthdate>
</row_element>

<row_element>
  148
  <name Title="Ms">
    <empty attribute1 = "blue"></empty>
    <last>Jordan</last>
  </name>
  <phone>6175557835</phone>
  <birthdate>11/12/1951</birthdate>
</row_element>
</root_element>
```

Result

<table>
<thead>
<tr>
<th>emp_id</th>
<th>emp_fname</th>
<th>emp_lname</th>
<th>phone</th>
<th>birth_date</th>
</tr>
</thead>
<tbody>
<tr>
<td>105</td>
<td>Matthew</td>
<td>Cobb</td>
<td></td>
<td>04/12/1960</td>
</tr>
<tr>
<td>148</td>
<td>Jordan</td>
<td></td>
<td>6175557835</td>
<td>11/12/1951</td>
</tr>
</tbody>
</table>

PowerScript enhancements for XML support

To support the ability to export DataWindow objects to XML, XSL-FO, and PDF, and import them from XML, the following PowerScript enhancements have been made in PowerBuilder 9:

- “New SaveAsType values for SaveAs method” on page 43
- “XML support in import methods” on page 44
- New DataWindow object properties:
  - Data.XML
  - Data.XMLDTD
  - Data.XMLSchema
  - Data.XSLFO
DataWindow support for XML

Export.XML.HeadGroups
Export.XML.MetaDataType
Export.XML.SaveMetaData
Export.XML.TemplateCount
Export.XML.Template[ ],Name
Export.XML.UseTemplate
Import.XML.UseTemplate

New SaveAsType values for SaveAs method

The SaveAs method for DataWindows now supports XML!, XSLFO!, and PDF! as SaveAsType values. The enumerated values have the string and numeric values shown in the following table:

<table>
<thead>
<tr>
<th>Enumerated value</th>
<th>String value</th>
<th>Numeric value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML!</td>
<td>XML</td>
<td>15</td>
<td>Extensible Markup Language (XML)</td>
</tr>
<tr>
<td>XSLFO!</td>
<td>XSLFO</td>
<td>16</td>
<td>XSL (Extensible Stylesheet Language) Formatting Objects (XSL-FO)</td>
</tr>
<tr>
<td>PDF!</td>
<td>PDF</td>
<td>17</td>
<td>Portable Document Format (PDF)</td>
</tr>
</tbody>
</table>

Syntax

dwcontrol.SaveAs(DocumentName, XML!, FALSE)
dwcontrol.SaveAs(DocumentName, XSLFO!, FALSE)
dwcontrol.SaveAs(DocumentName, PDF!, FALSE)

where dwcontrol is the name of a DataWindow control and DocumentName is the name of the file in which the document is saved.

The third parameter, which is a boolean value indicating whether the DataWindow object's column headings should be included at the beginning of the exported file, is not relevant in the context of these output formats.

Usage

For XML!, the XML logical structure used is based on the current XML export template for the DataWindow object. You can change the export template by setting the value of the Export.XML.UseTemplate property. If no export template is specified, the default template is used.

For XSLFO! and PDF!, the export template associated with the DataWindow object has no effect on the generated file.
PowerScript enhancements for XML support

Applies to graph controls
The SaveAs function for graph controls can also use XML! as an export format. However, like graph, composite, and OLE DataWindow objects, graph controls can only be exported using the default export template.

Examples
The following statements set the template used by the DataWindow dw_1 to t_report, specify that metadata in the XMLSchema! format should be generated in a separate file, and generate the files c:\myxml.xml containing the DataWindow row data in XML format, and c:\myxml.xsd containing the XML schema used in c:\myxml.xml:

```
  dw_1.Modify("DataWindow.Export.XML.UseTemplate = 't_report'")
  dw_1.Modify("DataWindow.Export.XML.MetaDataType = XMLSchema!")
  dw_1.Modify("DataWindow.Export.XML.SaveMetaData = MetaDataExternal!")
  dw_1.SaveAs("c:\myxml.xml", XML!, FALSE)
```

The following statements generate the files c:\dw_one.fo containing the DataWindow presentation and data in XSL-FO format, and c:\dw_one.pdf containing the DataWindow presentation and data in PDF format:

```
  dw_1.SaveAs("c:\dw_one.fo", XSLFO!, FALSE)
  dw_1.SaveAs("c:\dw_one.pdf", PDF!, FALSE)
```

See also
Export.XML.MetaDataType
Export.XML.SaveMetaData
Export.XML.UseTemplate

XML support in import methods
The ImportFile, ImportString, and ImportClipboard methods for DataWindows support XML as an import format. The ability to import XML has also been added to the ImportClipboard, ImportFile, and ImportString methods for the DataWindow Web Control for ActiveX, and the ImportStringEx method for the Web DataWindow PSWebDataWindowClass and server component.

Applies to graph controls
The changes described in this section also apply to Graph objects. However, like graph, composite, and OLE DataWindow objects, they can only be populated using the default import format.
These methods now have an optional first parameter that enables you to specify the type of data to be imported. The syntax shown here is for the PowerBuilder DataWindow:

```c
long control.ImportFile ( { saveastype importtype, } string filename {, long startrow {, long endrow {, long startcolumn {, long endcolumn {, long dwstartcolumn } } } } } )

long control.ImportString ( { saveastype importtype, } string string {, long startrow {, long endrow {, long startcolumn {, long endcolumn {, long dwstartcolumn } } } } } )

long control.ImportClipboard ( { saveastype importtype, } { long startrow {, long endrow {, long startcolumn {, long endcolumn {, long dwstartcolumn } } } } } )
```

For `ImportFile`, the valid import types are Text!, CSV!, XML!, DBase2!, and DBase3!. For `ImportString` and `ImportClipboard`, the valid import types are Text!, CSV!, and XML!.

If you specify the optional parameter, you do not need to include the extension in the import file name. For example:

```c
dw_1.ImportFile( XML!, "c:\data\customers" )
```

If you do not specify this parameter, PowerBuilder recognizes the extensions .TXT, .CSV, .XML, and, for `ImportFile`, .DBF and handles them correctly.

When you use the default mechanism to import data from an XML document, the number of DataWindow columns that take input determines the number of elements in the XML document that are assumed to make up one row. For example, if three columns in the DataWindow object can take input, each set of three consecutive elements in the XML document is imported as one row.

The following table shows how the optional parameters that determine start and end columns and rows are handled for XML data. For a complete description of the methods, see the online Help for each method.

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>startrow</code></td>
<td>The number of the first detail row in the file that you want to copy. The default is 1. For default XML import, if <code>startrow</code> is supplied, the first $N$ (<code>startrow</code> - 1) elements are skipped, where $N$ is the DataWindow row size. For template XML import, if <code>startrow</code> is supplied, the first (<code>startrow</code> - 1) occurrences of the repetitive row mapping defined in the template are skipped.</td>
</tr>
</tbody>
</table>
Data.XML

Description
A string containing the row data content of the DataWindow object in XML format.

Applies to
DataWindows

Syntax
PowerBuilder dot notation:

\[
\text{dw_control}.\text{Object}.\text{DataWindow}.\text{Data.XML}
\]

Describe argument:

"DataWindow.Data.XML"

Usage
If any of the Export.XML properties have been set, the string that is generated reflects the values of these properties.

Note
If Export.XML.SaveMetaData is set to MetaDataExternal!, no metadata is generated in the string.
The following statements set the template used by the DataWindow dw_1 to t_report, specify that metadata in the XMLSchema! format should be included in the generated XML, and return the generated XML document to the string ls_xml.

```plaintext
string ls_xml
  dw_1.Modify("DataWindow.Export.XML.UseTemplate = 't_report'")
  dw_1.Modify("DataWindow.Export.XML.SaveMetaData = MetaDataInternal!")
  dw_1.Modify
      ("DataWindow.Export.XML.MetaDataType = XMLSchema!")
ls_xml = dw_1.Object.DataWindow.Data.XML
```

**Data.XMLDTD**

**Description**
A string containing the full document type definition (DTD) of the XML output for a DataWindow object.

**Applies to**
DataWindows

**Syntax**
PowerBuilder dot notation:

```plaintext
dw_control.Object.DataWindow.Data.XMLDTD
```

**Describe argument:**

"DataWindow.Data.XMLDTD"

**Usage**
Use this property to return the full DTD of the XML output of a DataWindow object separately from the generated XML document itself. The export template used affects the generated DTD.

**Examples**
The following statements set the template used by the DataWindow dw_1 to t_report and return the generated DTD to the string ls_xml_dtd.

```plaintext
string ls_xml_dtd
  dw_1.Object.DataWindow.Export.XML.UseTemplate = 't_report'
  ls_xml_dtd = dw_1.Object.DataWindow.Data.XMLDTD
```

**Data.XMLSchema**

**Description**
A string containing the full schema of the XML output of a DataWindow object.

**Applies to**
DataWindows
**PowerScript enhancements for XML support**

**Syntax**

PowerBuilder dot notation:

```
dw_control.Object.DataWindow.Data.XMLSchema
```

Describe argument:

"DataWindow.Data.XMLSchema"

**Usage**

Use this property to return the full schema of the XML output of a DataWindow object separately from the generated XML document itself. The export template used affects the generated schema.

**Examples**

The following statements set the template used by the DataWindow `dw_1` to `t_report` and return the XML schema to the string `ls_xml_schema`.

```powerbuild
string ls_xml_schema

dw_1.Object.DataWindow.Export.XML.UseTemplate = 't_report'

ls_xml_schema = dw_1.Object.DataWindow.Data.XMLSchema
```

**Data.XSLFO**

**Description**

A string containing XSL Formatting Objects (XSL-FO) that represents the data and presentation of the DataWindow object.

**Applies to**

DataWindows

**Syntax**

PowerBuilder dot notation:

```
dw_control.Object.DataWindow.Data.XSLFO
```

Describe argument:

"DataWindow.Data.XSLFO"

**Usage**

Use this property to return the data and presentation of a DataWindow object in XSL-FO format. The export template associated with the DataWindow object does not affect the generated string.

**Examples**

The following statements return the data and presentation of the DataWindow object `dw_1` to the string `ls_xslfo` in XSL-FO format.

```powerbuild
string ls_xslfo

ls_xslfo = dw_1.Object.DataWindow.Data.XSLFO
```
Export.XML.HeadGroups

Description
Setting that causes elements, attributes, and all other items above the Detail Start element in an XML export template for a group DataWindow to be iterated for each group in the exported XML.

Applies to
DataWindow objects

Syntax
PowerBuilder dot notation:

dw_control.Object.DataWindow.Export.XML.HeadGroups

Describe and Modify argument:

"DataWindow.Export.XML.HeadGroups { = 'value' }"

Parameter | Description
---|---
value | *(exp)* Whether the contents of the header section in an export template iterate in the generated XML. Values are:
- Yes—The header section is repeated for each group (default)
- No—The header section is not repeated

Usage
This property must be set for group DataWindow objects if you want elements and other items added to the header section of an XML export template to be repeated before each group in the exported XML.

Examples

dw_1.Object.DataWindow.Export.XML.HeadGroups = "No"
dw_1.Modify("Export.XML.HeadGroups = 'Yes' ")

Export.XML.MetaDataType

Description
Setting that controls the type of metadata generated with the XML exported from a DataWindow object using the SaveAs method or a .Data.XML expression.

Applies to
DataWindow objects

Syntax
PowerBuilder dot notation:


dw_control.Object.DataWindow.Export.XML.MetaDataType

Describe and Modify argument:

"DataWindow.Export.XML.MetaDataType { = 'value' }"

Parameter | Description
---|---
value | *(exp)* A string specifying a value of the Export.XML.MetaDataType enumerated data type
Usage

This property must be set to specify the type of metadata generated before you call the SaveAs method with XML! as the SaveAsType to save data as an XML document, or use the .Data.XML expression to save data as an XML string. The metadata is saved into the exported XML itself or into an associated file, depending on the value of the Export.XML.SaveMetaData property.

The Export.XML.MetaDataType property is an enumerated datatype that can hold the following values:

<table>
<thead>
<tr>
<th>PowerBuilder enumerated value</th>
<th>Numeric value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>XMLNone!</td>
<td>0</td>
<td>Metadata (XML Schema or DTD) is not generated when XML is exported</td>
</tr>
<tr>
<td>XMLSchema!</td>
<td>1</td>
<td>XML Schema is generated when XML is exported</td>
</tr>
<tr>
<td>XMLDTD!</td>
<td>2</td>
<td>DTD is generated when XML is exported</td>
</tr>
</tbody>
</table>

Examples

This statement specifies that no metadata will be generated when the DataWindow is exported to XML:

```
   dw_1.object.DataWindow.Export.XML.MetaDataType = XMLNone!
```

These statements export the contents of dw_1 to the file c:\myxml.xml using the XML export template called t_schema, and generate an external XML schema file at c:\myxml.xsd:

```
   dw_1.Modify("DataWindow.Export.XML.UseTemplate = 't_schema'");
   dw_1.Modify("DataWindow.Export.XML.MetaDataType = 1");
   dw_1.Modify("DataWindow.Export.XML.SaveMetaData = 1");
   dw_1.SaveAs("c:\myxml.xml", XML!, FALSE)
```

See also

Export.XML.SaveMetaData

Export.XML.SaveMetaData

Description

Setting that controls the storage format for the metadata generated with the XML exported from a DataWindow object using the SaveAs method or a .Data.XML expression.

Applies to

DataWindow objects

Syntax

PowerBuilder dot notation:

```
   dw_control.Object.DataWindow.Export.XML.SaveMetaData
```
Describe and Modify argument:

"DataWindow.Export.XML.SaveMetaData { = 'value' }"

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>(exp) A string specifying a value of the</td>
</tr>
<tr>
<td></td>
<td>Export.XML.SaveMetaData enumerated data type</td>
</tr>
</tbody>
</table>

**Usage**

This property must be set to specify how to store the generated metadata before you call the `SaveAs` method with XML! as the `SaveAsType` to save data as an XML document, or use the `.Data.XML` expression to save data as an XML string. The metadata can be saved into the exported XML document or string or into an associated file.

**Note** If `Export.XML.MetaDataType` is set to XMLNone!, the value of the Export.XML.SaveMetaData property is not used.

The `Export.XML.SaveMetaData` property is an enumerated datatype that can hold the following values:

<table>
<thead>
<tr>
<th>PowerBuilder enumerated value</th>
<th>Numeric value</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>MetaDataInternal!</td>
<td>0</td>
<td>The metadata is saved into the generated XML document or string. To save metadata using the .Data.XML expression syntax, you must use this value.</td>
</tr>
<tr>
<td>MetaDataExternal!</td>
<td>1</td>
<td>With the SaveAs method, metadata is saved as an external file with the same name as the XML document but with the extension .xsd (for XMLSchema! type) or .dtd (for XMLDTD! type). A reference to the name of the metadata file is included in the output XML document. With .Data.XML, no metadata is generated in the XML string.</td>
</tr>
</tbody>
</table>

**Examples**

```
dw_1.Object.DataWindow.Export.XML.SaveMetaData = 0
```

```
dw_1.Modify("Export.XML.SaveMetaData = MetaDataExternal!")
```

**See also** Export.XML.MetaDataType
**PowerScript enhancements for XML support**

---

### Export.XML.TemplateCount

**Description**
The number of XML export templates associated with a DataWindow object.

**Applies to**
DataWindow objects

**Syntax**
PowerBuilder dot notation:

```
dw_control.Object.DataWindow.Export.XML.TemplateCount
```

Describe argument:

```
"DataWindow.Export.XML.TemplateCount { = 'number' }"
```

**Usage**
This property is used to get a count of the XML export templates associated with a DataWindow object. The count is used with the DataWindow.Export.XML.Template[].Name property to enable an application to select an export template at runtime.

**Examples**
This code in the open event of a window uses the TemplateCount property to get the number of templates associated with `dw_1`. It then uses the number returned as the upper limit in a FOR loop that populates a drop-down list box with the template names, using the DataWindow.Export.XML.Template[].Name property.

```powerscript
string ls_template_count, ls_template_name
long i

ls_template_count=dw_1.Describe
("DataWindow.Export.XML.TemplateCount")

for i=1 to Long(ls_template_count)
    ls_template_name=
        dw_1.Object.DataWindow.Export.XML.Template[i].Name
    ddlb_1.AddItem(ls_template_name)
next
```

Before generating the XML, set the export template using the text in the drop-down list box:

```
dw_1.Object.DataWindow.Export.XML.UseTemplate=
ddlb_1.text
```

**See also**
Export.XML.Template[].Name
Export.XML.UseTemplate

---

52 Sybase Preliminary and Confidential Beta 3 Draft
Export.XML.Template[ ].Name

Description: The name of an XML export template associated with a DataWindow object.

Applies to: DataWindow objects

Syntax: PowerBuilder dot notation:

\[ dw\_control.\text{Object}.\text{DataWindow}.\text{Export.XML.Template}[^{\text{num}}].\text{Name} \]

Describe argument:

"DataWindow.Export.XML.Template[^{\text{num}}].Name{ = 'value'}"

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>num</td>
<td>(exp) A long specifying the index of the export template</td>
</tr>
<tr>
<td>value</td>
<td>(exp) A string specifying the name of an export template previously saved in the DataWindow painter for the specified DataWindow object</td>
</tr>
</tbody>
</table>

Usage: This property is used to get the names of the XML export templates associated with a DataWindow object. The property is used with the DataWindow.Export.XML.TemplateCount property to enable an application to select an export template at runtime.

Examples: See Export.XML.TemplateCount.

See also: Export.XML.TemplateCount

Export.XML.UseTemplate

Description: Setting that optionally controls the logical structure of the XML exported from a DataWindow object using the SaveAs method or the .Data.XML property.

Applies to: DataWindow objects

Syntax: PowerBuilder dot notation:

\[ dw\_control.\text{Object}.\text{DataWindow}.\text{Export.XML.UseTemplate} \]

Describe and Modify argument:

"DataWindow.Export.XML.UseTemplate{ = 'value'}"

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>(exp) A string specifying the name of an export template previously saved in the DataWindow painter for the specified DataWindow object</td>
</tr>
</tbody>
</table>
**Usage**

This property should be set to specify the logical structure of the XML generated before you call the **SaveAs** method with **XML** as the **SaveAsType** to save data as an XML document, or use the **.Data.XML** expression to save data as an XML string.

**Examples**

This example stores the name of the export template used in **dw_1** in the string **ls_template**. If no template is selected in **dw_1**, an empty string is returned.

```powerbuilder
string ls_template_name
ls_template_name =
dw_1.Describe("DataWindow.Export.XML.UseTemplate")
```

This example sets the name of the current XML export template used in **dw_1** to **t_report**. If **t_report** does not exist, the current template is not changed.

```powerbuilder
dw_1.Modify("DataWindow.Export.XML.UseTemplate = 't_report' ")
```

See also

- Export.XML.MetaDataType
- Export.XML.SaveMetaData

---

**Import.XML.UseTemplate**

**Description**

Setting that optionally controls the logical structure of the XML imported from an XML file to a DataWindow object using the **ImportFile** method.

**Applies to**

DataWindow objects

**Syntax**

PowerBuilder dot notation:

```powerbuilder
dw_control.Object.DataWindow.Import.XML.UseTemplate
```

Describe and Modify argument:

```powerbuilder
"DataWindow.Import.XML.UseTemplate { = 'value' }"
```

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>value</td>
<td>(exp) A string specifying the name of an import template previously saved in the DataWindow painter for the specified DataWindow object</td>
</tr>
</tbody>
</table>

**Usage**

This property should be set to specify the logical structure of the XML imported before you call the **ImportFile** method to import data from an XML document. An import template is not required if the XML document from which data is imported corresponds to the DataWindow column definition.
If an export template for a DataWindow object exists, it can be used as an import template. Only the mapping of column names to element attribute names is used for import. The order of elements within the template is not significant, because import values are located by name match and nesting depth within the XML document. All other information in the template, such as controls and comments, is ignored.

**Examples**

This example sets the name of the current XML import template used in `dw_1` to `t_import_report`. If `t_import_report` does not exist, the current template is not changed.

```plaintext
dw_1.Modify("DataWindow.Export.XML.UseTemplate = 't_import_report' ")
```

**See also**

Export.XML.UseTemplate
PowerScript enhancements for XML support
JavaServer Pages targets

JSP authoring in PowerBuilder 9 will offer the same robust and rapid development that customers have come to expect from PowerBuilder. In particular, JSP authoring will automate as many of the development tasks as possible and offer users an easy-to-use development environment for JSP authoring.

In the beta 3 release of PowerBuilder 9, you can preview the following features of JSP authoring:

- **JSP Web Target wizard**
  
  This wizard creates a JSP target that has build and source folders and a deployment configuration.

- **JSP page authoring**
  
  You can add JSP pages to a JSP target and edit them. Menu options and dialog boxes are available for adding actions, directives, and scripting elements. Adding applets and JavaBeans to a JSP page inserts the appropriate JSP action.

- **JSP Web target object model**
  
  Several class and method changes were required for implementation of the Web Target object model in JSP targets. These include a 4GL server-side event model that provides foundation classes to handle many of the details of coding server scripts for Web pages, including server control classes, the DataWindow class, server variables, and parameter classes.

- **JSP deployment configuration**
  
  You can create a deployment configuration for deploying a JSP Web Application to either EAServer or the Apache Tomcat Server. Deployment works to either EAServer or Tomcat.

- **Custom tag library for the Web DataWindow**
  
  A custom tag library for the Web DataWindow is installed with this release.
About JavaServer Pages

JavaServer Pages (JSP) technology provides a quick, easy way to create Web pages with both static and dynamic content. JSPs are text-based documents that contain static markup, usually in HTML or XML, as well as Java content in the form of scripts and/or calls to Java components. JSPs extend the Java Servlet API and have access to all Java APIs and components.

You can use JSPs in many ways in Web-based applications. As part of the J2EE application model, JSPs typically run on a Web server in the middle tier, responding to HTTP requests from clients, and invoking the business methods of Enterprise JavaBeans (EJB) components on a transaction server.

JSP pages built with PowerBuilder 9 support version 1.2 of the JavaServer Pages specification, version 2.3 of the Java Servlet specification, JDK 1.3, and J2EE 1.3. PowerBuilder 9 also supports custom tag libraries that use the JSP 1.1 format.

In this release, you can choose to deploy a JSP target as a Web application to EAServer 4.1 and later or Apache Tomcat 4.0 and later.


How JavaServer Pages work

JSP pages are executed in a JSP engine (also called a JSP container) that is installed on a Web or application server. The JSP engine receives a request from a client and delivers it to the JSP page. The JSP page can create or use other objects to create a response. For example, it can forward the request to a servlet or an EJB component, which processes the request and returns a response to the JSP page. The response is formatted according to the template in the JSP page and returned to the client.

In PowerBuilder, JSP pages are deployed to the server in source form. If a JSP page is in source form, the JSP engine typically translates the page into a class that implements the servlet interface and stores it in the server’s memory. Depending on the implementation of the JSP engine, translation can occur at any time between initial deployment and the receipt of the first request. As long as the JSP page remains unchanged, subsequent requests reuse the servlet class, reducing the time required for those requests.
Requests and responses

Some JSP engines can handle requests and responses that use several different protocols, but all JSP engines can handle HTTP requests and responses. The JspPage and HttpServletRequest classes in the javax.servlet.jsp package define the interface for the compiled JSP, which has three methods:

- jspInit()
- jspDestroy()
- _jspService(HttpServletRequest request, HttpServletResponse response)

What a JSP contains

A JSP contains static template text that is written to the output stream. It also contains dynamic content that can take several forms:

- Directives provide global information for the page, or include a file of text or code.
- Scripting elements (declarations, scriptlets, and expressions) manipulate objects and perform computations.
- Standard tags, or actions, perform common actions such as instantiating a JavaBeans component or getting or setting its properties, downloading a plug-in, or forwarding a request.
- Custom tags perform additional actions defined in a custom tag library.

“JSP page authoring” on page 61 provides a brief description of each of these types of dynamic content. For more detailed information, see the JavaServer Pages specification, available at http://java.sun.com/products/jsp/index.html, or one of the many books about JavaServer Pages technology.

Application logic in JSPs

The application logic in JSPs can be provided by components such as servlets, JavaBeans, and EJBs, customized tag libraries, scriptlets, and expressions. Scriptlets and expressions hold the components and tags together in the page.

JavaBeans

You can easily use JavaBeans components in a JSP with the useBean tag. For more information, see “<jsp:useBean>” on page 62.

Enterprise JavaBeans

To use an EJB component, you need to use JNDI to establish an initial naming context for the EJB’s home interface. You could do this in a scriptlet, using a JavaBeans component, or using a custom tag.
Custom tag libraries

Custom tag libraries define a set of actions to be used within a JSP for a specific purpose, such as handling SQL requests. See “Custom tags” on page 70.

JSP Web Target wizard

Like the PowerDynamo Web Site wizard, the JSP Web Target wizard creates a target with Source and Build folders and a deployment configuration. JSP pages are deployed as a Web application in a Web Archive (WAR) file.

Many of the pages in the JSP Web Page wizard are similar to those in the PowerDynamo Web Site wizard. You can also specify the names and locations of any tag libraries the Web application uses, the type of server you want to deploy to, and the name and location of the WAR file that is used to deploy the Web application. The properties you specify in the wizard can be modified after target creation in the Deployment Configuration Properties dialog box for the JSP target.

Adding tag libraries

The tag library page of the wizard is not available in beta 3. To add a tag library to a target, use the Deployment Configuration Properties dialog box or drag a tag library from the Components tab of the System Tree to a target page in the HTML editor.

You will need to specify a prefix to identify the custom tag, and the location where the TLD file can be found relative to the root of the Web application. The folder that contains the deployed Web application has a `WEB-INF` subdirectory, and TLD files are typically deployed in a subdirectory of the `WEB-INF` directory called `tlds`.

By default, the taglib.tld custom tag library is included in your target. For more information about tag libraries, see “Custom tags” on page 70.

Specifying a server type

When you select a server type, the wizard presents a page where you specify how to connect to the server. For EAServer deployment, you use an EAServer profile. For Tomcat deployment, you specify a deployment folder, an HTTP server and port name, as well as a login name and password.
You can also select an option to have Tomcat Manager shut down and restart the Tomcat server after you deploy the JSP target. Otherwise, you will need to restart the Tomcat server manually. For this beta 3 release, Tomcat must be installed on the local machine.

JSP page authoring

JSP pages can be written in any well-formed language, including XML, but they are usually written in HTML. In PowerBuilder, you create JSP pages using any of the page wizards on the Web page of the New dialog box, and you edit them in much the same way as any other HTML page. When you create a new page, give it the extension .jsp instead of .htm.

Three new types of elements are available in the development environment:

- JSP actions
- Directives
- Scripting elements

You can also add the following to JSP pages:

- Custom tags
- Custom tags for Web services
- Comments
In the Page view, JSP standard actions and scripting elements are represented by icons showing the element’s delimiters. When you select a scripting element or a 4GL server-side event, Java is the only language available in the script editor.

### JSP actions

Actions are standard tags that perform common actions. All JSP standard actions use the prefix `jsp`.

You can insert any of the following actions:

- `<jsp:useBean>`
  - The `useBean`, `getProperty`, and `setProperty` actions are all used with JavaBeans components. The `useBean` id attribute is the name of the bean and corresponds to the name attribute for `getProperty` and `setProperty`. The `useBean` action locates or instantiates a JavaBeans component:

    ```
    <jsp:useBean id="labelLink" scope="session"
        class="LinkBean.labelLink" />
    ```

    The bean class and classes required by the bean class must be deployed under a JavaCode base that is available to the Web Application where the JSP is installed.

- `<jsp:getProperty>`
  - The `getProperty` action gets the value of a JavaBeans component property so that you can display it in a result page:

    ```
    <jsp:getProperty name="labelLink" property="url" />
    ```

- `<jsp:setProperty>`
  - The `setProperty` action sets a property value or values in a JavaBeans component:
JavaServer Pages targets

```html
<jsp:setProperty name="labelLink" property="url" value="<%= labelLink.getURL() %>"/>

<jsp:include>
The include action includes a static file or sends a request to a dynamic file:
```html
<jsp:include page="cart.html" flush="true" />
```

<jsp:forward>
The forward action forwards a client request to an HTML file, JSP file, or servlet for processing:
```html
<jsp:forward page="/jsp/datafiles/ListSort.jsp" />
```

<jsp:param>
The param action specifies request parameters in the body of an include or forward action. It can also be used in the body of a params action.
```html
<jsp:param name="bgColor" value="blue" />
```
```
</jsp:forward>
```

<jsp:params>
The params action can be used only in the body of a plugin action to enclose the applet parameters specified by param actions.
```
</jsp:params>
```

<jsp:plugin>
The plugin action downloads plug-in software to the Web browser to execute an applet or JavaBeans component. It generates HTML `<embed>` or `<object>` elements in the page. You can use the params and param actions to specify parameters required by the plug-in, and the fallback action to specify the text that displays if the browser does not support `<embed>` or `<object>` elements:
```
<jsp:plugin type="applet" code="Calc.class" codebase="/mathutils" >
  <jsp:params>
    <jsp:param name="multiplier" value="multipliers/tax.val"/>
  </jsp:params>
  <jsp:fallback>
    <p>unable to start plugin</p>
  </jsp:fallback>
</jsp:plugin>
```

<jsp:fallback>
The fallback action can be used only in the body of a plugin action to specify the text that displays if the browser does not support `<embed>` or `<object>` elements.
Inserting an action

❖ To insert an action in a JSP page:

1. Select Insert>JSP Standard Action from the menu bar and select an action:

2. In the dialog box that displays, specify the values of the action’s attributes. A √ in the Required column indicates that you must specify a value for the attribute:

For a description of each of the values available for the scope attribute of the <jsp:useBean> action, see “Scopes” on page 70.

Adding applets and JavaBeans

Adding applets and JavaBeans to a JSP page inserts the appropriate JSP action. To view JavaBeans and applets on the Components tab of the System Tree, you must make sure that the component that you want and the WTInfo90.jar file are included in the Java class path. The WTInfo90.jar is installed in the Sybase\Shared\Web Targets directory. It should be included in the class path by default.
Adding applets

When you drag an applet from the Components tab to a JSP page in Page view or Source view, the jsp:plugin Properties dialog box displays with default values for the applet you selected. When you click OK, the applet is added to the page in a jsp:plugin action tag.

When you add an applet to a JSP page, you must make sure the applet classes are stored in a location accessible to client browsers. You can assign this location, using a file or http protocol, to the codebase attribute of the jsp:plugin directive.

Adding JavaBeans and JavaBean properties

When you drag a JavaBean from the Components tab to a JSP page in Page view or Source view, the jsp:useBean Properties dialog box displays with default values for the JavaBean you selected. When you click OK, the JavaBean is added to the page in a jsp:useBean action tag. If the JavaBean is in a class file, the class file is added to the Web-Inf\classes directory for your target. If the JavaBean is in an archive file, the archive file is added to the Web-Inf\lib directory for your target.

When you drag a read-only JavaBean property from the Components tab to a JSP page, the jsp:getProperty Properties dialog box displays with default values for the JavaBean property you selected. When you drag a write-only JavaBean property from the Components tab to a JSP page, the jsp:setProperty Properties dialog box displays with default values for the JavaBean property you selected. If a JavaBean property has both read and write permissions, an intermediary dialog box displays, prompting you to add either a jsp:getProperty action or a jsp:setProperty action.

Read-Write JavaBean properties in beta 3

In beta 3, properties with both read and write permissions are listed twice on the Components tab: one time for the read property and another time for the write property. Dragging the read property to a JSP page has the same effect as dragging a read-only property to the page: the jsp:getProperty Properties dialog box displays. Dragging the write property to a JSP page has the same effect as dragging a write-only property to the page: the jsp:setProperty Properties dialog box displays.

Directives

Directives are messages to the JSP engine that provide global information for the page or include a file of text or code. Directives begin with the character sequence <%@ followed by the name of the directive and one or more attribute definitions. They end with the character sequence %>.
There are three directives: page, include, and taglib.

**Page directive**
The page directive defines attributes that apply to an entire JSP page, including language, the class being extended, packages imported for the entire page, the size of the buffer, and the name of an error page. For example:

```
<%@ page language="java" import="mypkg.*"  
   session="true" errorPage="ErrorPage.jsp" %>
```

For more information about error pages, see “Error handling” on page 73.

**Include directive**
The include directive includes a static file, parsing the file’s JSP elements:

```
<%@ include file="header.htm" %>
```

**Include directive and include standard tag** Note that the include directive parses the file’s contents, whereas the include tag does not.

**Taglib directive**
The taglib directive defines the name of a tag library and its prefix for any custom tags used in a JSP page:

```
<%@ taglib uri="http://www.mycorp/printtags"  
    prefix="print" %>
```

If the tag library with the prefix print includes an element called doPrintPreview, this is the syntax for using that element later in the page:

```
<print:doPrintPreview>
  ...
</print>
```

For more information, see “Custom tags” on page 70.

**Inserting a directive**

❖ **To insert a directive in a JSP page:**

1. Right-click inside a page in Page view and select Page Properties from the pop-up menu.
   or
   Right-click inside the `<BODY ...>` tag in Source view and select Properties from the pop-up menu.

2. In the Page Properties or Body Properties dialog box, select the JSP Directives tab and click the New icon.
3. Select the type of directive you want to add in the drop-down list box in the Name column.

![Image of drop-down list box and dialog box]

4. Click inside the Value column, then click the Browse (...) button that displays at the right of the Value column. Complete the dialog box that displays.

   The type of dialog box that displays depends on the type of directive you are adding. The Page Directive Attributes dialog box looks like this:

   ![Image of Page Directive Attributes dialog box]

**Scripting elements**

Scripting elements manipulate objects and perform computations. The character sequence that precedes a scripting element depends on the element’s type: `<%` for a scriptlet, `<%=` for an expression, and `<%!` for a declaration. Scriptlets, expressions, and declarations are all closed with the sequence `%>`.
JSP page authoring

Scriptlets
A scriptlet contains a code fragment valid in the page scripting language (usually Java, but other languages can be defined in the page directive):

```jsp
<% cart.processRequest(request); %>
```

Expressions
An expression contains an expression valid in the page scripting language:

```jsp
Value="<%= request.getParameter("amount") %>"
```

Declarations
A declaration declares variables or methods valid in the page scripting language:

```jsp
<%! Connection myconnection; String mystring; %>
```

Inserting a scripting element

❖ To insert a scripting element in a JSP page:

1. Open a JSP page, select the Page tab, and right-click in the Script editor.
2. From the pop-up menu, select New Script>Server>JSP and then the delimiters for the type of scripting element you want.
3. Type the script, expression, or declaration in the Script editor.

Implicit objects
When a JSP page processes a request, it has access to a set of implicit objects, each of which is associated with a given scope. Other objects can be created in scripts. These created objects have a scope attribute that defines where the reference to that object is created and removed.
Implicit objects display on the Language tab page in the System Tree.

References to the object are stored in the pageContext, request, session, or application object, according to the object’s scope.

Most of the following implicit objects are always available within scriptlets and expressions. If the JSP is an error page (the page directive’s isErrorPage attribute is set to true), the exception implicit object is also available.

<table>
<thead>
<tr>
<th>Implicit object</th>
<th>Description</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>request</td>
<td>The request triggering the servlet invocation.</td>
<td>Request</td>
</tr>
<tr>
<td>response</td>
<td>The response to the request that triggered the servlet invocation.</td>
<td>Page</td>
</tr>
<tr>
<td>pageContext</td>
<td>The page context for this JSP.</td>
<td>Page</td>
</tr>
<tr>
<td>session</td>
<td>The session object created for the requesting client (if any).</td>
<td>Session</td>
</tr>
<tr>
<td>application</td>
<td>The servlet context obtained from the servlet configuration, as in the call getServletConfig().getContext().</td>
<td>Application</td>
</tr>
<tr>
<td>out</td>
<td>An object that writes to the output stream.</td>
<td>Page</td>
</tr>
<tr>
<td>config</td>
<td>The ServletConfig instance for this JSP.</td>
<td>Page</td>
</tr>
<tr>
<td>page</td>
<td>The instance of this page’s implementation class that is processing the current request. A synonym for this when the programming language is Java.</td>
<td>Page</td>
</tr>
<tr>
<td>exception</td>
<td>The uncaught Throwable exception that caused the error page to be invoked.</td>
<td>Page</td>
</tr>
</tbody>
</table>
For more information about the exception implicit object, see “Error handling” on page 73.

Scopes

There are four scopes:

<table>
<thead>
<tr>
<th>Scope</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page</td>
<td>Accessible only in the page in which the object is created. Released when the response is returned or the request forwarded.</td>
</tr>
<tr>
<td>Request</td>
<td>Accessible from pages processing the request in which the object is created. Released when the request has been processed.</td>
</tr>
<tr>
<td>Session</td>
<td>Accessible from pages processing requests in the same session in which the object is created. Released when the session ends.</td>
</tr>
<tr>
<td>Application</td>
<td>Accessible from pages processing requests in the same application in which the object is created. Released when the runtime environment reclaims the ServletContext.</td>
</tr>
</tbody>
</table>

Custom tags

Custom tags, also called tag extensions or custom actions, extend the capabilities of JSP pages. Tag libraries define a set of actions to be used within a JSP page for a specific purpose, such as handling SQL requests. The tag libraries you use in PowerBuilder must be built using another tool such as PowerJ.

The URI identifying a tag library is associated with a Tag Library Descriptor (TLD) file and with tag handler classes.

Tag handlers

A tag handler is a Java class that defines the semantics of an action. The implementation class for the JSP instantiates a tag handler object for each action in the page. Tag handler objects implement the javax.servlet.jsp.tagext.Tag interface, which defines basic methods required by all tag handlers, including doStartTag and doEndTag. The BodyTag interface extends the Tag interface by adding methods that enable the handler to manipulate its body.
Tag libraries are usually packaged as JAR files in the META-INF subdirectory of the Web application. To associate a tag library with the page, you use a taglib directive that identifies the URI where the tag library’s TLD file can be located. The TLD file must be in the class path of the JSP container and is usually placed in the Web application’s WEB-INF directory, or in a tlds subdirectory of the WEB-INF directory. The class files for the tag library must also be in the class path of the JSP container. Typically they are placed in the WEB-INF/classes directory.

In PowerBuilder, you can add tag libraries to a target from the Components tab of the System Tree or in the Deployment Configuration Properties dialog box for a JSP target. To use a custom tag in a page, you must first insert a taglib directive (see “Taglib directive” on page 66). The Web application’s deployment descriptor must also include information about the tag (see “Tag Libraries” on page 84).

You can use Web services in your JSP by generating custom tags for them. PowerBuilder 9 provides a wizard that creates a custom tag with the information necessary for calling a Web service in a JSP.

The wizard collects information such as the location of a WSDL file, the service, and port. You can specify overrides to the WSDL file for a custom bean name, Java class name, Java package name, TLD name, Jar name, output variables, and the selection of operations within a service. The wizard creates a TLD file, the Java source to process the custom tag and the compiled Java class files of the source. See “Building a Web Service Client” on page 97 for details about Web services.

Once you have generated your custom tag, it appears in the system tree. When you drag it to a JSP, the tag library is automatically associated with the page. You must specify the input and output arguments for the custom tags for the Web services in the JSP. All arguments must be represented as objects as they are stored in the pageContext of the JSP container.

Deployment of the custom tag for Web services is the same as deployment of any custom tag in PowerBuilder 9. See “JSP deployment configuration” on page 78 for details.
Custom tags for Web services throw a JspTagException for nonrecoverable errors. The JspTagException contains information about the root cause of the exception and the point the error occurred in processing the custom tag. Using the PowerBuilder 9 error handling facility, this exception can be mapped to a specific error page. See “Error Mapping” on page 83 for details.

Example

This example demonstrates how custom tags for Web services on a JSP are defined to a JSP container.

First you declare the custom tag library to the JSP. This makes all of the tags in the library available to the JSP. The exchange prefix allows for easy reference to the tag library.

```jsp
<%@ taglib uri="WEB-INF/tlds/examples.publish.Exchange.tld" prefix="exchange" %>
```

Once the tag library is available, you declare the input and output arguments for the custom tags and put them into the pageContext to allow the custom bean to access them. This allows data types other than String to be used by the custom tag.

```jsp
<%
    String firstCountry = "usa";
    String secondCountry = "japan";
    Float returnValue;
    pageContext.setAttribute("firstCountry", firstCountry);
    pageContext.setAttribute("secondCountry", secondCountry);
%>
```

Next you invoke the Web service through the custom tag.

```jsp
<exchange:getRate country1="firstCountry" country2="secondCountry" />
```

Get the value of the returnValue variable from the custom tag and display it. The returnValue is set when the tag is executed.

```jsp
<% out.println("The exchange rate between " + firstCountry + " and " + secondCountry + "is: " + returnValue); %>
```

Comments

You can add two types of comments to a JSP file:

- HTML comments optionally contain an expression. They are sent to the client and can be viewed in the page source:

  ```jsp
  <!-- Copyright (C) 2002 Acme Software -->
  ```
• Hidden comments document the source file and are not sent to the client:

    <!-- Add new module here -->

To insert a comment, type it in the Source view.

**Error handling**

When a client request is processed, runtime errors can occur in the body of the implementation class for the JSP or in Java code that is called by the page. These exceptions can be handled in the code in the JSP page, using the Java language’s exception mechanism.

**Uncaught exceptions**

Exceptions that are thrown from the body of the implementation class that are not caught can be handled using an error page. You specify the error page using a page directive. Both the client request and the uncaught exception are forwarded to the error page. The java.lang.Throwable exception is stored in the javax.servlet.jsp.jspException instance for the client request using the putAttribute method, with the name javax.servlet.jsp.jspException.

**Using an error page JSP**

If you specify a JSP page as the error page, you can use its implicit exception variable to obtain information about the exception. The exception object is of type java.lang(Throwable and is initialized to the Throwable reference when the uncaught exception is thrown. For more information about the exception object, see “Implicit objects” on page 68.

To specify an error page for a JSP, set its errorPage attribute to the URL of the error page in a page directive:

    <%@ page errorPage="ErrorPage.jsp" %>

To define a JSP as an error page, set its isErrorPage attribute to true:

    <%@ page isErrorPage="true" %>

This sample error page uses the exception object’s toString method to return the name of the class of the object causing the exception and the result of the getMessage method for the object. If no message string was provided, toString returns only the name of the class.

The example also uses the getParameterNames and getAttributeNames methods of the request object to obtain information about the request.

    <%@ page language="java" import="java.util.*"
    isErrorPage="true" %>
    <H1 align="Center">Exceptions</H1>
    <br>
JSP Web target object model

The JSP object model is based on the existing PowerDynamo object model. Its classes handle the complexities of data transfer, HTML generation, and JavaScript generation for server scripts.

The non-4GL part of the JSP object model provides a set of utility Java classes that implement this functionality and encapsulate most of the JSP page's implicit objects. These object classes can be used on non-4GL Web pages as well as on 4GL Web pages.

Use of constructors

For ASP and PowerDynamo targets, you do not need to use a constructor for objects of type PSCommandClass, PSCursorClass, or PSConnectionClass. You can simply designate an untyped variable to reference an instance of the object that is returned by the CreateCommand, CreateConnection, or CreateCursor methods. For JSP targets you must assign a variable of the correct class type before you can create an instance of the object or call methods on it.

<% exception.toString() %>
<%! Enumeration parmNames; %>
<%! Enumeration attrNames; %>
<br>Parameters:
<%
parmNames = request.getParameterNames();
while (parmNames.hasMoreElements()) {
  <br><%= parmNames.nextElement().toString() %>
<%
}
%>
<br>
Attributes:<%
attrNames = request.getAttributeNames();
while (attrNames.hasMoreElements()) {
  <br><%= attrNames.nextElement().toString() %>
<%
}
%>
PSConnectionClass constructor

The PSConnectionClass constructor can use either of two distinct syntaxes. For either syntax, you can optionally include a boolean argument to allow tracing. In ASP and PowerDynamo targets, objects of the PSConnectionClass type allow you to connect to a database primarily through ODBC. The constructors in JSP targets are primarily designed for connection through JDBC. The JSP syntaxes are:

**PSConnectionClass syntaxes with user name and password**

```java
PSConnectionClass ( pageContext, Driver, URL, user, password,
{bTrace} )
```

**PSConnectionClass syntaxes with database properties**

```java
PSConnectionClass ( pageContext, Driver, URL, Properties,
{bTrace} )
```

<table>
<thead>
<tr>
<th>PSConnectionClass constructor</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pageContext</code></td>
<td>PageContext</td>
<td>The implicit pageContext object available to JSP targets (that does not require instantiation).</td>
</tr>
<tr>
<td><code>Driver</code></td>
<td>String</td>
<td>The name of the JDBC driver used to connect to the database.</td>
</tr>
<tr>
<td><code>URL</code></td>
<td>String</td>
<td>The location of the database to which you want to connect. The database URL is obtained from the database JDBC driver documentation.</td>
</tr>
<tr>
<td><code>user</code></td>
<td>String</td>
<td>The user name that the object uses to connect to the specified database.</td>
</tr>
<tr>
<td><code>password</code></td>
<td>String</td>
<td>The password that the object uses to connect to the specified database.</td>
</tr>
<tr>
<td><code>Properties</code></td>
<td>String</td>
<td>Any properties that your JDBC driver uses to connect to the database. If properties are defined, you must also define the user ID and password in the properties that you list.</td>
</tr>
<tr>
<td><code>bTrace</code> (Optional)</td>
<td>boolean</td>
<td>Allows tracing if set to true. The default is false.</td>
</tr>
</tbody>
</table>

PSCommandClass constructor

The PSCommandClass constructor uses the following syntax:

```java
PSCommandClass ( strSql, conn )
```
The PSCursorClass constructor uses the following syntax:

```
PSCursorClass( ResSet)
```

### Types and methods not implemented for JSP targets

The PSNamedConnectionParmsClass class type is not implemented for JSP targets. The psServer GetConnection and MapPath methods, and the SetSQL method on the PSCursorCommand object are not implemented for JSP targets.

The GetValue method on the PSCursor object does not return a value of a set data type and therefore cannot be used with JSP pages. This method has been replaced by a series of methods that return values of a specific data type:

<table>
<thead>
<tr>
<th>Method</th>
<th>Return value data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetColumn&lt;DataType&gt; (String strColName) where &lt;DataType&gt; can be Boolean, Byte, Double, Float, Int, Long, Short, or String</td>
<td>Corresponds to DataType used in method name</td>
</tr>
<tr>
<td>GetColumn&lt;DataType&gt; (int iColNo) where &lt;DataType&gt; can be Boolean, Byte, Double, Float, Int, Long, Short, or String</td>
<td>Corresponds to DataType used in method name</td>
</tr>
<tr>
<td>GetColumnLength (String strColName)</td>
<td>int</td>
</tr>
<tr>
<td>GetColumnLength (int iCol)</td>
<td>int</td>
</tr>
<tr>
<td>GetColumnName(int iCol)</td>
<td>String</td>
</tr>
<tr>
<td>GetColumnType(int iCol)</td>
<td>int</td>
</tr>
<tr>
<td>GetColumnTypeName (int iCol)</td>
<td>String</td>
</tr>
<tr>
<td>GetPrecision(int iCol)</td>
<td>int</td>
</tr>
<tr>
<td>GetResultSet()</td>
<td>ResultSet</td>
</tr>
<tr>
<td>GetResultSetMetaData()</td>
<td>ResultSetMetaData</td>
</tr>
<tr>
<td>GetScale(int iCol)</td>
<td>int</td>
</tr>
</tbody>
</table>
CreateConnection method

The psServer CreateConnection method has separate syntaxes for JSP targets that allow it to return objects of the PSConnectionClass type for these targets.

**Syntax with user name and password**

```java
psServer.CreateConnection ( pageContext, Driver, URL, user, password, {bTrace} )
```

**Syntax with database properties**

```java
PSConnectionClass ( pageContext, Driver, URL, Properties, {bTrace} )
```

The arguments for these syntaxes are described in the table for the PSConnectionClass constructor.

Server side events

When you enable 4GL functionality on a JSP page, you can rely on an event-driven infrastructure to handle many of the details of coding server scripts. The JSP 4GL object model provides foundation classes for the event-driven infrastructure, such as the server control classes, the DataWindow class, server variables, and parameter classes.

Unlike PowerDynamo or ASP Web targets, in JSP Web targets you must script a return value for the server-side events of a 4GL Web page that have a return value as part of their event signature. If there is no return statement, a servlet translation error occurs at runtime. The following server-side events on the 4GL psPage object have boolean return types: BeforeAction, BeforeGenerate, FirstTime, RequestStart, ServerError, and Validate.

New 4GL method

The doTrace property for the psPage object is not available for JSP Web targets. The errors property for the psPage object has a Vector data type (rather than a String data type, as in a Dynamo or ASP Web target). The Vector data type for the errors property is a collection of PageError objects. PageError is a value class with three string attributes (location, cause, message).

The psPage object has a new method available for JSP targets to check if tracing is on before calling the Trace method multiple times. The IsTrace method returns a boolean and takes no arguments. You can use the IsTrace and SetTrace methods in place of the doTrace property that is not available for JSP Web targets.

Variables

When you create variables on the Variables page of the Page Properties dialog box for 4GL JSP pages, you specify the same properties that you do for a 4GL HTML page in a Web target. You must also associate a data type with the variable. The following variable data types are supported in 4GL JSP pages: boolean, byte, char, double, float, int, long, short, and String.
JSP deployment configuration

When you deploy the JSP target, PowerBuilder builds a Web Archive (WAR) file in the deployment configuration folder created by the JSP Web Target wizard. The WAR file contains the JSP files you added to the target, any classes or JAR files you added to the Source folder, and a web.xml file that conforms to the Document Type Definition (DTD) for Web applications. The Web application is automatically deployed to the server you selected in the target wizard.

The web.xml file is the deployment descriptor for the Web application. The deployment configuration properties you define in the JSP Options section of the JSP Deployment Configuration Properties dialog box are written to the web.xml file.

In general, you should not edit the web.xml file manually. The changes you make in the Deployment Configuration Properties dialog box are propagated to copies of the web.xml file in the WAR file and appropriate subdirectories of your JSP target.

To open the Deployment Configuration Properties dialog box, select Properties from the JSP target’s pop-up menu and double-click the deployment configuration on the Deploy page.

The next two sections describe the deployment configuration properties that you can set, in the order in which they display in the Deployment Configuration Properties dialog box for a JSP target.

General deployment options

The general deployment options of a JSP target allow you to:

- Enter a description for the deployment configuration
- Enter server information
- Include the JSP object model in the deployment archive
- Select a build strategy and how you want to handle deployment errors

You cannot change the deployment configuration name from the Deployment Configuration Properties dialog box. If you want to create a different deployment configuration name, close this dialog box and start the Name New Deployment Configuration wizard from the Deploy page of the Properties dialog box for the target.
From the Deploy page you can create a test (local) configuration, a production (target) configuration, or both. Local configurations are stored in your registry, whereas target configurations are stored in the PBT file. Target configurations, but not local configurations, can be shared in a source control system.

The general deployment options for a JSP target consist of four main selection pages that you access from the tree view in the Deployment Configuration Properties dialog box:

<table>
<thead>
<tr>
<th>Deployment selection page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>Type a description for the deployment configuration.</td>
</tr>
<tr>
<td>Server Information</td>
<td>Lists the server types to which you can deploy. The current selection is highlighted. Additional selections are available depending on which server you select:</td>
</tr>
<tr>
<td></td>
<td><strong>EAServer</strong> Select a deployment profile.</td>
</tr>
<tr>
<td></td>
<td><strong>Tomcat</strong> Select the deployment folder and the HTTP server and port. You can also select a login name and password, and choose to stop and restart the server automatically after the target is deployed.</td>
</tr>
<tr>
<td>Object Model</td>
<td>Select whether you want to deploy the JSP object model with your JSP target. You can select only the default JSP object model.</td>
</tr>
<tr>
<td>Deploy What?</td>
<td>Select Deploy All Or Nothing to make sure that nothing gets deployed when one of the files selected for deployment fails the build or predeployment processing. Select Deploy Only Successful Files to prevent failure of a single file from affecting deployment of other files in the target. The Rebuild field lets you select whether to use an incremental or full rebuild of files you select for deployment with the current configuration.</td>
</tr>
<tr>
<td>Local Copy Folder</td>
<td>When you build the JSP target, PowerBuilder generates a WAR file containing JSP files and supporting objects in the folder you specify as the Local Copy Folder. You can clear the Make Local Copy Of Deployed Files check box only if you selected the Deploy Only Successful Files option.</td>
</tr>
</tbody>
</table>
### JSP deployment options

The JSP options that you specify in the Deployment Configuration Properties dialog box are added to the deployment descriptor for the WAR file that contains the Web application.

For beta 3, any changes you make to JSP deployment options apply to all deployment configurations in the same target.

<table>
<thead>
<tr>
<th>JSP options</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Web Application Name</strong></td>
<td>The Web Application Name is the display name used on the server to identify a deployed WAR file.</td>
</tr>
<tr>
<td><strong>Description</strong></td>
<td>Use the description box to provide any information that might be required by the consumer of the application.</td>
</tr>
<tr>
<td><strong>Session Timeout</strong></td>
<td>Session Timeout is a specified time in minutes after which the server will terminate servlet sessions. This value applies to all the servlets within an application. A value of 0 indicates that servlet sessions never expire.</td>
</tr>
<tr>
<td><strong>Distributable</strong></td>
<td>Web applications can run on only one Java VM at any one time. To override this rule, you must mark the Web application as distributable in the deployment descriptor. However, the application must conform to additional requirements. A distributable Web application cannot use <code>setAttribute</code> and <code>putValue</code> methods to place objects into a <code>javax.servlet.http.HttpSession</code> object unless the object is one of the following types:</td>
</tr>
</tbody>
</table>

- java.io.Serializable
- javax.ejb.EJBObject
- javax.ejb.EJBHome
- javax.transaction.UserTransaction
- javax.naming.Context object for the java:comp/env context

Context Params

The Context Params page is where you specify the value of parameters that convey initialization information for the Web application, such as a Web master’s address or the name of a system that holds critical data. They can be retrieved using the `getInitParameter` and `getInitParameterNames` methods of the `ServletContext` interface.

In a JSP page, the parameter can be retrieved in a scriptlet using the application implicit object, for example:

```<% String iURL = application.getInitParameter("iURL"); %>```

Filters

Filter Content

You can write a filter to modify requests and responses and then declare it on the Filters page. Filters implement the `javax.servlet.Filter` interface.

For more information about filters, see the Java Servlet specification or the EAServer Programmer’s Guide.

Filter Mapping

The container uses the filter mappings you specify on the Filter Mapping page to determine how to apply the filters that have been defined to requests. You can apply a filter to a single servlet by specifying its name, or to a group of servlets and other Web content by specifying a URL pattern. For example, `*` specifies that a filter applies to all servlets in the Web application. The filters are applied in the order in which they appear in the list of filter-mapping elements in the deployment descriptor.

Listeners

You can provide listener classes implementing one or more of the listener classes in the Servlet API. Listeners can support event notifications or manage resources or state. You package the listener classes in the WAR file and list them in the deployment descriptor in the order in which they are to be invoked.
Servlets

Servlet Details

Use the servlet pages to describe a servlet class or JSP page used in the Web application. Click New to give the servlet or page a short name that can be used to reference it. Then select Servlet Class or JSP Filename from the drop-down list box. For servlets, you must specify the fully qualified class name in the text box next to the drop-down list box.

If you want to see target JSPs listed in the management tool for your server, you must enter a short name for each JSP, select JSP Filename from the drop-down list box, and enter the JSP file name in the text box next to the drop-down list box. However, this information is not required for access to the JSPs from a client browser.

You can specify the following properties for each servlet or JSP from the Deployment Configuration Properties dialog box: Load on Startup, Init Param, Role Refs, and Servlet Mapping URL Pattern.

Load on Startup

Load on Startup indicates whether you want a servlet loaded and initialized when the application is deployed. Otherwise the servlet class is loaded when the first client requests it. Servlet classes that perform lengthy processing in the init method can be loaded at startup so that the first client to invoke the servlet does not experience increased response time.

A value of 0 or a positive integer requires the container to load the servlet when the application is deployed. Servlets with a low Load on Startup value are loaded before those with a higher value. If you do not specify a value, or if you specify a negative integer, the container can load the servlet at any time.

Init Param

Use the Init Param table to assign values of parameters specifying setup information for the servlet or JSP page. In a JSP page, the parameter can be retrieved in a scriptlet using the config implicit object, for example:

```<%
    String initVal = config.getInitParameter("initVal");
%>```

Role Refs

Role references provide a mechanism an application can use to map a role name used in the application’s code to a security role defined in its deployed environment.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Name of the security role used as a parameter to the IsCallerInRole method</td>
</tr>
<tr>
<td>Description</td>
<td>(Optional) A comment to explain how the property is used</td>
</tr>
<tr>
<td>Link</td>
<td>The security role (see “Roles” on page 87) to which this reference should be linked</td>
</tr>
</tbody>
</table>
Servlet Mapping URL Pattern

A servlet mapping defines the association between a URL pattern and a servlet. This mapping is used to map requests to servlets. The default is /ServletTargetName, for example /MyServlet.

If the container handling the request is a JSP container, a URL containing a .jsp extension is implicitly mapped.

Mime Mapping

Specify mime mappings to ensure that the Web container knows how to associate a file extension with a mime type. For example, if you specify .txt as the extension, you must specify a predefined mime type such as text/plain.

Welcome Files

The welcome file list contains an ordered list of welcome file elements to be used when the container receives a valid partial request. A valid partial request is a request for a URI that corresponds to a directory entry in the WAR not mapped to a Web component.

For example, if the container receives a request for //myhost:8080/myapp/mydir, and mydir is not mapped to a servlet or JSP file, then if the welcome file list includes the mapping mydir/index.html, index.html is displayed.

Error Mapping

You can customize what the client sees when an error is generated by specifying the locations of error pages for different kinds of errors. Error pages you specify here are used for servlets and for any JSP pages that do not specify an error page for the error type.

In the left column, you can specify an HTTP error code, for example 404, or a fully qualified class name of a Java exception type. In the right column, specify where to find the resource in the Web application relative to the root of the Web application. The value of the location must have a leading forward slash (/). For example, /404.html.
Tag Libraries

If the Web application uses one or more tag libraries, you can make sure that the Web container can locate them by specifying a mapping for each tag library in the deployment descriptor. If you selected tag libraries in the JSP Web Target wizard, they display here.

You use a taglib directive to refer to a tag library in a JSP page. For example:

```html
<%@ taglib uri="/WEB-INF/tlds/mycalc.tld" prefix="mc" %>
```

The uri attribute specifies the uniform resource locator (URI) for the TLD file relative to the root of the Web application. You can map this path to a short name in the deployment descriptor. Specify the name you want to use in the Tag Library URI column, and the location relative to the root of the Web application in the Descriptor File Location column. The value of the location must have a leading forward slash (/). For example, `/WEB-INF/tlds/Testlibrary_1_3.tld`.

If you specify `/mycalc` as the short name for the `/WEB-INF/tlds/mycalc.tld`, the taglib directive can be written like this:

```html
<%@ taglib uri="/mycalc" prefix="mc" %>
```

Resource References

To be platform independent, an application should refer to resources within the operating environment in which it is deployed, rather than having a specific location coded within the application. The J2EE specification defines a mechanism for an application to obtain resource references in its deployed environment. Resource references are used to obtain database connections, JavaMail sessions, URL factories, and JMS connection factories.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify the JNDI name used to refer to a resource. Use the prefix <code>mail/</code> for JavaMail references, <code>jdbc/</code> for data source references, <code>url/</code> for <code>java.net.URL</code> references, and <code>jms/</code> for <code>javax.jms</code> references. For example, if your code refers to <code>java:comp/env/jdbc/MyDatabase</code>, enter <code>jdbc/MyDatabase</code>.</td>
</tr>
</tbody>
</table>
### Env References

Resource environment references allow the JSP page to use logical names to refer to administered objects associated with resources. These references must be bound to administered objects in the deployment environment.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Use one of these resources:</td>
</tr>
<tr>
<td></td>
<td>• javax.sql.DataSource for JDBC connections</td>
</tr>
<tr>
<td></td>
<td>• java.net.URL for URL factories</td>
</tr>
<tr>
<td></td>
<td>• javax.mail.Session for mail sessions</td>
</tr>
<tr>
<td></td>
<td>• javax.jms.QueueConnectionFactory for a JMS queue</td>
</tr>
<tr>
<td></td>
<td>• javax.jms.TopicConnectionFactory for a JMS topic</td>
</tr>
<tr>
<td>Authentication</td>
<td>Enter:</td>
</tr>
<tr>
<td></td>
<td>• Container if the container signs on to the resource manager on behalf of the servlet component. The methodology used to sign on is server specific.</td>
</tr>
<tr>
<td></td>
<td>• Application if the application signs on programmatically to the resource manager.</td>
</tr>
<tr>
<td></td>
<td>• Servlet if the servlet (not the container) signs on programmatically to the resource manager.</td>
</tr>
<tr>
<td>Sharing Scope</td>
<td>By default, connections to a resource manager can be shared by other components that use the resource in the same transaction context, optimizing the use of connections. Select Unshareable if the application cannot share connections to the resource.</td>
</tr>
<tr>
<td>Description</td>
<td>(Optional) A comment to explain how the property is used.</td>
</tr>
</tbody>
</table>

### Security

Security constraints let you control access to a Web resource collection. A Web resource collection identifies the resources, defined by URL patterns, and the HTTP methods on those resources, to which the security constraints apply. The security constraints define the roles authorized to use the Web resource collection (authorization constraint) and the level of transport security required of the client server (user data constraint).

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Security</td>
<td></td>
</tr>
<tr>
<td>Security Constraints</td>
<td>Security constraints let you control access to a Web resource collection. A Web resource collection identifies the resources, defined by URL patterns, and the HTTP methods on those resources, to which the security constraints apply. The security constraints define the roles authorized to use the Web resource collection (authorization constraint) and the level of transport security required of the client server (user data constraint).</td>
</tr>
</tbody>
</table>
You define the Web resource collection and its constraints on the Security Constraints page.

If you do not assign a user role, no user has access to the resources in the specified collection. If you do not specify HTTP methods, the constraints apply to all methods.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specify a name for the Web resource collection.</td>
</tr>
<tr>
<td>URL Pattern</td>
<td>Select one or more URL patterns to specify the resources in this Web application to which the constraints apply.</td>
</tr>
<tr>
<td>HTTP Methods</td>
<td>(Optional) Specify the HTTP methods to which the constraints apply. If you do not specify any methods, the constraints apply to all methods.</td>
</tr>
<tr>
<td>Authorized Roles</td>
<td>Select the roles authorized to access the collection of Web resources defined in the URL Pattern and HTTP Methods boxes. You can define roles on the Roles page of the Deployment Configuration Properties dialog box.</td>
</tr>
<tr>
<td>Transport Guarantee</td>
<td>Establish a level of transport security appropriate for the Web resources you are protecting. If you use basic or form-based authentication, passwords and other sensitive information are not protected for confidentiality. If you have sensitive information that you want to protect, establish a security constraint that uses a greater level of protection:</td>
</tr>
<tr>
<td></td>
<td>• NONE – uses insecure HTTP. SSL-protected sessions require more overhead than insecure HTTP sessions. Use none for transport guarantee if you do not need the added confidentiality of SSL.</td>
</tr>
<tr>
<td></td>
<td>• INTEGRAL – uses an SSL-protected session that checks for data integrity.</td>
</tr>
<tr>
<td></td>
<td>• CONFIDENTIAL – uses an SSL-protected session to ensure that all message content, including the client authenticators, is protected for confidentiality as well as data integrity. A confidential transport guarantee has more overhead than none.</td>
</tr>
</tbody>
</table>

Protected resources on a server can be partitioned into separate protection spaces. Each protection space can be configured with a specific security scheme, such as an authentication protocol or authorization database. When a Web server asks a client to authenticate a user, it passes a realm to the client. A realm is a string that defines a protection space.
Use of the term realm
In J2EE applications, the term realm is also used to refer to a security policy domain. In this deployment descriptor, it refers to the string passed as part of HTTP basic authentication.

The client passes the user name and password to the Web server, and the Web server authenticates the user in the specified realm. The login-config element is used to configure the authentication method, the realm name that should be used for this application, and the attributes that are needed by the form login mechanism.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Authentication Method</td>
<td>Select the authentication method to be used to configure the authentication mechanism for the Web application:</td>
</tr>
<tr>
<td></td>
<td>• BASIC – the server asks the client for a user name and password. You must also provide a realm name.</td>
</tr>
<tr>
<td></td>
<td>• DIGEST – advanced form of BASIC authentication using an MD5 message-digest hash of the credentials and a unique value supplied by the server: the password is not sent in clear, unencrypted text as with BASIC authentication.</td>
</tr>
<tr>
<td></td>
<td>• FORM – the Web application developer creates an HTML login page, where the client enters a user name and password. The entire HTML page is sent to the server. You also create an error page that is returned to the client in the event of a server error.</td>
</tr>
<tr>
<td></td>
<td>• CLIENT-CERT – the client connects to the server using SSL tunneled within HTTP. The client must provide a certificate that the server accepts and authenticates.</td>
</tr>
<tr>
<td>Realm Name</td>
<td>Specify the realm name to be used in HTTP basic authentication.</td>
</tr>
<tr>
<td>Form Login Page</td>
<td>Specify the location in the Web application where the page to be used for login can be found. The path begins with a leading / and is interpreted relative to the root of the Web application.</td>
</tr>
<tr>
<td>Form Error Page</td>
<td>Specify the location in the Web application where the error page that is displayed when login fails can be found. The path begins with a leading / and is interpreted relative to the root of the Web application.</td>
</tr>
</tbody>
</table>

Roles
A security role is a grouping of permissions that a given type of user of an application must have to successfully use an application and its components. The Roles page allows you to define security roles, for example admin or user, that you can associate with specific resources on the Security Constraints page.
Environment

Environment properties allow you to specify global read-only data for use by all the JSP pages in the Web application.

Servlets and JSP pages must use JNDI to retrieve environment properties, using the prefix `java:comp/env` in JNDI lookups. Unlike context initialization properties, environment properties can have data types other than `java.lang.String`.

The deployment descriptor catalogs the environment properties used by your servlets and JSP pages, as well as each property’s Java data type and default value. You can tailor the values to match a server’s configuration. For example, you may have environment properties to specify the name of a logging file or to tune cache usage.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Specifies the JNDI name, relative to the <code>java:comp/env</code> prefix, used in servlet and JSP code to refer to this resource.</td>
</tr>
<tr>
<td>Type</td>
<td>Select the Java data type of the property from the drop-down list box. The specified type must have a constructor that takes a single <code>java.lang.String</code> argument.</td>
</tr>
<tr>
<td>Value</td>
<td>The initial or post-deployment value of the property, specified as text that is valid for the type constructor that takes a single <code>java.lang.String</code> argument.</td>
</tr>
<tr>
<td>Description</td>
<td>(Optional) A comment to explain how the property is used.</td>
</tr>
</tbody>
</table>

EJBs

EJBs that support the EJB 2.0 specification can have both remote and local interfaces.

EJB References

When servlets and JSP pages reference remote EJBs, the EJB reference in the deployment descriptor is used to instantiate proxies for EJB home interfaces. EJB references must be catalogued in the deployment descriptor so that the Web application does not depend on a specific naming configuration. When deploying the Web application, a site administrator can specify site-specific EJB home names.

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name (New button)</td>
<td>Click New to create a new remote reference to an enterprise bean. Specifies the JNDI name used to refer to this EJB.</td>
</tr>
<tr>
<td>Type</td>
<td>Choose Session for session beans or Entity for entity beans.</td>
</tr>
<tr>
<td>Home Interface</td>
<td>The Java class name of the EJB home interface, specified in dot notation. For example, <code>com.sybase.myBeanHome</code>.</td>
</tr>
</tbody>
</table>
Servlets and JSP pages can reference EJBs running in the same Java VM using local interfaces. The settings for EJB local references are analogous to the settings for EJB references, which are used when the EJB is not running in the same Java VM.

### Local References

<table>
<thead>
<tr>
<th>Setting</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Remote Interface</td>
<td>The Java class name of the EJB remote interface, specified in dot notation. For example, com.sybase.myBeanRemote.</td>
</tr>
<tr>
<td>Description</td>
<td>(Optional) A comment to describe the EJB reference.</td>
</tr>
<tr>
<td>Link</td>
<td>The JNDI name of an instance of the specified EJB that is installed in the server where the Web application is to be deployed.</td>
</tr>
</tbody>
</table>

### Custom tag library for the Web DataWindow

You can use the Web DataWindow custom tag library to specify the parameters and values required by a Web DataWindow on a JSP page. The tag library is defined in the file `DataWindow90.tld`. To use the tag library, place the `DataWindow90.tld` file in a `WEB-INF/tlds` directory in your Web applications Source directory. The tag classes are included in the `jspobject.jar` file that is deployed with all PowerBuilder JSP Web applications.

The tag library contains two tags, DataWindow and DWCOLUMNLink. The DWCOLUMNLink tag is an inner tag—it can be used only inside the DataWindow tag.

Attributes have three subelements: name, required, and rtexprvalue. The rtexprvalue element is optional and indicates whether the attribute’s value can be dynamically calculated at runtime.
**DataWindow**

**Description**
Sets parameters for a Web DataWindow on a JSP page.

All DataWindow tag attributes are required unless noted in the Description column. The value of the rtexprvalue subelements is true for all attributes.

<table>
<thead>
<tr>
<th>Attributes of DataWindow tag</th>
<th>Java type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>String</td>
<td>Optional identifier.</td>
</tr>
<tr>
<td>libName</td>
<td>String</td>
<td>See the sourceFileName property for PSDataWindowSourceClass in the Web Target Reference.</td>
</tr>
<tr>
<td>dwName</td>
<td>String</td>
<td>See the dwName property for PSDataWindowSourceClass in the Web Target Reference.</td>
</tr>
<tr>
<td>allowForm</td>
<td>boolean</td>
<td>See the bAllowForm argument for PSDataWindowClass.setWeight in the Web Target Reference.</td>
</tr>
<tr>
<td>clientValidation</td>
<td>boolean</td>
<td>See the bClientValidation argument for PSDataWindowClass.setWeight in the Web Target Reference.</td>
</tr>
<tr>
<td>clientEvents</td>
<td>boolean</td>
<td>See the bClientEvents argument for PSDataWindowClass.setWeight in the Web Target Reference.</td>
</tr>
<tr>
<td>clientScriptable</td>
<td>boolean</td>
<td>See the bclientScriptable argument for PSDataWindowClass.setWeight in the Web Target Reference.</td>
</tr>
<tr>
<td>clientFormatting</td>
<td>boolean</td>
<td>See the bClientFormatting argument for PSDataWindowClass.setWeight in the Web Target Reference.</td>
</tr>
<tr>
<td>dbms</td>
<td>String</td>
<td>See the dbms constructor for PSConnectionParmsClass in the Web Target Reference.</td>
</tr>
<tr>
<td>dbparm</td>
<td>String</td>
<td>See the dbparm constructor for PSConnectionParmsClass in the Web Target Reference.</td>
</tr>
<tr>
<td>lock</td>
<td>String</td>
<td>See the lock constructor for PSConnectionParmsClass in the Web Target Reference.</td>
</tr>
<tr>
<td>logid</td>
<td>String</td>
<td>See the user constructor for PSConnectionParmsClass in the Web Target Reference.</td>
</tr>
<tr>
<td>Attributes of DataWindow tag</td>
<td>Java type</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------------</td>
<td>-----------</td>
<td>-------------</td>
</tr>
<tr>
<td>logpass</td>
<td>String</td>
<td>See the password constructor for PSCConnectionParmsClass in the Web Target Reference.</td>
</tr>
<tr>
<td>database</td>
<td>String</td>
<td>See the database constructor for PSCConnectionParmsClass in the Web Target Reference.</td>
</tr>
<tr>
<td>servername</td>
<td>String</td>
<td>See the serverName constructor for PSCConnectionParmsClass in the Web Target Reference.</td>
</tr>
<tr>
<td>jagservername</td>
<td>String</td>
<td>See the serverName constructor for PSJaguarConnection in the Web Target Reference.</td>
</tr>
<tr>
<td>jaglogid</td>
<td>String</td>
<td>(Optional) See the userid constructor for PSJaguarConnection in the Web Target Reference.</td>
</tr>
<tr>
<td>jaglogpass</td>
<td>String</td>
<td>(Optional) See the password constructor for PSJaguarConnectionClass in the Web Target Reference.</td>
</tr>
<tr>
<td>selfLink</td>
<td>String</td>
<td>The URL for the current page.</td>
</tr>
<tr>
<td>selfLinkArg</td>
<td>String</td>
<td>Page parameters to be passed to the server. The syntax is: argname='exp'</td>
</tr>
<tr>
<td>action</td>
<td>String</td>
<td>See the action argument for the SetAction DataWindow method in the DataWindow Reference.</td>
</tr>
<tr>
<td>context</td>
<td>String</td>
<td>See the context argument for the SetAction DataWindow method in the DataWindow Reference.</td>
</tr>
<tr>
<td>argument</td>
<td>String</td>
<td>See the argument argument for the RetrieveEx DataWindow method in the DataWindow Reference.</td>
</tr>
<tr>
<td>dwHTMLObjectName</td>
<td>String</td>
<td>See the objectname argument for the SetHTMLObjectNam DataWindow method in the DataWindow Reference.</td>
</tr>
<tr>
<td>pageSize</td>
<td>String</td>
<td>(Optional) See the pagesize argument for the SetPageSize DataWindow method in the DataWindow Reference.</td>
</tr>
</tbody>
</table>
**DWColumnLink**

**Description**

Establishes a link on a column that is passed from the database to the Web DataWindow control. This link lets the Web DataWindow DTC pass data to another page.

**Attributes**

All DWColumnLink tag attributes are required. The value of the rtexprvalue subelements is unspecified for all attributes.

<table>
<thead>
<tr>
<th>Attributes of DWColumnLink tag</th>
<th>Java type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sColumnName</td>
<td>String</td>
<td>The name of the column that you want to link to a target page.</td>
</tr>
<tr>
<td>sColLink</td>
<td>String</td>
<td>The URL target of a link from a data item in the column.</td>
</tr>
<tr>
<td>sColLinkArgs</td>
<td>String</td>
<td>The arguments passed with the link argument.</td>
</tr>
<tr>
<td>sColLinkTarget</td>
<td>String</td>
<td>The name of a target frame or window for the link specified in the Link argument. The target is included in the HTML element using the HTML TARGET attribute. You can use sColLinkTarget to link from a master to a detail page by specifying a different window or frame for the detail page.</td>
</tr>
</tbody>
</table>

**Example using the DataWindow tag**

This example uses two JSP pages. Both use the DataWindow tag. The first, Departments.jsp, uses two nested DWColumnLink tags to pass data to the Employees.jsp page.

The deployment descriptor for the application must include a taglib element that associates the short name DW90 with the DataWindow90.tld file in the Web application’s /WEB-INF/tlds directory:

```xml
<taglib>
  <taglib-uri>/DW90</taglib-uri>
  <taglib-location>/WEB-INF/tlds/DataWindow90.tld
</taglib-location>
</taglib>
```

The deployment descriptor for the application is the file web.xml, which resides in the Web application’s /WEB-INF directory. For more information, see “JSP deployment configuration” on page 78.

**Departments.jsp**

```jsp
<%@ page import="com.sybase.powerbuilder.jspobject.*"
```
%>
<%  // global instance for the page
    PSDocumentClass psDocument = new PSDocumentClass
      (request, response, out, application);
    PSSessionClass psSession = new
      PSSessionClass(session);
%>
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0
  Transitional//EN">

<!-- Import tag class--%>
<%@ taglib prefix="webdw" uri="/DW90" %>
<HTML>
<HEAD>
<META HTTP-EQUIV="PowerSiteData" NAME="SERVERLANGUAGE"
  CONTENT="JavaScript">
<TITLE></TITLE>
<META http-equiv="Content-Type" content="text/html">
<META content="MSHTML 5.50.4522.1800" name="GENERATOR">
</HEAD>
<BODY PSPARAMS="">

<!-- Use DataWindow custom tag--%>
<webdw:DataWindow argument=""
    selfLinkArg=""
    logpass=""
    jaglogpass=""
    dbms="ODBC"
    servername=""
    clientScriptable="true"
    clientFormatting="true"
    action=""
    selfLink="dwpage2.jsp"
    jaglogid="jagadmin"
    dwHtmlObjectName="dwTest"
    logid="" lock=""
    clientEvents="true"
    clientValidation="true"
    libName="f:\Mywork\Pbjsp\dw_departments.srd"
    database=""
    dbparm="ConnectString='DSN=EAS Demo DB V4;
      UID=dba;PWD=sql',ConnectOption=
      'SQL_DRIVER_CONNECT,SQL_DRIVER_NOPROMPT'"
    jagservername="localhost:9000"
    dwName=""
Custom tag library for the Web DataWindow

context=""
  allowForm="true"
>
<webdw:DWColumnLink
  sColLink="/Employees.jsp"
  sColLinkArgs="dept_id='dept_id'"
  sColLinkTarget=""
  sColumnName="dept_id">
</webdw:DWColumnLink>
</webdw:DataWindow>
</BODY>
</HTML>

Employees.jsp

<%@ page import="com.sybase.powerbuilder.jspobject.*" %>

<%! String strDept; %>
<%  strDept = psDocument.GetParam("dept_id");%>

<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<%@ taglib prefix="webdw" uri="/DW90" %>
<HTML>
<HEAD><TITLE>
DataWindowJSP Example
</TITLE></HEAD>
<BODY>
<H2>
Basic JSP Datawindow: Employee List Report
</H2>
<%! String strDept; %>
<%  strDept = psDocument.GetParam("dept_id");%>

<webdw:DataWindow
  libName="d:\Mywork\Pbjsp\dw_employees.srd"
  dwName=""
  allowForm="true"
  clientValidation="true"
  clientEvents="true"
  clientScriptable="true"
  clientFormatting="true"
  dbms="ODBC"
  dbparm="ConnectString='DSN=EAS Demo DB V4;UID=dba;PWD=sql';ConnectOption="
  "
'SQL_DRIVER_CONNECT,SQL_DRIVER_NOPROMPT''
lock=""
logid=""
logpass=""
database=""
servername=""
jagservername="localhost:9000"
jaglogid="jagadmin"
jaglogpass=""
selfLink="Employees.jsp"
selfLinkArg=""
action=""
context=""
argument="<%=strDept%>"
dwHtmlObjectName="dwTest"
pageSize="10"
>

</webdw:DataWindow>
</BODY></HTML>
Building a Web Service Client

This section includes the following topics:

• About Web services
• About building a Web services client
• Adding PBSoapClient90.pbd to the library search path
• Generating Web service proxy objects
• Connecting to a SOAP server
• Invoking the Web service method
• Exception handling
• Reference information

About Web services

Web services are loosely defined as use of Internet technologies to make distributed software components talk to each other, without human intervention. Those software components might perform such business logic as getting a stock quote, searching the inventory for a catalog on the Internet, or integrating the reservation services for an airline and a car rental agency. You can reach across the Internet and use preexisting components, instead of having to write them for your application.

Web services were born when the Simple Object Access Protocol (SOAP) was introduced. SOAP leverages Extensible Markup Language (XML) and usually employs Hypertext Transfer Protocol (HTTP) as the transport. Invoking Web services through SOAP requires serialization and deserialization of datatypes, and the building and parsing of SOAP messages.
Part of the value of Web services comes from the Web Services Description Language (WSDL), which enables a service to be self-describing. WSDL defines an XML grammar for describing Web services as collections of communication endpoints capable of exchanging messages. WSDL service definitions provide documentation for distributed systems and serve as a recipe for automating the details involved in applications communication. These descriptions can be used by applications and/or registered on a Web site that uses Universal Description, Discovery, and Integration (UDDI). You can search UDDI registry sites and find the services you need for your application.

**Note** UDDI search functionality is not implemented in this release.

With SOAP, WSDL, and UDDI, using third party components is easier because interfaces between applications become standardized across disparate platforms.

PowerBuilder 9.0 supports the following Web services standards:

- SOAP 1.1
- WSDL 1.1
- UDDI version 1 (not supported in this release)
- HTTP or HTTPS

**About building a Web services client**

A PowerBuilder application can act as a client consuming a Web service that is accessed through the Internet. Using SOAP and WSDL, a collection of functions published remotely as a single entity can become part of your PowerBuilder application. A Web service accepts and responds to requests sent by applications or other Web services.

Invoking Web services through SOAP requires serialization and deserialization of data types, and the building and parsing of XML-based SOAP messages. Using the **PBSoapClient90.pbd** and the **PBSoapClient90.dll**, the Web services client proxy performs these tasks for you, thereby eliminating the need to have extensive knowledge of the SOAP specification and schema, the XML Schema specification, or the WSDL specification and schema.
Adding PBSoapClient90.pbd to the library search path

The PBSoapClient90.dll and PBSoapClient90.pbd files are installed in the Shared/PowerBuilder directory when you install PowerBuilder. When you create a Web service client application, you do not need to copy PBSoapClient90.dll to another location, but you do need to deploy it with the client executable in a directory in the application’s search path.

To add PBSoapClient90.pbd to the application’s search path, right-click the client target in the System Tree and select Properties from the pop-up menu. In this beta release, you need to type in the full path and name of the PBSoapClient90.pbd file.

After you add the PBSoapClient90.pbd, the following objects display in the System Tree:

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>soapconnection</td>
<td>Used to connect to a SOAP server</td>
</tr>
<tr>
<td>soapexception</td>
<td>Used to catch exceptions thrown from soapconnection</td>
</tr>
</tbody>
</table>

Generating Web service proxy objects

To create a new Web service proxy, select the Web Service Proxy Wizard icon from the Projects page in the New dialog box. The Web Service Proxy Wizard helps you create the proxy so you can use the Web service in PowerScript. One proxy is created for each port.

In the wizard you specify:

- which WSDL file you want to access
- which service within the WSDL file you want to select
- which port or ports you want to use
- a prefix which is appended to a port name, and becomes the proxy name
- to which PowerBuilder library you want the proxy deployed
Generating Web service proxy objects

You can also select the Web Service Proxy icon from the Projects page in the New dialog box. The Web Service Proxy icon opens the Project painter for Web services so that you can create a project, specify options, and build the proxy library. The new project lists the Web service and their ports for which proxies will be generated and specifies the name of the output library that will contain the generated proxy objects.

Whether you create the Web service project using the wizard or in the painter, the final step is to build the proxy objects, by clicking the Build icon on the painter bar or selecting Design>Deploy project from the menu bar.

Generated proxies

The generated proxies display in the System Tree. You can expand the proxy nodes to display the signatures of the methods.

You cannot open proxy objects in a painter or the Source editor, but you can examine the source of the object.

❖ Exporting the source of a proxy object

1. In the System Tree, select Export from the proxy object’s pop-up menu
   or
   In the Library painter, select the proxy object then select Entry>LibraryItem>Export.
2. Click the Save button to save the source to a file with the extension .srx.
PowerBuilder is case insensitive, whereas XML and SOAP are case sensitive. To ensure that PowerScript code can call XML methods correctly, each method in the proxy uses an alias. The string that follows alias for contains the name and the signature of the corresponding XML or SOAP method in case-sensitive mode.

For example:

```powerbuilder
function real getquote(string ticker) alias for getQuote(xsd:string symbol)
    return xsd:float StockPrice@urn:xmethods-delayed-quotes@SoapAction
```

**Data type mappings**

The Web service proxy generator maps data types between XML and PowerBuilder. All XML data types are based on a schema at `www.w3.org/1999/XMLSchema` and a schema at `www.w3.org/2001/XMLSchema`. The mappings appear in the following table:

<table>
<thead>
<tr>
<th>XML Type</th>
<th>PowerBuilder Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>byte</td>
<td>int(-128 ~ 127)</td>
</tr>
<tr>
<td>unsignedByte</td>
<td>uint(0 ~ 255)</td>
</tr>
<tr>
<td>short</td>
<td>int</td>
</tr>
<tr>
<td>unsignedShort</td>
<td>uint</td>
</tr>
<tr>
<td>int</td>
<td>long</td>
</tr>
<tr>
<td>unsignedInt</td>
<td>ulong</td>
</tr>
<tr>
<td>long</td>
<td>longlong</td>
</tr>
<tr>
<td>unsignedLong</td>
<td>longlong(0 ~ (2^63)-1)</td>
</tr>
<tr>
<td>integer(-2^63 ~ 2^63-1)</td>
<td>longlong</td>
</tr>
<tr>
<td>decimal(-999999999999999999 ~ 999999999999999999)</td>
<td>decimal</td>
</tr>
<tr>
<td>float</td>
<td>real</td>
</tr>
<tr>
<td>double</td>
<td>double</td>
</tr>
<tr>
<td>string</td>
<td>string</td>
</tr>
<tr>
<td>date</td>
<td>date</td>
</tr>
<tr>
<td>time</td>
<td>time</td>
</tr>
<tr>
<td>dateTime</td>
<td>datetime</td>
</tr>
<tr>
<td>base64</td>
<td>blob</td>
</tr>
<tr>
<td>base64Binary</td>
<td>blob</td>
</tr>
<tr>
<td>hexBinary</td>
<td>blob</td>
</tr>
</tbody>
</table>

*Note Based on a schema at schemas.smlsoap.org/soap/encoding*
Arrays of arrays

Unlike XML, PowerBuilder can only support unbounded one-dimensional arrays. If an array in a WSDL file is bounded and one-dimensional, PowerBuilder automatically converts it to an unbounded array. If an array in a WSDL file is multidimensional, the return type is invalid and cannot be used.

In function prototypes, PowerBuilder displays an array type as a PowerBuilder any type. You must declare an array of the appropriate type to hold the return value.

Connecting to a SOAP server

The SoapConnection class is used to connect to the SOAP server that hosts the Web service you want to access. It has three methods: CreateInstance(proxyObj, proxyName), CreateInstance(proxyObj, proxyName, endpoint), SetOptions(optionsString).

Example

The following script shows a connection to a Web service on a SOAP server. It sets the connection properties using a default URL stored in the proxy if no other URL is specified. It identifies the specified options: whether or not to use the SoapLog, user ID, password, or timeout. Then the script creates an instance of the SoapConnection object, invokes a proxy object for a Web service, and checks for errors.

```pascal
Long ret
Conn = create SoapConnection

Conn.SetOptions("SoapLog=-"FilePath-"")
// Set trace file to record soap interchange data, if string is "", // disable the feature
Ret = Conn.CreateInstance(proxy_obj, str_proxy_name, str_end_point)
// or Conn.CreateInstance(proxy_obj, str_proxy_name) to use default server // url
Try
   ReturnVal = proxy_obj.Method(arg,...)
Catch ( SoapException e )
   ...
end try

Destroy Conn
```
Invoking the Web service method

SoapConnection is used to create the Soap_proxy object and populate the Soap connection options that you set in SoapConnection.SetOptions(). Once a proxy object for a Web service is created, the client application can begin accessing the Web service. To invoke a Web service method, the proxy object must contain the following information:

- end point of service, obtained from a WSDL file
- name space definition used in the SOAP method call
- any struct definition, when applicable
- an instance variable for each returned structure array, since all returned arrays are any
- one or more SOAP methods and corresponding alias string

Producing a Web service

PowerBuilder provides tools for developing custom class (nonvisual) user objects and deploying them as EAServer components. You can deploy the component to an EAServer host running on Windows and UNIX operating systems. You generate a Java bean for the component using Jaguar Manager.

Using EAServer’s Web Services Toolkit, you can generate a WSDL file for the EAServer component, describing your Web service and its location. You also create a client proxy for the Web service, accessible to a client application. The client proxy uses the WSDL document that describes your Web service/EAServer component. The client application can then use the Web service. For more information see the EAServer Programmer’s Guide and the EAServer Web Services Toolkit User’s Guide.
Exception handling

Errors that occur in the execution of a method of a Web service are converted to SoapException objects and thrown to the calling script. The methods of all the classes in PBSoapClient90.dll can also throw SoapException objects when, for example, connection to the server failed, or the Web service could not be located or created.

Catching exceptions

A client application can handle communications errors in a number of ways. For example, if a client connects to a server and tries to invoke a method for an object that does not exist, the client can disconnect from the server, connect to a different server, or retry the operation. Alternatively, the client can display a message to the user and give the user the opportunity to control what happens next.

When an error occurs, if the client connects to a new server to retry the operation, it must instantiate the remote object on the new server before invoking a method of the remote object.

Unhandled exceptions

If no exception handler exists, or if the existing exception handlers do not handle the exception, the SystemError event on the Application object is executed. If the SystemError event has no script, an application error occurs and the application is terminated.
Reference information

The following section provides reference information for each of the Web service client classes and their methods.

SoapConnection

Description

The SoapConnection class is used to create a proxy object for a specific Web service and populate the Soap connection options.

It has the following functions: SetOptions and CreateInstance.

SetOptions

Description

Sets connection options for SoapConnection class. There are currently three options available: SoapLog, UserID, and Password. The string values for the options names are case insensitive.

Syntax

\texttt{conn.SetOptions (string Options)}

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoapLog</td>
<td>The file path for SoapLog. To disable the log, enter &quot;.&quot;.</td>
</tr>
<tr>
<td>UserID</td>
<td>A string value for an https connection.</td>
</tr>
<tr>
<td>Password</td>
<td>A string value for an https connection.</td>
</tr>
<tr>
<td>Timeout</td>
<td>The maximum wait time in seconds.</td>
</tr>
</tbody>
</table>

Note: Timeout is not implemented in Beta 2.

Return value

None.

Examples

In this example, the application enables the logging function and is connecting to an endpoint for which no user ID or password has been set. Each option is separated by a comma.

\texttt{Conn.SetOptions("SoapLog="sybase/powerbuilder/mysoaplog-", UserID="administrator", Password="password")}

Usage

When the UserID and Password is not specified in SetOptions(), PowerBuilder uses the default arguments in the endpoint, when they exist.

When the UserID and Password is specified in SetOptions():
• If SoapConnection.CreateInstance() specifies an endpoint, and the endpoint contains a combination of UserID and Password, PowerBuilder uses the arguments in the endpoint.

• If SoapConnection.CreateInstance() specifies an endpoint, and the endpoint does not contain a combination of UserID and Password, PowerBuilder uses the arguments specified in SetOptions().

• If the SoapConnection.CreateInstance() has the default endpoint stored in the proxy, PowerBuilder uses the combination of UserID and Password specified in SetOptions().

UserID and Password can be specified individually. If the Password is not specified, the value is taken as an empty string.

See also

CreateInstance

CreateInstance

Description
Creates a proxy instance with a default URL for a Web server, which comes from a user-supplied WSDL file. The client application must create a proxy instance before it can access a Web service.

Syntax
CreateInstance has two possible syntaxes:

```
conn.CreateInstance (ref proxy_obj, string proxy_name) throws SoapException
conn.CreateInstance (ref proxy_obj, string proxy_name, string portname)
throws SoapException
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>conn</td>
<td>The name of your SoapConnection object that establishes the connection.</td>
</tr>
<tr>
<td>proxy_object</td>
<td>The referenced name of your proxy object.</td>
</tr>
<tr>
<td>proxy_name</td>
<td>The name of the proxy, based on the port name from a URL in the WSDL file stored in the proxy.</td>
</tr>
<tr>
<td>portname</td>
<td>The port name from a URL not stored in the proxy.</td>
</tr>
</tbody>
</table>

**Note** If a *portname* is not specified, PowerBuilder uses the default in the proxy.

Return value

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Successful</td>
</tr>
</tbody>
</table>
Examples

In this example, the client application creates a proxy to access the Web services at: "http://my.server/soap/myport".

```
Ret = Conn.CreateInstance(myproxy, syb_myport, "http://my.server/soap/myport")
```

Usage

You can select which SOAP methods you want to call from each available Web service. For each port you specify, a client proxy is created.

See also

SetOptions

---

SoapException

Description

The SoapException class is a PBNI class that inherits from PowerBuilder RuntimeError class. When an exception occurs in a Web service method call, it is converted into a SoapException and thrown. The methods of the classes in PBSSoapClient90.dll can also throw SoapException.

Properties

<table>
<thead>
<tr>
<th>Exception property</th>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassDefinition</td>
<td>PowerObject</td>
<td>An object of type PowerObject containing information about the class definition of the object or control.</td>
</tr>
<tr>
<td>Text</td>
<td>String</td>
<td>Contains the text of the error message.</td>
</tr>
</tbody>
</table>

Usage

The following tables define the usage of the SoapException class.

<table>
<thead>
<tr>
<th>Exception event</th>
<th>Occurs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor</td>
<td>Immediately before the exception is thrown.</td>
</tr>
<tr>
<td>Destructor</td>
<td>Immediately after the exception is thrown.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Exception function</th>
<th>Data type returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ClassName</td>
<td>String</td>
<td>Returns the name assigned to the object.</td>
</tr>
<tr>
<td>GetContextService</td>
<td>Integer</td>
<td>Creates a reference to a context-specific instance of the specified service.</td>
</tr>
</tbody>
</table>
The following example demonstrates using the SoapException class. The ServiceProxy fails to be invoked and returns the error message.

```java
string s1, s2
s1 = "abcd"

try
    conn = create SoapConnection
    ret = conn.CreateInstance(proxy, "ServiceProxy")
    if (ret <> 0)
        MessageBox("Fail", "Can't create proxy ServiceProxy")
        return
    end if

    s2 = proxy.EchoString(s1)
catch (SoapException e1)
    ...
catch (PBXRuntimeError e2)
    ...
catch (RuntimeError e3)
    ...
end try
See also                RuntimeException
```

### SoapException

<table>
<thead>
<tr>
<th>Exception function</th>
<th>Data type returned</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetMessage</td>
<td>String</td>
<td>Returns the error message from objects of type Throwable.</td>
</tr>
<tr>
<td>GetParent</td>
<td>PowerObject</td>
<td>Returns a reference to the name of the parent object.</td>
</tr>
<tr>
<td>PostEvent</td>
<td>Boolean</td>
<td>Adds an event to the end of the message queue for the object.</td>
</tr>
<tr>
<td>SetMessage</td>
<td>—</td>
<td>Sets an error message for an object of type Throwable.</td>
</tr>
<tr>
<td>TriggerEvent</td>
<td>Integer</td>
<td>Triggers a specific event in the object and executes the script for the event.</td>
</tr>
<tr>
<td>TypeOf</td>
<td>Object</td>
<td>Returns the type of the object.</td>
</tr>
</tbody>
</table>

The following example demonstrates using the SoapException class. The ServiceProxy fails to be invoked and returns the error message.
PowerBuilder Document Object Model

This section describes the Power Builder Document Object Model (PBDOM). It includes the following topics:

- About PBDOM
- Using PBDOM
- PBDOM methods

About PBDOM

Note  Internal testing of PBDOM has not been completed, and PBDOM is not recommended for beta 3 users. The documentation for this feature is incomplete.

PBDOM is the PowerBuilder implementation of the Document Object Model (DOM), a programming interface defining the means by which XML documents can be accessed and manipulated. Although PBDOM is not an implementation of the World Wide Web Consortium (W3C) DOM API, the PBDOM PowerBuilder API can be used for reading, writing, and manipulating standard-format XML from within PowerScript code. PBDOM portrays an XML document as a collection of interconnected objects and provides intuitive methods indicating the use and functionality of each object.
PBDOM is strictly an extension of PowerScript. However, the PowerBuilder graphical user interface (GUI) will display PBDOM objects, as shown in the following illustration of a PowerBuilder system tree.

These objects are visible in the PowerBuilder GUI because PBDOM is packaged as a PowerBuilder Native Interface (PBNI) extension DLL.

**Node trees**

PBDOM interacts with XML documents according to a tree-view model consisting of parent and child nodes. A document element represents the top-level node of a standalone XML document. This element has one or many child nodes that represent the branches of the tree. Elements in the node tree are accessible through the appropriate PowerScript class methods.

**XML parser**

The PBDOM XML parser is used to load and parse an XML document, and also to generate XML based on user-specified DOM nodes.
The PBDOM provides all the necessary methods for the PowerBuilder developer to traverse the node tree, access the nodes and attribute values (if any), insert and delete nodes, and serialize the node tree back to XML.

**PBDOM objects**

The following table shows the cognate W3C DOM and JDOM objects for each PBDOM object. Note that while these W3C DOM and JDOM objects correspond to PBDOM objects, they are not equivalent to the PBDOM objects.

<table>
<thead>
<tr>
<th>PBDOM</th>
<th>W3C DOM</th>
<th>JDOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBDOM_ATTRIBUTE</td>
<td>ATTRIBUTE_NODE</td>
<td>Attribute</td>
</tr>
<tr>
<td>PBDOM_BUILDER</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>PBDOMCDATA</td>
<td>CDATA_SECTION_NODE</td>
<td>CDATA</td>
</tr>
<tr>
<td>PBDOM_COMMENT</td>
<td>COMMENT_NODE</td>
<td>Comment</td>
</tr>
<tr>
<td>PBDOM_DOCUMENT</td>
<td>DOCUMENT_NODE</td>
<td>Document</td>
</tr>
<tr>
<td>PBDOM_DOCTYPE</td>
<td>DOCUMENT_TYPE_NODE</td>
<td>DocType</td>
</tr>
<tr>
<td>PBDOM_ELEMENT</td>
<td>ELEMENT_NODE</td>
<td>Element</td>
</tr>
<tr>
<td>PBDOM_NAMESPACE</td>
<td>ENTITY_NODE</td>
<td>Namespace</td>
</tr>
<tr>
<td>PBDOM_PROCESSINGINSTURCTION</td>
<td>PROCESSING_INSTRUCTION_NODE</td>
<td>Processinginstructio</td>
</tr>
<tr>
<td>PBDOM_TEXT</td>
<td>TEXT_NODE</td>
<td>N/A</td>
</tr>
</tbody>
</table>
About PBDOM

Object hierarchy

The W3C DOM and JDOM object hierarchies also differ from the PBDOM object hierarchy, which is shown in the following illustration.

For information on the W3C DOM and JDOM objects and hierarchies, refer to their respective specifications.

Different node types are represented in PBDOM by the following Non-visual Object (NVO) classes:

- PBDOM_ATTRIBUTE
- PBDOM_CDATA
- PBDOM_CHARACTERDATA
- PBDOM_COMMENT
- PBDOM_DOCTYPE
- PBDOM_ELEMENT
- PBDOM_PROCESSINGINSTRUCTION
- PBDOM_TEXT
- PBDOM_NAMESPACE
- PBDOM_DOCUMENT
- PBDOM_BUILDER

Methods from these classes, which are derived from PBDOM_OBJECT, are used in PowerScript to access objects in a PBDOM node tree. The PBDOM_NAMESPACE and PBDOM_BUILDER classes do not represent DOM nodes but are required PBDOM classes.

The methods for all PBDOM classes are described at the end of this chapter.
The **PBDM_OBJECT** class abstractly represents a node in an XML node tree and serves as the base class for all the PBDM classes. The DOM cognate of PBDM is the Node object. PBDM_OBJECT contains all the basic functionalities for derived classes. Therefore, a node can be an element node, a document node, or any of the node types listed above that derive from PBDM_OBJECT.

### PBDM_OBJECT inheritance

The PBDM_OBJECT class is similar to a virtual class in C++ in that it is not expected to be directly instantiated and used. For example, although a PBDM_OBJECT may be created using the PowerScript CREATE keyword, its methods cannot be used directly:

```powerbuilder
PBDM_OBJECT pbdom_obj
pbdom_obj = CREATE PBDM_OBJECT
pbdom_obj.SetName ("VIRTUAL_PBDM_OBJ")
```

The third line of PowerScript above throws an exception because the code attempts to directly access the SetName method for the base class PBDM_OBJECT. A similar implementation is valid, however, when the SetName method is accessed from a derived class, such as PBDM_ELEMENT:

```powerbuilder
PBDM_OBJECT pbdom_obj
pbdom_obj = CREATE PBDM_ELEMENT
pbdom_obj.SetName ("VIRTUAL_PBDM_OBJ")
```

### Using base PBDM_OBJECT as a placeholder

The PBDM_OBJECT class can be used as a placeholder for the object of a derivative class, as in the following example:

```powerbuilder
PBDM_DOCUMENT pbdom_doc
PBDM_OBJECT pbdom_obj
pbdom_doc = CREATE PBDM_DOCUMENT
pbdom_doc.NewDocument ("", "Root_Element_From_Doc_1", "", "")
pbdom_obj = pbdom_doc.GetRootElement()
pbdom_obj.SetName ("Root_Element_From_Doc_1_Now_Changed")
```

The instantiated PBDM_OBJECT pbdom_obj is assigned to a PBDM_DOCUMENT object, which holds the return value of the GetRootElement method. Here, pbdom_obj holds a reference to a PBDM_ELEMENT and can be legally operated on like any object of a class derived from PBDM_OBJECT.

### Stand-alone objects

A PBDM_OBJECT may be created as a self-contained object independent of any document or parent PBDM_OBJECT. Such a PBDM_OBJECT is known as a stand-alone object, illustrated in the following example:
About PBDOM

```
PBDOM_ELEMENT pbdom_elem_1
pbdom_elem_1 = Create PBDOM_ELEMENT
pbdom_elem_1.SetName ("pbdom_elem_1")
```

Here, pbdom_elem_1 is instantiated in the derived class PBDOM_ELEMENT using the Create keyword. The SetName method can then be invoked from the pbdom_elem_1 object, which is a stand-alone object not contained within any document.

While stand-alone objects may perform any legal PBDOM operations, stand-alone status bestows no special privileges or disadvantages for a PBDOM_OBJECT.

A PBDOM_OBJECT can be assigned a parent by appending it to another stand-alone PBDOM_OBJECT, as in the following example:

```
PBDOM_ELEMENT pbdom_elem_1
PBDOM_ELEMENT pbdom_elem_2

pbdom_elem_1 = Create PBDOM_ELEMENT
pbdom_elem_2 = Create PBDOM_ELEMENT

pbdom_elem_1.SetName ("pbdom_elem_1")
pbdom_elem_2.SetName ("pbdom_elem_2")
pbdom_elem_1.AddContent(pbdom_elem_2)
```

Here, two PBDOM_ELEMENT objects, pbdom_elem_1 and pbdom_elem_2, are instantiated. The pbdom_elem_2 object is appended as a child object of pbdom_elem_1 using the AddContent method.

In the example above, both pbdom_elem_1 and pbdom_elem_2 are not owned by any document, and the pbdom_elem_1 object is still stand-alone. If pbdom_elem_1 were assigned to a parent PBDOM_OBJECT owned by a document, pbdom_elem_1 would cease to be a stand-alone object.

**PBDOM_DOCUMENT**

The PBDOM_DOCUMENT class derives from PBDOM_OBJECT and represents an XML DOM document. The PBDOM_DOCUMENT methods allow access to the root element, processing instructions, and other document-level information.
The PBDOM_DOCTYPE class represents the document type declaration object of an XML DOM document. The PBDOM_DOCUMENT methods allow access to the root element name, the internal subset, and the system and public IDs.

The PBDOM_ELEMENT represents an XML element modeled in PowerScript. The PBDOM_ELEMENT methods allow access to element attributes, children, and text.

The PBDOM_ATTRIBUTE class represents an XML attribute modeled in PowerScript. The PBDOM_ATTRIBUTE methods allow access to element attributes and namespace information.

The PBDOM_NAMESPACE class represents an XML namespace modeled in PowerScript. The PBDOM_ATTRIBUTE class defines the mapping between a prefix string and a namespace URI. The PBDOM_ATTRIBUTE class is most useful when combined with PBDOM_ELEMENT and PBDOM_ATTRIBUTE objects.

The PBDOM_CHARACTERDATA class derives from PBDOM_OBJECT and represents character-based content (not markup) within an XML document. The PBDOM_CHARACTERDATA class extends PBDOM_OBJECT with methods specifically designed for manipulating character data.

The PBDOM_TEXT class derives from PBDOM_CHARACTERDATA and represents a DOM text node in an XML document. The PBDOM_TEXT class extends PBDOM_CHARACTERDATA with methods designed specifically for manipulating DOM text nodes.
Using PBDOM

PBDOM_CDATA

The PBDOM_CDATA class derives from PBDOM_TEXT and represents an XML DOM CDATA section.

PBDOM_COMMENT

The PBDOM_CDATA class derives from PBDOM_CHARACTERDATA and represents a DOM Comment node in an XML document. PBDOM_CDATA extends PBDOM_CHARACTERDATA with methods designed specifically for manipulating DOM Comment nodes.

PBDOM_PROCESSINGINSTRUCTION

The PBDOM_PROCESSINGINSTRUCTION class represents an XML processing instruction. The PBDOM_PROCESSINGINSTRUCTION methods allow access to the processing instruction target and its data. The data can be accessed as a string or, where appropriate, as name/value pairs.

PBDOM_BUILDER

The PBDOM_BUILDER class is used to create PBDOM_DOCUMENT objects from an input source, such as a string or a DataStore. PBDOM_BUILDER is not derived from PBDOM_OBJECT, and there are no DOM objects to which the PBDOM_BUILDER can map.

Using PBDOM

This section provides examples for accomplishing basic tasks using PowerScript and PBDOM classes and methods.

Loading an XML file

The following example loads the file sample.xml for reading:

```powerbuilder
integer xml_FileNum
string xml_Input
xml_FileNum = FileOpen("sample.xml", StreamMode!) FileRead(xml_FileNum, xml_Input)
```

116
The integer variable xml_FileNum is assigned a file number for sample.xml. The file contents are streamed into the string variable xml_Input.

### Loading pure XML

XML can also be loaded directly into a string variable, as in the following example:

```plaintext
string Xml_doc
Xml_doc = "<?xml version="1.0" ?>"
Xml_doc = Xml_doc + "<WHITEPAPER>"
Xml_doc = Xml_doc + "<TITLE>Document Title</TITLE>"
Xml_doc = Xml_doc + "<AUTHOR>Author Name</AUTHOR>"
Xml_doc = Xml_doc + "<PARAGRAPH>Document text.</PARAGRAPH>"
Xml_doc = Xml_doc + "</WHITEPAPER>"
```

### Creating an XML file

The following example uses an XML string, the PBDOM_OBJECT and its descendant classes PBDOM_BUILDER and PBDOM_DOCUMENT, and various methods to create a PBDOM_DOCUMENT. The XML string used is that from the previous example, “Loading pure XML”.

First the objects are declared.

```plaintext
PBDOM_BUILDER pbdom_builder_new
PBDOM_DOCUMENT pbdom_doc
```

The objects are then instantiated using the constructor and the PBDOM_BUILDER Build method.

```plaintext
pbdom_builder_new = Create PBDOM_Builder
pbdom_doc = pbdom_builder_new.Build (Xml_doc)
```

### Using BuildFromFile()

An XML file may be created using the BuildFromFile() method and a string containing the path to a file from which to create a PBDOM_DOCUMENT:

```plaintext
PBDOM_BUILDER pbdombuilder_new
PBDOM_DOCUMENT pbdom_doc
pbdombuilder_new = Create PBDOM_Builder
pbdom_doc = pbdombuilder_new.BuildFromFile ("c:\pbdom_doc_1.xml")
```
Using PBDOM

For more information on these and other subclasses and methods, see “PBDOM methods”.

Creating an XML document from scratch

An XML document can be created from within a script using the appropriate PBDOM_OBJECT subclasses and methods. The following example code uses the PBDOM_ELEMENT and PBDOM_DOCUMENT classes and some of their methods to create a simple XML document.

First, the objects are declared and instantiated:

```pascal
PBDOM_ELEMENT pbdom_elem_1
PBDOM_ELEMENT pbdom_elem_2
PBDOM_ELEMENT pbdom_elem_3
PBDOM_ELEMENT pbdom_elem_root
PBDOM_DOCUMENT pbdom_doc1

pbdom_elem_1 = Create PBDOM_ELEMENT
pbdom_elem_2 = Create PBDOM_ELEMENT
pbdom_elem_3 = Create PBDOM_ELEMENT
```

The instantiated objects are assigned names. Note that the PBDOM_DOCUMENT object pbdom_doc1 is not named.

```pascal
pbdom_elem_1.SetName ("pbdom Elem_1")
pbdom_elem_2.SetName ("pbdom Elem_2")
pbdom_elem_3.SetName ("pbdom Elem_3")
```

The objects are arranged into a node tree using the AddContent method. The AddContent method adds the referenced object as a child node under the object from which AddContent is invoked.

```pascal
pbdom_elem_1.AddContent (pbdom_elem_2)
pbdom_elem_2.AddContent (pbdom_elem_3)
```

A new XML document is created using the NewDocument method. The parameter value supplied to the NewDocument method becomes the name of the root element. This name is then accessed from the PBDOM_DOCUMENT object pbdom_doc1 and assigned to the PBDOM_ELEMENT object pbdom_elem_root using the GetRootElement method.

```pascal
pbdom_doc1.NewDocument ("Root Element From Doc_1")
pbdom_elem_root = pbdom_doc1.GetRootElement()
```
The ELEMENT object pbdom_elem_1 and all its child nodes are then placed in the new XML document node tree under the root element using the AddContent method. Note that as the ancestor node pbdom_elem_1 is placed in the node tree, all its child nodes move as well.

\[
\text{pbdom}\_\text{elem}\_\text{root}.\text{AddContent} \left( \text{ref pbdom}\_\text{elem}\_1 \right)
\]

The XML document created looks like this:

\[
\text{<!DOCTYPE Root\_Element\_From\_Doc\_1>}
\text{<Root\_Element\_From\_Doc\_1>}
\text{<pbdom\_elem\_1>}
\text{<pbdom\_elem\_2>}
\text{<pbdom\_elem\_3>}
\text{<pbdom\_elem\_2>}
\text{<pbdom\_elem\_1>}
\text{</Root\_Element\_From\_Doc\_1>}
\]

For more information on these and other subclasses and methods, see “PBDOM methods”.

**Accessing node data**

An XML document can be read by accessing the elements of its node tree using the appropriate PBDOM_OBJECT subclasses and methods. The following example code uses an array, the PBDOM_OBJECT and its descendant class PBDOM_DOCUMENT, and the GetContent and GetRootElement methods of the PBDOM_DOCUMENT class to access node data on an XML document.

A PBDOM_DOCUMENT object named pbdom_doc contains the following XML document:

\[
<\text{Root}>
<\text{Element\_1}>
<\text{Element\_1\_1>}</\text{Element\_1\_1}>
<\text{Element\_1\_2}</\text{Element\_1\_2}>
<\text{Element\_1\_3}</\text{Element\_1\_3}>
</\text{Element\_1}>
<\text{Element\_2}</\text{Element\_2}>
<\text{Element\_3}</\text{Element\_3}>
</\text{Root}>
\]

An array is declared to hold the elements returned from the GetContent method, invoked to read the PBDOM_DOCUMENT object named pbdom_doc.
Using PBDOM

```
PBDOM_OBJECT pbdom_obj_array[]
...
pbdom_doc.GetContent(ref pbdom_obj_array)
```

The `pbdom_obj_array` array now contains one value representing the root element of `pbdom_doc`: `<Root>`. To access the other nodes in `pbdom_doc`, the `GetRootElement` method is used with the `GetContent` method.

```
pbdom_doc.GetRootElement().GetContent(ref pbdom_obj_array)
```

The `pbdom_obj_array` array now contains three values corresponding to the three child nodes of the root element of `pbdom_doc`: `<Element_1>`, `<Element_2>`, and `<Element_3>`.

PBDOM provides other methods for accessing data, including `InsertContent`, `AddContent`, `RemoveContent`, and `SetContent`. For more information on these and other methods, see “PBDOM methods”.

Changing node content with arrays

The `AddContent` method can also be used to change node content:

```
pbdom_obj_array[3].AddContent ("This is Element 3."")
```

This line of code will change the node tree as follows:

```
<Root>
  <Element_1>
    <Element_1_1></Element_1_1>
    <Element_1_2></Element_1_2>
    <Element_1_3></Element_1_3>
  </Element_1>
  <Element_2></Element_2>
  <Element_3> This is Element 3. </Element_3>
</Root>
```

**Note** PBDOM_OBJECT references returned in an array (as by a method that returns such an array, like the `GetContent` method of the PBDOM_DOCUMENT class) refer to instantiated PBDOM objects. Modifications made to any of these objects through its respective array item are permanent and are reflected on any other arrays that hold the same object reference.
Manipulating the node-tree hierarchy

An XML node tree can be restructured by rearranging its nodes. One means of manipulating nodes involves detaching a child node from its parent node. This can be accomplished with the Detach method, as in the following example.

The root element of a PBDOM_DOCUMENT object named pbdom_doc is obtained using the GetRootElement method.

```
pbdom_obj = pbdom_doc.GetRootElement()
```

The root element is detached from the PBDOM_DOCUMENT object, which is the parent node of the root element.

```
pbdom_obj.Detach()
```

PBDOM also provides the SetParentObject method to an object a child of another object. For more information on these and other methods, see “PBDOM methods”.

Checking for parent node

The GetParentObject method can be used to determine if an element has a parent object, as in the following example.

```
pbdom_parent_obj = pbdom_obj.GetParentObject()
if (IsNull(pbdom_parent_obj)) then
    MessageBox ("IsNull", "Root Element has no Parent")
end if
```

If the object from which GetParentObject has no parent object, the function returns NULL.

PBDOM provides similar methods that return information about an element’s place in an XML node tree. These methods include HasChildren, which returns a boolean indicating whether an object has children objects, and IsAncestorOf, which indicates whether an object is the ancestor of another object.

For more information on these and other methods, see “PBDOM methods”.

PBDOM methods

The remainder of this chapter describes the PBDOM NVO classes and their methods.
**PBDM_OBJECT**

**Description**

A PBDM_OBJECT serves as the base class for all the PBDM classes. It contains all the basic functionalities for all the derived classes. The derived classes of a PBDM_OBJECT each inherit the base methods of a PBDM_OBJECT, and additionally contain their own specialized functionalities.

**Note** A PBDM_OBJECT cannot be instantiated directly. See “Virtual Nature Of a PBDM_OBJECT” for details. BRETT--ADD CROSS REF

PBDM_OBJECT has the following functions:

- GetName
- GetText
- GetContent
- AddContent
- InsertContent
- RemoveContent
- SetContent
- SetName
- Clone
- Equals
- GetObjectClass()
- GetObjectClassString()
- GetDocument()
- Detach()
- GetParentObject()
- HasChildren()
- GetTextNormalize()
- GetTextTrim()
- IsAncestorOf()
- SetParentObject()
**GetName**

**Description**

The GetName() method allows you to obtain the name of the current PBDOM_OBJECT. This returned string depends on the type of DOM Object that is contained within a PBDOM_OBJECT.

**Syntax**

```
pbdom_object_name.GetName()
```

**Argument**

| pbdom_object_name | The name of your PBDOM_OBJECT. |

**Return value**

The following table lists the return values based on the type of DOM Object contained within PBDOM_OBJECT:

<table>
<thead>
<tr>
<th>DOM Object Type</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBDOM_DOCTYPE</td>
<td>&quot;#document&quot;</td>
</tr>
<tr>
<td>PBDOM_ELEMENT</td>
<td>The local tag name of the element, without any namespace prefixes. For example, if the element is: <code>&lt;abc&gt;Value&lt;/abc&gt;</code> then the string returned from GetName() is &quot;abc&quot;. Also, if the tag name of the element contains a namespace prefix, the prefix is not included in the returned string. For example, if the element is: <code>&lt;Tower:CD xmlns:Tower=&quot;http://www.Tower_Records.com&quot;/&gt;</code> then GetName() returns the string &quot;CD&quot;.</td>
</tr>
<tr>
<td>PBDOM_ATTRIBUTE</td>
<td>The local name of the attribute itself, without a namespace. For example, if the element with the attribute is: <code>&lt;abc ATTRIBUTE_1=&quot;My Attribute&quot;&gt;</code> then GetName() returns &quot;ATTRIBUTE_1&quot;. If the name of the attribute contains a namespace prefix, then the prefix is not included in the returned string. For example, if the element with an attribute is: <code>&lt;Tower:CD xmlns:Tower=&quot;http://www.Tower_Records.com&quot; Tower:Type=&quot;Jazz&quot;/&gt;</code> then GetName() returns the string &quot;Type&quot;.</td>
</tr>
<tr>
<td>PBDOM_CDATA</td>
<td>&quot;#cdata-section&quot;</td>
</tr>
</tbody>
</table>
Examples

TBD

Usage

A PBDOM_OBJECT cannot be instantiated directly.

See also

TBD

GetText

Description

The GetText() method allows you to obtain the text data that is contained within the current PBDOM_OBJECT.

Syntax

\[ pbdom\_object\_name.GetText() \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( pbdom_object_name )</td>
<td>The name of your PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

Return value

String

The following table lists the return values based on the type of DOM Object contained within a PBDOM_OBJECT:

<table>
<thead>
<tr>
<th>DOM Object Type</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBDOM_COMMENT</td>
<td>&quot;#comment&quot;</td>
</tr>
<tr>
<td>PBDOM_DOCTYPE</td>
<td>The name that was given to the doctype object itself. For example, if the DOCTYPE declaration is: &lt;!DOCTYPE d_grid_object &gt; then GetName() returns &quot;d_grid_object&quot;.</td>
</tr>
<tr>
<td>PBDOM_PROCESSINGINSTRUCTION</td>
<td>The name that was given to the processing instruction itself. For example, if the processing instruction definition is: &lt;?works document=&quot;hello.doc&quot; data=&quot;hello.wks&quot; ?&gt; then GetName() returns &quot;works&quot;.</td>
</tr>
<tr>
<td>PBDOM_TEXT</td>
<td>&quot;#text&quot;</td>
</tr>
</tbody>
</table>
### PowerBuilder Document Object Model

<table>
<thead>
<tr>
<th>DOM Object Type</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBDOM_ELEMENT</td>
<td>The concatenation of the text values of all the TEXT nodes contained within the PBDOM_ELEMENT. If the PBDOM_ELEMENT definition is:</td>
</tr>
<tr>
<td></td>
<td><code>&lt;abc&gt;Root Element Data&lt;data&gt;ABC Data &lt;/data&gt; now with extra info &lt;/abc&gt;</code> then GetText() returns &quot;Root Element Data now with extra info &quot;.</td>
</tr>
<tr>
<td></td>
<td><strong>Note</strong> There are extra spaces between the word “Data” and “now” and again after the word “info”. They are there because they originally exist in the text.</td>
</tr>
<tr>
<td></td>
<td>If the PBDOM_ELEMENT definition is: <code>&lt;abc&gt;Root Element Data&lt;/abc&gt;</code> then GetText() returns &quot;Root Element Data&quot;.</td>
</tr>
<tr>
<td>PBDOM_ATTRIBUTE</td>
<td>The text data contained within the PBDOM_ATTRIBUTE object. If the element with an attribute is <code>&lt;abc ATTRIBUTE_1=&quot;My Attribute&quot;&gt;</code> then GetText()</td>
</tr>
<tr>
<td></td>
<td>returns &quot;My Attribute&quot;.</td>
</tr>
<tr>
<td>PBDOM_TEXT</td>
<td>The text data contained within the PBDOM_TEXT object itself. If we have the following element:</td>
</tr>
<tr>
<td></td>
<td><code>&lt;abc&gt;MY TEXT&lt;/abc&gt;</code> If we have a PBDOM_TEXT object to represent the TEXT NODE “MY TEXT”, then calling GetText() on the PBDOM_TEXT will return the</td>
</tr>
<tr>
<td></td>
<td>string &quot;MY TEXT&quot;.</td>
</tr>
<tr>
<td>PBDOMCDATA</td>
<td>The string data that is contained within the CDATA section itself. For example, if we have the following CDATA:</td>
</tr>
<tr>
<td></td>
<td><code>&lt;![CDATA[ They’re saying “x &lt; y” &amp; that “z &gt; y” so I guess that means that z &gt; x ]]&gt;</code> If we have a PBDOMCDATA to represent the above CDATA section,</td>
</tr>
<tr>
<td></td>
<td>then calling GetText() on it will return the string: &quot; They’re saying “x &lt; y” &amp; that “z &gt; y” so I guess that means that z &gt; x &quot;.</td>
</tr>
</tbody>
</table>
### Examples

TBD

### Usage

This method only returns meaningful data if the PBDOM_OBJECT is of a type that can contain text nodes, CDATA sections, or basic text. These include:

- PBDOM_ELEMENT
- PBDOM_ATTRIBUTE
- PBDOM_TEXT
- PBDOM_CDATA
- PBDOM_COMMENT

The PBDOM_TEXT, PBDOM_CDATA, and PBDOM_COMMENT objects are special cases which cause the GetText() method to return the text data that is intrinsically contained within the objects. A PBDOM_TEXT object is basically a DOM text node and consequently it does not hold any child text nodes. A PBDOM_CDATA object represents a DOM CDATA object and so it does not hold any child DOM nodes. The same rule is true for a PBDOM_COMMENT object.

### See also

TBD

---

<table>
<thead>
<tr>
<th>DOM Object Type</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBDOM_COMMENT</td>
<td>The string data that is contained within the COMMENT itself. For example, if we have the following COMMENT:</td>
</tr>
<tr>
<td></td>
<td>&lt;![CDATA[&lt;!—This is some comment. --&gt;]]</td>
</tr>
<tr>
<td></td>
<td>If we a PBDOM_COMMENT to represent the above COMMENT, then calling GetText() on it will return the string:</td>
</tr>
<tr>
<td></td>
<td>“ This is some comment. ”.</td>
</tr>
<tr>
<td></td>
<td>The string data that is contained within the COMMENT itself. For example, if we have the following COMMENT:</td>
</tr>
<tr>
<td></td>
<td>&lt;![CDATA[&lt;!—This is some comment. --&gt;]]</td>
</tr>
<tr>
<td></td>
<td>If we a PBDOM_COMMENT to represent the above COMMENT, then calling GetText() on it will return the string:</td>
</tr>
<tr>
<td></td>
<td>“ This is some comment. ”.</td>
</tr>
</tbody>
</table>
GetContent

Description
The GetContent() method allows you to obtain an array of PBDOM_OBJECTs each of which is a child node of the called PBDOM_OBJECT.

Syntax

```
pbdom_object_name.GetContent(pbdom_object_array)
```

Argument | Description
---|---
`pbdom_object_name` | The name of your PBDOM_OBJECT.
`pbdom_object_array` | The referenced name of an array of PBDOM_OBJECTs that receives PBDOM_OBJECTs.

Return value
Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>Successful</td>
</tr>
<tr>
<td>False</td>
<td>Unsuccessful</td>
</tr>
</tbody>
</table>

Examples
TBD

Usage
The returned array is passed by reference, with items in the same order as they appear in the PBDOM_OBJECT. Any changes to any item of the array will affect the actual item to which it refers.

See also
TBD

AddContent

Description
The AddContent() method allows you to add a new PBDOM_OBJECT into the current PBDOM_OBJECT.

Syntax

```
pbdom_object_name.AddContent(pbdom_object_ref)
```

Argument | Description
---|---
`pbdom_object_name` | The name of your PBDOM_OBJECT.
`pbdom_object_ref` | The referenced name of a PBDOM_OBJECT you want to add.

Return value
PBDOM_OBJECT

The return value is the newly modified PBDOM_OBJECT.

Examples
TBD
**PBDOM_OBJECT**

**Usage**
When a new PBDOM_OBJECT is added to the current one, the new PBDOM_OBJECT becomes a child node of the current PBDOM_OBJECT.

**See also**
TBD

**InsertContent**

**Description**
The InsertContent() method allows you to insert a new PBDOM_OBJECT into the current PBDOM_OBJECT.

**Syntax**
```
insert_object_name.InsertContent(insert_object_new, insert_object_ref)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert_object_name</td>
<td>The name of your PBDOM_OBJECT.</td>
</tr>
<tr>
<td>insert_object_new</td>
<td>The referenced name of a PBDOM_OBJECT you want to insert.</td>
</tr>
<tr>
<td>insert_object_ref</td>
<td>The referenced name of the PBDOM_OBJECT in front of which you want to insert the new PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

**Return value**
PBDOM_OBJECT

The return value is the newly modified PBDOM_OBJECT.

**Examples**
TBD

**Usage**
When a new PBDOM_OBJECT is inserted into the current PBDOM_OBJECT, the new PBDOM_OBJECT becomes a child node of the current PBDOM_OBJECT. Also, the new PBDOM_OBJECT is to be positioned specifically before another PBDOM_OBJECT, specified using second parameter.

If the second PBDOM_OBJECT is specified as NULL, then the new PBDOM_OBJECT is to be inserted at the end of the list of children of the current PBDOM_OBJECT.

**Note** Some derived classes may implement different behavior. Some classes, such as PBDOM.TEXT, have trivial implementations that do nothing with the input parameters and will simply return the instance objects themselves, unmodified in any way.

**See also**
TBD
RemoveContent
Description
The RemoveContent() method allows you to remove a child PBDOM_OBJECT from the current PBDOM_OBJECT.

Syntax
pbdom_object_name.RemoveContent(pbdom_object_ref)

Argument | Description
---|---
 pbdom_object_name | The name of your PBDOM_OBJECT.
 pbdom_object_ref | The referenced name of a PBDOM_OBJECT you want to remove.

Return value
Boolean

Value | Description
---|---
 True | The content was removed.
 False | The content was not removed.

Examples
TBD

Usage
When a new PBDOM_OBJECT is removed from the current one, all children under the removed PBDOM_OBJECT are also removed.

See also
TBD

SetContent
Description
The SetContent() method sets the entire content of the PBDOM_OBJECT.

Syntax
pbdom_object_name.SetContent(pbdom_object_array)

Argument | Description
---|---
 pbdom_object_name | The name of your PBDOM object.
 pbdom_object_array | An array of PBDOM_OBJECTs set as the contents of the PBDOM_OBJECT.

Return value
PBDOM_OBJECT

Examples
TBD
The supplied array contains PBDOM_OBJECTs that are legal for the particular derived PBDOM_OBJECT that is associated with this PBDOM_DOCUMENT only accepts an array that contains PBDOM_ELEMENT, PBDOM_COMMENT or PBDOM_PROCESSINGINSTRUCTION objects. The array is restricted to contain only one PBDOM_ELEMENT object that it sets as its root element.

If illegal objects are included in the array, exceptions (specific to the particular derived PBDOM_OBJECT) will be thrown. For more details, please refer to the SetContent() method of the derived PBDOM_OBJECTs.

In the event of an exception, the original contents of this PBDOM_OBJECT will be unchanged and the PBDOM_OBJECTs contained in the supplied array will be unaltered.

See also TBD

**Set Name**

**Description**
The SetName() method sets the name of the PBDOM_OBJECT.

**Syntax**

```cpp
pbdom_object_name.SetName(strName)
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_object_name</td>
<td>The name of your PBDOM_OBJECT.</td>
</tr>
<tr>
<td>strName</td>
<td>The new name you want to set for PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

**Return value**

Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The PBDOM_OBJECT's name was changed.</td>
</tr>
<tr>
<td>False</td>
<td>The PBDOM_OBJECT's name was not changed.</td>
</tr>
</tbody>
</table>

**Examples**

TBD

**Usage**

This name refers to the name of the particular derived PBDOM_OBJECT to which this PBDOM_OBJECT refers. Certain types of PBDOM_OBJECT do not have any name associated with them. See the description of GetName.

For example, PBDOM_DOCUMENT does not have any name and so calling PBDOM_DOCUMENT::SetName() returns false.

See also TBD
Clone
Description
The `Clone()` method creates a deep clone of the PBDOM_OBJECT—the original object and all child objects are cloned.

Syntax
```
pbdom_object_name.Clone()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_object_name</td>
<td>The name of your PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

Return value

PBDOM_OBJECT

The return value is the deep clone of PBDOM_OBJECT.

Examples

TBD

Usage

This function clones the original object and all of the child PBDOM_OBJECTs.

See also

TBD

Equals
Description
The `Equals()` method tests for the equality of a referenced PBDOM_OBJECT.

Syntax
```
pbdom_object_name.Equals(pbdom_object_ref)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_object_name</td>
<td>The name of your PBDOM_OBJECT.</td>
</tr>
<tr>
<td>pbdom_object_ref</td>
<td>A reference to a PBDOM_OBJECT to test for equality with the current PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

Return value

Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The current PBDOM_OBJECT is equivalent to the referenced PBDOM_OBJECT.</td>
</tr>
<tr>
<td>False</td>
<td>The current PBDOM_OBJECT is not equivalent to the referenced PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

Examples

TBD

See also

TBD
**GetObjectClass()**

**Description**

The `GetObjectClass()` method returns a long integer code that indicates the class of this `PBDOM_OBJECT`.

**Syntax**

```
pbdom_object_name.GetObjectClass()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_object_name</code></td>
<td>The name of your <code>PBDOM_OBJECT</code>.</td>
</tr>
</tbody>
</table>

**Return value**

Long

The `GetObjectClass()` method returns a long integer code that indicates the class of the current `PBDOM_OBJECT`.

**Examples**

TBD

**Usage**

This function returns the following possible values:

<table>
<thead>
<tr>
<th>Class</th>
<th>Long integer value</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNKNOWN (indicates an error)</td>
<td>0</td>
</tr>
<tr>
<td><code>PBDOM_OBJECT</code> (the base class)</td>
<td>1</td>
</tr>
<tr>
<td><code>PBDOM_DOCUMENT</code></td>
<td>2</td>
</tr>
<tr>
<td><code>PBDOM_ELEMENT</code></td>
<td>3</td>
</tr>
<tr>
<td><code>PBDOMDOCTYPE</code></td>
<td>4</td>
</tr>
<tr>
<td><code>PBDOM_ATTRIBUTE</code></td>
<td>5</td>
</tr>
<tr>
<td><code>PBDOM_CHARACTERDATA</code></td>
<td>6</td>
</tr>
<tr>
<td><code>PBDOM_TEXT</code></td>
<td>7</td>
</tr>
<tr>
<td><code>PBDOMCDATA</code></td>
<td>8</td>
</tr>
<tr>
<td><code>PBDOM_COMMENT</code></td>
<td>9</td>
</tr>
<tr>
<td><code>PBDOMPROCESSINGINSTRUCTION</code></td>
<td>10</td>
</tr>
</tbody>
</table>

**See also**

TBD

**GetObjectClassString()**

**Description**

The `GetObjectClassString()` method returns a string form of the class of the `PBDOM_OBJECT`.

**Syntax**

```
pbdom_object_name.GetObjectClassString()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_object_name</code></td>
<td>The name of your <code>PBDOM_OBJECT</code>.</td>
</tr>
</tbody>
</table>

**Return value**

String
The `GetObjectClassString` returns a string that indicates the class of the current `PBDOM_OBJECT`.

### Examples
TBD

### Usage
This function returns the following possible values:

<table>
<thead>
<tr>
<th>Class</th>
<th>String returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBDOM_OBJECT</td>
<td><code>pbdom_object</code></td>
</tr>
<tr>
<td>PBDOM_DOCUMENT</td>
<td><code>pbdom_document</code></td>
</tr>
<tr>
<td>PBDOM_ELEMENT</td>
<td><code>pbdom_element</code></td>
</tr>
<tr>
<td>PBDOM_DOCTYPE</td>
<td><code>pbdom_doctype</code></td>
</tr>
<tr>
<td>PBDOM_ATTRIBUTE</td>
<td><code>pbdom_attribute</code></td>
</tr>
<tr>
<td>PBDOM_CHARACTERDATA</td>
<td><code>pbdom_characterdata</code></td>
</tr>
<tr>
<td>PBDOM_TEXT</td>
<td><code>pbdom_text</code></td>
</tr>
<tr>
<td>PBDOMCDATA</td>
<td><code>pbdom_cdata</code></td>
</tr>
<tr>
<td>PBDOM_COMMENT</td>
<td><code>pbdom_comment</code></td>
</tr>
<tr>
<td>PBDOM_PROCESSINGINSTRUCTION</td>
<td><code>pbdom_processinginstruction</code></td>
</tr>
</tbody>
</table>

### See also
TBD

---

**GetDocument()**

### Description
The `GetDocument()` method returns the owning `PBDOM_DOCUMENT` of the current `PBDOM_OBJECT`.

### Syntax
```
pbdom_object_name.GetDocument()
```

### Arguments
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_object_name</code></td>
<td>The name of your <code>PBDOM_OBJECT</code>.</td>
</tr>
</tbody>
</table>

### Return value
`PBDOM_OBJECT`

### Examples
TBD

### Usage
The owning `PBDOM_DOCUMENT` of the current `PBDOM_OBJECT` will be `NULL` if `PBDOM_OBJECT` is not owned by any `PBDOM_DOCUMENT` or if the current `PBDOM_OBJECT` itself is a `PBDOM_DOCUMENT` object.

### See also
TBD
**PBDOM_OBJECT**

### Detach()

**Description**  
The Detach() method detaches a PBDOM_OBJECT from its parent.

**Syntax**  
```
pbdm_object_name.Detach()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_object_name</code></td>
<td>The name of your PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

**Return value**  
PBDOM_OBJECT

**Examples**  
The root element of a PBDOM_DOCUMENT called `pbdom_doc` is detached from its parent object, that is, the PBDOM_DOCUMENT itself. Then, its parent PBDOM_OBJECT is called and asked if it is NULL, using the IsNull() function. IsNull() returns True and the message box is displayed.

```plaintext
pbdom_obj = pbdom_doc.GetRootElement()
pbdom_obj.Detach()
pbdom_parent_obj = pbdom_obj.GetParentObject()
if (IsNull(pbdom_parent_obj)) then
    MessageBox ("IsNull", "Root Element has no Parent")
end if
```

**Usage**  
If the PBDOM_OBJECT has no parent, this method does nothing.

**See also**  
TBD

### GetParentObject()

**Description**  
The GetParentObject() method returns the parent PBDOM_OBJECT of the current PBDOM_OBJECT.

**Syntax**  
```
pbdm_object_name.GetParentObject()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_object_name</code></td>
<td>The name of your PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

**Return value**  
PBDOM_OBJECT

**Examples**  
The root element of a PBDOM_DOCUMENT called `pbdom_doc`, is gotten using the GetRootElement() method and stored a PBDOM_OBJECT called `pbdom_obj`. Next, the GetParentObject() method is invoked on `pbdom_obj` and the returned PBDOM_OBJECT is stored in `pbdom_parent_obj`.

This returns the parent of the root element which is the PBDOM_DOCUMENT itself. Then the GetObjectClassString() method is called. The messagebox displays the class name of `pbdom_parent_obj` as `pbdom_document`.

---

134
pbdom_document pbdom_doc
pbdom_object pbdom_obj
pbdom_object pbdom_parent_obj
string strClassName

pbdom_doc = pbdombuilder_new.Build (strXML)
pbdom_obj = pbdom_doc.GetRootElement()
pbdom_parent_obj = pbdom_obj.GetParentObject()
strClassName = pbdom_parent_obj.GetObjectClassString()
MessageBox ("Parent Class Name", strClassName)

Usage
If the PB DOM OBJECT has no parent, a null is returned.

See also
TBD

HasChildren()

Description
The HasChildren() method returns True if the current PB DOM OBJECT has at least one child PB DOM OBJECT, and False if it has none.

Syntax
pbdom_object_name.HasChildren()

Argument | Description
--- | ---
pbdom_object_name | The name of your PB DOM OBJECT.

Return value
Boolean

Value | Description
--- | ---
True | The current PB DOM OBJECT has at least one child PB DOM OBJECT.
False | The current PB DOM OBJECT has no child PB DOM OBJECTs.

Examples
In the following example, a PB DOM DOCUMENT is created out of a simple XML string. The root element <abc> has a child TEXT Node which encapsulates the text “abc data”. Calling HasChildren() on the root element returns a True. The message box displays Has Children. If the method returns a False, the message box displays Has No Children

PB DOM_Builder pbdombuilder_new
pbdom_document pbdom_doc
pbdom_object pbdom_root_element
string strXML = "<abc>abc data</abc>"

pbdombuilder_new = Create PB DOM_Builder
pbdom_doc = pbdombuilder_new.Build (strXML)
pbdom_root_element = pbdom_doc.GetRootElement()
if (pbdom_root_element.HasChildren()) then
    MessageBox ("pbdom_root_element", "Has Children")
else
    MessageBox ("pbdom_root_element", "Has No Children")
end if
Destroy pbdombuilder_new

Usage
If the PBDM_OBJECT has at least one child True is returned, and False if there are no children.

See also
TBD

GetTextNormalize()

Description
The GetTextNormalize() method gets the text data that is contained in the current PBDM_OBJECT with all surrounding whitespace removed and internal whitespace normalized to a single space.

Syntax

\[ pbdom_object_name.GetTextNormalize() \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_object_name</td>
<td>The name of your PBDM_OBJECT.</td>
</tr>
</tbody>
</table>

Return value
String

The GetTextNormalize returns the normalized text content of the current PBDM_OBJECT, or an empty string if there is no text content.

Examples
TBD

Usage
This method only returns meaningful data if the PBDM_OBJECT is of a type that can contain TEXT NODEs or CDATA Sections, or of a type that intrinsically contains basic text. These types are:

- PBDM_ELEMENT
- PBDM_ATTRIBUTE
- PBDM_TEXT
- PBDM_CDATA
- PBDM_COMMENT
The PBDOM_TEXT, PBDOM_CDATA, and PBDOM_COMMENT classes are special cases which cause the GetTextNormalize() method to return the intrinsic text data contained within their instances. A PBDOM_TEXT object represents a DOM TEXT NODE, so it does not hold any child DOM Nodes. PBDOM_CDATA object is a representation of a DOM CDATA object and does not hold any child DOM Nodes. Nor does PBDOM_COMMENT contain any child DOM Nodes.

The following table lists the return values based on the type of actual DOM Object contained within PBDOM_OBJECT:

<table>
<thead>
<tr>
<th>DOM Object Type</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBDOM_ELEMENT</td>
<td>The concatenation of the text values of all the TEXT Nodes and CDATA Sections contained within the PBDOM_ELEMENT.</td>
</tr>
<tr>
<td></td>
<td>If we have a PBDOM_ELEMENT defined as follows:</td>
</tr>
<tr>
<td></td>
<td>&lt;abc&gt;Root Element Data&lt;data&gt;ABC Data &lt;/data&gt; now with extra info&lt;/abc&gt;</td>
</tr>
<tr>
<td></td>
<td>GetTextNormalize() will return “Root Element Data now with extra info”.</td>
</tr>
<tr>
<td></td>
<td>If we have a PBDOM_ELEMENT defined as follows:</td>
</tr>
<tr>
<td></td>
<td>&lt;abc&gt; Root Element Data &lt;/abc&gt;</td>
</tr>
<tr>
<td></td>
<td>GetTextNormalize() will return “Root Element Data”.</td>
</tr>
<tr>
<td></td>
<td>If we have a PBDOM_ELEMENT defined as follows:</td>
</tr>
<tr>
<td></td>
<td>&lt;abc&gt;Root Element Data &lt;![CDATA[ with some cdata text]]&gt;&lt;/abc&gt;</td>
</tr>
<tr>
<td></td>
<td>GetTextNormalize() will return “Root Element Data with some cdata text”.</td>
</tr>
<tr>
<td>DOM Object Type</td>
<td>Return Value</td>
</tr>
<tr>
<td>-----------------------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| PBDOM_ATTRIBUTE       | The text data contained within the PBDOM_ATTRIBUTE object. If we have an element with an attribute as follows: <abc ATTRIBUTE_1="My Attribute ">
|                       | GetTextNormalize() will return “My Attribute”.                                                                                               |
| PBDOM_TEXT            | The text data contained within the PBDOM_TEXT object itself. For example, if we have the following element: <abc> MY TEXT </abc>
|                       | If we have a PBDOM_TEXT object to represent the TEXT NODE “MY TEXT”, then calling GetTextNormalize() on the PBDOM_TEXT will return the string “MY TEXT”. |
| PBDOM_CDATA           | The string data that is contained within the CDATA section itself. For example, if we have the following CDATA: <![CDATA[ They’re saying “x < y” & that “z > y” so I guess that means that z > x ]]> |
|                       | If we have a PBDOM_CDATA to represent the above CDATA section, then calling GetTextNormalize() on it will return the string: “They’re saying “ x < y ” & that “z > y” so I guess that means that z > x ”. Note that the initial spaces before “They’re” and the trailing space after the last “x” are removed. Additionally, the spaces between the word “guess” and “that” are reduced to just one space. |
GetTextTrim()

Description
The GetTextTrim() method gets the text data that is contained in the current PBDOM_OBJECT with all surrounding whitespace removed.

Syntax

```powershell
pbdom_object_name.GetTextTrim()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_object_name</td>
<td>The name of your PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

Return value
String

The GetTextTrim returns the trimmed text content of the current PBDOM_OBJECT, or an empty string if there is no text content or only whitespace.

Examples
TBD

Usage
This method only returns meaningful data if the PBDOM_OBJECT is of a type that can contain TEXT NODEs or CDATA Sections, or of a type that intrinsically contains basic text. These types are:

- PBDOM_ELEMENT
- PBDOM_ATTRIBUTE
- PBDOM_TEXT
- PBDOM_CDATA
- PBDOMCOMMENT

See also
TBD
The PBDOM_TEXT, PBDOMCDATA, and PBDOM_COMMENT classes are special cases which cause the GetTextTrim method to return the intrinsic text data contained within their instances. A PBDOM_TEXT object represents a DOM TEXT NODE, so it does not hold any child DOM Nodes. PBDOMCDATA object is a representation of a DOM CDATA object and does not hold any child DOM Nodes. Nor does PBDOM_COMMENT contain any child DOM Nodes.

The following table lists the return values based on the type of actual DOM Object contained within PBDOM_OBJECT:

<table>
<thead>
<tr>
<th>DOM Object Type</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBDOM_ELEMENT</td>
<td>The concatenation of the text values of all the TEXT Nodes and CDATA Sections contained within the PBDOM_ELEMENT. Surrounding whitespaces are removed.</td>
</tr>
<tr>
<td></td>
<td>If we have a PBDOM_ELEMENT defined as follows:</td>
</tr>
<tr>
<td></td>
<td>&lt;abc&gt; Root Element Data&lt;data&gt;ABC Data &lt;/data&gt; now with extra info &lt;/abc&gt;</td>
</tr>
<tr>
<td></td>
<td>GetTextTrim() will return “Root Element Data now with extra info”.</td>
</tr>
<tr>
<td></td>
<td>If we have a PBDOM_ELEMENT defined as follows:</td>
</tr>
<tr>
<td></td>
<td>&lt;abc&gt; Root Element Data &lt;/abc&gt;</td>
</tr>
<tr>
<td></td>
<td>GetTextTrim() will return “Root Element Data”.</td>
</tr>
<tr>
<td></td>
<td>If we have a PBDOM_ELEMENT defined as follows:</td>
</tr>
<tr>
<td></td>
<td>&lt;abc&gt; Root Element Data &lt;![CDATA[ with some cdata text]]&gt;&lt;/abc&gt;</td>
</tr>
<tr>
<td></td>
<td>GetTextTrim() will return “Root Element Data with some cdata text”</td>
</tr>
<tr>
<td>DOM Object Type</td>
<td>Return Value</td>
</tr>
<tr>
<td>-------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
</tbody>
</table>
| PBDOM_ATTRIBUTE   | The text data contained within the PBDOM_ATTRIBUTE object with surrounding whitespaces removed. If we have an element with an attribute as follows:  
                      &lt;abc ATTRIBUTE_1="My Attribute ",&gt;  
                      GetTextTrim() will return “My Attribute”. Note, however, that the spaces between “My” and “Attribute” will still be there. |
| PBDOM_TEXT        | The text data contained within the PBDOM_TEXT object itself with surrounding whitespaces removed. For example, if we have the following element:  
                      &lt;abc&gt; MY TEXT &lt;/abc&gt;  
                      If we have a PBDOM_TEXT object to represent the TEXT NODE “ MY TEXT ”, then calling GetTextTrim() on the PBDOM_TEXT will return the string “MY TEXT”. |
| PBDOM_CDATA       | The string data that is contained within the CDATA section itself with surrounding whitespaces removed. For example, if we have the following CDATA:  
                      &lt;![CDATA[ They’re saying “x < y” & that “z > y” so I guess that means that z > x ]]&gt;  
                      If we a PBDOM_CDATA to represent the above CDATA section, then calling GetTextTrim() on it will return the string:  
                      “They’re saying “ x < y ” & that “z > y” so I guess that means that z > x ”.  
                      Note that the initial spaces before “They’re” and the trailing space after the last “x” are removed. |
See also TBD

IsAncestorOf()

Description
The IsAncestorOf() method determines if the current PBDOM_OBJECT is the ancestor of another PBDOM_OBJECT.

Syntax
```
pbdom_object_name.IsAncestorOf(pbdom_object_ret)
```

Arguments
- `pbdom_object_name`: The name of your PBDOM_OBJECT.
- `pbdom_object_ref`: A reference to a PBDOM_OBJECT to check against.

Return value
Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The current PBDOM_OBJECT is the ancestor of the referenced PBDOM_OBJECT.</td>
</tr>
<tr>
<td>False</td>
<td>The current PBDOM_OBJECT is not the ancestor of the referenced PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

Examples
The following code fragment uses the IsAncestorOf method and creates a structured document. In the fragment, `pbdom Elem_1` represents the `<pbdom Elem_1>` element. Because it is an ancestor of `pbdom Elem_3`, which represents the `<pbdom Elem_3>` element, the call—
```
pbdom Elem_1.IsAncestorOf(ref pbdom Elem_3) —returns True.
```

PBDOM_ELEMENT pbdom Elem_1
PBDOM_ELEMENT pbdom Elem_2
PBDOM_ELEMENT pbdom Elem_3
PowerBuilder Document Object Model

```plaintext
PBDOM_ELEMENT pbdom_elem_root
PBDOM_DOCUMENT pbdom_doc1

pbdom_doc1 = Create PBDOM_DOCUMENT
pbdom_elem_1 = Create PBDOM_ELEMENT
pbdom_elem_2 = Create PBDOM_ELEMENT
pbdom_elem_3 = Create PBDOM_ELEMENT

pbdom_elem_1.SetName ("pbdom_elem_1")
pbdom_elem_2.SetName ("pbdom_elem_2")
pbdom_elem_3.SetName ("pbdom_elem_3")

pbdom_elem_1.AddContent (pbdom_elem_2)
pbdom_elem_2.AddContent (pbdom_elem_3)

pbdom_doc1.NewDocument ("", "Root_Element_From_Doc_1", ", ", ", ")

pbdom_elem_root = pbdom_doc1.GetRootElement()

IF (pbdom_elem_1.IsAncestorOf(ref pbdom_elem_3)) THEN
    MessageBox ("Ancestry", "pbdom_elem_1 Is The Ancestor Of pbdom_elem_3")
ELSE
    MessageBox ("Ancestry", "pbdom_elem_1 Is NOT The Ancestor Of pbdom_elem_3")
END IF

destroy pbdom_elem_1
destroy pbdom_elem_2
destroy pbdom_elem_3
destroy pbdom_elem_root
destroy pbdom_doc1
```

The above code fragment creates the following document.

```xml
<!DOCTYPE Root_Element_From_Doc_1>
<Root_Element_From_Doc_1>
    <pbdom_elem_1>
        <pbdom_elem_2>
            <pbdom_elem_3 />
        </pbdom_elem_2>
    </pbdom_elem_1>
</Root_Element_From_Doc_1>
```

Usage

The IsAncestorOf() method determines if the current PBDOM_OBJECT is the ancestor of another PBDOM_OBJECT.

See also

TBD
**SetParentObject()**

**Description**
The `SetParentObject()` method sets the referenced PBDOM_OBJECT to be the parent of the current PBDOM_OBJECT.

**Syntax**
```
pbdom_object_name.SetParentObject(pbdom_object_ref)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_object_name</td>
<td>The name of your PBDOM_OBJECT.</td>
</tr>
<tr>
<td>pbdom_object_ref</td>
<td>A reference to a PBDOM_OBJECT to be set as the parent of the current PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

**Return value**
PBDOM_OBJECT

The current PBDOM_OBJECT is appended as a child node of the referenced parent.

**Examples**
In the following code example, a PBDOM_ELEMENT object is created and called `pbdom_elem_1`. Its parent is set to be the root element of the PBDOM_DOCUMENT called `pbdom_doc`. Once this is done, `pbdom_elem_1` is immediately transferred to the `pbdom_doc` document and `pbdom_elem_1` is immediately appended as a child node of the root element of `pbdom_doc`.

The function call `pbdom_elem_1.GetParentObject().GetObjectClassString()` returns the string "pbdom_element" because the root element is a PBDOM_ELEMENT.

The function call `pbdom_elem_1.GetParentObject().GetName()` returns the string "Root_Element"—the name of the root element.

```pascal
PBDOM_ELEMENT pbdom_elem_1
PBDOM_ELEMENT pbdom_elem_root
PBDOM_DOCUMENT pbdom_doc1

pbdom_doc1 = Create PBDOM_DOCUMENT
pbdom_elem_1 = Create PBDOM_ELEMENT
pbdom_elem_1.SetName ("pbdom_elem_1")

pbdom_doc1.NewDocument ("", "Root_Element", "", ")
pbdom_elem_root = pbdom_doc1.GetRootElement()
pbdom_elem_1.SetParentObject(ref pbdom_elem_root)

MessageBox ("Parent Class",
pbdom_elem_1.GetParentObject().GetObjectClassString())
MessageBox ("Parent Name", pbdom_elem_1.GetParentObject().GetName())
```
Usage

The caller is responsible for ensuring that the current PBDOM_OBJECT and the referenced PBDOM_OBJECT can have a legal parent-child relationship. You are also responsible for making sure pre-existing parentage is legal.

See the SetParentObject() documentation of derived PBDOM_OBJECT classes for more details on implementation of specific classes.

See also

TBD

PBDOM_DOCUMENT

Description

The PBDOM_DOCUMENT class defines behavior for an XML DOM document. Methods allow access to the root element, processing instructions, and other document-level information.

The PBDOM_DOCUMENT class inherits from a PBDOM_OBJECT and so provides specialized implementations for most of the PBDOM_OBJECT class.

Default or trivial functionalities

Some of the inherited methods from PBDOM_OBJECT serve no meaningful objective and so only default or trivial functionalities will result. These are noted in the individual function descriptions that follow.

PBDOM_DOCUMENT has the following functions:

- GetName()
- GetText()
- GetContent
- AddContent
- InsertContent
- RemoveContent
- SetContent
- SetName
- Clone

The caller is responsible for ensuring that the current PBDOM_OBJECT and the referenced PBDOM_OBJECT can have a legal parent-child relationship. You are also responsible for making sure pre-existing parentage is legal.

See the SetParentObject() documentation of derived PBDOM_OBJECT classes for more details on implementation of specific classes.

See also

TBD
PBDOM_DOCUMENT

- Equals
- GetObjectClass()
- GetObjectClassString()
- GetDocument()
- Detach()
- GetParentObject()
- HasChildren()
- GetTextNormalize()
- GetTextTrim()
- IsAncestorOf()
- SetParentObject()
- NewDocument()
- NewDocument() overloaded
- GetRootElement()
- GetDocType()
- SaveDocument()
- DetachRootElement()
- HasRootElement()
- SetDocType()
- SetRootElement()

**GetName()**

**Description**

**Trivial Implementation**

Calling the GetName() method always returns the string "#document".

**Syntax**

`pbdom_document_name.GetName()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_document_name</code></td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
</tbody>
</table>
PowerBuilder Document Object Model

Return value
String

GetName() always always returns the string "#document".

Examples
TBD

Usage
A PBDOM_OBJECT cannot be instantiated directly.

See also
TBD

GetText()

Description
Trivial Implementation
Calling the GetText() method always returns an empty string.

Syntax

\[
\text{pbdom\_document\_name}.\text{GetText()}
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{pbdom_document_name}</td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
</tbody>
</table>

Return value
Empty String

Examples
TBD

Usage
A GetText() method always returns an empty string because PBDOM_DOCUMENT cannot contain any text DOM nodes.

See also
TBD

GetContent

Description
The GetContent() method returns all child content of the current PBDOM_DOCUMENT.

Syntax

\[
\text{pbdom\_document\_name}.\text{GetContent(pbdom\_object\_array)}
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{pbdom_document_name}</td>
<td>The name of your PBDOM_DOCUMENT,</td>
</tr>
<tr>
<td>\text{pbdom_object_array}</td>
<td>The referenced name of an array of PBDOM_OBJECTs that will receive PBDOM_OBJECTs.</td>
</tr>
</tbody>
</table>

Return value
Boolean
Examples

Assume a PBDOM_DOCUMENT object called pbdom_doc contains the following XML document.

```xml
<Root>
  <Element_1>
    <Element_1_1/>
    <Element_1_2/>
    <Element_1_3/>
  </Element_1>
  <Element_2/>
  <Element_3/>
</Root>
```

In the following PowerScript code fragment, the array `pbdom_obj_array` contains `pbdom_obj_array[1] - <Root>`. It contains just one PBDOM_ELEMENT which represents the element `<Root>`.

```powerscript
PBDOM_DOCUMENT pbdom_doc
PBDOM_OBJECT pbdom_obj_array[]
...
pbdom_doc.GetContent(ref pbdom_obj_array)
```

`pbdom_doc.GetRootElement().GetContent(ref pbdom_obj_array)` yields an array that contains:

- `pbdom_obj_array[1] - <Element_1>`
- `pbdom_obj_array[2] - <Element_2>`

The following is an example of the returned PBDOM_OBJECTArray. `pbdom_obj_array[2].AddContent("Element 2 Text")` causes `<Element_2>` to contain the Text node "Element 2 Text" and so the tree is as follows:

```xml
<Root>
  <Element_1>
    <Element_1_1/>
    <Element_1_2/>
    <Element_1_3/>
  </Element_1>
  <Element_2>Element 2 Text</Element_2>
  <Element_3/>
</Root>
```
Usage
The returned array is passed by reference, with items in the same order as they appear in the PBDOM_OBJECT. Any changes to any item of the array will affect the actual item to which it refers.

See also
TBD

AddContent
Description
The AddContent() method allows you to add a new PBDOM_OBJECT into the current PBDOM_DOCUMENT.

Syntax

```
pbdom_document_name.AddContent(pbdom_object_ref)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_document_name</td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
<tr>
<td>pbdom_object_ref</td>
<td>The referenced name of a PBDOM_OBJECT you want to add.</td>
</tr>
</tbody>
</table>

Return value
PBDOM_OBJECT
The return value is the newly modified PBDOM_DOCUMENT returned as a PBDOM_OBJECT.

Examples
The document pbdom_doc1 is created with three elements: pbdom_elem_1, Pbdom_elem_2, and pbdom_elem_3. pbdom_elem_2 and pbdom_elem_3 are set as children of pbdom_element_1.

```
pbdom_doc1.GetRootElement().Detach() detaches the root element from pbdom_doc1. pbdom_elem_1 is added as a child of pbdom_doc1 with :pbdom_doc1.AddContent(ref pbdom_elem_1).
```

```
PBDOM_ELEMENT pbdom_elem_1
PBDOM_ELEMENT pbdom_elem_2
PBDOM_ELEMENT pbdom_elem_3
PBDOM_DOCUMENT pbdom_doc1

pbdom_doc1 = Create PBDM_DOCUMENT
pbdom_elem_1 = Create PBDM_ELEMENT
pbdom_elem_2 = Create PBDM_ELEMENT
pbdom_elem_3 = Create PBDM_ELEMENT

pbdom_elem_1.SetName ("pbdom_elem_1")
pbdom_elem_2.SetName ("pbdom_elem_2")
pbdom_elem_3.SetName ("pbdom_elem_3")

pbdom_elem_1.AddContent(pbdom_elem_2)
pbdom_elem_1.AddContent(pbdom_elem_3)
```
The original root element `<Root_Element>` has been detached and replaced by `<pbdom_elem_1>`. The document is transformed to:

```xml
<!DOCTYPE Root_Element>
<pbdom_elem_1>
  <pbdom_elem_2/>
  <pbdom_elem_3/>
</pbdom_elem_1>
```

If the root element detachment statement `pbdom_doc1.GetRootElement().Detach()` is omitted, then the following exception statement is thrown:

![Exception Occurred]

Adding `pbdom_object` will cause `pbdom_document` to contain multiple root elements.

Exception Information:
- Function Name: `pbdom_document: AddContent(ref pbdom_object pbdom_object_ref)`
- `PBDOM_OBJECT` name: `pbdom_elem_1`.

**Usage**

The new `PBDOM_OBJECT` becomes a child `PBDOM_OBJECT` of the current `PBDOM_DOCUMENT`. The following table lists the `PBDOM_OBJECTs` that can be added to a `PBDOM_DOCUMENT` and the restrictions for their addition.

<table>
<thead>
<tr>
<th><code>PBDOM_OBJECT</code></th>
<th>Restrictions</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>PBDOM_ELEMENT</code></td>
<td>Allowed to be added only if this document currently does not contain any root element. Otherwise the exception <code>EXCEPTION_MULTIPLE_ROOT_ELEMENT</code> will be thrown. Furthermore, the <code>PBDOM_ELEMENT</code> to be added must not already have a parent <code>PBDOM_OBJECT</code>. If it does, the exception <code>EXCEPTION_PBDOM_OBJECT_ALREADY_HAS_PARENT</code> will be thrown.</td>
</tr>
</tbody>
</table>
PowerBuilder Document Object Model

InsertContent

Description
The InsertContent() method allows you to insert a new PBDOM_OBJECT into the current PBDOM_DOCUMENT.

Syntax

\[ pbdom\_document\_name.\text{InsertContent}(\text{pbdom\_object\_new, pbdom\_object\_ref}) \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_document_name</td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
</tbody>
</table>
**PBDM\_DOCUMENT**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_object_new</td>
<td>The referenced name of a PBDM_OBJECT you want to insert.</td>
</tr>
<tr>
<td>pbdom_object_ref</td>
<td>The referenced name of the PBDM_OBJECT in front of which you want to insert the new PBDM_OBJECT.</td>
</tr>
</tbody>
</table>

**Return value**

PBDM\_OBJECT

The modified PBDM\_DOCUMENT returned as a PBDM\_OBJECT.

**Examples**

A PBDM\_DOCUMENT is created from an XML string. The PBDM\_ELEMENT pbdom\_elem\_1 is also created and set as Elem\_1. The PBDM\_DOCTYPE pbdom\_doctype\_1 and the root element pbdom\_root\_elem are set.

The root element is detached from its parent which is also the PBDM\_DOCUMENT itself. This makes it possible to insert pbdom\_elem\_1 into the document specifically before pbdom\_doctype\_1.

```c
pbdom\_builder pbdom\_builder\_1
pbdom\_document pbdom\_doc
pbdom\_doctype pbdom\_doctype\_1
pbdom\_element pbdom\_elem\_1
pbdom\_element pbdom\_elem\_root
string strXML = "<!DOCTYPE abc[<!-- internal subset --> <!ELEMENT abc (#PCDATA)>
> <!ELEMENT data (#PCDATA)> <!ELEMENT inner_data (#PCDATA)>]><abc>Root Element Data<data>ABC Data</data>My Inner Data</inner_data>My Data</data> now with extra info</abc>"

pbdom\_builder\_1 = Create PBDM\_Builder
pbdom\_elem\_1 = Create PBDM\_Element

pbdom\_doc = pbdom\_builder\_1.Build (strXML)
pbdom\_elem\_1.SetName ("Elem\_1")
pbdom\_doctype\_1 = pbdom\_doc.GetDocType()
pbdom\_elem\_root = pbdom\_doc.GetRootElement()

pbdom\_elem\_root.Detach()
pbdom\_doc.InsertContent(ref pbdom\_elem\_1, ref pbdom\_doctype\_1)
```

The result is the following, not well-formed document:

```xml
<Elem\_1/>
<!DOCTYPE abc[<!-- internal subset --> <!ELEMENT abc (#PCDATA)> <!ELEMENT data (#PCDATA)> <!ELEMENT inner_data (#PCDATA)>]>
```
Usage
When a new PBDOM_OBJECT is inserted into the current PBDOM_DOCUMENT, the new PBDOM_OBJECT becomes a child node of the current PBDOM_DOCUMENT. Also, the new PBDOM_OBJECT is to be positioned specifically before another PBDOM_OBJECT, specified using the second parameter.

If the second PBDOM_OBJECT is specified as null, then the new PBDOM_OBJECT is to be inserted at the end of the list of children of the current PBDOM_DOCUMENT.

See also
TBD

RemoveContent
Description
The RemoveContent() method allows you to remove a child PBDOM_OBJECT from the current PBDOM_DOCUMENT.

Syntax
```
pbdom_document_name.RemoveContent(pbdom_object_ref)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_document_name</td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
<tr>
<td>pbdom_object_ref</td>
<td>The referenced name of a PBDOM_OBJECT you want to remove.</td>
</tr>
</tbody>
</table>

Return value
Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The content was removed.</td>
</tr>
<tr>
<td>False</td>
<td>The content was not removed.</td>
</tr>
</tbody>
</table>

Examples
TBD

Usage
When a PBDOM_OBJECT is removed from the current PBDOM_DOCUMENT, all children under the removed PBDOM_OBJECT are also removed.

See also
TBD

SetContent
Description
The SetContent() method sets the entire content of the PBDOM_DOCUMENT.

Syntax
```
pbdom_document_name.SetContent(pbdom_object_array)
```
**PBDOM_DOCUMENT**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBDOM_DOCUMENT</td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
<tr>
<td>PBDOM_OBJECT_ARRAY</td>
<td>An array of PBDOM_OBJECTs set as the contents of the PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

Return value

PBDOM_OBJECT

The return value is the newly modified PBDOM_OBJECT.

Examples

TBD

Usage

The supplied array contains PBDOM_OBJECTs that are legal to be set as the content of a PBDOM_DOCUMENT.

For example, a PBDOM_DOCUMENT only accepts an array that contains PBDOM_ELEMENT, PBDOM_COMMENT, PBDOM_PROCESSINGINSTRUCTION, or objects. The array is restricted to contain only one PBDOM_ELEMENT object that it sets as its root element. It is also restricted to contain at most one PBDOM_DOCTYPE object that it sets as its DOCTYPE.

In the event of an exception, the original contents of this PBDOM_DOCUMENT will be unchanged and the PBDOM_OBJECTs contained in the supplied array will be unaltered.

See also

TBD

**SetName**

Description

**Trivial Implementation**

The PBDOM_DOCUMENT does not have any name associated with it so calling the SetName() method always returns False.

Syntax

PBDOM_DOCUMENT.SetName(strName)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBDOM_DOCUMENT</td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
<tr>
<td>strName</td>
<td>The new name you want to set for PBDOM_DOCUMENT.</td>
</tr>
</tbody>
</table>

Return value

Boolean

The current implementation always returns False for the SetName() method.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The PBDOM_DOCUMENTs name was changed.</td>
</tr>
</tbody>
</table>
### Clone

**Description**
The `Clone()` method creates a deep clone of the current PBDOM_DOCUMENT.

**Syntax**

```powershell
pbdom_document_name.Clone()
```

**Argument** | **Description**
--- | ---
`pbdom_document_name` | The name of your PBDOM_DOCUMENT.

**Return value**
PBDOM_OBJECT

**Examples**
The return value is the deep clone of PBDOM_OBJECT.

**Usage**
The `Clone()` method creates a deep clone of the current PBDOM_DOCUMENT, housed as a PBDOM_OBJECT. This function clones all the child PBDOM_OBJECTs contained within the original PBDOM_DOCUMENT in the clone documents.

**See also**
TBD

### Equals

**Description**
The `Equals()` method tests for the equality of the current PBDOM_DOCUMENT and a referenced PBDOM_OBJECT.

**Syntax**

```powershell
pbdom_document_name.Equals(pbdom_object_ref)
```

**Argument** | **Description**
--- | ---
`pbdom_document_name` | The name of your PBDOM_OBJECT.
`pbdom_object_ref` | A reference to a PBDOM_OBJECT to test for equality with the current PBDOM_OBJECT.

**Return value**
Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>The PBDOM DOCUMENTs name was not changed.</td>
</tr>
</tbody>
</table>
**GetObjectClass()**

**Description**

The GetObjectClass() method returns a long integer code that indicates the class of the current PBDOM_DOCUMENT.

**Syntax**

```plaintext
pbdom_document_name.GetObjectClass()
```

**Argument**

| pbdom_document_name | The name of your PBDOM_DOCUMENT. |

**Return value**

Long

The GetObjectClass returns a long integer code that indicates the class of the current PBDOM_DOCUMENT. The return value is 2 for PBDOM_DOCUMENT.

**Examples**

TBD

**See also**

TBD

---

**GetObjectClassString()**

**Description**

The GetObjectClassString() method returns a string form of the class of the PBDOM_DOCUMENT.

**Syntax**

```plaintext
pbdom_document_name.GetObjectClassString()
```

**Argument**

| pbdom_document_name | The name of your PBDOM_DOCUMENT. |

---

**Examples**

TBD

**See also**

TBD
Return value: String

The `GetObjectClassString` returns a string that indicates the class of the current `PBDOM_DOCUMENT`.

Examples: TBD

Usage: The returned string is "pbdom_document".

See also: TBD

### GetDocument()

**Description**

**Trial Implementation**

A `PBDOM_DOCUMENT` has no owner document. Therefore calling the `GetDocument()` method always returns a null.

**Syntax**

```plaintext
pbdom_document_name.GetDocument()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_document_name</code></td>
<td>The name of your <code>PBDOM_DOCUMENT</code>.</td>
</tr>
</tbody>
</table>

**Return value**: `PBDOM_OBJECT`

**Examples**: TBD

**See also**: TBD

### Detach()

**Description**

**Trivial Implementation**

This method does nothing and returns the `PBDOM_DOCUMENT` unmodified in any way, because the a `PBDOM_DOCUMENT` does not have a parent `PBDOM_OBJECT`.

**Syntax**

```plaintext
pbdom_document_name.Detach()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_document_name</code></td>
<td>The name of your <code>PBDOM_DOCUMENT</code>.</td>
</tr>
</tbody>
</table>

**Return value**: `PBDOM_OBJECT`

**See also**: TBD
GetParentObject()

Description
The GetParentObject() method always returns a null because PBDOM_DOCUMENT does not have a parent.

Syntax
```
pbdom_document_name.GetParentObject()
```

Return value
PBDOM_OBJECT

Usage
If the PBDOM_OBJECT has no parent, a NULL is returned.

See also
TBD

HasChildren()

Description
The HasChildren() method returns True if the current PBDOM_DOCUMENT has at least one child PBDOM_OBJECT, and False if it has none.

Syntax
```
pbdom_document_name.HasChildren()
```

Return value
Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The current PBDOM_DOCUMENT has at least one child PBDOM_OBJECT.</td>
</tr>
<tr>
<td>False</td>
<td>The current PBDOM_DOCUMENT has no child PBDOM_OBJECTs.</td>
</tr>
</tbody>
</table>

Examples
TBD

Usage
If the PBDOM_DOCUMENT has at least one child PBDOM_OBJECT True is returned, and False if there are no children.

See also
TBD
**GetTextNormalize**

**Description**
Trivial Implementation
The GetTextNormalize() method always returns an empty string because PBDOM_DOCUMENT cannot contain any text or CDATA DOM objects.

**Syntax**

```
pbdom_document_name.GetTextNormalize()
```

**Argument Description**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_document_name</td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
</tbody>
</table>

**Return value**
String

**See also**
TBD

**GetTextTrim**

**Description**
Trivial Implementation
The GetTextTrim() method always returns an empty string because PBDOM_DOCUMENT cannot contain any text or CDATA DOM objects.

**Syntax**

```
pbdom_document_name.GetTextTrim()
```

**Argument Description**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_document_name</td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
</tbody>
</table>

**Return value**
String

**See also**
TBD

**IsAncestorOf**

**Description**
The IsAncestorOf() method determines if the current PBDOM_DOCUMENT is the ancestor of another PBDOM_OBJECT.

**Syntax**

```
pbdom_document_name.IsAncestorOf(pbdom_object_ret)
```

**Argument Description**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_document_name</td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
<tr>
<td>pbdom_object_ref</td>
<td>A reference to a PBDOM_OBJECT to check against.</td>
</tr>
</tbody>
</table>

**Return value**
Boolean
**SetParentObject()**

**Description**  
Trivial Implementation  
The `SetParentObject()` method does nothing meaningful. It returns the current PBDOM_DOCUMENT because it does not have a parent.

**Syntax**  
`pbdom_document_name.SetParentObject(pbdom_document_ret)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_document</code></td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
<tr>
<td><code>pbdom_object_ref</code></td>
<td>A reference to a PBDOM_OBJECT to be set as the parent of the current PBDOM_DOCUMENT.</td>
</tr>
</tbody>
</table>

**Return value**  
PBDOM_OBJECT

**See also**  
TBD

---

**NewDocument()**

**Description**  
The `NewDocument()` method allows you to create a new XML DOM document from scratch.

**Syntax**  
`pbdom_document_name.NewDocument(strRootElementName)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_document_name</code></td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
<tr>
<td><code>strRootElementName</code></td>
<td>The name of the root element to be contained with in the DOM document.</td>
</tr>
</tbody>
</table>

**Return value**  
Boolean

---

**Examples**  
TBD

**See also**  
TBD
NewDocument() overloaded

Description
The NewDocument() method allows you to create a new XML DOM document from scratch.

Syntax
```
pbdom_document_name.NewDocument(strRootElementNamespaceURI,strRootElementName,strDocTypePublicId,strDocTypeSystemId)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_document_name</td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
<tr>
<td>strRootElementNamespaceURI</td>
<td>The namespace URI of the root element to be contained with in the DOM document. This can be an empty string.</td>
</tr>
<tr>
<td>strRootElementName</td>
<td>The name of the root element to be contained with in the DOM document.</td>
</tr>
<tr>
<td>strDocTypePublicId</td>
<td>The external subset public identifier.</td>
</tr>
<tr>
<td>strDocTypeSystemId</td>
<td>The external subset system identifier.</td>
</tr>
</tbody>
</table>

Return value
Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>A new document is successfully created</td>
</tr>
<tr>
<td>False</td>
<td>A new document is not successfully created.</td>
</tr>
</tbody>
</table>

Examples
TBD

Usage
There are four parameters to set which provides more control over the DOCTYPE definition of the document.

See also
TBD
**GetRootElement()**

**Description**
The `GetRootElement()` method allows you to retrieve the root element of the current XML DOM document.

**Syntax**
```
pbdom_document_name.GetRootElement()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_document_name</code></td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
</tbody>
</table>

**Return value**
PBDOM_ELEMENT

The `GetRootElement()` method returns the root element of the PBDOM_DOCUMENT housed in a PBDOM_ELEMENT object.

**Examples**
TBD

**Usage**
The return value is the root element encapsulated in a PBDOM_ELEMENT object.

**See also**
TBD

---

**GetDocType()**

**Description**
The `GetDocType()` method allows you to retrieve the DOCTYPE declaration of the current XML DOM document.

**Syntax**
```
pbdom_document_name.GetDocType()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_document_name</code></td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
</tbody>
</table>

**Return value**
PBDOM_ELEMENT

**Examples**
TBD

**Usage**
The DOCTYPE declaration is housed in a PBDOM_OBJECT.

**See also**
TBD

---

**SaveDocument()**

**Description**
The `SaveDocument()` method allows you to serialize the DOM tree contained within the PBDOM_DOCUMENT into a disk file.

**Syntax**
```
pbdom_document_name.SaveDocument(strFileName)
```

---

162
**DetachRootElement()**

**Description**
The `DetachRootElement()` method detaches the root element of this document and returns it.

**Syntax**
```
pbdom_document_name.DetachRootElement()
```

**Argument** | **Description**
--- | ---
`pbdom_document_name` | The name of your PBDOM_DOCUMENT.

**Return value**
Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>A new document is successfully serialized to a disk file.</td>
</tr>
<tr>
<td>False</td>
<td>A new document is not successfully serialized to a disk file.</td>
</tr>
</tbody>
</table>

**Examples**
TBD

**Usage**
TBD

**See also**
TBD

---

**HasRootElement()**

**Description**
The `HasRootElement()` method returns True if this document has a root element.

**Syntax**
```
pbdom_document_name.HasRootElement()
```

**Argument** | **Description**
--- | ---
`pbdom_document_name` | The name of your PBDOM_DOCUMENT.

**Return value**
PBDOM_ELEMENT

**Examples**
TBD

**Usage**
TBD

**See also**
TBD
Return value

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>This document has a root element.</td>
</tr>
<tr>
<td>False</td>
<td>This document does not have a root element.</td>
</tr>
</tbody>
</table>

Examples  TBD
Usage      TBD
See also   TBD

**SetDocType()**

Description  The SetDocType() method sets the DOCTYPE declaration of this document.

Syntax  
```
pbdom_document_name.SetDocType(pbdom_doctype_ref)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_document_name</td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
<tr>
<td>pbdom_doctype_ref</td>
<td>A reference to a PBDOM_DOCTYPE object to be set as the DOCTYPE of this document.</td>
</tr>
</tbody>
</table>

Return value  PBDOM_DOCUMENT

This PBDOM_DOCUMENT returns modified.

Examples  TBD
Usage  If this document already contains a DOCTYPE declaration, the new PBDOM_DOCTYPE replaces it. The DOCTYPE of a PBDOM_DOCUMENT may be set multiple times and it is legal for a user to call PBDOM_DOCUMENT::SetDocType() multiple times.

A DOM DOCTYPE object:
- can have no owner document
- can have an owner document but no parent node.
- that has an owner document as well as a parent node is the actual doctype of that document.

See also   TBD
SetRootElement()

Description
The SetRootElement() method sets the root element for this document.

Syntax
`pbdom_document_name.SetRootElement(pbdom_element_ref)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_document_name</code></td>
<td>The name of your PBDOM_DOCUMENT.</td>
</tr>
<tr>
<td><code>pbdom_element_ref</code></td>
<td>A reference to a PBDOM_ELEMENT object to be set as the root element for this document.</td>
</tr>
</tbody>
</table>

Return value
PBDOM_DOCUMENT
This PBDOM_DOCUMENT returns modified.

Examples
TBD

Usage
If this document already has a root element, the existing root element is replaced. The root element of a PBDOM_DOCUMENT may be set multiple times and it is legal for a user to call PBDOM_DOCUMENT::SetRootElement() multiple times.

See also
TBD

PBDOM_DOCTYPE

Description
The PBDOM_DOCTYPE class represents the Document Type Declaration Object of an XML DOM Document. The PBDOM_DOCTYPE class provides access to the name of the root element which is constrained within the DOCTYPE as well as the internal subset, system and public IDs.

Default or trivial functionalities
Some of the inherited methods from PBDOM_OBJECT serve no meaningful objective and so only default or trivial functionalities will result. These are noted in the individual function descriptions that follow.

PBDOM_DOCTYPE has the following functions:

- GetName()
- GetText()
- GetContent
- AddContent
**GetName()**

**Description**
The `GetName()` method allows you to obtain the name of the element which is being constrained within the current PBDOM_DOCTYPE.

**Syntax**

```
pbdom_doctype_name.GetName()
```
### GetName()

**Description**

If you have the following DOCTYPE declaration, the GetName() method returns "abc".

```
<!DOCTYPE abc [<!-- internal subset -->
  <!ELEMENT abc (#PCDATA)>
  <!ELEMENT data (#PCDATA)>
  <!ELEMENT inner_data (#PCDATA)>]
```

**Examples**

- If you have the following DOCTYPE declaration, the GetName() method returns "abc".

**Usage**

TDB

**See also**

TBD

---

### GetText()

**Description**

**Trivial Implementation**

Calling the GetText() method always returns an empty string.

**Syntax**

```
pbdom_doctype_name.GetText()
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_doctype_name</td>
<td>The name of your PBDO DOCTYPE.</td>
</tr>
</tbody>
</table>

**Return value**

Empty String

**Usage**

A GetText() method always returns an empty string because PBDO DOCTYPE cannot contain any text DOM nodes.

**See also**

TBD

---

### GetContent

**Description**

**Trivial Implementation**

The GetContent() method always returns False and no PBDO OBJECTs are set into the input array.

**Syntax**

```
pbdom_doctype_name.GetContent(pbdom_object_array)
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_doctype_name</td>
<td>The name of your PBDO DOCTYPE.</td>
</tr>
</tbody>
</table>

---
**PBDO_40C45TYPE**

Return value

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>Successful</td>
</tr>
<tr>
<td>False</td>
<td>Unsuccessful</td>
</tr>
</tbody>
</table>

See also

TBD

---

**AddContent**

Description

Trivial Implementation

The AddContent() method does nothing with the input PBDO_40C45BJECT and returns the current PBDO_40C45TYPE.

Syntax

```plaintext
pbdom_doctype_name.AddContent(pbdom_object_ref)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_doctype_name</code></td>
<td>The name of your PBDO_40C45TYPE.</td>
</tr>
<tr>
<td><code>pbdom_object_ref</code></td>
<td>The referenced name of a PBDO_40C45BJECT you want to add.</td>
</tr>
</tbody>
</table>

Return value

PBDO_40C45TYPE

See also

TBD

---

**InsertContent**

Description

Trivial Implementation

The InsertContent() method does nothing with the input parameters and returns the current PBDO_40C45TYPE without any modifications.

Syntax

```plaintext
pbdom_doctype_name.InsertContent(pbdom_object_new, pbdom_object_ref)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_doctype_name</code></td>
<td>The name of your PBDO_40C45TYPE.</td>
</tr>
<tr>
<td><code>pbdom_object_new</code></td>
<td>The referenced name of a PBDO_40C45BJECT you want to insert.</td>
</tr>
<tr>
<td><code>pbdom_object_ref</code></td>
<td>The referenced name of the PBDO_40C45BJECT in front of which you want to insert the new PBDO_40C45BJECT.</td>
</tr>
</tbody>
</table>

Return value

PBDO_40C45BJECT
See also TBD

**RemoveContent**

**Description**

**Trivial Implementation**

The RemoveContent() method does not remove anything from the current PBDOM_DOCTYPE.

**Syntax**

```powershell
dom_doctype_name.RemoveContent(pbdom_object_ref)
```

**Argument** | **Description**
---|---
`pbdom_doctype_name` | The name of your PBDOM_DOCTYPE.
`pbdom_object_ref` | The referenced name of a PBDOM_OBJECT you want to remove.

**Return value**

Boolean

The return value is always `False`.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>True</code></td>
<td>The content was removed.</td>
</tr>
<tr>
<td><code>False</code></td>
<td>The content was not removed.</td>
</tr>
</tbody>
</table>

See also TBD

**SetContent**

**Description**

**Trivial Implementation**

The SetContent() method does nothing with the input PBDOM_OBJECT array and returns the current DOCTYPE without modifications.

**Syntax**

```powershell
dom_doctype_name.SetContent(pbdom_object_array)
```

**Argument** | **Description**
---|---
`pbdom_doctype_name` | The name of your PBDOM_DOCTYPE.
`pbdom_object_array` | An array of PBDOM_OBJECTs set as the contents of the PBDOM_OBJECT.

**Return value**

PBDOM_DOCTYPE

See also TBD
**SetName**

**Description**
The SetName() method sets the name of the root element which is declared by this PBDOM_DOCTYPE.

**Syntax**

```plaintext
pbdom_doctype_name.SetName(strName)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_doctype_name</td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
<tr>
<td>strName</td>
<td>The new name you want to set for the root element that is declared by the current PBDOM_DOCTYPE.</td>
</tr>
</tbody>
</table>

**Return value**

Boolean

The current implementation always returns False for the SetName() method.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The of the root element declared by the current PBDOM_DOCTYPE name was changed.</td>
</tr>
<tr>
<td>False</td>
<td>The of the root element declared by the current PBDOM_DOCTYPE name was not changed.</td>
</tr>
</tbody>
</table>

**Examples**

TBD

**See also**

TBD

---

**Clone**

**Description**
The Clone() method creates and returns a clone of the current PBDOM_DOCTYPE.

**Syntax**

```plaintext
pbdom_doctype_name.Clone()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_doctype_name</td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
</tbody>
</table>

**Return value**

PBDOM_OBJECT

The return value is a deep clone of the current PBDOM_DOCTYPE housed in a PBDOM_OBJECT.

**Examples**

TBD
Usage

The Clone() method creates a deep clone of the current PBDOM_DOCTYPE, housed as a PBDOM_OBJECT. This function clones all the child PBDOM_OBJECTs contained within the original PBDOM_DOCTYPE in the clone documents.

See also

TBD

Equals

Description

The Equals() method tests for the equality of the current PBDOM_DOCTYPE and a referenced PBDOM_OBJECT.

Syntax

```
pbdom_doctype_name.Equals(pbdom_object_ref)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_doctype_name</td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
<tr>
<td>pbdom_object_ref</td>
<td>A reference to a PBDOM_OBJECT to test for equality with the current PBDOM_DOCTYPE.</td>
</tr>
</tbody>
</table>

Return value

Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The current PBDOM_DOCUMENT is equivalent to the referenced PBDOM_OBJECT.</td>
</tr>
<tr>
<td>False</td>
<td>The current PBDOM_DOCUMENT is not equivalent to the referenced PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

Examples

TBD

Usage

Only if the referenced PBDOM_OBJECT is also a PBDOM_DOCUMENT and refers to the same DOM document as the current PBDOM_DOCUMENT is a True returned.

See also

TBD

GetObjectClass()

Description

The GetObjectClass() method returns a long integer code that indicates the class of the current PBDOM_DOCTYPE.

Syntax

```
pbdom_doctype_name.GetObjectClass()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_doctype_name</td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
</tbody>
</table>
**PBDOM_DOCTYPE**

Return value
Long

The `GetObjectClass` returns a long integer code that indicates the class of the current PBDOM_DOCTYPE. The return value is 4 for PBDOM_DOCTYPE.

Examples
TBD

See also
TBD

---

**GetObjectClassString()**

Description
The `GetObjectClassString()` method returns a string form of the class of the PBDOM_DOCTYPE.

Syntax
`pbdom_doctype_name.GetObjectClassString()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_doctype_name</code></td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
</tbody>
</table>

Return value
String

Examples
TBD

Usage
The returned string is "pbdom_doctype".

See also
TBD

---

**GetDocument()**

Description
The `GetDocument()` method returns the owning PBDOM_DOCUMENT of the current PBDOM_DOCTYPE.

Syntax
`pbdom_doctype_name.GetDocument()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_doctype_name</code></td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
</tbody>
</table>

Return value
PBDOM_OBJECT

Examples
TBD

Usage
If there is no owning PBDOM_DOCUMENT, null is returned.

See also
TBD
**Detach()**

**Description**  
**Trivial Implementation**  
This method does nothing and returns the PBDOM_DOCTYPE without modifications.

**Syntax**  
`pbdom_doctype_name.Detach()`  

**Argument** | **Description**  
---|---  
`pbdom_doctype_name` | The name of your PBDOM_DOCTYPE.

**Return value**  
PBDOM_OBJECT  

**Usage**  
A PBDOM_DOCTYPE object is not allowed to be detached from its owner PBDOM_DOCUMENT. You can clone a DOCTYPE and then set the clone as the DOCTYPE of another document.

**See also**  
TBD

---

**GetParentObject()**

**Description**  
The GetParentObject() method returns the parent PBDOM_OBJECT of the current PBDOM_DOCTYPE.

**Syntax**  
`pbdom_doctype_name.GetParentObject()`  

| **Argument** | **Description**  
---|---  
`pbdom_doctype_name` | The name of your PBDOM_DOCTYPE.

**Return value**  
PBDOM_OBJECT  

**Usage**  
The parent is also a PBDOM_DOCUMENT object. If the PBDOM_OBJECT has no parent, a null is returned.

**See also**  
TBD

---

**HasChildren()**

**Description**  
The HasChildren() method returns `True` if the current PBDOM_DOCTYPE has at least one child PBDOM_OBJECT, and `False` if it has none.

**Syntax**  
`pbdom_doctype_name.HasChildren()`  

| **Argument** | **Description**  
---|---  
`pbdom_doctype_name` | The name of your PBDOM_DOCTYPE.
**PBDOM_DOCTYPE**

Return value  Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The current PBDOM_DOCTYPE has at least one child PBDOM_OBJECT.</td>
</tr>
<tr>
<td>False</td>
<td>The current PBDOM_DOCTYPE has no child PBDOM_OBJECTs.</td>
</tr>
</tbody>
</table>

Examples  TBD

Usage  If the PBDOM_DOCTYPE has at least one child PBDOM_OBJECT True is returned, and False if there are no children.

See also  TBD

---

**GetTextNormalize()**

Description  **Trivial Implementation**

The GetTextNormalize() method always returns an empty string because PBDOM_DOCTYPE cannot contain any text or CDATA DOM objects.

Syntax  \[ pbdom_doctype_name.GetTextNormalize() \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( pbdom_doctype_name )</td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
</tbody>
</table>

Return value  String

See also  TBD

---

**GetTextTrim()**

Description  **Trivial Implementation**

The GetTextTrim() method always returns an empty string because PBDOM_DOCTYPE cannot contain any text or CDATA DOM objects.

Syntax  \[ pbdom\_doctype\_name.GetTextTrim() \]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( pbdom_doctype_name )</td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
</tbody>
</table>

Return value  String

See also  TBD

174
IsAncestorOf()

Description
The IsAncestorOf() method determines if the current PBDOM_DOCTYPE is the ancestor of another PBDOM_OBJECT.

Syntax
`pbdom_doctype_name.IsAncestorOf(pbdom_object_ret)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_doctype_name</code></td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
<tr>
<td><code>pbdom_object_ret</code></td>
<td>A reference to a PBDOM_OBJECT to check against.</td>
</tr>
</tbody>
</table>

Return value
Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The current PBDOM_DOCTYPE is the ancestor of the referenced PBDOM_OBJECT.</td>
</tr>
<tr>
<td>False</td>
<td>The current PBDOM_DOCTYPE not the ancestor of the referenced PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

Examples
TBD

See also
TBD

SetParentObject()

Description
The SetParentObject() method sets the referenced PBDOM_OBJECT to be the parent of the current PBDOM_OBJECT, and so sets the DOCTYPE represented by this PBDOM_DOCTYPE to be the DOCTYPE represented by the referenced PBDOM_DOCUMENT.

Syntax
`pbdom_doctype_name.SetParentObject(pbdom_object_ref)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_doctype_name</code></td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
<tr>
<td><code>pbdom_object_ref</code></td>
<td>A reference to a PBDOM_OBJECT to be set as the parent of the current PBDOM_DOCTYPE.</td>
</tr>
</tbody>
</table>

Return value
PBDOM_OBJECT

Usage
The PBDOM_OBJECT that you set to be the parent of the current PBDOM_OBJECT, must be a PBDOM_DOCUMENT. If it is not, an exception is thrown. This method is exactly the same as the PBDOM_DOCTYPE::SetDocument() method.

A DOM DOCTYPE object:
• can have no owner document.
• can have an owner document but no parent node.
• that has an owner document as well as a parent node is the actual doctype of that document.

See also TBD

SetDocument()

Description
The SetDocument() method sets the owning PBDOM_DOCUMENT of the current PBDOM_DOCTYPE.

Syntax
```
pbdom_doctype_name.SetDocument(pbdom_document_ref)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_doctype_name</td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
<tr>
<td>pbdom_document_ref</td>
<td>The name of the referenced PBDOM_DOCUMENT to be set as the owner of this DOCTYPE.</td>
</tr>
</tbody>
</table>

Return value
PBDOM_DOCTYPE
The current PBDOM_DOCTYPE is modified to be the DOCTYPE of the referenced PBDOM_DOCUMENT.

Examples
TBD

Usage
A DOM DOCTYPE object:
• can have no owner document.
• can have an owner document but no parent node.
• that has an owner document as well as a parent node is the actual doctype of that document.

See also TBD

GetInternalSubset()

Description
The GetInternalSubset() method returns the internal subset data of the DOCTYPE.

Syntax
```
pbdom_doctype_name.GetInternalSubset()
```
### GetPublicID()

**Description**

The `GetPublicID()` method retrieves the public ID of an externally reference DTD declared in the DOCTYPE.

**Syntax**

```powershell
pbdom_doctype_name.GetPublicID()
```

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_doctype_name</code></td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
</tbody>
</table>

**Return value**

String

If no DTD is referenced, an empty string is returned.

**Examples**

If you have the following DTD declaration:

```xml
```

and you have the following PowerScript:

```powershell
pbdom_doctype_1 = pbdom_doc.GetDocType()
MessageBox ("DocType Public ID", pbdom_doctype_1.GetPublicID())
MessageBox ("DocType System ID", pbdom_doctype_1.GetSystemID())
```

then the returned string from `pbdom_doctype_1.GetPublicID()` is:

"-//MyCompany//DTD//EN"

and the returned string from `pbdom_doctype_1.GetSystemID()` is:

"http://mycompany.com/dtd/mydoctype.dtd"

**See also**

TBD
**PBDOM_DOCTYPE**

**GetSystemID()**

Description: The GetSystemID() method retrieves the system ID of an externally referenced DTD declared in the DOCTYPE.

Syntax: `pbdom_doctype_name.GetSystemID()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_doctype_name</code></td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
</tbody>
</table>

Return value: String

If no DTD is referenced, an empty string is returned.

Examples

If you have the following DTD declaration:

```xml
<!DOCTYPE Books SYSTEM
     "http://mycompany.com/dtd/mydoctype.dtd">
```

and you have the following PowerScript:

```powerscript
pbdom_doctype_1 = pbdom_doc.GetDocType()
MessageBox ("DocType System ID", pbdom_doctype_1.GetSystemID())
```

then the returned string from `pbdom_doctype_1.GetSystemID()` is:

```
"http://mycompany.com/dtd/mydoctype.dtd"
```

See also: TBD

**SetInternalSubset()**

Description: The SetInternalSubset() method sets the data for the internal subset of the PBDOM_DOCTYPE.

Syntax: `pbdom_doctype_name.SetInternalSubset()`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_doctype_name</code></td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
</tbody>
</table>

Return value: PBDOM_DOCTYPE

The SetInternalSubset() method modifies the current PBDOM_DOCTYPE with the new internal subset.

Examples

If you have the following DTD declaration:

```xml
<!DOCTYPE abc [<!ELEMENT abc (#PCDATA)> <!ELEMENT data (#PCDATA)> <!ELEMENT inner_data (#PCDATA)>]
```

and you have the following PowerScript:
strInternalSubset = pbdom_doc.GetDocType().GetInternalSubset()
strInternalSubset = strInternalSubset + "<!ELEMENT another_data (#PCDATA)>"
pbdom_doc.GetDocType().SetInternalSubset(strInternalSubset)
MessageBox("Get Internal Subset",
pbdom_doc.GetDocType().GetInternalSubset())

The returned string from
pbdom_doc.GetDocType().GetInternalSubset() in the MessageBox() call is:

"<!-- internal subset --> <!ELEMENT abc (#PCDATA)>
<!ELEMENT data (#PCDATA)> <!ELEMENT inner_data (#PCDATA)> <!ELEMENT another_data (#PCDATA)>"

The new ELEMENT declaration for “another_data” is included in the final internal subset.

See also TBD

SetPublicID()

Description
The SetPublicID() method sets the public ID of an externally referenced DTD.

Syntax
pbdom_doctype_name.SetPublicID(strPublicID)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_doctype_name</td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
<tr>
<td>strPublicID</td>
<td>A string that specifies the new public ID.</td>
</tr>
</tbody>
</table>

Return value
PBDOM_DOCTYPE

Examples
If you have the following DTD declaration:

`<!DOCTYPE abc [<!ELEMENT abc (#PCDATA)> <!ELEMENT data (#PCDATA)> <!ELEMENT inner_data (#PCDATA)>]>`

and you have the following PowerScript:

pbdom_doc.GetDocType().SetPublicID("-//MyCompany//DTD//EN")
MessageBox("Get Public ID", pbdom_doc.GetDocType().GetPublicID())

The returned string from pbdom_doc.GetDocType().GetPublicID() in the MessageBox() call is "-//MyCompany//DTD//EN", as specified. The final DOCTYPE definition in the document is:

`<!DOCTYPE abc PUBLIC "-//MyCompany//DTD//EN" [<!ELEMENT abc (#PCDATA)> <!ELEMENT data (#PCDATA)> <!ELEMENT inner_data (#PCDATA)>]>`
About Public ID
The PUBLIC ID is usually accompanied by a SYSTEM ID, so the DOCTYPE declaration in this example (with a PUBLIC ID but no SYSTEM ID) may be considered invalid by some parsers.

See also
TBD

SetSystemID()
Description
The SetSystemID() method sets the system ID of an externally referenced DTD.

Syntax
```
pbdom_doctype_name.SetSystemID(strSystemID)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_doctype_name</td>
<td>The name of your PBDOM_DOCTYPE.</td>
</tr>
<tr>
<td>strSystemID</td>
<td>A string that specifies the new system ID.</td>
</tr>
</tbody>
</table>

Return value
PBDOM_DOCTYPE

Examples
If you have the following DTD declaration:

```
<!DOCTYPE abc [<!ELEMENT abc (#PCDATA)> <!ELEMENT data (#PCDATA)> <!ELEMENT inner_data (#PCDATA)>]
```

And you have the following PowerScript:

```
pbdom_doc.GetDocType().SetSystemID
("http://www.sybase.com/dtd/datadef.dtd")
MessageBox ("Get System ID", pbdom_doc.GetDocType().GetSystemID())
```

the returned string from `pbdom_doc.GetDocType().GetSystemID()` in the `MessageBox()` call is "http://www.sybase.com/dtd/datadef.dtd", as specified. The final DOCTYPE definition in the document will be:

```
<!DOCTYPE abc SYSTEM "http://www.sybase.com/dtd/datadef.dtd" [<!ELEMENT abc (#PCDATA)> <!ELEMENT data (#PCDATA)> <!ELEMENT inner_data (#PCDATA)>]
```

See also
TBD
PB DOM_ CHARACTERDATA

Description

The PB DOM_ CHARACTERDATA class represents character-based content (not markup) within an XML document. It extends the PB DOM_ OBJECT class with a set of methods specifically for manipulating character data in the DOM.

The PB DOM_ CHARACTERDATA class is the parent class of three other PB DOM classes:

• PB DOM_ TEXT
• PB DOM_ CDATA
• PB DOM_ COMMENT

The PB DOM_ CHARACTERDATA class, like its parent class PB DOM_ OBJECT, is a “virtual” class (similar to a virtual C++ class) in that it is not expected to be directly instantiated and used.

Default or trivial functionalities

Some of the inherited methods from PB DOM_ OBJECT serve no meaningful objective and so only default or trivial functionalities will result. These are noted in the individual function descriptions that follow.

PB DOM_ CHARACTERDATA has the following functions:

• GetName()
• GetText()
• GetContent
• AddContent
• InsertContent
• RemoveContent
• SetContent
PBDM_CHARACTERDATA

- SetName

*Author comment:* Documentation of the functionality for PBDM_CHARACTERDATA is not complete.

### GetName()

**Description**
The `GetName()` method allows you to obtain the name of the current PBDM_CHARACTERDATA.

**Syntax**
```
pbdom_characterdata_name.GetName()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_characterdata_name</code></td>
<td>The name of your PBDM_CHARACTERDATA.</td>
</tr>
</tbody>
</table>

**Return value**
String

**Examples**
TBD

**Usage**
The returned string depends on the specific type of DOM object that is contained within PBDM_CHARACTERDATA. Because a PBDM_CHARACTERDATA is abstract and is not to be instantiated into an object of its own, there is no name returned as “#characterdata”.

The following table lists the return values based on the type of DOM Object contained within PBDM_CHARACTERDATA.

<table>
<thead>
<tr>
<th>DOM Object</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBDM_CDATA</td>
<td>“#data-section”</td>
</tr>
<tr>
<td>PBDM_COMMENT</td>
<td>“#comment”</td>
</tr>
<tr>
<td>PBDM_TEXT</td>
<td>“#text”</td>
</tr>
</tbody>
</table>

**See also**
TBD

### GetText()

**Description**
Calling the `GetText()` method allows you to obtain text data that is contained within the current PBDM_CHARACTERDATA.
### Syntax

```powershell
pbdom_characterdata_name.GetText()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_characterdata_name</code></td>
<td>The name of your PBDOM_CHARACTERDATA.</td>
</tr>
</tbody>
</table>

### Return value

String

The `GetText()` method returns the text of the current PBDOM_CHARACTERDATA-derived object.

### Usage

The following table lists the return values based on the type of DOM Object contained within PBDOM_CHARACTERDATA.

<table>
<thead>
<tr>
<th>DOM Object</th>
<th>Return Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBDOM_TEXT</td>
<td>The text data contained within the PBDOM_TEXT object itself.</td>
</tr>
<tr>
<td></td>
<td>For example, if you have the following element:</td>
</tr>
<tr>
<td></td>
<td><code>&lt;abc&gt;MY TEXT&lt;/abc&gt;</code></td>
</tr>
<tr>
<td></td>
<td>If you have a PBDOM_TEXT object to represent the TEXT NODE “MY TEXT”, then calling GetText() on the PBDOM_TEXT returns the string “MY TEXT”.</td>
</tr>
<tr>
<td>PBDOMCDATA</td>
<td>The string data that is contained within the CDATA section itself. For example, if you have the following CDATA:</td>
</tr>
<tr>
<td></td>
<td><code>&lt;![CDATA[ They’re saying “x &lt; y” &amp; that “z &gt; y” so I guess that means that z &gt; x ]]]&gt;</code></td>
</tr>
<tr>
<td></td>
<td>If we a PBDOMCDATA to represent the above CDATA section, then calling GetText() on it will return the string:</td>
</tr>
<tr>
<td></td>
<td>“They’re saying “x &lt; y” &amp; that “z &gt; y” so I guess that means that z &gt; x”.</td>
</tr>
<tr>
<td>PBDOM_COMMENT</td>
<td>The comment itself. For example, if you have the following comment:</td>
</tr>
<tr>
<td></td>
<td><code>&lt;!--This is a comment. --&gt;</code></td>
</tr>
<tr>
<td></td>
<td>Calling GetText() on the comment will return the string:</td>
</tr>
<tr>
<td></td>
<td>“This is a comment.”</td>
</tr>
</tbody>
</table>

### See also

TBD
**GetContent**

**Description**
Trivial Implementation
The GetContent() method always returns False and no PBDOM_OBJECTs are set into the input array.

**Syntax**

```
pbdom_characterdata_name.GetContent(pbdom_object_array)
```

**Return value**
Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>Successful</td>
</tr>
<tr>
<td>False</td>
<td>Unsuccessful</td>
</tr>
</tbody>
</table>

**See also**
TBD

---

**AddContent**

**Description**
Trivial Implementation
The AddContent() method does nothing with the input PBDOM_OBJECT and returns the current PBDOM_CHARACTERDATA.

**Syntax**

```
pbdom_characterdata_name.AddContent(pbdom_object_ref)
```

**Return value**
PBDOM_DOCTYPE

**See also**
TBD

---

**InsertContent**

**Description**
Trivial Implementation
The InsertContent() method does nothing with the input parameters and returns the current PBDOM_CHARACTERDATA without any modifications.
Syntax

```
pbdom_characterdata_name.InsertContent(pbdom_object_new, pbdom_object_ref)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_characterdata_name</td>
<td>The name of your PBDOM_CHARACTERDATA.</td>
</tr>
<tr>
<td>pbdom_object_new</td>
<td>The referenced name of a PBDOM_OBJECT you want to insert.</td>
</tr>
<tr>
<td>pbdom_object_ref</td>
<td>The referenced name of the PBDOM_OBJECT in front of which you want to insert the new PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

Return value

PBDOM_OBJECT

Usage

A PBDOM_CHARACTERDATA object does not contain any child nodes.

See also

TBD

---

**RemoveContent**

Description

**Trivial Implementation**

The `RemoveContent()` method does nothing with the referenced PBDOM_OBJECT. `False` is always returned.

Syntax

```
pbdom_characterdata_name.RemoveContent(pbdom_object_ref)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_characterdata_name</td>
<td>The name of your PBDOM_CHARACTERDATA.</td>
</tr>
<tr>
<td>pbdom_object_ref</td>
<td>The referenced name of a PBDOM_OBJECT you want to remove.</td>
</tr>
</tbody>
</table>

Return value

Boolean

The return value is always `False`.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The content was removed.</td>
</tr>
<tr>
<td>False</td>
<td>The content was not removed.</td>
</tr>
</tbody>
</table>

See also

TBD
**SetContent**

**Description**

**Trivial Implementation**
The `SetContent()` method does nothing with the input `PBDOM_OBJECT` array and returns the current `PBDOM_CHARACTERDATA` without modifications.

**Syntax**

```
pbdom_characterdata_name.SetContent(pbdom_object_array)
```

**Argument** | **Description**
--- | ---
`pbdom_characterdata_name` | The name of your `PBDOM_CHARACTERDATA`.
`pbdom_object_array` | An array of `PBDOM_OBJECTs`.

**Return value**

`PBDOM_CHARACTERDATA`

**See also**

TBD

---

**SetName**

**Description**

**Trivial Implementation**
The `SetName()` method does nothing with the input string and always returns `False`.

**Syntax**

```
pbdom_characterdata_name.SetName(strName)
```

**Argument** | **Description**
--- | ---
`pbdom_characterdata_name` | The name of your `PBDOM_CHARACTERDATA`.
`strName` | The new name you want to set for the root element that is declared by the current `PBDOM_CHARACTERDATA`.

**Return value**

`Boolean`

The current implementation always returns `False` for the `SetName()` method.

**Value** | **Description**
--- | ---
`True` | The of the root element declared by the current `PBDOM_CHARACTERDATA` name was changed.
`False` | The of the root element declared by the current `PBDOM_CHARACTERDATA` name was not changed.

**Examples**

TBD

**See also**

TBD
# PBDOM_ATTRIBUTE

**Description**

Some of the PBDOM_ATTRIBUTE functions have not yet been documented.

The PBDOM_ATTRIBUTE class defines the behavior for an XML attribute, modeled in PowerScript. Methods allow the user to obtain the value of the attribute as well as namespace information.

PBDOM_ATTRIBUTE has the following functions:

- `GetName`
- `GetText`
- `GetContent`
- `AddContent`
- `InsertContent`
- `RemoveContent`
- `setContent`
- `SetName`
- `Clone`
- `Equals`
- `GetObjectClass`
- `GetObjectClassString`
- `GetDocument`
- `Detach`
- `GetParentObject`
- `HasChildren`
- `GetTextNormalize`
- `GetTextTrim`
- `IsAncestorOf`
- `SetParentObject`
- `GetAttributeType`
- `GetBooleanValue`
PBOM_ATTRIBUTE

- GetDoubleValue
- GetRealValue
- GetIntValue
- GetLongValue
- GetUintValue
- GetDateValue
- GetDateTimeValue
- GetDecValue
- GetTimeValue
- GetUlongValue
- GetNamespace
- GetNamespacePrefix
- GetNamespaceUri
- GetQualifiedName
- SetAttributeType
- SetNamespace
- SetText
- SetBooleanValue
- SetDoubleValue
- SetRealValue
- SetIntValue
- SetLongValue
- SetUintValue
- SetDateValue
- SetDateTimeValue
- SetDecValue
- SetTimeValue
- SetUlongValue
**GetText**

Description: The GetText method returns the text value of the PBDOM_ATTRIBUTE object. The returned value includes all text within quotation marks.

Syntax: `pbdom_attribute_name.GetText()`

**Argument** | **Description**
--- | ---
`pbdom_attribute_name` | The name of the PBDOM_ATTRIBUTE.

Return value: String

Examples

**Example 1** The GetText method is invoked for the attribute name in the following element:

```xml
<abc ATTRIBUTE_1="My Attribute">
    The GetText method returns the following string:
    ATTRIBUTE_1
</abc>
```

**Example 2** The GetText method is invoked for the name of the Tower:Type attribute in the following element:

```xml
    The GetText method returns the following string:
    Type
```

Usage: For an XML attribute that appears in the form `[namespace_prefix]:[attribute_name]`, the local attribute name is `attribute_name`. Where the XML attribute has no namespace prefix, the local name is simply the attribute name.

See also: Use the GetNamespacePrefix method to obtain the namespace prefix for a PBDOM_ATTRIBUTE object. Use the GetQualifiedName method to obtain the fully qualified name for a PBDOM_ATTRIBUTE object.
**PBDOM_ATTRIBUTE**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_attribute_name</td>
<td>The name of the PBDOM_ATTRIBUTE.</td>
</tr>
</tbody>
</table>

**Return value**  
String

**Examples**  
The GetText method is invoked for the attribute in the following element:

```xml
<abc ATTRIBUTE_1="My Attribute">
```

The GetName method returns the following string:

```
My Attribute
```

**Usage**

**See also**

---

**GetContent**

Description  
The GetContent method always returns False. Nothing is returned in the input array.

**Syntax**

`pbdom_attribute_name.GetContent(pbdom_object_array)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_attribute_name</td>
<td>The name of the PBDOM_ATTRIBUTE.</td>
</tr>
<tr>
<td>pbdom_object_array</td>
<td>The referenced name of an array of PBDOM_OBJECTs that will receive PBDOM_OBJECTs.</td>
</tr>
</tbody>
</table>

**Return value**  
Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>False</td>
<td>Unsuccessful</td>
</tr>
</tbody>
</table>

**Examples**  
The GetContent method is invoked for the following element:

```xml
<abc ATTRIBUTE_1="My Attribute">
```

The GetContent method returns the boolean value False.

**Usage**

**See also**

---

190
AddContent

Description
The AddContent method does nothing with the input PBDOM_OBJECT and leaves the PBDOM_ATTRIBUTE unmodified.

Syntax
`pbdom_attribute_name.AddContent(pbdom_object_ref)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_attribute_name</code></td>
<td>The name of the PBDOM_ATTRIBUTE.</td>
</tr>
<tr>
<td><code>pbdom_object_ref</code></td>
<td>A reference to a PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

Return value
PBDOM_OBJECT
The PBDOM_ATTRIBUTE is not modified.

Examples

Usage
If the text value of the PBDOM_ATTRIBUTE is to be set, use the `SetText` method instead.

See also

InsertContent

Description
The InsertContent method does nothing with the input parameters and leaves the PBDOM_ATTRIBUTE unmodified.

Syntax
`pbdom_attribute_name.InsertContent(pbdom_object_new, pbdom_object_ref)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_attribute_name</code></td>
<td>The name of the PBDOM_ATTRIBUTE.</td>
</tr>
<tr>
<td><code>pbdom_object_new</code></td>
<td>A reference to a PBDOM_OBJECT.</td>
</tr>
<tr>
<td><code>pbdom_object_ref</code></td>
<td>A reference to a PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

Return value
PBDOM_OBJECT
The PBDOM_ATTRIBUTE is returned as a PBDOM_OBJECT.

Examples
**RemoveContent**

**Description**

The `RemoveContent` method does nothing with the input parameter and leaves the PBDOM_ATTRIBUTE unmodified.

**Syntax**

```
pbdom_attribute_name.RemoveContent(pbdom_object_ref)
```

**Return value**

Boolean

**Examples**

**Argument Description**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_attribute_name</code></td>
<td>The name of the PBDOM_ATTRIBUTE.</td>
</tr>
<tr>
<td><code>pbdom_object_ref</code></td>
<td>A reference to a PBDOM_OBJECT.</td>
</tr>
</tbody>
</table>

**SetContent**

**Description**

The `SetContent` method does nothing with the input parameter and leaves the PBDOM_ATTRIBUTE unmodified.

**Syntax**

```
pbdom_attribute_name.SetContent(pbdom_object_array)
```

**Argument Description**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_attribute_name</code></td>
<td>The name of the PBDOM_ATTRIBUTE.</td>
</tr>
<tr>
<td><code>pbdom_object_array</code></td>
<td>The referenced name of an array of PBDOM_OBJECTs.</td>
</tr>
</tbody>
</table>

**Return value**

PBDOM_OBJECT

The PBDOM_ATTRIBUTE is not modified.
Usage
See also

**SetName**

**Description**
The `SetName` method sets the local name of the PBDOM_ATTRIBUTE object.

**Syntax**
```plaintext
pbdom_attribute_name.SetName(strName)
```

**Argument**  | **Description**
---|---
`pbdom_attribute_name` | The name of the PBDOM_ATTRIBUTE.
`strName` | The new local name for the PBDOM_ATTRIBUTE.

**Return value**
Boolean

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>True</td>
<td>The local name of the PBDOM_ATTRIBUTE has been changed.</td>
</tr>
<tr>
<td>False</td>
<td>The local name of the PBDOM_ATTRIBUTE has not been changed.</td>
</tr>
</tbody>
</table>

**Examples**

**Usage**
See also

**Clone**

**Description**
The `Clone` method creates a clone of the PBDOM_ATTRIBUTE object.

**Syntax**
```plaintext
pbdom_attribute_name.Clone()
```

**Argument**  | **Description**
---|---
`pbdom_attribute_name` | The name of the PBDOM_ATTRIBUTE.

**Return value**
PBDOM_OBJECT

The PBDOM_ATTRIBUTE clone is returned as a PBDOM_OBJECT.

**Examples**

**Example 1** Consider the following element `<abc>` with an attribute named "My_Attr" and a value of "An Attribute":

---

**PowerBuilder Document Object Model**

193
This element can be cloned with the Clone method as follows:

```csharp
pbdom_attr = pbdom_obj.GetAttribute("My_Attr")
pbdom_attr_clone = pbdom_attr.Clone()
MessageBox ("Attribute Text", pbdom_attr_clone.GetText())
```

The GetAttribute method of the PBDOM_ELEMENT class returns a reference to the "My_Attr" attribute, which is subsequently cloned with the Clone method of the PBDOM_ATTRIBUTE class. A message box titled "Attribute Text" reports the attribute text "An_Attribute".

**Example 2** An attribute can also be cloned as follows:

```csharp
pbdom_attr = Create PBDOM_Attribute
pbdom_attr.SetName("My_Attribute")
pbdom_attr_clone = pbdom_attr.Clone()
```

The SetName method names the newly created PBDOM_ATTRIBUTE, which is subsequently cloned with the Clone method. Note that the new PBDOM_ATTRIBUTE object pbdom_attr must be assigned a name before it can be cloned. Otherwise, an exception is raised.

**Usage**

A newly created PBDOM_ATTRIBUTE cannot be cloned until assigned a name using the SetName method.

**See also**

### Equals

**Description**  
The Equals method tests for equality between the supplied PBDOM_OBJECT and the PBDOM_ATTRIBUTE from which the method is invoked.

**Syntax**

```csharp
pbdom_attribute_name.Equals(pbdom_object_ref)
```

**Argument** | **Description**
---|---
`pbdom_attribute_name` | The name of the PBDOM_ATTRIBUTE.
`pbdom_object_ref` | A reference to a PBDOM_OBJECT.

**Return value**

**Value** | **Description**
---|---
`True` | The PBDOM_ATTRIBUTE is equivalent to the referenced PBDOM_OBJECT.
PowerBuilder Document Object Model

Examples

Example 1 The following code uses the Equals method to test for equivalence between a referenced PBDOM_OBJECT and a cloned object.

```
pbdom_attr = Create PBDOM_Attribute
pbdom_attr.SetName("My_Attr")

pbdom_attr_clone = pbdom_attr.Clone()

if (pbdom_attr_clone.Equals(ref pbdom_attr)) then
  MessageBox ("Equals", "Yes")
else
  MessageBox ("Equals", "No")
end if
```

The SetName method names the newly created PBDOM_ATTRIBUTE, which is subsequently cloned with the Clone method. The Equals method tests for equality between the cloned PBDOM_ATTRIBUTE `pbdom_attr_clone` and the referenced PBDOM_OBJECT `pbdom_attr`. A message box displays the result returned from the Equals method.

Note here that because a cloned object is never equivalent to the object from which it is cloned, the Equals method returns False:

Example 2 The following code uses the Equals method to test for equivalence between two cloned objects.

```
pbdom_attr = Create PBDOM_Attribute
pbdom_attr.SetName("My_Attr")

pbdom_attr_clone = pbdom_attr.Clone()

pbdom_attr_2 = pbdom_attr_clone

if (pbdom_attr_clone.Equals(ref pbdom_attr_2)) then
  MessageBox ("Equals", "Yes")
else
  MessageBox ("Equals", "No")
end if
```

A newly created PBDOM_ATTRIBUTE is cloned, and a reference to this clone is assigned to `pbdom_attr_2`. The Equals method tests for equality between the cloned PBDOM_ATTRIBUTE `pbdom_attr_clone` and the reference to it, `pbdom_attr_2`. A message box displays the result returned from the Equals method.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
</table>
| False | The PBDOM_ATTRIBUTE is not equivalent to the referenced PBDOM_OBJECT.
PBDO

**PBDO**

Here the `Equals` method returns `True`.

**Usage**

Note that the clone of a `PBDO` is not considered equal to itself.

**See also**

**GetObjectClass**

**Description**

The `GetObjectClass` method returns a long integer value indicating the class of the `PBDO`. A value of 5 indicates a `PBDO` class.

**Syntax**

```c
pbdom_attribute_name.GetObjectClass()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_attribute_name</code></td>
<td>The name of the <code>PBDO</code>.</td>
</tr>
</tbody>
</table>

**Return value**

Long

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>This <code>PBDO</code> belongs to the <code>PBDO</code> class.</td>
</tr>
</tbody>
</table>

**Examples**

The `GetObjectClass` method returns a value specific to the class of the object from which the method is invoked.

```c
PBDO pbdom_obj

pbdom_obj = Create PBDO
MessageBox ("Class", string(pbdom_obj.GetObjectClass()))
```

This example illustrates polymorphism: `pbdom_obj` is declared as `PBDO` but instantiated as `PBDO`. A message box returns the result of the `GetObjectClass` method invoked for `PBDO`. Here the result is 5, indicating that `pbdom_obj` is a `PBDO` object.

**Usage**

This method can be used for diagnostic purposes to dynamically determine the actual type of a `PBDO` at runtime.
See also

**GetObjectClassString**

The `GetObjectClassString` method returns a string indicating the class of the PBDOM_OBJECT. A value of `pbdom_attribute` indicates a PBDOM_ATTRIBUTE class.

**Syntax**

```plaintext
pbdom_attribute_name.GetObjectClassString()
```

**Argument**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_attribute_name</code></td>
<td>The name of the PBDOM_ATTRIBUTE.</td>
</tr>
</tbody>
</table>

**Return value**

String

**Value**

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbdom_attribute</code></td>
<td>This PBDOM_OBJECT belongs to the PBDOM_ATTRIBUTE class.</td>
</tr>
</tbody>
</table>

**Examples**

The `GetObjectClass` method returns a string specific to the class of the object from which the method is invoked.

```plaintext
PBDOM_OBJECT pbdom_obj

pbdom_obj = Create PBDOM_ATTRIBUTE
MessageBox ("Class", pbdom_obj.GetObjectClassString())
```

This example illustrates polymorphism: `pbdom_obj` is declared as PBDOM_OBJECT but instantiated as PBDOM_ATTRIBUTE. A message box returns the result of the `GetObjectClassString` method invoked for PBDOM_ATTRIBUTE. Here the result is `pbdom_attribute`, indicating that `pbdom_obj` is a PBDOM_ATTRIBUTE object.

**Usage**

This method can be used for diagnostic purposes to dynamically determine the actual type of a PBDOM_OBJECT at runtime.

See also

**GetDocument**

The `GetDocument` method returns the PBDOM_DOCUMENT object that owns the PBDOM_ATTRIBUTE.
**PBDOM_ATTRIBUTE**

Syntax

```
pbdom_attribute_name.GetDocument()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_attribute_name</td>
<td>The name of the PBDOM_ATTRIBUTE.</td>
</tr>
</tbody>
</table>

Return value

PBDOM_OBJECT

The PBDOM_OBJECT returned is the PBDOM_DOCUMENT that owns the PBDOM_ATTRIBUTE object from which the GetDocument method is invoked.

A return value of NULL indicates the PBDOM_ATTRIBUTE object is not owned by any PBDOM_DOCUMENT.

Examples

The GetDocument method can be used to identify the PBDOM_DOCUMENT object that owns a PBDOM_ATTRIBUTE object.

```c
PBDOM_Builder pbdombuilder_new
pbdom_document pbdom_doc
pbdom_document pbdom_doc_2
PBDOM_ATTRIBUTE pbdom_attr
string strXML = "<abc My_Attr="My Attribute Value"></data>Data
</data></abc>"

TRY
pbdombuilder_new = Create PBDOM_Builder
pbdom_doc = pbdombuilder_new.Build (strXML)

pbdom_attr = pbdom_doc.GetRootElement().GetAttribute("My_Attr")
pbdom_doc_2 = pbdom_attr.GetDocument()

if (pbdom_doc.Equals(pbdom_doc_2)) then
    MessageBox ("Equals", "pbdom_doc equals pbdom_attr.GetDocument()")
end if

Destroy pbdombuilder_new

CATCH (PBDOM_Exception except)
MessageBox ("Exception Occurred", except.Text)
END TRY
```

Here, the Build method is used to create the following PBDOM_DOCUMENT object, pbdom_doc, using an XML string:

```
<abc My_Attr="My Attribute Value">
<data>Data </data>
</abc>
```
The GetAttribute method is used to obtain the attribute from the root element of pbdom_doc. This value is assigned to the PBDOM_ATTRIBUTE object pbdom_attr. The GetDocument method is used to obtain the name of pbdom_doc, which owns pbdom_attr. The result of the GetDocument method is assigned to the PBDOM_DOCUMENT object pbdom_doc_2. Then pbdom_doc_2 is compared to pbdom_doc using the Equals method, and the result is displayed in the following message box:

Usage
See also

Detach

Description
The Detach method detaches a PBDOM_ATTRIBUTE from its parent PBDOM_OBJECT, a PBDOM_ELEMENT.

Syntax

```
pbdom_attribute_name.Detach()
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_attribute_name</td>
<td>The name of the PBDOM_ATTRIBUTE.</td>
</tr>
</tbody>
</table>

Return value
PBDOM_OBJECT
The returned PBDOM_OBJECT is the PBDOM_ATTRIBUTE object detached from its parent object.
If the PBDOM_ATTRIBUTE object has no parent, the Detach method does nothing.

Examples
The Detach method can be used to manipulate an XML document as follows:

```powershell
PBDOM_BUILDER pbdombuilder_new
PBDOM_DOCUMENT pbdom_doc
PBDOM_ATTRIBUTE pbdom_attr
PBDOM_ELEMENT pbdom_elem
string strXML = "<abc My_Attr="My Attribute Value="">\<data>Data</data></abc>"

TRY
pbdombuilder_new = Create PBDOM_Builder
pbdom_doc = pbdombuilder_new.Build (strXML)

pbdom_attr = pbdom_doc.GetRootElement().GetAttribute("My_Attr")
pbdom_attr.Detach()
```
Here, the Build method is used to create the following PBDOM_DOCUMENT object, pbdom_doc, using an XML string:

```xml
<abc My_Attr="My Attribute Value">
  <data>Data</data>
</abc>
```

The GetAttribute method is used to obtain the attribute from the root element of pbdom_doc. This value is assigned to the PBDOM_ATTRIBUTE object pbdom_attr. The pbdom_attr object is detached from its parent element, and the <data> element is obtained from pbdom_doc using the GetChildElement method. The <data> element is then assigned to the PBDOM_ELEMENT object pbdom_elem. The attribute assigned to pbdom_attr is assigned to pbdom_elem, yielding the following modified pbdom_doc:

```xml
<abc>
  <data My_Attr="My Attribute Value">Data</data>
</abc>
```

### GetParentObject

**Description**

The GetParentObject method returns the parent PBDOM_ELEMENT object for the PBDOM_ATTRIBUTE.

The GetParentObject method returns the parent PBDOM_ELEMENT of this PBDOM_ATTRIBUTE. If there is no parent, a NULL is returned.

**Syntax**

```
pbdom_attribute_name.GetParentObject() 
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbdom_attribute_name</td>
<td>The name of the PBDOM_ATTRIBUTE.</td>
</tr>
</tbody>
</table>
Return value

PBDM_OBJECT

The PBM_OBJECT returned is the parent PBDOM_ELEMENT of the
PBDM_ATTRIBUTE object from which the GetParentObject method is
invoked.

A return value of NULL indicates the PBDOM_ATTRIBUTE object has no
parent PBDOM_ELEMENT.

Examples

The GetParentObject method can be used to identify a parent
PBDM_ELEMENT and display information about it.

PBDM_BUILDER pbdombuilder_new
PBDM_DOCUMENT pbdom_doc
PBDM_ATTRIBUTE pbdom_attr
PBDM_ELEMENT pbdom_elem
string strXML = "<abc  My_Attr="My Attribute
Value"> <data>Data</data></abc>"

TRY
pbdombuilder_new = Create_Builder
pbdom_doc = pbdombuilder_new.Build(strXML)
pbdom_attr = pbdom_doc.GetRootElement().GetAttribute("My_Attr")
MessageBox ("pbdom_attr parent name",
pbdom_attr.SetParentObject().GetName())
pbdom_attr.Detach()
pbdom_elem = pbdom_doc.GetRootElement().GetChildElement("data")
pbdom_elem.SetParentAttribute (pbdom_attr)
MessageBox ("pbdom_attr parent name",
pbdom_attr.SetParentObject().GetName())
Destroy pbdombuilder_new
Destroy pbdom_doc
CATCH (PBDM_Exception except)
MessageBox ("Exception Occurred", except.Text)
END TRY

Here, the Build method is used to create the following PBM_DOCUMENT
object, pbdom_doc, using an XML string:

<abc  My_Attr="My Attribute Value">
<data>Data</data>
</abc>

************************
The `Detach` method can be used to manipulate an XML document as follows:

```csharp
PBDOM_BUILDER pbdombuilder_new
PBDOM_DOCUMENT pbdom_doc
PBDOM_ATTRIBUTE pbdom_attr
PBDOM_ELEMENT pbdom_elem

string strXML = "<abc My_Attr="My Attribute Value"> <data>Data</data> </abc>"

TRY
pbdombuilder_new = Create PBDOM_Builder
pbdom_doc = pbdombuilder_new.Build (strXML)

pbdom_attr = pbdom_doc.GetRootElement().GetAttribute("My_Attr")
pbdom_attr.Detach()

pbdom_elem = pbdom_doc.GetRootElement().GetChildElement("data")
pbdom_elem.SetAttribute (pbdom_attr)

Destroy pbdombuilder_new
Destroy pbdom_doc

CATCH (PBDOM_Exception except)

MessageBox ("Exception Occurred", except.Text)

END TRY
```

Here, the `Build` method is used to create the following `PBDOM_DOCUMENT` object, `pbdom_doc`, using an XML string:

```
<abc My_Attr="My Attribute Value">
<data>Data</data>
</abc>
```

The `GetAttribute` method is used to obtain the attribute from the root element of `pbdom_doc`. This value is assigned to the `PBDOM_ATTRIBUTE` object `pbdom_attr`. The `pbdom_attr` object is detached from its parent element, and the `<data>` element is obtained from `pbdom_doc` using the `GetChildElement` method. The `<data>` element is then assigned to the `PBDOM_ELEMENT` object `pbdom_elem`. The attribute assigned to `pbdom_attr` is assigned to `pbdom_elem`, yielding the following modified `pbdom_doc`:

```
<abc
<data My_Attr="My Attribute Value">Data</data>
</abc>
```
PBDOM_ELEMENT

Description

Note The PBDOM_ELEMENT functions have not yet been documented.

PBDOM_ELEMENT has the following functions:

- GetName
- GetText
- GetContent
- AddContent
- InsertContent
- RemoveContent
- SetContent
- SetName
- Clone
- Equals
- GetObjectClass
- GetObjectClassString
- GetDocument
- Detach
- GetParentObject
- HasChildren
- GetTextNormalize
- GetTextTrim
- IsAncestorOf
- SetParentObject
• AddContent
• SetText
• SetAttribute
• SetAttribute (overloaded)
• SetAttribute (overloaded)
• SetAttributes
• HasChildElements
• IsRootElement
• AddNamespaceDeclaration
• GetAdditionalNamespaces
• GetAttribute
• GetAttribute (overloaded)
• GetAttributes
• GetAttributeValue
• GetAttributeValue (overloaded)
• GetAttributeValue (overloaded)
• GetAttributeValue (overloaded)
• GetChildElement
• GetChildElement (overloaded)
• GetChildElements
• GetChildElements (overloaded)
• GetChildElements (overloaded)
• GetNamespace
• GetNamespace (overloaded)
• GetNamespacePrefix
• GetNamespaceUri
• GetQualifiedName
• RemoveAttribute
• RemoveAttribute (overloaded)
• RemoveAttribute (overloaded)
• RemoveChildElement
• RemoveChildElement (overloaded)
• RemoveChildElements
• RemoveChildElements (overloaded)
• RemoveChildElements (overloaded)
• RemoveNamespaceDeclaration
• SetDocument
• SetNamespace
Building an EJB Client

This section describes building a PowerBuilder client for an Enterprise JavaBeans component running on an application server. It contains the following topics:

- About building an EJB client
- Adding pbejbclient90.pbd to the library search path
- Generating EJB proxy objects
- Creating a Java VM
- Connecting to the server
- Invoking component methods
- Exception handling
- Client-managed transactions
- Debugging the client
- Reference information

About building an EJB client

A PowerBuilder application can act as a client to an EJB 1.1 or 2.0 component running on an application server that is J2EE compliant. Tested servers include Sybase EAServer 4.1, IBM WebSphere 4.0, BEA WebLogic 6.1, and BEA WebLogic 7.

To connect to the server and communicate with the EJB component, clients use a set of classes implemented in a DLL file, pbejbclient90.dll. To use this DLL, you must place it in the application’s path, and you must add the pbejbclient90.pbd file to the client application’s library search path. The PBD acts as a wrapper for the DLL, enabling the PowerBuilder client to use the classes in the DLL as though they were PowerBuilder custom class user objects.
About building an EJB client

These files were developed using the PowerBuilder Native Interface (PBNI). For information about PBNI, see the documentation provided with this beta release. However, you do not need to know anything about PBNI to create EJB clients.

About EJB proxy objects

The PowerBuilder client uses local proxy objects for the EJB component to delegate calls to methods on the remote EJB component. Each EJB component has two proxy objects in the client application—one for the home interface and one for the remote interface. For example, an EJB component named Cart has two proxies, CartHome and Cart. If the EJB component uses other Java classes as method arguments or return values, or if it uses exceptions, proxies for those classes are also generated.

Overview of the process

To build an EJB client, you need to complete the following steps:

1. Create a workspace and a PowerScript target
2. Add pbejbclient90.pbd to the target’s library search path
3. Create a project for building proxy objects
4. Build the project to generate the proxy objects
5. Create the windows required to implement the user interface of the client application
6. Create a Java VM
7. Establish a connection to the server and look up the EJB
8. Create an instance of the EJB component and call one or more component methods from the client
9. Test and debug the client

Path and classpath requirements

To build EJB client applications, you must add the following directories to the system PATH environment variable, where %JDK% is the path to the Sun JDK installation (for EAServer and WebLogic) or the IBM JDK installation (for WebSphere):

- %JDK%\bin
- %JDK%\jre\bin\classic

Note To develop EJB client applications, you must have JDK 1.2.2 or JDK 1.3.1 installed on the development computer. Deployed EJB clients require the equivalent Java Runtime Environment (JRE).
Either your system CLASSPATH environment variable or the classpath argument to createJavaVM must include the following files or directories:

- **pbejbclient90.jar**
  This file is installed in your Sybase\Shared\PowerBuilder folder.

- **The stub files for the components for which you want to create proxies and the stub files for any classes used by the components**
  If the stub files are in a JAR file, add the JAR file to the class path. If the stub files are in a directory and the fully-qualified name of the EJB is `packagename.beanname`, add the directory that contains `packagename` to the class path.

- **%JDK%\jre\lib\rt.jar**
  To ensure that the Java VM can locate the classes in this JAR file, you should add it to the CLASSPATH environment variable.

- **For EAServer, easclient.jar and easj2ee.jar**
  These files are installed in `Sybase\EAServer\java\lib` on the server.

- **For WebLogic, weblogic.jar**
  This file is installed in `wlserver6.1\lib` or `weblogic700\server\lib` on the server.

- **For WebSphere, JAR files installed on the server in websphere\appserver\lib**
  For detailed information about the files required on the client by each application server, see the documentation for the server.

### Adding pbejbclient90.pbd to the library search path

The `pbejbclient90.dll` and `pbejbclient90.pbd` files are placed in the `Shared\PowerBuilder` directory when you install PowerBuilder. When you are building an EJB client application, you do not need to copy `pbejbclient90.dll` to another location, but you do need to deploy it with the client executable in a directory in the application’s search path.
To add `pbejbclient90.pbd` to the application’s search path:

1. Right-click the client target in the System Tree and select Properties from the pop-up menu.
2. Select Browse on the Library List page and browse to the Shared\PowerBuilder directory.
3. Select PB Dynamic Libraries from the Files of Type drop-down list, select `pbejbclient90.pbd`, and close the dialog boxes.

After you add `pbejbclient90.pbd`, the following objects display in the System Tree:

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EJBConnection</td>
<td>Used to connect to an EJB server and locate an EJB.</td>
</tr>
<tr>
<td>EJBHandle</td>
<td>Maps to the <code>javax.ejb.Handle</code> interface. Used as the return type of the <code>getHandle</code> method of EJBObject.</td>
</tr>
<tr>
<td>EJBHome</td>
<td>Maps to the <code>javax.ejb.EJBHome</code> interface. The base class of the EJB home interface.</td>
</tr>
<tr>
<td>EJBHomeHandle</td>
<td>Maps to the <code>javax.ejb.HomeHandle</code> interface. Used as the return type of the <code>getHomeHandle</code> method of EJBHome.</td>
</tr>
<tr>
<td>EJBMetaData</td>
<td>Maps to the <code>javax.ejb.EJBMetaData</code> interface. Used as the return type of the <code>getEJBMetaData</code> method of EJBHome.</td>
</tr>
<tr>
<td>EJBObject</td>
<td>Maps to the <code>javax.ejb.EJBObject</code> interface. The base class of the EJB remote interface.</td>
</tr>
<tr>
<td>EJBTransaction</td>
<td>Maps to the <code>javax.transaction.UserTransaction</code> interface. Used to control transactions from the EJB client.</td>
</tr>
<tr>
<td>JavaVM</td>
<td>Used to create an instance of the Java VM.</td>
</tr>
</tbody>
</table>

---

**Generating EJB proxy objects**

**Using an EJB Proxy project**

To create a new EJB Client Proxy project, select the EJB Client Proxy icon or the EJB Client Proxy Wizard icon from the Projects page of the New dialog box.

The EJB Client Proxy icon opens the Project painter for EJB proxies so you can create a project, specify options, and build the proxy library.
To create an EJB Client Proxy project in the Project painter:

1. Double-click the EJB Client Proxy icon on the Projects page of the New dialog box.

2. To specify the EJB, select Edit> Select Objects and enter the fully-qualified name of the component’s remote interface in the text box, for example com.sybase.jaguar.sample.svu.SVULoginImpl or portfolio.MarketMaker.

3. Enter the path of the directory or JAR file that contains the EJB’s stubs in the Classpath box and click OK.

   If the stub files are in a directory and the fully-qualified name of the EJB is packagename.beanname, enter the directory that contains packagename.

4. To specify the PBL where the proxy objects should be stored, select Edit> Properties and browse to the location of a library in the target’s library list or enter a new library name.

   You can specify an optional prefix that is added to the beginning of each generated proxy name. Adding a prefix makes it easier to identify the proxies associated with a specific EJB. However, it will result in extra copies of some common proxies, such as proxies for exception objects, if you create proxies for multiple EJB components in the same target.

   **Note** The second tab in this dialog box is inactive and will be removed in a later beta release.

5. Close the dialog box and select File> Save to save the project.

   The new project lists the EJB component for which a proxy will be generated and specifies the name of the output library that will contain the generated proxy objects.

   The EJB Client Proxy Wizard helps you create the project.

To create an EJB Client Proxy project using the wizard:

1. Double-click the EJB Client Proxy Wizard icon on the Projects page of the New dialog box and click Next on the first page of the wizard.

2. Select a library in which to store the project object and click Next.

3. Specify a name and optional description for the project and click Next.
Generating EJB proxy objects

4 Enter the fully-qualified name of the component’s remote interface in the text box, for example com.sybase.jaguar.sample.svu.SVULoginImpl.

5 Browse to select the JAR file that contains the EJB’s stubs or the directory that contains the stub package.

   If the stub files are in a directory and the fully-qualified name of the EJB is packagename.beanname, enter the directory that contains packagename.

6 Specify an optional prefix that is added to the beginning of each generated proxy name and click Next.

   Adding a prefix makes it easier to identify the proxies associated with a specific EJB. However, it will result in extra copies of some common proxies, such as proxies for exception objects, if you create proxies for multiple EJB components in the same target.

7 Browse to select an existing library or enter the name of a new library, then click Next and Finish.

   The proxy objects are generated and stored in this library, which must be added to the target’s library list.

After the wizard has created the project, you can use the Project painter to modify your project settings.

Building proxies

Whether you create the EJB Proxy project using the wizard or the painter, the final step is to build the proxy objects. To do so, click the Build icon on the painter bar or select Design->Deploy Project from the menu bar.
Using the **ejb2pb90** tool

You can also use the ejb2pb90 command-line tool to generate proxies. The tool generates `.srx` files for the home and remote interfaces of the EJB you specify, as well as a text file that lists the proxies generated. If the EJB depends on other classes, the tool generates `.srx` files for them. The home and remote classes for the EJB and any dependent classes must be in the class path that you specify.

The syntax is:

```
ejb2pb90 [-classpath pathlist ] EJBName [ prefix ]
```

If you specify the optional `prefix`, it is added to the beginning of the generated proxy name. If the `pathlist` argument contains spaces, for example `D:\Program Files`, the `pathlist` must be enclosed in quotes.

After generating the proxies, you import them into your target by selecting the library that contains the client, selecting Import from its pop-up menu, and selecting the `.srx` files from the dialog box that displays. The order in which you import `.srx` files is significant—you cannot import proxies that depend on other classes until you have imported the proxies for those classes.
Generating EJB proxy objects

Viewing the generated proxies

The generated proxies display in the System Tree. You can expand the proxy nodes to display the signatures of the methods on the home and remote interfaces.

In addition to proxies for the EJB, the project creates proxies for exception objects. For more information, see “Exception handling” on page 221.

You cannot open proxy objects in a painter or the Source editor, but you can examine the source of the object.

❖ To export the source of a proxy object:
1. Do one of the following:
   • In the System Tree, select Export from the proxy object’s pop-up menu.
   • In the Library painter, select the proxy object, then select Entry>LibraryItem>Export.
2. Click Save to save the source to a file with the extension .srx.
Example

The following are the exported sources of a simple EJB that has only a create method on its home interface and a single method, displayMessage, on its remote interface. This is HelloEJBHome:

```
$PBExportHeader$helloejbhome.srx
$PBExportComments$Proxy for EJB.
global type HelloEjbHome from ejbhome
dtype

global HelloEjbHome HelloEjbHome

forward prototypes
public:
  function HelloEjb create() alias for
  "create,()LejbSample/HelloEjb;"
end prototypes
```

This is HelloEJB (the proxy for the remote interface):

```
$PBExportHeader$helloejb.srx
$PBExportComments$Proxy for EJB.
global type HelloEjb from NonVisualObject
dtype

global HelloEjb HelloEjb

forward prototypes
public:
  function string displayMessage() alias for
  "displayMessage,()Ljava/lang/String;"
end prototypes
```

Aliases for Java methods

PowerBuilder is case insensitive, whereas Java is case sensitive. To ensure that PowerScript code can call Java methods correctly, each method in the proxy uses an alias. The string that follows alias for contains the name and the signature of the corresponding Java method in case-sensitive mode.

Datatype mappings

The EJB Proxy generator maps datatypes between Java and PowerBuilder as shown in the following table:

<table>
<thead>
<tr>
<th>Java type</th>
<th>PowerBuilder type</th>
</tr>
</thead>
<tbody>
<tr>
<td>short</td>
<td>Integer</td>
</tr>
<tr>
<td>int</td>
<td>Long</td>
</tr>
</tbody>
</table>
### Arrays of arrays

Unlike Java, PowerBuilder does not support unbounded multidimensional arrays. If a Java method takes an array of arrays as a parameter, the corresponding PowerBuilder proxy method takes a parameter of type `Any`. To call the method in PowerBuilder, declare a PowerBuilder array with the same dimensions as the Java array, and pass the array as the parameter.

### Creating a Java VM

Before calling an EJB component, you need to create a Java VM using the `createJavaVM` method of the JavaVM class. The first argument is a string that specifies a class path to be appended to the path defined in the system `CLASSPATH` variable. The second argument to `createJavaVM` is a boolean that specifies whether debug information is written to a text file. See “Debugging the client” on page 224.

The JavaVM class also has a `GetJavaVMVersion` method that you can use to determine which version of the Java VM is running.

The JavaVM should be a global variable for the client application. It should not be destroyed explicitly.

<table>
<thead>
<tr>
<th>Java type</th>
<th>PowerBuilder type</th>
</tr>
</thead>
<tbody>
<tr>
<td>long</td>
<td>LongLong</td>
</tr>
<tr>
<td>float</td>
<td>Real</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>byte</td>
<td>Char</td>
</tr>
<tr>
<td>char (16-bit unsigned)</td>
<td>Char</td>
</tr>
<tr>
<td>java.lang.String</td>
<td>String</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>java.util.Date</td>
<td>Datetime</td>
</tr>
</tbody>
</table>

**Array of primitive type**
- Parameters: Array of primitive type
- Return values: Any

**Array of java.lang.String or java.util.Date objects**
- Parameters: Array of string, date, or time
- Return values: Any

**Array of arrays**
- Any (see "Arrays of arrays" next)

**Java class arguments or return values**
- Java class proxies

**Other**
- Powerobject
To create and run a Java VM, the system PATH environment variable must contain the bin and jre\bin\classic directories from a JDK installation, and the system CLASSPATH environment variable or the classpath argument to createJavaVM must contain the rt.jar file from the JDK installation. For more information, see “Path and classpath requirements” on page 208.

Example
This example demonstrates the creation of an instance of the Java VM for JDK 1.3 on EAServer:

```powershell
// global variables javavm g_jvm, boolean gb_jvm_started
boolean isdebug
string classpath

if NOT gb_jvm_started then
  //create JAVAVM
  g_jvm = create javavm
  classpath = "D:\hex_test\pbejbcl;" + 
    "D:\Sybase\Shared\Sun\jdk\jdk1.3\jre\lib\rt.jar;" + 
    "D:\Sybase\Shared\PowerBuilder\pbejbclient90.jar;"

  isdebug = true
  choose case g_jvm.createJavaVM(classpath, isdebug)
    case 0
      gb_jvm_started = true
    case -1
      MessageBox("Error", "Failed to load JavaVM")
    case -2
      MessageBox("Error", "Failed to load EJBLocator")
  end choose
end if
```

Connecting to the server
The EJBConnection class is used to connect to an EJB server and locate an EJB. It has four methods: ConnectToServer, DisconnectServer, Lookup, and GetEJBTransaction.

To establish a connection to the server, you need to execute the PowerScript statements required to perform these operations:

1. Declare an instance of the EJBConnection class.
2. Set properties for the EJBConnection object.
Connecting to the server

3 Use the CREATE statement to instantiate the EJBConnection object.
4 Invoke the ConnectToServer method to establish a connection to the server.
5 Check for errors.

Classpath requirements

To connect to the application server and create an EJB object, the system CLASSPATH environment variable or the classpath argument of createJavaVM must contain the location of the EJB stub files, either a directory or a JAR file. JAR files used by the server must also be available on the client computer and added to the class path. For more information, see “Path and classpath requirements” on page 208.

Setting the initial context

The string used to establish the initial context depends on the EJB server. The following table shows the string values for the servers tested for this beta release:

<table>
<thead>
<tr>
<th>Server</th>
<th>INITIAL_CONTEXT_FACTORY value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAServer 3.6.1, 4.1.x</td>
<td>com.sybase.ejb.InitialContextFactory</td>
</tr>
<tr>
<td>WebLogic 6.1, 7</td>
<td>weblogic.jndi.WLInitialContextFactory</td>
</tr>
<tr>
<td>WebSphere 4.0</td>
<td>com.ibm.websphere.naming.WsnInitialContextFactory</td>
</tr>
</tbody>
</table>

Example

The following script shows a connection to EAServer. It sets connection properties to create an initial context, to identify the host name and port number of the server, and to identify the user ID and password. Then, the script creates an instance of the EJBConnection object, invokes the ConnectToServer method to establish a connection to the server, and checks for errors:

```
ejbconnection conn
string properties[]

properties[3]="javax.naming.Context.SECURITY_PRINCIPAL=jagadmin"

conn = CREATE ejbconnection
TRY
    conn.connectToServer(properties)
CATCH (exception e)
    MessageBox("exception", e.getMessage())
END TRY
```
Invoking component methods

After a connection to the server has been established and a proxy object or objects created, the client application can begin using the EJB components. To invoke an EJB component method, you need to execute the PowerScript statements required to perform these operations:

1. Use the lookup method of EJBConnection to access the component’s home interface.

2. Invoke the create or findByPrimaryKey method on the home interface to create or find an instance of the component and get a reference to the component’s remote interface.

3. Invoke the business methods on the remote interface.

To enable the client to locate the EJB, the phejbclient90.jar file must be in the system CLASSPATH environment variable, or in the classpath argument passed to the CreateJavaVM method.

The lookup method takes three string arguments: the name of the proxy, the JNDI name of the EJB component, and the home interface name of the EJB component.

The format in which you specify the JNDI name depends on the EJB server. The following table shows the format used on each of the servers tested for this beta. The EJB component is called HelloEJB, its location relative to the Java naming context is ejbsample, and on EAServer it resides in the Hello package:

<table>
<thead>
<tr>
<th>Server</th>
<th>JNDI name format</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAServer</td>
<td>The value of com.sybase.jaguar.component.bind.naming in the component properties file:</td>
</tr>
<tr>
<td></td>
<td>com.sybase.jaguar.component.bind.naming= Hello/HelloEJB</td>
</tr>
<tr>
<td></td>
<td>Or (if bind.naming is not set), PackageName/ComponentName obtained from Jaguar Manager or display-name.ejb-name obtained from the EAServer ejb-jar.xml file:</td>
</tr>
<tr>
<td></td>
<td>&lt;ejb-jar&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;display-name&gt;Hello&lt;/display-name&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;ejb-name&gt;HelloEJB&lt;/ejb-name&gt;</td>
</tr>
<tr>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>

Using the lookup method
Invoking component methods

The home interface name is the fully-qualified class name of the EJB home interface, for example:

```
ejbsample.HelloEJBHome
```

The following example shows the invocation of the lookup method for HelloEJB on WebSphere.

```
HelloEJBHome homeobj

homeobj = conn.lookup("HelloEJBHome", "HelloEJB", "ejbsample.HelloEJBHome")
```

Lookup is case sensitive

Lookup in EJB servers is case sensitive. Make sure that the case in the string you specify for the arguments to the lookup method matches the case on the server.

Creating or finding an instance of an EJB

A session bean is created in response to a client request. A client usually has exclusive use of the session bean for the duration of that client session. An entity bean represents persistent information stored in a database. A client uses an entity bean concurrently with other clients. Since an entity bean persists beyond the lifetime of the client, you must use a primary key class name to identify or find this preexisting component.

For a session bean, you use the proxy object’s create method to create the instance of the EJB. The create method can throw CreateException and RemoteException. Assuming that you have obtained a reference to the home interface in homeobj, create is used in the same way on all EJB servers:

```
HelloEJB beanobj
```
try
  beanobj = homeobj.create()
catch (remoteexception re)
  MessageBox("Remote exception", re.getmessage())
  return -1
catch (createexception ce)
  MessageBox("Create exception", ce.getmessage())
  return -2
end try

For an entity bean, you provide a primary key. The FindByPrimaryKey method can throw FinderException and RemoteException. In this example, the key is the ID of a specific customer that is passed as an argument to the function:

try
  beanobj = homeobj.findByPrimaryKey(customerID)
catch (remoteexception re)
  MessageBox("Remote exception", re.getmessage())
  return -1
catch (finderexception re)
  MessageBox("Finder exception", re.getmessage())
  return -2
end try

Invoking EJB component methods

When the bean instance has been created or found, you can invoke its methods. For example:

```java
string msg
msg = beanobj.displaymessage()
```

Disconnecting from the server

When your application has finished using the EJB component, it should disconnect from the server:

```java
conn.disconnectserver()
```

Exception handling

Errors that occur in the execution of a method of an EJB component are mapped to exception proxies and thrown to the calling script. The methods of all the classes in `pbejbclient90.dll` can also throw exceptions when, for example, connection to the server failed, or the component could not be located or created.
Building EJB proxy projects generates the proxies for the home and remote interfaces, proxies for any Java classes referenced by the EJB, proxies for ancestor classes, and proxies for any exceptions that can be thrown by the EJB and its supporting classes. The following exception proxies are among those that may display in the System Tree:

<table>
<thead>
<tr>
<th>Proxy name</th>
<th>Java object name</th>
</tr>
</thead>
<tbody>
<tr>
<td>createexception</td>
<td>javax.ejb.CreateException</td>
</tr>
<tr>
<td>ejbexception</td>
<td>javax.ejb.EJBException</td>
</tr>
<tr>
<td>finderexception</td>
<td>javax.ejb.FinderException</td>
</tr>
<tr>
<td>remoteexception</td>
<td>java.rmi.RemoteException</td>
</tr>
<tr>
<td>removeexception</td>
<td>javax.ejb.RemoveException</td>
</tr>
</tbody>
</table>

Catching exceptions

A client application can handle communications errors in a number of ways. For example, if a client connects to a server and tries to invoke a method for an object that does not exist, the client can disconnect from the server, connect to a different server, or retry the operation. Alternatively, the client can display a message to the user and give the user the opportunity to control what happens next.

When an error occurs, if the client connects to a new server to retry the operation, it must instantiate the remote object on the new server before invoking a method of the remote object.

In the following example, the script simply displays a message box when a specific exception occurs:

```java
// function char getChar() throws RemoteException
try
    conn.connectToServer(properties)
    mappinghome = conn.lookup("pbEjbMappingHome",
    "pbEjbTest/pbEjbMappingBeanSL",
    "pbejb.pbEjbMappingHome")
    mapping = mappinghome.create()
    ret = mapping.getChar()
    messagebox("char from EJB", ret)
catch (remoteexception re)
    messagebox("remoteexception", re.getMessage())
catch (createexception ce)
    messagebox("createexception", ce.getMessage())
end try
```

Unhandled exceptions

If no exception handler exists, or if the existing exception handlers do not handle the exception, the SystemError event on the Application object is executed. If the SystemError event has no script, an application error occurs and the application is terminated.
Client-managed transactions

EJB client applications can control transactions on the server using the EJBTransaction object. This object has methods that enable the client to begin, commit, or roll back a transaction. The client can also get the status of a transaction, change its timeout value, or modify the transaction so that it cannot be committed.


Clients can obtain access to the methods of the EJBTransaction class by calling the getEJBTransaction method of the EJBConnection class:

```java
ejbconnection conn
ejbtransaction trans
string properties[]

conn = create ejbconnection
TRY
  conn.connectToServer(properties)
  trans = conn.getEJBTransaction()
CATCH (exception e)
  messagebox("exception", e.getmessage())
END TRY
```

If an EJBTransaction instance is obtained successfully, you use its begin method to start the transaction and its commit or rollback methods to end it:

```java
TRY
  // Start the transaction
  trans.begin()
  // Create a component and call methods to be executed
  // within the transaction
  ...
  // Commit the transaction
  trans.commit();
CATCH (exception e)
  messagebox("exception", e.getmessage())
  trans.rollback()
END TRY
```

GetStatus returns an integer that indicates whether the transaction is active, has been marked for rollback, is in the prepare phase or commit phase, or has been committed or rolled back.
Setting a timeout period for transactions

A calling thread can specify a timeout period after which a transaction will be rolled back. This example sets the timeout period to three minutes (180 seconds):

```java
trans.SetTimeout(180)
trans.Begin()
```

Debugging the client

The `createJavaVM` method of the JavaVM class takes a boolean value as a second argument. If this second argument is "true", execution information, including class loads, are logged to the file `vm.out` in the directory where the application resides:

```java
// global variable: JavaVM g_jvm
string classpath
boolean isdebug

classpath = "d:\tests\ejbsample;"
classpath += "D:\Sybase\Shared\PowerBuilder\pbejbclient90.jar;"
isdebug = true
g_jvm.createJavaVM(classpath, isdebug)
```

Reference information

The rest of this chapter provides reference information for each of the EJB client classes and their methods.

EJBConnection

Description

Connects to an EJB server and locates an EJB.
ConnectToServer

Description
Connects a client application to an EJB server. The client application must call ConnectToServer before it can use a remote object on the server.

Syntax
connection.ConnectToServer ( string properties[] )

Argument | Description
----------|-------------------
connection | The name of the EJBConnection object you want to use to establish the connection
properties[] | A string array used to pass name/value pairs that specify how the connection will be established

Return value
None.

Examples
In this example, the client application connects to a WebLogic server application using the Connection object called conn:

```java
    ejbconnection conn
    helloejbhome hellohome
    helloejb hello
    string properties[ ]
    string msg
    properties[3]="javax.naming.Context.SECURITY_PRINCIPAL=myid"

    conn = create ejbconnection
    TRY
        conn.connectToServer (properties)
    CATCH (exception e)
        messagebox("exception", e.getMessage())
    END TRY
```

Usage
Before calling ConnectToServer, declare a string array variable and assign name/value pairs to each element in the array. The names for which you must supply values are:

- javax.naming.Context.INITIAL_CONTEXT_FACTORY
- javax.naming.Context.PROVIDER_URL
- javax.naming.Context.SECURITY_PRINCIPAL
- javax.naming.Context.SECURITY_CREDENTIALS
For commonly used servers, the values for javax.naming.Context.INITIAL_CONTEXT_FACTORY are shown in the following table:

<table>
<thead>
<tr>
<th>Server</th>
<th>INITIAL_CONTEXT_FACTORY value</th>
</tr>
</thead>
<tbody>
<tr>
<td>EAServer</td>
<td>com.sybase.ejb.InitialContextFactory</td>
</tr>
<tr>
<td>WebLogic</td>
<td>weblogic.jndi.WLInitialContextFactory</td>
</tr>
<tr>
<td>WebSphere</td>
<td>com.ibm.websphere.naming.WsnInitialContextFactory</td>
</tr>
</tbody>
</table>

See also
- CreateJavaInstance
- Lookup

CreateJavaInstance

Description
Creates an instance of a Java object from a proxy name.

Syntax
\[
\text{connection.CreateJavaInstance} (\text{proxyobject proxyobject}, \text{string proxyname})
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection</td>
<td>The name of the EJBConnection object used to establish the connection.</td>
</tr>
<tr>
<td>proxyobject</td>
<td>PowerObject into which the function places a reference to the object specified by proxyname. This argument is passed by reference.</td>
</tr>
<tr>
<td>proxyname</td>
<td>The name of the proxy object for the EJB component.</td>
</tr>
</tbody>
</table>

Return value
Long. Returns 0 for success and one of the following values for failure:

-1 Failed to create Java class.
-2 Invalid proxy name.
-3 Failed to create proxy object.

Examples

```
ejbconnection conn
customer cus_1

conn = create ejbconnection
comm.connectToServer(properties)
mappinghome = conn.lookup("pbEjbMappingHome",
  "pbEjbTest/pbEjbMappingBeanSL",
  "pbejb.pbEjbMappingHome")

conn.createJavaInstance(cus_1, "customer")
cus_1.setname("Jones")
```
See also ConnectToServer

## DisconnectServer

**Description**
Disconnects a client application from an EJB server application.

**Syntax**

```java
connection.DisconnectServer()
```

**Argument** | **Description**
--- | ---
`connection` | The name of the EJBConnection object used to establish the connection you want to delete

**Return value**
None.

**Examples**
In this example, the client application disconnects from the server application using the EJBConnection object `myconnect`:
```
myconnect.DisconnectServer()
```

See also ConnectToServer

## GetEJBTransaction

**Description**
Returns a reference to the EJBTransaction object associated with the client.

**Syntax**

```java
connection.GetEJBTransaction()
```

**Argument** | **Description**
--- | ---
`connection` | The name of the EJBConnection object used to establish the connection

**Return value**
EJBTransaction.

**Examples**
This example shows the use of `GetEJBTransaction` to return a reference to the EJBTransaction object so that you can control transactions from the client:
```
// Instance variables:
// EJBConnection myconnect
EJBTransaction mytrans
long ll_status

mytrans = myconnect.GetEJBTransaction()
ll_status = mytrans.GetStatus()
```
EJBConnection

Usage
The PowerBuilder client can control the transaction demarcation of EJBs. After a transaction has been started with the EJBTransaction Begin method, GetEJBTransaction can be used to return the name of the transaction.

See also
Begin
Commit
GetStatus
Rollback
SetRollbackOnly
SetTransactionTimeout

Lookup

Description
Allows a PowerBuilder client to obtain the home interface of an EJB component in an application server in order to create an instance of the component.

Syntax
connection.lookup (string proxyname, string JNDIname, string homeinterfacename )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>connection</td>
<td>The name of the EJBConnection object used to establish the connection</td>
</tr>
<tr>
<td>proxyname</td>
<td>The name of the proxy object for the EJB component</td>
</tr>
<tr>
<td>JNDIname</td>
<td>The JNDI name of the EJB component</td>
</tr>
<tr>
<td>homeinterfacename</td>
<td>The fully-qualified class name of the EJB home interface</td>
</tr>
</tbody>
</table>

Return value
EJBHome.

Examples
The following example uses lookup to locate the home interface of the Multiply session EJB in the Java package abc.xyz.math. The example assumes the connection to the EJB server has already been established. The EJB server is EAServer:

```powerbuilder
// Instance variable:
// EJBConnection myconnect
Multiply myMultiply
MultiplyHome myMultiplyHome
long ll_product

TRY
    myMultiplyHome = myconnect.lookup("MultiplyHome",
                                        "Math/Multiply", "abc.xyz.math.MultiplyHome")
    myMultiply = myMultiplyHome.create()
    ll_product = myMultiply.multiply(1234, 4567)
```

228 Sybase Preliminary and Confidential Beta 3 Draft
catch (remoteexception re)
    messagebox("remoteexception", re.GetMessage())
catch (createexception ce)
    messagebox("createexception", ce.GetMessage())
CATCH (exception e)
    MessageBox("Exception", e.getmessage())
END TRY

The style used for the JNDI name depends on the EJB server. In the previous example, the `lookup` call for WebLogic would look like this:

```java
myMultiplyHome = myconnect.lookup("MultiplyHome",
    "abc.xyz.math.Multiply", "abc.xyz.math.MultiplyHome")
```

For WebSphere, the `lookup` call would look like this:

```java
myMultiplyHome = myconnect.lookup("MultiplyHome",
    "Multiply", "abc.xyz.math.MultiplyHome")
```

See also

ConnectToServer

---

**EJBHandle**

**Description**

EJBHandle inherits from NonVisualObject and is used as the return type for the EJBObject `GetHandle` function. EJBHandle maps closely to the `javax.ejb.Handle` interface. It has one member function: `GetEJBObject`.

**GetEJBObject**

**Description**

Obtains the EJB object reference represented by this handle.

**Syntax**

```java
ejbhandle.GetEJBObject()
```

**Return value**

EJBObject.

**EJBHome**

**Description**

The base class of the EJB Home interface. EJBHome inherits from NonVisualObject, and all EJB home interface proxies inherit from EJBHome. EJBHome maps closely to the `javax.ejb.EJBHome` interface.
EJBHome is used as the return type for the lookup method of the EJBConnection class and has two member functions: GetEJBMetaData and Remove.

**GetEJBMetaData**

Description: Obtains the EJBMetaData interface for the EJB.

Syntax: 
```
ejbhome.GetEJBMetaData ( )
```

Return value: EJBMetaData.

Usage: Throws remoteexception if the method fails due to a system-level failure.

**Remove**

Description: Removes the object identified by the name of its proxy.

Syntax: 
```
ejbhome.Remove ( NonVisualObject nvo)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvo</td>
<td>The name of the object to be removed</td>
</tr>
</tbody>
</table>

Return value: None.

Usage: Throws removeexception if the EJB or its container does not allow the client to remove the object.

Throws remoteexception if the method fails due to a system-level failure.

**EJBHomeHandle**

Description: EJBHomeHandle inherits from NonVisualObject and is used as the return type for the EJBHome GetHomeHandle function (not supported in this beta release). EJBHandle maps closely to the javax.ejb.HomeHandle interface. It has one member function: GetEJBHome.
**GetEJBHome**

Description: Obtains the home object represented by this handle.
Syntax: `ejbhomehandle.GetEJBHome ( )`
Return value: EJBHome.

**EJBMetaData**

Description: EJBMetaData inherits from NonVisualObject and is used as the return type for the EJBHome GetMetaData function. EJBMetaData maps closely to the `javax.ejb.EJBMetaData` interface. It has five member functions:

- GetHomeInterfaceClass
- GetPrimaryKeyClass
- GetRemoteInterfaceClass
- IsSession
- IsStatelessSession

**GetHomeInterfaceClass**

Description: Obtains the class object for the EJB’s home interface.
Syntax: `ejbmetadata.GetHomeInterfaceClass ( )`
Return value: NonVisualObject.

**GetPrimaryKeyClass**

Description: Obtains the class object for the EJB’s primary key class.
Syntax: `ejbmetadata.GetPrimaryKeyClass ( )`
Return value: NonVisualObject.

**GetRemoteInterfaceClass**

Description: Obtains the class object for the EJB’s remote interface.
Syntax: `ejbmetadata.GetRemoteInterfaceClass ( )`
EJBObject

Return value
NonVisualObject.

IsSession
Description
Returns true if the EJB is a session bean and false otherwise.
Syntax
ejbmetadata.IsSession ( )
Return value
Boolean.

IsStatelessSession
Description
Returns true if the EJB is a stateless session bean and false otherwise.
Syntax
ejbmetadata.IsStatelessSession ( )
Return value
Boolean.

EJBOBJect
Description
EJBOBJect inherits from NonVisualObject and is the base class of the EJB remote interface. All EJB remote interface proxies inherit from EJBOBJect. EJBHandle maps closely to the javax.ejb.HomeHandle interface. EJBOBJect has four member functions:
GetEJBHome
GetPrimaryKey
IsIdentical
Remove

GetEJBHome
Description
Obtains a reference to the EJB’s home interface.
Syntax
ejbobject.GetEJBHome ( )
Return value
EJBHome.
GetPrimaryKey
Description Obtains the EJB’s primary key.
Syntax ejbobject.GetPrimaryKey ( )
Return value NonVisualObject.

IsIdentical
Description Returns true if the specified EJB object is identical to the invoked EJB object.
Syntax ejbobject.IsIdentical (EJBObject ejbobject)
Return value Boolean.

Remove
Description Removes the EJB object.
Syntax ejbobject.Remove ( )
Return value None.

EJBTransaction
Description The EJB transaction class enables PowerBuilder clients to control a transaction on an EJB server. EJBTransaction maps closely to the javax.transaction.UserTransaction interface. EJBTransaction has six member functions:
   Begin
   Commit
   GetStatus
   Rollback
   SetRollbackOnly
   SetTransactionTimeout

Begin
Description Creates a new transaction and associates it with the current thread.
EJBTransaction

Syntax

```java
ejbtrans.Begin ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ejbtrans</td>
<td>The name of an EJBTransaction object</td>
</tr>
</tbody>
</table>

Return value

None.

Examples

The following example shows the use of begin to create a transaction from a client:

```java
EJBTransaction trans
EJBConnection conn
string properties[ ]

// set properties
.....
conn = create ejbconnection
trans = create ejbtransaction
TRY
    conn.connectToServer(properties)
    trans.begin()
CATCH (exception e)
    messagebox("exception", e.getmessage())
END TRY
```

See also

Commit
GetStatus
GetEJBTransaction (EJBConnection class)
Rollback
SetRollbackOnly
SetTransactionTimeout

Commit

Description

 Declares that the transaction associated with the calling thread should be committed.

Syntax

```java
ejbtrans.Commit ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ejbtrans</td>
<td>The name of an EJBTransaction object.</td>
</tr>
</tbody>
</table>

Return value

None.
Examples

In this example, the client calls the dopayroll method on the CmpnyAcct EJB component, which processes a company payroll. If the company has sufficient funds to meet the payroll, the client commits the transaction. Otherwise, an exception is thrown and the client rolls back the transaction:

```
// Instance variables:
// EJBTransaction trans
// EJBConnection conn
// CmpnyAcctHome AcctHome
// CmpnyAcct Acct

TRY
    trans.begin()
    AcctHome = conn.lookup("CmpnyAcctHome",
        "Sample/CmpnyAcct", "sample.CmpnyAcctHome")
    Acct = AcctHome.create()
    Acct.dopayroll()
    trans.commit()
CATCH (remoteexception re)
    messagebox("remoteexception", re.GetMessage())
CATCH (createexception ce)
    messagebox("createexception", ce.GetMessage())
CATCH (exception e1)
    MessageBox ("exception", e1.getmessage() )
    TRY
        trans.rollback();
    CATCH (exception e2)
        MessageBox ("exception", e2.getmessage() )
    END TRY
    END TRY
```

Usage

The Commit method completes the transaction associated with the calling thread. The transaction is not completed if any other participants in the transaction vote to roll back the transaction.

See also

Begin
GetStatus
GetEJBTransaction (EJBConnection class)
Rollback
SetRollbackOnly
SetTransactionTimeout

GetStatus

Description

Returns the status of the EJB transaction associated with the client.
EJBTransaction

Syntax

```java
ejbtrans.GetStatus ( )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ejbtrans</td>
<td>The name of an EJBTransaction object</td>
</tr>
</tbody>
</table>

Return value

A long value representing the transaction status.

Possible values are:

1. Status active
2. Status marked rollback
3. Status prepared
4. Status committed
5. Status rolled back
6. Status unknown
7. Status no transaction
8. Status preparing
9. Status committing
10. Status rolling back

Examples

This example shows the use of `GetStatus` to obtain the state of the current transaction:

```java
// Instance variables:
// EJBConnection myconnect
EJBTransaction mytrans
long ll_status

mytrans = myconnect.GetEJBTransaction()
ll_status = mytrans.GetStatus()
```

Usage

The `GetStatus` method can be used to determine the current status of a transaction by the client that initiated the transaction using the `Begin` method.

See also

- `Begin`
- `Commit`
- `GetEJBTransaction(EJBConnection class)`
- `Rollback`
- `SetRollbackOnly`
- `SetTransactionTimeout`
Rollback

Description
Rolls back the transaction associated with the calling thread.

Syntax
`ejbtrans.Rollback()`  

Return value
None.

Examples
This example shows the use of Rollback to roll back a transaction when an update does not succeed:

```java
// Instance variables:
// EJBTransaction trans

TRY
    trans.begin()
    acct.updateChecking(amount)
    trans.commit()
CATCH (exception e1)
    TRY
        trans.rollback()
    CATCH (exception e2)
        MessageBox("Rollback failed", e2.getMessage())
    END TRY
    MessageBox("Transaction failed", e1.getMessage())
END TRY
```

See also
Begin  
Commit  
GetStatus  
GetEJBTransaction (EJBConnection class)  
SetRollbackOnly  
SetTransactionTimeout

SetRollbackOnly

Description
Modifies a transaction associated with a calling thread so that the only possible outcome is to roll back the transaction.

Syntax
`ejbtrans.SetRollbackOnly()`  

```
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ejbtrans</code></td>
<td>The name of an EJBTransaction object</td>
</tr>
</tbody>
</table>
```
**EJBTransaction**

Return value

None.

Examples

In this example, a participant in a transaction has determined that it should be rolled back. It gets a reference to the current transaction and votes to roll back the transaction:

```java
// Instance variables:
// EJBConnection conn
// EJBTransaction trans

trans = conn.GetEJBTransaction();
trans.SetRollbackOnly();
```

Usage

Rollback is typically called by the originator of the transaction, but another participant in a transaction can call SetRollbackOnly to vote that the transaction should be rolled back.

See also

Begin
Commit
GetStatus
GetEJBTransaction (EJBConnection class)
Rollback
SetTransactionTimeout

**SetTransactionTimeout**

Description

Sets the timeout value for subsequent transactions. The transaction is rolled back if it does not complete before the timeout expires.

Syntax

```
ejbtrans.SetTransactionTimeout ( seconds )
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ejbtrans</td>
<td>The name of an EJBTransaction object</td>
</tr>
<tr>
<td>seconds</td>
<td>A long that specifies the number of seconds that elapse before a transaction is rolled back</td>
</tr>
</tbody>
</table>

Return value

None.

Examples

This example shows the use of SetTransactionTimeout to set the timeout period to five minutes:

```java
// Instance variables:
// EJBConnection conn
// EJBTransaction trans

TRY
```
trans.SetTransactionTimeout(300)
trans.begin()
CATCH (exception e)
    MessageBox("Exception", e.getMessage())
END TRY

Usage
The SetTransactionTimeout method specifies the number of seconds that can elapse before a transaction is rolled back. The timeout period applies to transactions created by subsequent invocations of Begin. If seconds is 0, no timeout period is in effect.

See also
Begin
Commit
GetStatus
GetEJBTransaction (EJBConnection class)
Rollback
SetRollbackOnly

JavaVM
Description
Enables PowerBuilder client applications to gain access to EJB components on a server.

CreateJavaVM
Description
Loads and initializes a Java VM.

Syntax
javavm.createJavaVM(string classpath, boolean isdebug)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>javavm</td>
<td>An instance of the JavaVM class</td>
</tr>
<tr>
<td>classpath</td>
<td>A string specifying the class path that contains files required by the EJB server such as pbejbclient90.jar and J2EE classes</td>
</tr>
<tr>
<td>isdebug</td>
<td>A boolean that determines whether debug information is saved to a file called VM.out in the directory where the current application is located</td>
</tr>
</tbody>
</table>

Return value
Returns 0 for success and one of the following negative integers for failure:
-1 Failed to load Java VM, possibly because jvm.dll was not found in the classpath.
Failed to find the *pbejbclient90.jar* file.

### Examples

```java
// global variable: JavaVM g_jvm
boolean isdebug
string classpath

g_jvm = create javavm

isdebug = true

classpath = ".;d:\j2sdkee1.3\lib\j2ee.jar;"
classpath += "cd:\pb90\pbni\java\ejbclient;"
classpath += "D:\Sybase\EAServer\html\classes;;"

IF g_jvm.createJavaVM(classpath, isdebug) < 0 THEN
    MessageBox("Error", "Failed to load JavaVM")
END IF
```

### Usage

The `isdebug` argument is used to record information about the Java VM, including class loads, into the file `VM.out` in the directory where the current application is located.

The `classpath` argument must include the classes, JAR files, and so forth required by the server.

### See also

- `ConnectToServer`
- `GetJavaVMVersion`

---

### GetJavaVMVersion

#### Description

Gets the version number of the current Java VM.

#### Syntax

```
javavm.getJavaVMVersion()
```

#### Return value

String representing the Java VM version. For example, for JDK 1.3.1, `GetJavaVMVersion` returns `1.3.1_01`.

#### Examples

This example shows how to use `GetJavaVMVersion`:

```java
string ls_javaVMVersion
ls_javaVMVersion = getJavaVMVersion()
```

### See also

- `CreateJavaVM`
Source control enhancements

PowerBuilder 9 includes enhancements to improve the speed and performance of source control integration. The following enhancements are available for testing in the beta 3 release:

- Performing source control operations on a PBL
- Ability to suppress overwrite prompt
- Checklist dialog box enhancements
- Source control history enhancements
- Status determination by version number
- PBNative configuration enhancements
- Library painter list view enhancement
- Source control logging enhancement
- Initialization mode performance enhancement

Performing source control operations on a PBL

Although you cannot register a PBL in source control through the PowerBuilder UI, you can now right-click a PBL in the System Tree or in the Library painter and add all the objects in that PBL to source control. The pop-up menu for PBLs now includes the Get Latest Version, Check In, Check Out, Undo Check Out, and Add To Source Control menu items.

When you select a source control menu item on a PBL, a checklist dialog box displays all the objects in the current PBL to which the operation you selected can apply. For a single-PBL target, the list is the same as when you click on the source control menu item for the PBT; however, the PBT is never listed in the checklist dialog box that you open from a PBL pop-up menu.
Ability to suppress overwrite prompt

For certain SCC providers, PowerBuilder makes an SccDiff method call when it refreshes the source control status of your workspace files. In order for the SccDiff method to provide accurate information, PowerBuilder must export the objects in your workspace PBLs to the local project path. The exported files must overwrite any identically named files already existing in the local project path subsequent to a source control operation.

Because many SCC providers leave object files on the local project path with the read-only attribute flag set (when the objects are not checked out), PowerBuilder prompts you for confirmation before overwriting these files. This can be time consuming if there are many files to overwrite.

By selecting the Suppress Prompts To Overwrite Read-Only Files option on the Source Control tab of the Workspace properties dialog box, you set permission for PowerBuilder to overwrite read-only files silently whenever it needs to export object source from the PBL to the local project path in order to communicate with the SCC provider. This option is not selected by default.

Because the SccDiff method is not called for Perforce or ClearCase, selecting the Suppress Prompts To Overwrite Read-Only Files option does not have any impact on performance for these SCC providers. This option also has minimal or no impact if you select the Delete PowerBuilder Generated Object Files option, clear the Perform Diff On Status Update option, or if your source control provider implements the SccQueryInfoEx method.

For information on the SccQueryInfoEx method, see “Status determination by version number” on page 243.

Checklist dialog box enhancements

The checklist dialog boxes that you open for source control operations are now bigger than in earlier versions of PowerBuilder, and you can resize the dialog boxes by grabbing and dragging resize handles. You can open checklist dialog boxes by selecting a source control item from the pop-up menu for a PBW, PBT, or PBL file. You can also open a checklist dialog box for a multiple object selection in the Library painter list view.
Source control enhancements

Importing earlier versions of an object

For certain source control systems, PowerBuilder 9 lets you get earlier versions of an object from the source control repository. Previously you could do this only with the Check Out operation. This functionality has been expanded to let you get earlier versions of the object while performing other source control operations.

For source control systems that return an SCC_I_RELOADFILE return code on an SccHistory or an SccProperties call, PowerBuilder 9 lets you automatically import an earlier version of an object. This happens when you select Show History from an object pop-up menu, or when you select the object in the Library painter, then select Entry>Source Control>SCC System Properties from the Library painter menu.

If an SCC system supports this functionality, you can select the version of the object you want from the Show History dialog box or from the dialog box for the SCC system properties, then click a Get button. The Get button imports the version of the object you selected to the current target.

PBNative implements ShowHistory

PBNative supports the Show History command in PowerBuilder 9. The command returns a simple dialog box showing the file name on the local machine, the current source control status, the version number of the current object, and the most recent check-in date to source control.

Status determination by version number

PowerBuilder 9 provides third-party SCC providers with an extension to the SCC API that allows them to enhance their product's integration with PowerBuilder. Typically, calls to the SccDiff method are required to determine if an object is out-of-sync with the SCC repository. (This is not required for Perforce and ClearCase.)

However, SCC providers can implement SccQueryInfoEx as a primary file comparison method instead of SccDiff. The SccQueryInfoEx method returns the most recent version number for each object requested. This allows PowerBuilder to compare the version number associated with the object in the PBL with the version number of the tip revision in the SCC repository in order to determine whether an object is in sync.
### PBNative configuration enhancements

Since `SccQueryInfoEx` is a much simpler request than `SccDiff`, the performance of the PowerBuilder IDE improves noticeably when this feature is implemented by the SCC provider. For these providers, the `SccDiff` call is used as a backup strategy only when a version number is not returned on an object in the repository.

Once the new API method is implemented in an SCC provider DLL and exported, PowerBuilder automatically begins to use the `SccQueryInfoEx` call with that provider. The `SccQueryInfoEx` method is currently used by PBNative.

For information on PBNative version number support, see "PBNative configuration enhancements" next.

<table>
<thead>
<tr>
<th>PBNative configuration enhancements</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Version number support</strong></td>
<td>PowerBuilder 9 adds a version number to the PRP file for an object in the PBNative archive directory when you register that object with PBNative source control. PowerBuilder increments the version number when you check in a new revision. The version number is visible in the Show History dialog box that you open from the pop-up menu for the object, or in the Library painter when you display the object version numbers.</td>
</tr>
<tr>
<td><strong>Determining out-of-sync status by version number</strong></td>
<td>At workspace initialization, PowerBuilder 9 determines the most efficient way to judge whether objects are in sync with the source control repository. With PBNative source control, PowerBuilder uses version numbers to make this determination. Although this change does not affect the PowerBuilder UI, it dramatically improves performance by increasing the speed at which objects are synched.</td>
</tr>
<tr>
<td><strong>Relative directory specification</strong></td>
<td>In PowerBuilder 9, PBNative is able to archive objects successfully in a target even if the target includes a “..\” character sequence for any of the PBL files specified in its library list.</td>
</tr>
</tbody>
</table>
PowerBuilder 9 includes options that you can set for a visual differences utility that you use with PBNative. You access the PBNative Command Options dialog box by right-clicking the workspace file, selecting PBNative for your source control system on the Source Control tab, and clicking the Advanced button.

![PBNative Command Options dialog box]

You can select any or all of the following options:

<table>
<thead>
<tr>
<th>Option</th>
<th>Select this if</th>
</tr>
</thead>
<tbody>
<tr>
<td>Enclose file names in double quotes</td>
<td>Your visual difference utility does not handle spaces in file names.</td>
</tr>
<tr>
<td>Refer to local PBL entry as argument #1</td>
<td>You do not want the visual difference utility to use the repository object as the first file in a file comparison.</td>
</tr>
<tr>
<td>Generate short (8.3) file names</td>
<td>Your visual difference utility does not handle long file names.</td>
</tr>
<tr>
<td>Generate an extra space prior to file arguments</td>
<td>Your visual difference utility requires an extra space between files that are listed as arguments when you open the utility from a command line. This option has been added for backward compatibility only, as an extra space was automatically added by PowerBuilder 8.</td>
</tr>
</tbody>
</table>
Library painter list view enhancement

In PowerBuilder 9, the Library painter list view can display the version number of all files registered in source control. The ability to display the version number in the file listing is controlled by the SCC Version Number option in the Options dialog box for the Library painter. This option is selected by default. You open the Options dialog box by selecting Design>Options from the Library painter menu.

Source control logging enhancement

You can set the diagnostic level for source control logging activity by adding the SccLogLevel property to the Library section of the PB.INI file and assigning it a value from 1 to 3. (There are three different diagnostic levels.)

By default, the diagnostic level is set to level 1. Level 2 logging adds diagnostic information when errors are encountered, and level 3 adds metadata and status cache detail for certain source control operations.
Initialization mode performance enhancement

Obtaining the SCC status for many source-controlled objects is a time-consuming operation that occurs during workspace start-up, and whenever objects such as PBLs or Folders are expanded in the PowerBuilder System Tree or Library Painter.

During periods of high SCC activity, PowerBuilder 9 automatically spawns a child thread to communicate with the SCC provider asynchronously. The title bar in the Library Painter displays the word Initialization while this thread is running. SCC operations are temporarily disabled as status information is being obtained, but you can still work with objects as you would in offline mode, including the editing of objects that are already checked out.

Because status requests issued by PowerBuilder 9 are much more efficient than in previous releases, typical initialization lasts only for a few seconds when expanding a medium-sized PBL or folder. For example, when a PBL containing 200 objects is expanded, requests to the server can be reduced from 400 in PowerBuilder 8 to as few as 8 in PowerBuilder 9. This results in noticeable improvement when you are connecting to a source control repository over a dial-up modem.
Initialization mode performance enhancement
The OrcaScript Language

OrcaScript is a scripting language that allows you to build PowerBuilder workspaces and targets without operator intervention. The full ORCA tool kit is available to Sybase partners only, but OrcaScript can be used by any PowerBuilder customer.

This chapter covers the following topics:

- About OrcaScript
- OrcaScript Commands
- Usage notes for OrcaScript commands and parameters

About OrcaScript

OrcaScript allows you to write batch scripts to process PowerBuilder applications and files without using the PowerBuilder development environment. You can use OrcaScript to get the latest version of a PowerScript target from source control, build the target PBLs, deploy components to EAServer, and compile PowerBuilder executable files—all without operator intervention.

The targets you obtain from source control using OrcaScript could be placed on a network build machine that is shared by PowerBuilder developers. This is especially advantageous for large shops with fixed working hours: the builds could be done nightly by running an OrcaScript batch file, and an up-to-date version of the targets and libraries would be available at the start of the next work day.

Developers could then use OrcaScript or operating system commands to copy the shared files directly to their local machines. Although developers would still connect directly to source control from their local workspaces, refreshing the targets in the workspaces would be much faster since compilation times for complex PowerScript targets would be greatly minimized.
Batch file order

If you include OrcaScript commands in a batch file, the file is read line by line. Each OrcaScript batch file must begin with a start session command and end with an end session command. You can save the batch file with any extension. You run the batch file by calling the OrcaScript executable on a command line and passing the batch file name as an argument:

```
OrcaScr9 myOrcaBat.dat
```

If you use relative directories in the OrcaScript batch file, the directories are relative to the location of the OrcaScr9.exe file. The command to start the OrcaScript executable can also take the following parameters:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>/D</td>
<td>Sets variables that are valid in the batch file</td>
<td>OrcaScr9 /D myVar=value myOrca.dat</td>
</tr>
<tr>
<td>/H or /?</td>
<td>Prints syntax help to screen</td>
<td>OrcaScr9 /H</td>
</tr>
<tr>
<td>/T</td>
<td>Runs with tracing on</td>
<td>OrcaScr9 /T myOrca.dat</td>
</tr>
</tbody>
</table>

Caution

You should not run an OrcaScript batch file if PowerBuilder 9.0 is currently running on the same machine. If the PowerBuilder development environment is not shut down while OrcaScript is running, your PowerBuilder libraries can become corrupted. For this reason, casual use of OrcaScript is not recommended.

Error handling

Each line of an OrcaScript batch file either succeeds or fails. If a command fails, subsequent commands are not processed and the OrcaScript session is ended. An error message is printed to the command window.

OrcaScript Commands

OrcaScript commands are not case sensitive. The generic command parameters can include only strings delimited by quotation marks, or predefined variables and constants without quotation marks. White space is used to separate multiple parameters for a single command. Any place a string is expected, a name that has been previously defined (set) in an OrcaScript command can be used.
In the OrcaScript command prototype syntax that follows, brackets indicate a parameter is optional. A pipe character inside angle brackets ( < | > ) indicates that a selection must be made from one of the values inside the angle brackets. As elsewhere in the PowerBuilder documentation, text in italic type indicates a variable.

For commands where a string variable is required by the command syntax but is not essential to the command function (such as pbrName for the build library command), you can use an empty string inside quotation marks for the string value. Most of the OrcaScript commands and parameters are self-explanatory. For usage notes and an example of an OrcaScript batch file for obtaining a PowerScript target from source control, see "Usage notes for OrcaScript commands and parameters" next.

**OrcaScript commands**

OrcaScript supports the following commands:

- start session
- end session
- set liblist pbl_list [pbl_list ...]
- set application pblName applicationName
- set name = value
- set name += value
- echo value [value ...]
- file copy fromFile toFile
- regenerate pblName entryName entryType
- copy entry pblName entryName entryType toPblName
- build library pblName pbrName <pbd | 32>
- build executable exeName iconName pbrName pbdflags [machinecode]
- build application <full | migrate | incremental>
- build project pblName projectName [ serverName serverPort ]
- create library pblName pblComments
- scc get connect properties workspaceName
- scc set connect property provider sccProvider
- scc set connect property userid userID
- scc set connect property logfile logFileName
- scc set connect property project projectPath
- scc set connect property localprojpath localProjectPath
- scc set connect property auxproject auxProjectPath
- scc set connect property logappend < true | false >
- scc set connect
- scc set target targetName [refreshType][refreshOption][refreshOption]
- scc get latest version file_list [file_list ...]
- scc exclude liblist pblName [ pblName ...]
### OrcaScript Commands

- `scc refresh target targetName <full | migrate | incremental >`
- `scc close`

**Argument description**

Arguments for OrcaScript commands are described in the table below:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pbl_list</code></td>
<td>String containing the list of PBLs for the session application. PBL names can be separated by semicolons in a single string, or separated by a blank space in multiple strings.</td>
</tr>
<tr>
<td><code>pblName</code></td>
<td>Name of a PBL for an OrcaScript action or for the OrcaScript session application.</td>
</tr>
<tr>
<td><code>applicationName</code></td>
<td>Name of the application for an OrcaScript action.</td>
</tr>
<tr>
<td><code>name</code></td>
<td>String you define for an OrcaScript session.</td>
</tr>
<tr>
<td><code>value</code></td>
<td>Value of a string that you set for the OrcaScript session.</td>
</tr>
<tr>
<td><code>fromFile</code></td>
<td>File that you want to copy during an OrcaScript session.</td>
</tr>
<tr>
<td><code>toFile</code></td>
<td>File name for a file that you copy during an OrcaScript session.</td>
</tr>
<tr>
<td><code>entryName</code></td>
<td>Pointer to a string whose value is the name of the referenced object.</td>
</tr>
<tr>
<td><code>entryType</code></td>
<td>Value specifying the type of the referenced object. Values can be: app, dw, fn, menu, query, struct, uo, win, pipe, project, or proxy.</td>
</tr>
<tr>
<td><code>toPblName</code></td>
<td>Name of the PBL to which you copy an entry.</td>
</tr>
<tr>
<td><code>pbrName</code></td>
<td>Name of a resource file you want to include in a build.</td>
</tr>
<tr>
<td>`pbd</td>
<td>32`</td>
</tr>
<tr>
<td><code>exeName</code></td>
<td>Name of the executable you want to build.</td>
</tr>
<tr>
<td><code>iconName</code></td>
<td>Name of an icon to use for an executable you build with OrcaScript.</td>
</tr>
<tr>
<td><code>pbdFlags</code></td>
<td>String composed of a series of Y and N values for each library in the library list. A value of “nmy” indicates that there are four libraries in the library list, the last two being PBDs. Objects from PBLs are copied into the executable; objects from PBDs are not copied.</td>
</tr>
<tr>
<td><code>machinecode</code></td>
<td>Use to compile a project as machine code.</td>
</tr>
<tr>
<td>`full</td>
<td>migrate</td>
</tr>
<tr>
<td><code>projectName</code></td>
<td>Name of the project object you want to build and deploy.</td>
</tr>
<tr>
<td><code>serverName</code></td>
<td>Name of the server where you want to deploy a project.</td>
</tr>
<tr>
<td><code>serverPort</code></td>
<td>Port for the server where you want to deploy a project.</td>
</tr>
<tr>
<td><code>pbdName</code></td>
<td>Name of a PBD you append to an EXE.</td>
</tr>
<tr>
<td><code>pblComments</code></td>
<td>Comments for a PBL you create in an OrcaScript session.</td>
</tr>
</tbody>
</table>
Arguments for source control commands

In addition to some of the arguments listed in the preceding table, OrcaScript source control commands use the following argument:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>workspaceName</code></td>
<td>Name of the workspace to connect to source control.</td>
</tr>
<tr>
<td><code>sccProvider</code></td>
<td>Name of the source control provider.</td>
</tr>
<tr>
<td><code>userID</code></td>
<td>Name of the user registered to source control.</td>
</tr>
<tr>
<td><code>logFileName</code></td>
<td>Name of a log file used to record SCC transactions.</td>
</tr>
<tr>
<td><code>projectPath</code></td>
<td>Path to the source control project.</td>
</tr>
<tr>
<td><code>localProjectPath</code></td>
<td>Local root directory for the project.</td>
</tr>
<tr>
<td><code>auxProjectPath</code></td>
<td>Contains any string that the SCC provider wants to associate with the project. It has a different meaning for every SCC vendor.</td>
</tr>
<tr>
<td><code>targetName</code></td>
<td>Name of the target for source control operations.</td>
</tr>
<tr>
<td>`true</td>
<td>false`</td>
</tr>
<tr>
<td><code>refreshType</code></td>
<td>Value can be:</td>
</tr>
<tr>
<td></td>
<td>• <strong>refresh_all</strong> Gets latest version of all objects from the SCC provider and refreshes all target libraries. Does not perform comparisons.</td>
</tr>
<tr>
<td></td>
<td>• <strong>outofdate</strong> Performs comparisons and updates objects that are out of date. If no refreshType value is specified, the refreshType defaults to outofdate. This value cannot be used with the <code>refreshOption</code> variable set to importonly.</td>
</tr>
<tr>
<td><code>refreshOption</code></td>
<td>Value can be:</td>
</tr>
<tr>
<td></td>
<td>• <strong>importonly</strong> Does not perform comparisons and does not refresh. Use to build targets if you refreshed the local path using the SCC provider’s administration tool.</td>
</tr>
<tr>
<td></td>
<td>• <strong>exclude_checkout</strong> Prevents objects that are currently checked out by the current user from being overwritten. Can be used with outofdate parameter in the same OrcaScript command.</td>
</tr>
<tr>
<td><code>file_list</code></td>
<td>String containing one or more file names using relative or absolute path specification. File names can be separated by semicolons in a single string, or separated by a blank space in multiple strings.</td>
</tr>
</tbody>
</table>
Usage notes for OrcaScript commands and parameters

Before calling any other ORCA functions, you need to open a session:

    start session

You can start and end multiple OrcaScript sessions in the same batch file.

Copying files, objects, and properties

If you want to use OrcaScript simply to move objects among libraries, you do not need to set a library list or application. You can use the copy commands to copy files, objects, and properties. This example copies the d_labels DataWindow from the source.pbl library to the destin.pbl library:

    copy "c:\app\source.pbl""d_labels" PBORCA_DATALOAD "c:\app\destin.pbl"

Setting a library list and an application

If you want to use OrcaScript to build targets or deploy components, you need to set the library list and the current application. You can set the library list and current application only once in an OrcaScript session. To use another library list and application, end the OrcaScript session and start a new session. The following OrcaScript commands build target libraries and compile an executable file:

    start session
    set liblist ".\qadbtest\qadbtest.pbl; .\shared_obj\shared_obj.pbl; .\datatypes\datatype.pbl; .\chgreqs\chgreqs.pbl"
    set application ".\qadbtest\qadbtest.pbl" "qadbtest"
    build library ".\shared_obj\shared_obj.pbl" "" pbd
    build library ".\datatypes\datatype.pbl" "" pbd
    build library ".\chgreqs\chgreqs.pbl" "" pbd
    build executable ".\qadbtest\qadbtest.exe" ".\emp.ico" ".\qadbtest.pbr" "nyyy"
    end session

Source control example

You can use OrcaScript source control commands instead of the commands to set the library list and application. The following is an example of an OrcaScript session that builds the same libraries as the previous example, but uses the target properties to set a library list and application:

    start session
    scc get connect properties "c:\testbld\testbld.pbw"
    scc connect
    scc set target "c:\testbld\qadbtest\qadbtest.pbt" "outofdate exclude_checkout"
    scc refresh target "incremental"
    build library ".\shared_obj\shared_obj.pbl" "" pbd
    build library ".\datatypes\datatype.pbl" "" pbd
build library ".\chgreqs\chgreqs.pbl" "" pbd
build executable ".\qadbtest\qadbtest.exe" ".\emp.ico"
".\qadbtest.pbz" "nyyy"
scc close
end session

You can call the scc connect command only after getting connection properties, and you must call it before you set or refresh the source-controlled targets. You must call the scc close command before you end your OrcaScript session.

SCC connection properties

The SCC get connect properties command is an easy way to populate the Orca SCC connection structure with the source control properties of a local workspace. However, to create OrcaScript batch files that are portable from one workstation to another, the recommended technique is to set each property explicitly. Many of these properties are vendor-specific. The best way to obtain correct values is to copy them directly from the SCC log file for your PowerBuilder workspace.

Build command failures

OrcaScript build commands for an executable or a library fail if the executable or library already exists in the build directory. To prevent an OrcaScript batch file containing these commands from failing, move or delete existing executables and libraries from the build directory before running the batch script.

Escape characters for string variables

OrcaScript, like PowerScript, uses the tilde (~) as an escape character. If you need to include a special character, such as a quotation mark, inside a string, you must place a tilde in front of it. A character in an OrcaScript batch file with a tilde in front of it is processed as a literal character.

Ending an OrcaScript session

You must close an OrcaScript session after you finish calling other OrcaScript commands. You close an OrcaScript session by calling:

    end session
Usage notes for OrcaScript commands and parameters
Introduction to PNI

This section provides a brief introduction to PNI.

Samples
This documentation contains two complete simple examples that illustrate the basic principles of using PNI: “PowerBuilder extension example” on page 271 and “Example: Calling PowerBuilder from C++” on page 284. Additional samples will be provided on the Sybase Web site, and may be available from the Beta Resource page during the beta cycle.

About PNI and PowerBuilder extensions

The PowerBuilder Native Interface (PNI) is a standard interface that enables developers to extend the functionality of PowerBuilder. In previous versions of PowerBuilder, you could use C or C++ functions by declaring them as external functions, but this technique requires you to make an external function declaration whenever you need to use a function. It also has limitations: you cannot use functions that require call backs, and some C datatypes do not map to PowerBuilder datatypes.

A PowerBuilder extension is a DLL, written in C++, that exposes one or more native classes so that they can be used in a PowerBuilder application as if they were custom class user objects created in PowerBuilder. A native class is a PowerScript class that is implemented in C++.

A PowerBuilder extension runs faster than a PowerBuilder custom class user object because it is compiled into native machine code instead of Pcode. This solution does not have the portability of Pcode, but PNI complies with the C++ specification, so that well-programmed code should be portable at the source code level.
**The elements of PBNI**

**_marshaller extensions**
Marshaler extensions act as bridges between PowerBuilder and other components, such as CORBA components, Java classes, Web services, and so on. PowerBuilder 9 uses PBNI in a marshaler extension for creating clients for EJB components in various Web application servers.

**Calling PowerBuilder from C++**
PBNI enables you to use PowerBuilder custom class user objects in a C++ application. To do so, you load the PowerBuilder VM and invoke functions through an interface called IPB_Session.

**Interacting with Java**
To call Java classes from PowerBuilder, you can build a marshaler extension that invokes Java methods through the Java Native Interface (JNI), as described in “Creating marshaler extensions” on page 288. You can also use JNI to enable Java to call into PowerBuilder through C or C++.

**The elements of PBNI**

**Ancestor classes**
All PowerBuilder native classes inherit from IPBX_NonVisualObject, which in turn inherits from IPBX_UserObject. You must implement the **Invoke** method in the inherited class to enable PowerBuilder to invoke methods in the native class.

Marshaler extensions contain a class that inherits from IPBX_Marshaler. You must implement the **InvokeRemoteMethod** method in the inherited class to enable PowerBuilder to invoke methods on remote objects represented by a proxy.

**Interfaces**
The IPB_VM interface is used to load PowerBuilder applications in third-party applications and interoperate with the PowerBuilder virtual machine (PBVM).

IPB_Session is an abstract interface that defines methods for accessing PowerScript data and calling PowerScript functions.

The IPB_Value and IPB_Arguments interfaces enable you to pass values between the PowerBuilder VM and PowerBuilder extension modules. The IPB_Value interface provides information about each variable, including its type, null flag, access privileges, array or simple type, and reference type. The IPB_Arguments struct is used to access the data.
Introduction to PBNI

Structures
The PBCallInfo class holds arguments and return type information for function calls between PBNI and PowerBuilder. You access the information in PBCallInfo using the IPB_Arguments interface. The PBArrrayInfo structure holds information about arrays.

Global functions
Every PowerBuilder extension object must export two global functions that enable the PowerBuilder VM to create instances of the object and use its methods. See “Expose standard functions” on page 266.

Helper classes
Several helper classes, such as PBObjectCreator, PBArrrayAccess, and PBEventTrigger, make it easier to program with PBNI.

Interaction between PBNI and the PowerBuilder VM
The following diagram shows how PBNI interacts with the PowerBuilder VM:

The PBNI SDK
When you install PowerBuilder, the Software Development Kit (SDK) for PBNI is installed in the PowerBuilder 9.0\SDK\PBNI directory. The SDK contains the following components:

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbx2pbd90.exe</td>
<td>A tool that generates a PBD file from a C++ DLL. The DLL must export a set of PBNI functions.</td>
</tr>
</tbody>
</table>
Comparing PBNI and JNI

If you have used the Java Native Interface (JNI), which enables Java applications and C and C++ modules to interoperate, you may find it helpful to be aware of the similarities and differences between the two interfaces.

The IPB_VM interface in PBNI is analogous to the JavaVM type, and the IPB_Session interface in PBNI is analogous to JNIEnv. For JNI, you use the javap command to obtain a string that encodes the signature of each method in a native class. For PBNI, the pbsig90 tool performs the same function.

The major difference between the two interfaces is in how a native function or class is declared.

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbsig90.exe</td>
<td>A tool that generates a set of strings representing the return type and arguments of each function in a PBL. The strings are used to call PowerBuilder functions from external modules.</td>
</tr>
<tr>
<td>include\pbn.h</td>
<td>A header file that defines the structures and interfaces used to build PowerBuilder extensions.</td>
</tr>
<tr>
<td>include\pbarray.h</td>
<td>A header file that contains helper classes that make it easier to create arrays and access data in them.</td>
</tr>
<tr>
<td>include\pbfield.h</td>
<td>A header file that contains helper classes that make it easier to access data in fields.</td>
</tr>
<tr>
<td>include\pbtraits.h</td>
<td>A header file used by pbarray.h and pbfield.h that provides specializations for the PBValue enumerated types.</td>
</tr>
<tr>
<td>include\pbext.h</td>
<td>A header file that defines the functions PowerBuilder extension functions must export.</td>
</tr>
<tr>
<td>include\pbnimd.h</td>
<td>A header file that defines machine-dependent datatypes used in pbni.h.</td>
</tr>
<tr>
<td>lib\pbn.lib</td>
<td>A library that must be linked into PowerBuilder extension DLLs.</td>
</tr>
<tr>
<td>wizards\pbext.awx</td>
<td>A Microsoft Visual C++ wizard that makes it easier to create PBNI projects.</td>
</tr>
<tr>
<td>pbni90.hlp, pbni90.cnt</td>
<td>Help files for PBNI.</td>
</tr>
</tbody>
</table>
In JNI, you must use the native keyword to declare that a function is native, but you cannot simply declare a class as native. You must define your classes in Java source code, use the javah tool to generate a C header file that defines a C prototype for each native method, then implement the individual C or C++ functions, using #include to include the generated header file.

PBNI provides an object-oriented approach—you declare a class as native in the C++ code by inheriting from the IPBX_NonVisualObject struct.
Building PowerBuilder Extensions

This section provides a brief introduction to building PowerBuilder extensions and a simple application that demonstrates the basic principles.

• “Creating a PowerBuilder extension” on page 263
• “Using a PowerBuilder extension” on page 269
• “PowerBuilder extension example” on page 271
• “Using the Visual C++ wizard” on page 276

Creating a PowerBuilder extension

This section gives an overview of the steps involved in building a PowerBuilder extension.

1 Decide on a feature to implement
2 Declare native classes
3 Implement native classes
4 Expose standard functions
5 Build a DLL

Decide on a feature to implement

Before you build a PowerBuilder extension, you first need to identify a problem that can best be solved by writing an extension. This might be a feature that can be coded more efficiently and easily in C++ than in PowerScript, or that requires the use of callback functions or nonstandard datatypes. You may also have access to existing C++ classes that perform the tasks you want to add to a PowerBuilder application.
For some examples of PowerBuilder extensions, see the PowerBuilder 9.0 Code Samples page at http://www.sybase.com/pb9_samples/index.html. (Not available for beta.)

**Declare native classes**

For each native class that the extension will support, declare and implement an ANSI C++ class that inherits from IPBX_NonVisualObject, which is the ancestor class for all PowerBuilder native classes.

The declaration of the class can be placed in a header file. This is a simple prototype:

```cpp
#include "pbext.h"

class MyClass: public IPBX_NonVisualObject
{
public:
    MyClass();
    virtual ~MyClass();
    PBXRESULT Invoke(
        IPB_Session *session,
        pboject obj,
        pbmethodID mid,
        PBCallInfo *ci);

    PBXRESULT Invoke(
        IPB_Session *session,
        pboject obj,
        LPCTSTR MethodName,
        PBCallInfo *ci);

    // function declarations go here

protected:
    // protected arguments and functions go here

private:
    virtual void Destroy();
};
```
Implement native classes

The implementation of the class must include the implementation of the Invoke and Destroy functions, as well as all the methods declared for the class. The Invoke function must be coded to invoke each function in the class, depending on the method ID returned from the GetMethodID function.

```c
enum MethodIDs
{
    mFunc1=0,
    mFunc2
};

PBXRESULT MyClass::Invoke(IPB_Session *session, pbobject obj, pbmethodID mid, PBCallInfo *ci)
{
    PBXRESULT result = PBX_OK;

    switch (mid)
    {
    case mFunc1:
        result = func1(session, obj, ci);
        break;

    case mFunc2:
        result = func2(session, obj, ci);
        break;

    default:
        result = PBX_E_INVOKE_FAILURE;
        break;
    }

    return result;
}

void Destroy()
{
    delete this;
}
```
Creating a PowerBuilder extension

Expose standard functions

Your C++ code must expose a set of standard functions that enable PowerBuilder to recognize each native class, and create, destroy, and invoke methods on instances of the class.

PBX_GetDescription

PBX_GetDescription is used to pass the descriptions of classes in the PowerBuilder extension to PowerBuilder. You use the pbx2pbd90 tool to extract this information from the DLL file for the PBNI class and create a PBD file that you can then use in a PowerBuilder application.

The syntax of the description is as follows. The syntax element \([\text{Desc}]^*\) indicates that one description can contain multiple classes:

\[
\text{Desc ::= class\_desc [Desc]*} \\
\text{class\_desc ::= class className from parentClass newline} \\
\text{ [methods\_desc]* end class newline} \\
\text{className ::= a PowerBuilder token, cannot duplicate existing group name} \\
\text{parentClass ::= any class inherited from NonVisualObject} \\
\text{newline ::= a newline character} \\
\text{methods\_desc ::= method\_desc [methods\_desc]*} \\
\text{method\_desc ::= func\_desc | sub\_desc | event\_desc} \\
\text{func\_desc ::= function returnType funcName(args\_desc) newline} \\
\text{returnType ::= pbType} \\
\text{pbType ::= any PowerBuilder type | previous declared PBNI class} \\
\text{funcName ::= a PowerBuilder token} \\
\text{args\_desc ::= None | arg\_desc, [args\_desc]*} \\
\text{arg\_desc ::= [ ref | readonly ] pbType argName [array\_desc]} \\
\text{argName ::= a PowerBuilder token}
\]
array_desc ::= 
    array declaration of PowerBuilder

sub_desc ::= 
    subroutine subName(args_desc) newline

event_desc ::= 
    event returnType eventName(args_desc) [event_id] newline

event_id ::= 
    number

Here is a typical description in an extension that includes three classes:

PBXEXPORT LPCTSTR PBXCALL PBX_GetDescription()
{
    static const TCHAR desc[] = {
        "class ClassName1 from NonVisualObject\n"
        "function integer objectFunction(integer a[])\n"
        "subroutine objectSubroutine(integer ai_ref)\n"
        "event integer eventName(integer b) event_id\n"
        "end class\n"

        "class ClassName2 from Exception\n"
        "function integer objectFunction(readonly integer ai)\n"
        "subroutine objectSubroutine(integer arg)\n"
        "event integer eventName(integer arg)\n"
        "event_token_id\n"
        "end class\n"

        "class ClassName3 from Transaction\n"
        "function integer objectFunction(integer arg)\n"
        "subroutine objectSubroutine(integer arg)\n"
        "event integer eventName(integer arg)\n"
        "end class\n"
    };
    return desc;
}

Referencing classes
A PowerBuilder extension module can provide more than one class, but a given class can reference only classes declared before it in the module. Only classes that inherit from NonVisualObject or a descendant of NonVisualObject can be declared.
Creating a PowerBuilder extension

A PowerBuilder native class can provide several functions, subroutines, and events. The syntax follows PowerScript syntax. In the example above, ClassName1 inherits from NonVisualObject, ClassName2 inherits from Exception, and ClassName3 inherits from Transaction.

Creating instances

The PBX_CreateNonVisualObject method enables the PowerBuilder application to create instances of the extension object. The PowerScript CREATE statement maps to this method.

```cpp
PBXEXPORT PBXRESULT PBXCALL PBX_CreateNonVisualObject
{
    IPB_Session* pbsession,
    pbobject pbobj,
    LPCTSTR xtraName,
    IPBX_NonVisualObject **obj
}
{
    PBXRESULT result = PBX_OK;
    
    if ( strcmp( xtraName, "pbnitest" ) == 0 )
    {
        *obj = new PBNITest;
    }
    else
    {
        *obj = NULL;
        result = PBX_E_NO_SUCH_CLASS;
    }
    return result;
};
```

Build a DLL

Using your C++ development tool, build a DLL from your C++ classes.

When you compile and link the C++ code, the include directory for the PBNI SDK, typically PowerBuilder 9.0\SDK\PBNI\include, must be in your include path, the lib directory for the PBNI SDK must be in your library search path, and the PBNI.lib library must be linked into your DLL.
Using a PowerBuilder extension

When the PowerBuilder extension DLL has been created, follow these steps to use the PowerBuilder extension in a PowerBuilder application.

1. Generate a PBD from the DLL
2. Add the PBD to the target’s library list
3. Use the extension

Generate a PBD from the DLL

Use the `pbx2pbd90` tool to create a PBD from the DLL. The syntax of the command is:

```
px2pbd90 [+] destination.pbd src1.dll [ src2.dll src3.dll ...srcn.dll ]
```

You can include multiple DLLs in a single PBD file. If you want to add additional DLLs to an existing PBD, use the optional plus (+) sign before the name of the PBD.

For examples showing how to use the command, see `pbx2pbd90` on page 386.

Using an absolute path

If you execute the `pbx2pbd90` command in the directory where the DLLs reside, the PowerBuilder application will search the system path for the files when you use the extension in the application. If you specify a path, the application searches only in that path.

Add the PBD to the target’s library list

In PowerBuilder, add the PBD to the library list of your PowerScript target, and make sure the C++ DLL is in the system path. Each class in the DLL displays in the System Tree so that you can expand it and drag its function prototypes into your scripts.
Using a PowerBuilder extension

Use the extension

In PowerScript, use the classes in the extension just as you would a custom class user object: declare an instance of the object, use the CREATE statement to create the instance, invoke the object’s functions, and destroy the instance when you have finished with it. You can inherit from the native classes if you want to add functions or events to the class.

Creating a PJNI object instance

At runtime, instances of the native class are created as normal PowerBuilder objects.

In this example, the extension module contains two native classes: fontcallback and fontenumerator. A PowerBuilder custom class user object, nvo_font, inherits from the fontcallback class. These statements create instances of both classes:

```plaintext
fontenumerator fe
nvo_font uf
fe = create fontenumerator
uf = create nvo_font
```

Once an instance of a PJNI class has been created, the PowerBuilder application can call methods on the object. Each PJNI class should implement an `Invoke` method that is called by the PowerBuilder VM when the PowerBuilder application calls one of the PJNI class’s methods. The `Invoke` method then dispatches the method call based on the method ID or method name. The method name is used when a PJNI method is called dynamically.

Using the example above, this statement invokes the `enumprinterfonts` method of the instance of the fontenumerator class:

```plaintext
fe.enumprinterfonts(uf)
```

Destroying the PJNI object instance

When the PowerBuilder application no longer needs an instance of a PJNI class and issues a DESTROY statement, the PowerBuilder VM destroys the instance by calling the native class’s Destroy method.

Debugging

You cannot edit a PJNI object in the PowerBuilder development environment, and you cannot enter PJNI methods in the PowerBuilder debugger because the methods are C++ methods. You must use a C/C++ debugger to debug a PJNI object.
PowerBuilder extension example

This example creates a PowerBuilder extension to perform a simple arithmetic operation. This is not a task that needs PBNI, but it illustrates the basic steps outlined above:

• “Building the pbadd PowerBuilder extension” on page 271
• “Using the extension in PowerBuilder” on page 274

Building the pbadd PowerBuilder extension

In this example, the C++ code is in three files. The class declaration is in a header file, pbadd.h. The standard functions that every PowerBuilder extension must expose are in main.cpp. The implementation of the class is in pbadd.cpp.

❖ To implement the pbadd extension:

1 Code the pbapp.h header file.

The pbadd.h header file declares the pbadd class. It includes pbext.h, which must be included in all PowerBuilder extensions because it declares the ancestor classes for native classes and the standard functions that the extension must expose.

```cpp
#include "pbext.h"

class pbadd: public IPBX_NonVisualObject
{
    public:
        pbadd();
        virtual ~pbadd();
        PBXRESULT Invoke(
            IPB_Session *session,
            pobject obj,
            pbmethodID mid,
            PBCallInfo *ci);

        PBXRESULT Invoke(
            IPB_Session *session,
            pobject obj,
            LPCTSTR MethodName,
            PBCallInfo *ci);

        int f_add(IPB_Session*, pbint, pbint);
```
private:
    virtual void Destroy();
};

The main.cpp file includes pbadd.h and implements the standard functions, PBX_GetDescription and PBX_CreateNonvisualObject.

PBX_GetDescription is used to pass the descriptions of classes in the extension to PowerBuilder. The PBX_CreateNonVisualObject method creates the object instance. The PowerScript CREATE statement maps to this PBNI method.

#include "pbadd.h"

BOOL APIENTRY DllMain(HANDLE hModule, 
    DWORD ul_reason_for_all, 
    LPVOID lpReserved
)
{
    switch(ul_reason_for_all)
    {
    case DLL_PROCESS_ATTACH:
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}

PBXEXPORT LPCTSTR PBXCALL PBX_GetDescription()
{
    static const TCHAR desc[]={{
        "class pbadd from nonvisualobject \n"
        "function int f_add(int a,int b)\n"
        "end class \n"
    }};
    return desc;
}

PBXEXPORT PBXRESULT PBXCALL PBX_CreateNonVisualObject
( 
    IPB_Session*    pbSession,
    pboject        pbobj,
    LPCSTR          xtraName,
    IPBX_NonVisualObject  **obj
)
Building PowerBuilder Extensions

```cpp
{ 
  if (strcmp(xtraName,"pbadd")==0) 
    { 
    *obj=new pbadd; 
  } 
  return 0; 
};

The pbadd.cpp file includes pbadd.h and contains the implementation of the pbadd class and its single method, f_add.

#include "pbadd.h"

PBXRESULT pbadd::Invoke(IPB_Session* Session, pbobject obj, pbmethodID mid, PBCallInfo* ci) 
{ 
  pbint in_a,in_b,ret; 
  IPB_Value*pArg0 = ci->pArgs->GetAt(0); 
  IPB_Value*pArg1 = ci->pArgs->GetAt(1); 
  pbint in_a = pArg0->GetInt(); 
  pbint in_b = pArg1->GetInt(); 
  switch (mid) 
  { 
    case 0: 
      ret=f_add(Session, in_a,in_b); 
      break; 
    default: 
      break; 
  } 
  ci->returnValue.SetInt(Session,ret); 
  return 0; 
};

PBXRESULT pbadd::Invoke(IPB_Session *Session, pbobject obj, LPCTSTR methodName, PBCallInfo *ci) 
{ 
  return 0; 
}

pbadd:: pbadd() 
{ 
}

pbadd:: ~pbadd() 
{ 
}

int pbadd:: f_add(IPB_Session* session, pbint arg1, pbint arg2) 
{ 
}
PowerBuilder extension example

```cpp
return arg1+arg2;
}
void pbadd::Destroy()
{
    delete this;
}

❖ To compile and link the DLL:
  • In your C++ development tool or on the command line, compile and link the DLL.
    For this example, the generated DLL is called PBNIAdd.DLL.

Using the extension in PowerBuilder

To use the PowerBuilder native class in a PowerBuilder application, you create a PBD file that provides PowerBuilder with the information it needs to instantiate the classes in the DLL and invoke their methods. Then you create a PowerScript target, add the PBD to its library list, and invoke its functions.

❖ To generate a PBD and add it to your library list:
1 The pbx2pbd90 tool generates the PBD. At a command prompt, change directory to the location of the PBNIAdd.DLL file and type the following:
   pbx2pbd90 PBNIAdd.pbd PBNIAdd.dll
   The pbx2pbd90 tool is installed in your Shared\PowerBuilder directory and is therefore in the system PATH environment variable.
2 In PowerBuilder, create a simple application, add PBNIAdd.PBD to the library list, and make sure that PBNIAdd.dll is in the system path.

❖ To invoke the f_add function in PowerBuilder:
1 Create a new window called w_add, and add three single-line edit boxes and a command button to it.
2 Declare an instance variable called mypbadd for the pbadd native class, then add this script to the button’s Clicked event:
   ```cpp
   TRY
       mypbadd = CREATE pbadd
   CATCH (runtimeerror re)
       MessageBox("Application terminating", &
   re.getmessage() )
   Halt
   ```
TRY
    sle_3.text = string (mypbadd.f_add( &
        integer(sle_1.text), integer(sle_2.text)))
CATCH (pbxruntimeerror pbxre)
    MessageBox("Method invocation failed", &
        pbxre.getmessage() )
    Halt
END TRY

The pbadd class displays in the System Tree. You could expand its
function list and drag the f_add function into the script.

3 Add open(w_add) to the application’s Open event.
4 Add DESTROY mypbadd to the window’s Close event.
5 Run the application.

The application should run just as it would if you had created a custom
class user object in PowerBuilder with an f_add function. If PowerBuilder
cannot find PBNIAdd.dll, the runtime error in the Clicked event script will
be triggered and caught.
Using the Visual C++ wizard

Note This feature is new in this beta release and is still under development. It has not yet undergone full internal testing, and users are not expected to test the feature in this beta.

If you use Microsoft Visual C++, you can use a wizard to create a PBNI extension project. The wizard creates a project with .cpp and .h files that contain required code as well as template code to help you get started.

The wizard is installed in the Sybase\PowerBuilder 9.0\SDK\PBNI\wizards directory. To use it with Microsoft Visual C++, the pbext.awx file must be in your Microsoft Visual Studio\Common\MSDev98\Template directory. The installation program attempts to copy the file to this location. You can copy it by hand if necessary.

❖ To create a new PBNI project in Microsoft Visual C++:

2. Select PBNI Extension Wizard, enter a name and location for the project, and click OK.
3. Enter a name for the native class and click Next.
4. If this PowerBuilder extension will be used as an interface by other PowerBuilder native classes, select the Generate Interface check box.
5. Click Finish and review your selections, then click OK to generate the project.

The generated project contains the following files, where Projectname and ClassName are the names you supplied in the wizard:

<table>
<thead>
<tr>
<th>File name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projectname.h, Projectname.cpp</td>
<td>Main header and source file containing required functions for the Projectname extension module: PBX_GetDescription and PBX_CreateNonVisualObject.</td>
</tr>
<tr>
<td>ClassName.h, ClassName.cpp</td>
<td>Skeleton header and source file for the ClassName class. If Generate Interface was selected, ClassName inherits from IClassName. Otherwise, it inherits directly from IPBX_NonVisualObject.</td>
</tr>
<tr>
<td>ClassName_IF.h (if Generate Interface selected)</td>
<td>Skeleton header file containing a declaration, ClassNameDecl, that enables the class to be referenced in the PBX_GetDescription function of another extension interface, and a declaration of the IClassName interface.</td>
</tr>
</tbody>
</table>
Building PowerBuilder Extensions

Setting project options

The wizard adds pbni.lib to the list of libraries that are linked in to the generated DLL, but you must set the path for the library and for the PBNI include files yourself.

❖ To set project options for PBNI library and include files:

1. In Microsoft Visual C++, select Tools>Options and select the Directories tab.

2. Select Include files from the Show Directories For drop-down list, click the New icon, click the browse (...) button, browse to the location of the PowerBuilder 9.0\SDK\PBNI\include directory, and click OK.

3. Select Library files from the Show Directories For drop-down list, click the New icon, click the browse (...) button, browse to the location of the PowerBuilder 9.0\SDK\PBNI\lib directory, and click OK.

4. Click OK to close the Options dialog box.

Building and using the DLL

When you have finished coding the project, select Build>Build Projectname.pbx, where Projectname is the name you specified for the project, to create a DLL with the extension .pbx in the Debug directory.

You can then generate a PBD from the .pbx file and use it as described in “Generate a PBD from the DLL” on page 269.
Using the Visual C++ wizard
Working with PNI

This section describes how to work with PNI:

- Exchanging data with PowerBuilder
- Calling PowerBuilder functions
- Calling PowerBuilder from C++
- Exception handling
- Creating marshaler extensions

Exchanging data with PowerBuilder

You can use the IPB_Session interface or the IPB_Value and IPB_Arguments interfaces to exchange data between PowerBuilder and PBNI. IPB_Session provides most flexibility, but using IPB_Value and IPB_Arguments can help maintain data integrity and consistency.

IPB_Session

IPB_Session is an abstract interface. It defines methods for accessing PowerScript variables, calling PowerScript methods, handling exceptions, and setting a marshaler to convert PowerBuilder data formats to the user’s communication protocol.

IPB_Session contains virtual methods that are implemented in the PowerBuilder VM. The code you write in a PowerBuilder native class interacts with the PowerBuilder VM through the IPB_Session interface.

The IPB_Session interface has sections that correspond to different kinds of PowerBuilder methods:

- PowerBuilder class accessor methods are used to find PowerBuilder classes, call PowerBuilder methods and events, and get and set instance variables of PowerBuilder objects.
Exception-handling methods communicate with the PowerBuilder exception handling mechanism.

Array accessor methods create, destroy, and access PowerBuilder bounded and unbounded arrays.

Typed data access methods create, destroy, and access data of the PowerBuilder types string, double, decimal, blob, date, time, datetime, and so forth.

Proxy access methods provide an interface for the implementation of new protocols.

The Release method releases the IPB_Session object itself.

**IPB_Value**

The IPB_Value interface has a set of helper methods that provide access to information about variables and arguments, including the value’s type, whether the null flag is set, whether it is an array or simple type, and whether it is set by reference.

IPB_Value also has methods that set and get the values of variables and arguments.

**Helper methods**

IPB_Value has the following helper methods:

```cpp
pbint GetType() const;
pbboolean IsArray() const;
pbboolean IsByRef() const;
pbboolean IsObject() const;
pbboolean IsNull() const;
void SetToNull();
```

**Set methods**

The IPB_Value SetTypedData methods include SetInt, SetUint, SetLong, SetUlong, and so forth. These methods automatically set the value represented by IPB_Value to not null. The syntax is:

```cpp
virtual PBXRESULT SetTypedData(TypedData arg);
```

For example, the SetLong method takes an argument of type pblong.

The IPB_Value SetTypedData methods set the datatype of the value represented by IPB_Value to a specific type. If the original type of the value is any, you can set it to any other type. Because the value then has a specific type, setting it later to another type throws an exception.
Get methods

The set of GetTypedData methods includes GetInt, GetUint, GetLong, GetUlong, and so forth. The syntax is:

```cpp
virtual TypedData GetTypedData();
```

For example, the GetString method returns a value of type pbstring.

If the value is null, or if you use a get method that is expected to return one datatype when the value is of a different datatype (such as using GetLong when the datatype is pbarray), the result returned is undetermined.

**IPB_Arguments**

The IPB_Arguments interface enables you to pass arguments between the PowerBuilder VM and PowerBuilder extension modules.

IPB_Arguments has two methods, GetCount to obtain the number of arguments in a method call, and GetAt to obtain the value at a specific index of the PBCallInfo structure. Each argument is represented by a pointer to the IPB_Value interface.

```cpp
PBCallInfo *ci
...
pblong longval = NULL;
IPB_Value* pArg = ci->pArgs->GetAt(0);
if (!pArg->IsNull())
    longval = pArg->GetLong();
```

For examples of the use of these interfaces, see “PNI Reference” on page 295.

**Calling PowerBuilder functions**

You can call PowerBuilder system functions through IPB_Session. The InitCallInfo method simplifies the process of setting up the call information. You need to provide the arguments to the InitCallInfo method, including an identifier for the PowerBuilder function you want to call.

In calls to PowerBuilder functions, you identify the function by its ID and not its name. This is designed to provide superior performance. To get the function’s ID, use the GetMethodID method.
Calling PowerBuilder functions

PbmethodID GetMethodID(pbclass cls, LPCTSTR methodName, PBRoutineType rt, LPCTSTR signature);

The signature argument in this method call is a string representing the method’s return type and arguments. You can obtain this string by expanding the object that contains the function in the System Tree, selecting the function, and selecting Properties from its pop-up menu:

You can also use the pbsig90 command-line tool. For more information about using pbsig90, see pbsig90 on page 383.

The following methods are those you use most frequently to invoke PowerBuilder functions. For descriptions of each method, see IPB_Session interface on page 302.

PbmethodID GetMethodID(pbclass cls, LPCTSTR methodName, PBRoutineType rt, LPCTSTR signature, pbboolean publiconly)
PBXRESULT InitCallInfo(pbclass cls, pbmethodID mid, PBCallInfo *ci)
void FreeCallInfo(PBCallInfo *ci)
PBXRESULT InvokeClassFunction(pbclass cls, pbmethodID mid, PBCallInfo *ci)
PBXRESULT InvokeObjectFunction(pbobject obj, pbmethodID mid, PBCallInfo *ci)
PBXRESULT TriggerEvent(pbobject obj, pbmethodID mid, PBCallInfo *ci)
Example: Calling PowerBuilder functions

In this code fragment, the class and method ID returned by calls to the IPB_Session GetClass and GetMethodID methods are used to initialize a PBCallInfo structure, called “ci”, using the IPB_Session InitCallInfo method.

After a new pbstring variable is created, the value of that string is set to the value of the first argument in the PBCallInfo structure.

```c
BOOL CALLBACK CFontEnumerator::EnumFontProc
{
    LPLOGFONT lplf,
    LPNEWTEXTMETRIC lpntm,
    DWORD FontType,
    LPVOID userData
}
{
    UserData* ud = (UserData*)userData;
    pbclass clz = ud->session->GetClass(ud->object);
    pbmethodID mid = ud->session->GetMethodID
        (clz, "onnewfont", PBRT_EVENT, "IS");

    PBCallInfo ci;
    ud->session->InitCallInfo(clz, mid, &ci);

    // create a new string variable and set its value
    // to the value in the first argument in the
    // PBCallInfo structure
    pbstring str = ud->session->NewString
        (lplf->lfFaceName);
    ci.pArgs->GetAt(0)->SetString(str);

    //
    ud->session->TriggerEvent(ud->object, mid, &ci);

    pbint ret = ci.returnValue->GetInt();

    ud->session->FreeCallInfo(&ci);

    return ret == 1 ? TRUE : FALSE;
}
```
Calling PowerBuilder from C++

A third-party application or server written in C++ can load the PowerBuilder VM and use the PowerBuilder objects in a PowerBuilder application. The process is illustrated by the following example.

Example: Calling PowerBuilder from C++

The PowerBuilder application

To keep the code for this example simple, the PowerBuilder application contains only one custom class user object, nvo_mult. nvo_mult has a single method, f_mult, that returns the product of two integers. The name of the PowerBuilder application is loadpbvm, and the PBL is named loadpbvm.pbl.

Getting the method ID

Before you write the C++ code that invokes the f_mult method, you need its method signature. You pass this signature as the last argument to the GetMethodID method. To get the signature, expand nvo_mult in the System Tree, right-click on the f_mult function, and select Properties from the pop-up menu. The signature displays in the Properties dialog box in the Signature text box.

You can also use the pbsig90 tool:

```
pbsig90 d:\pbls\loadpbvm.pbl
```

In the output of pbsig90, the comment on the last line contains the signature to be passed as the method ID argument to GetMethodID:

```
PB Object Name: loadpbvm

PB Object Name: nvo_mult
  public function integer f_mult (integer arg1, integer arg2)
     /* III */
```

For more information about the pbsig90 tool, see pbsig90 on page 383.

Invoking the method from C++

In your C++ development tool, create a new console application project. The include directory for the PBNI SDK, typically PowerBuilder 9.0/SDK/PBNI/include, must be in your include path, the lib directory for the PBNI SDK must be in your library search path, and the PBNLlib library must be linked into the executable file.
The code for the third-party application creates an IPB_VM object using the PB_GetVM method. Then it creates an IPB_Session object within IPB_VM, using the PowerBuilder application’s name and library list as arguments. Once the session has been created, the C++ application can create PowerBuilder objects and call PowerBuilder functions in that session.

```c
#include "pbext.h"
#include "stdio.h"

typedef PBXEXPORT PBXRESULT (*P_PB_GetVM)(IPB_VM** vm);

int main(int argc, char *argv[]) {
  IPB_Session* session;
  IPB_VM* pbvm = NULL;

  //Load the PowerBuilder VM module
  HINSTANCE hinst = LoadLibrary("pbvm90.dll");
  if ( hinst== NULL) return 0;
  fprintf(stderr, "Loaded PBVM successfully
");

  P_PB_GetVM getvm = (P_PB_GetVM)GetProcAddress
                     (hinst,"PB_GetVM");
  if (getvm == NULL) return 0;
  getvm(&pbvm);
  if (pbvm == NULL) return 0;

  // loadpbvm.pbl must contain an application object
  // named loadpbvm
  LPCTSTR LibList[] = {"D:\pbls\loadpbvm.pbl");
  if ( pbvm->CreateSession("loadpbvm", LibList, 1,
                     &session) != PBX_OK )
    {
      fprintf(stderr, "Error in CreateSession\n");
      return 1;
    }
  fprintf(stderr, "Created session successfully\n");

  // Create the PowerBuilder object contained
  // in loadpbvm.pbl.
  // First find the group that contains the
  // user object nvo_mult
  pgproup group = session->FindGroup("nvo_mult",
                                    pbgroup_userobject);
```
if (group == NULL) return 0;

// Now find the class nvo_mult in the group
pbclass cls = session->FindClass(group,"nvo_mult");
if (cls == NULL) return 0;

// Create an instance of the PowerBuilder object
pbobject pobj = session->NewObject(cls);

// PBCallInfo contains arguments and return value
PBCallInfo ci;

// To call the class member function f_mult, you
// use the pbsig90 tool to get its method ID.
// You provide the ID as the last argument
// of GetMethodID
pbmethodID mid = session->GetMethodID(cls, "f_mult",
PBRI_FUNCTION, "III");

// Initialize call info structure based on method ID
session->InitCallInfo(cls, mid, &ci);

// Set IN arguments. The prototype of the function is
// integer f_mult(integer arg1, integer arg2)
ci.pArgs->GetAt(0)->SetInt(123);
ci.pArgs->GetAt(1)->SetInt(45);

// Call the function
try
{
    session->InvokeObjectFunction(pobj, mid, &ci);

    // Was PB exception thrown?
    if (session->HasExceptionThrown())
    {
        // Handle PB exception
        session->ClearException();
    }
}
catch (...)
{
    // Handle C++ exception
}

// Get the return value and print it to the console
pbint ret = ci.returnValue->GetInt();
When you run the compiled executable file at the command prompt, if the PowerBuilder VM is loaded and the session created successfully, the following output displays in the command window:

- Loaded PBVM successfully
- Created session successfully
- The product of 123 and 45 is 5535

Exception handling

To handle errors, you use the error codes returned from PNI functions. Some functions of the IPB_Session interface return detailed error codes in order to make debugging easier.

Native functions, such as the IPBX_UserObject Invoke function, return either PBX_OK or PBX_FAIL if the extension encounters a serious problem from which it cannot recover.

Whenever the PowerBuilder VM gets PBX_FAIL from a native function, it throws a PBXRuntimeError in the PowerBuilder application. PBXRuntimeError inherits from the PowerBuilder RuntimeError system object and can be caught and handled in a script in the same way as any exception in PowerBuilder.

The IPB_Session interface provides a set of functions to handle exceptions that occur in native code. Use HasExceptionThrown to determine whether an exception occurred. If it has, use GetException to get the current exception object so that it can be handled. If necessary, you can throw exceptions to PowerBuilder with ThrowException. When an exception has been handled, use ClearException to clear it.
Creating marshaler extensions

Marshaler extensions can act as bridges between PowerBuilder and other components, such as CORBA components, Java classes, Web services, and so on, as long as those components can be called from C++.

To create a marshaler extension, build a DLL that contains a class that implements the IPBX_Marshaler interface as well as one or more native classes. The extension must contain code that associates the marshaler with a proxy for the component that you want to call.

The following example provides an overview and shows some of the code used to develop a marshaler extension to call Java classes from PowerScript.

Developing proxies

You need to develop PowerBuilder proxies for the Java classes you want to invoke from PowerBuilder. You might do this using Java reflection, from Java source code directly, or using the `javap` tool. For example, suppose you want to invoke this Java class:

```java
public class Converter {
    public double dollarToYen(double dollar);
    public double yenToEuro(double yen);
}
```

The PowerBuilder proxy for this Java class could be stored in a file called `converter.srx` that looks like this:

```xml
$PBExportHeader$converter.srx
$PBExportComments$Proxy generated for Java class
global type Converter from nonvisualobject
double Converter
forward prototypes
    public: function double dollarToYen(double ad_1) alias for "dollarToYen,(D)D"
    function double yenToEuro(double ad_1) alias for "yenToEuro,(D)D"
end prototypes
```

Notice that both PowerBuilder proxy methods have an alias containing the Java method name and method signature. This is necessary because Java is case sensitive, but PowerBuilder is not. The alias information is used by the extension to find the corresponding Java methods.
To add the proxy to a PowerScript target, select the library where the proxy will be stored in the System Tree, select Import from the pop-up menu, and browse to select `converter.srx`.

A PowerBuilder extension can contain several classes. This example shows some of the code for two of the classes in the Java marshaler extension.

The JavaMarshaler class implements the IPBX.Marshaler interface. This interface is responsible for marshaling requests to a Java object and unmarshaling the result returned from the Java object:

```cpp
#include "JavaMarshaler.h"
#include "JMethod.h"
#include "JavaVMWrapper.h"

// GetModuleHandle enables the PowerBuilder VM
// to determine which DLL contains the extension
extern pbulong g_dll_hModule;

pbulong JavaMarshaler::GetModuleHandle()
{
    return g_dll_hModule;
}

JavaMarshaler::JavaMarshaler
{
    JNIEnv* env,
    pbproxyObject pbobj,
    jobject ejbobj
}

: d_jobject(env->NewGlobalRef(ejbobj)),
  d_pbobject(pbobj)
{
}

JavaMarshaler::~JavaMarshaler()
{
    JNIEnv* env = JavaVMWrapper::instance()->getEnv();

    if (d_object != NULL && env != NULL)
        env->DeleteGlobalRef(d_object);
}

PBXRESULT JavaMarshaler::InvokeRemoteMethod
{
    IPB_Session* session,
    pbproxyObject obj,
    LPCSTR szMethodDesc,
```
Creating marshaler extensions

PBCallInfo* ci
)
{ static char* eFailedToInvokeJavaMethod = "Failed to invoke the EJB method."

JNIEnv* env = JavaVMWrapper::instance()->getEnv();
JMethod method(this, szMethodDesc);
try
{
 if (d_jobject != NULL)
 {
 method.invoke(session, env, d_jobject, ci);
 if (env->ExceptionCheck() == JNI_TRUE)
 {
 string error(eFailedToInvokeJavaMethod);
 error += "\n";
 // Throw PB Exception here
 return PBX_E_INVALID_ARGUMENT;
 }
 }
}
catch(...) {
}

return PBX_OK;
}

void JavaMarshaler::Destroy() {
 delete this;
}

The CJavaVM native class has a CreateJavaVM method that creates an instance of the Java VM and a CreateJavaObject method that associates the Java marshaler with the proxy. This class and its methods display in the System Tree when you add the PBD file for the extension to a PowerScript target.

The CreateJavaObject method is shown here:

#include "CJavaVM.h"
#include "JavaVMWrapper.h"
#include "JavaMarshaler.h"

....

PBXRESULT CJavaVM::CreateJavaObject
{
    IPB_Session *session,
    pbobject obj,
    PBCallInfo *ci
}
{
    enum
    {
        kSuccessful = 0,
        kInvalidJavaClassName = -1,
        kFailedToCreateJavaClass = -2,
        kInvalidProxyName = -3,
        kFailToCreateProxy = -4
    }

    // Get java class name.
    string jclassName;

    {
        pbstring jcn = ci->pArgs->GetAt(1)->GetString();
        if (jcn == NULL)
        {
            ci->returnValue->SetLong(kInvalidJavaClassName);
            return PBX_OK;
        }
        else
        {
            jclassName = session->GetString(jcn);
        }
    }

    // Create java object
    JavaVMWrapper* jvm = JavaVMWrapper::instance();
    JNIEnv* env = jvm->getEnv();
    jclass jcls = env->FindClass(jclassName.c_str());
    jobject jobj = NULL;
    if (jcls != NULL)
    {
    }
Creating marshaler extensions

JLocalRef lrClz(env, jcls);

jmethodID mid =
    env->GetMethodID(jcls, "<init>", "()V");
if (mid != NULL)
{
    jobj = env->NewObject(jcls, mid);
}

// Get PB proxy name
string proxyName;
{
    pbstring pn = ci->pArgs->GetAt(2)->GetString();
    if (pn == NULL)
    {
        ci->returnValue->SetLong(kInvalidProxyName);
        return PBX_OK;
    } else
    {
        proxyName = session->GetString(pn);
    }
}

// Find proxy class
pbgroup group = session->FindGroup
    (proxyName.c_str(), pbgroup_proxy);
if (group == NULL)
{
    ci->returnValue->SetLong(kInvalidProxyName);
    return PBX_OK;
}

pbclass cls =
    session->FindClass(group, proxyName.c_str());
if (cls == NULL)
{
    ci->returnValue->SetLong(kInvalidProxyName);
    return PBX_OK;
}

// Create PB proxy object.
pbproxyObject proxy = session->NewProxyObject(cls);
if (proxy == NULL) {
    ci->returnValue->SetLong(kFailToCreateProxy);
    return PBX_OK;
}

// Create JavaMarshaler
JavaMarshaler* marshaler = new JavaMarshaler(env, proxy, jobj);

// Associate the JavaMarshaler with the PB proxy
session->SetMarshaler(proxy, marshaler);

ci->pArgs->GetAt(0)->SetObject(proxy);

mci->returnValue->SetLong(kSuccessful);

return PBX_OK;

void CJavaVM::Destroy()
{
    delete this;
}

When the code for all the classes in the extension has been written, build the extension DLL, generate a PBD, and add the PBD to your PowerBuilder application’s library list, just as for any other PowerBuilder extension. You also need to import the proxy for the Converter Java class into your PowerScript target as described in “Developing proxies” on page 288.

In the application’s open event, create an instance of the extension’s JavaVM class:

// instance variable: javavm i_jvm
string properties[]
i_jvm = create javavm
string classpath
i_jvm.createjavavm(classpath, properties)

Now you can create an instance of the Converter class using the CreateJavaObject method of the JavaVM instance, and call the conv method on the Converter class:

converter conv
double yen
i_jvm.createjavaobject(conv, "Converter", "converter")
yen = conv.dollarToYen(100.0)
messagebox("Yen", string(yen))

When the `createJavaObject` method of the JavaVM class is called in PowerScript, the PowerBuilder VM calls the corresponding C++ method in the extension. The C++ method creates a Java Converter object through JNI. If successful, the method creates an instance of the PowerBuilder Converter proxy and a JavaMarshaler object, and associates the JavaMarshaler object with the PowerBuilder proxy.

When `conv.dollarToYen(100.0)` is called, the PowerBuilder VM calls the `InvokeRemoteMethod` method on the JavaMarshaler object. This method then delegates the call to the Java Converter object though JNI and returns the result to PowerBuilder.
This section contains reference information for the PowerBuilder Native Interface. The information is presented in the following sections:

- “Error information” on page 295
- “PowerBuilder datatypes” on page 296
- “Datatypes for access to PowerBuilder data” on page 296
- “PNI enumerated types” on page 297
- “Classes, structures, and methods” on page 298

**Error information**

The following table shows the PBXRESULT return values returned from PNI functions.

<table>
<thead>
<tr>
<th>Value of PBXResult</th>
<th>Error code</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBX_OK</td>
<td>0</td>
</tr>
<tr>
<td>PBX_SUCCESS</td>
<td>0</td>
</tr>
<tr>
<td>PBX_FAIL</td>
<td>-1</td>
</tr>
<tr>
<td>PBX_E_NO_REGISTER_FUNCTION</td>
<td>-1</td>
</tr>
<tr>
<td>PBX_E_REGISTRATION_FAILED</td>
<td>-2</td>
</tr>
<tr>
<td>PBX_E_BUILD_GROUP_FAILED</td>
<td>-3</td>
</tr>
<tr>
<td>PBX_E_INVALID_ARGUMENT</td>
<td>-4</td>
</tr>
<tr>
<td>PBX_E_INVOKE_METHOD_INACCESSIBLE</td>
<td>-5</td>
</tr>
<tr>
<td>PBX_E_INVOKE_WRONG_NUM_ARGS</td>
<td>-6</td>
</tr>
<tr>
<td>PBX_E_INVOKE_REFARG_ERROR</td>
<td>-7</td>
</tr>
<tr>
<td>PBX_E_INVOKE_METHOD_AMBIGUOUS</td>
<td>-8</td>
</tr>
<tr>
<td>PBX_E_INVOKE_FAILURE</td>
<td>-9</td>
</tr>
<tr>
<td>PBX_E_MISMATCHED_DATA_TYPE</td>
<td>-10</td>
</tr>
<tr>
<td>PBX_E_OUTOF_MEMORY</td>
<td>-11</td>
</tr>
<tr>
<td>PBX_E_GET_PBVM_FAILED</td>
<td>-12</td>
</tr>
</tbody>
</table>
PowerBuilder datatypes

The following table maps PowerBuilder datatypes to predefined types used in PBNI C++ modules.

<table>
<thead>
<tr>
<th>PowerBuilder type</th>
<th>Predefined type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>pbint</td>
</tr>
<tr>
<td>Uint</td>
<td>pbuint</td>
</tr>
<tr>
<td>Long</td>
<td>pblong</td>
</tr>
<tr>
<td>Longlong</td>
<td>pblonglong</td>
</tr>
<tr>
<td>Ulong</td>
<td>pbulong</td>
</tr>
<tr>
<td>Boolean</td>
<td>pbboolean</td>
</tr>
<tr>
<td>Real</td>
<td>pbreal</td>
</tr>
<tr>
<td>Double</td>
<td>pbdouble</td>
</tr>
<tr>
<td>Decimal</td>
<td>pbdec</td>
</tr>
<tr>
<td>Date</td>
<td>pbdate</td>
</tr>
<tr>
<td>Time</td>
<td>pbtime</td>
</tr>
<tr>
<td>Datetime</td>
<td>pbdatetime</td>
</tr>
<tr>
<td>Char</td>
<td>pbchar</td>
</tr>
<tr>
<td>Blob</td>
<td>pbblob</td>
</tr>
<tr>
<td>String</td>
<td>pbstring</td>
</tr>
<tr>
<td>Powerobject</td>
<td>pbobject</td>
</tr>
</tbody>
</table>

Datatypes for access to PowerBuilder data

The following datatypes enable access to PowerBuilder data.
The pbgroup_type enumerated types are used in IPB_Session FindGroup calls to identify the type of group required.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbgroup</td>
<td>Access PowerBuilder group information. A group is a container of PowerBuilder classes.</td>
</tr>
<tr>
<td>pbclass</td>
<td>Access PowerBuilder class definition information.</td>
</tr>
<tr>
<td>pbmethodID</td>
<td>Access the method ID of a PowerBuilder global or member function.</td>
</tr>
<tr>
<td>pbfieldID</td>
<td>Access an instance variable.</td>
</tr>
<tr>
<td>pbarray</td>
<td>Access array information and data items.</td>
</tr>
</tbody>
</table>

The pbvalue_type enumerated types are used in functions such as the IPB_Value GetType function and the IPB_Session NewUnboundedSimpleArray function to identify the type of PowerBuilder data.

<table>
<thead>
<tr>
<th>Value</th>
<th>PowerBuilder datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbvalue_notype</td>
<td>Undetermined datatype.</td>
</tr>
<tr>
<td>pbvalue_int</td>
<td>int</td>
</tr>
<tr>
<td>pbvalue_uint</td>
<td>uint</td>
</tr>
<tr>
<td>pbvalue_long</td>
<td>long</td>
</tr>
<tr>
<td>pbvalue_longlong</td>
<td>longlong</td>
</tr>
<tr>
<td>pbvalue_ulong</td>
<td>ulong</td>
</tr>
<tr>
<td>pbvalue_real</td>
<td>real</td>
</tr>
</tbody>
</table>
The `pbvalue` enumerated types are used in IPB_Session GetMethodID calls to identify the type of routine required.

<table>
<thead>
<tr>
<th>Value</th>
<th>PowerBuilder datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbvalue_double</td>
<td>double</td>
</tr>
<tr>
<td>pbvalue_dec</td>
<td>decimal</td>
</tr>
<tr>
<td>pbvalue_string</td>
<td>string</td>
</tr>
<tr>
<td>pbvalue_boolean</td>
<td>boolean</td>
</tr>
<tr>
<td>pbvalue_any</td>
<td>any (the type is changed to another type when set explicitly)</td>
</tr>
<tr>
<td>pbvalue_blob</td>
<td>blob</td>
</tr>
<tr>
<td>pbvalue_date</td>
<td>date</td>
</tr>
<tr>
<td>pbvalue_time</td>
<td>time</td>
</tr>
<tr>
<td>pbvalue_datetime</td>
<td>datetime</td>
</tr>
<tr>
<td>pbvalue_char</td>
<td>char</td>
</tr>
</tbody>
</table>

The `pbroutine_type` enumerated types are used in IPB_Session GetMethodID calls to identify the type of routine required.

<table>
<thead>
<tr>
<th>Value</th>
<th>Routine type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PBRT_FUNCTION</td>
<td>Function</td>
</tr>
<tr>
<td>PBRT_EVENT</td>
<td>Event</td>
</tr>
</tbody>
</table>

Classes, structures, and methods

The classes, structures, and methods defined in the header file `pbni.h` enable PowerBuilder extension modules to interact with PowerBuilder. `pbni.h` also includes the `pbarray.h`, `pbfield.h`, and `pbnimd.h` header files.

`pbarray.h` contains helper classes that make it easier to create arrays and access data in them. `pbfield.h` contains a helper class that makes it easier to access fields. Both header files rely on `pbtraits.h`, which provides specializations for the IPB_Value enumerated types. `pbnimd.h` contains machine-specific datatype definitions. These files should not be included directly in your code.
The classes, structures, and methods defined in the header file `pbext.h` must be implemented in PowerBuilder extension modules to enable PowerBuilder applications to use the extension modules.

**Note** In addition to the helper classes added to `pbarray.h` and `pbfield.h`, several additional classes (including PBArrayInfoHolder, PBCallInfoHolder, PBOBJECTCreator, PBEventTrigger, PBOBJECTFunctionInvoker, and PBGlobalFunctionInvoker) have been added to `pbni.h`. These classes have not yet been documented.

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPB_Arguments interface</td>
<td>Used to access the arguments of the PBCallInfo class.</td>
<td><code>pbni.h</code></td>
</tr>
<tr>
<td>IPB_Session interface</td>
<td>Used to interoperate with PowerBuilder. IPB_Session is an abstract interface. It defines methods for accessing PowerScript data, calling PowerScript functions, catching and throwing PowerScript exceptions, and setting a marshaler to convert PowerBuilder data formats to the user’s communication protocol.</td>
<td><code>pbni.h</code></td>
</tr>
<tr>
<td>IPB_Value interface</td>
<td>Used to hold PowerBuilder data. IPB_Value contains information about each variable, including its type, null flag, access privileges, array or simple type, and reference type.</td>
<td><code>pbni.h</code></td>
</tr>
<tr>
<td>IPB_VM struct</td>
<td>Used to load PowerBuilder applications in third-party applications and interoperate with the PowerBuilder virtual machine (PBVM).</td>
<td><code>pbni.h</code></td>
</tr>
<tr>
<td>PBArrayInfo struct</td>
<td>Used to hold information about arrays.</td>
<td><code>pbni.h</code></td>
</tr>
<tr>
<td>PBCallInfo class</td>
<td>Used to hold arguments and return type information in function calls between PBNI and PowerBuilder.</td>
<td><code>pbni.h</code></td>
</tr>
<tr>
<td>PBArrayAccessor template class</td>
<td>Used to access items in an array.</td>
<td><code>pbarray.h</code></td>
</tr>
<tr>
<td>PBOBJECTArrayAccessor class</td>
<td>Used to access items in an object array.</td>
<td><code>pbarray.h</code></td>
</tr>
<tr>
<td>PBObjectArrayCreator template class</td>
<td>Used to create bounded arrays.</td>
<td><code>pbarray.h</code></td>
</tr>
<tr>
<td>PBObjectArrayCreator class</td>
<td>Used to create bounded object arrays.</td>
<td><code>pbarray.h</code></td>
</tr>
<tr>
<td>PBObjectArrayCreator class</td>
<td>Used to create unbounded object arrays.</td>
<td><code>pbarray.h</code></td>
</tr>
<tr>
<td>PBObjectArrayCreator class</td>
<td>Used to create unbounded object arrays.</td>
<td><code>pbarray.h</code></td>
</tr>
</tbody>
</table>
**IPB_Arguments interface**

<table>
<thead>
<tr>
<th>Object</th>
<th>Description</th>
<th>Defined in</th>
</tr>
</thead>
<tbody>
<tr>
<td>IPB_Marshaler interface</td>
<td>Used to invoke remote methods and convert PowerBuilder data formats to the user’s communication protocol. A marshaler extension is a PowerBuilder extension that acts as the bridge between PowerBuilder and other components, such as EJBs, Java classes, CORBA objects, Web services, and so on.</td>
<td>pbext.h</td>
</tr>
<tr>
<td>IPB_NonVisualObject struct</td>
<td>Inherits from IPBX_UserObject and is the direct ancestor class of the PowerBuilder native classes.</td>
<td>pbext.h</td>
</tr>
<tr>
<td>IPBX_UserObject struct</td>
<td>The ancestor class of PowerBuilder native classes. It has two methods, Destroy and Invoke.</td>
<td>pbext.h</td>
</tr>
</tbody>
</table>

**Exported methods**

The following methods must be implemented in the PowerBuilder extension module:
- PBX_CreateNonVisualObject
- PBX_GetDescription

Methods exported by PowerBuilder VM

The PB_GetVM method is exported by the PowerBuilder VM and is used to pass the IPB_VM interface to the user. | pbni.h |

**IPB_Arguments interface**

**Description**

The IPB_Arguments and IPB_Value interfaces are used to pass values between the PowerBuilder VM and PowerBuilder extension modules. Each argument is represented by a pointer to the IPB_Value interface.

The IPB_Arguments interface has two functions, GetAt and GetCount.

**GetAt**

**Description**

Returns a pointer to the IPB_Value interface representing an argument whose order in the list of arguments is indicated by a specified index.

**Syntax**

`GetAt ( pbint index )`

**Argument** | **Description**
--- | ---
`index` | A valid index into the PBCallInfo structure

**Return value**

IPB_Value*.

**Examples**

In the following fragment, GetAt obtains the first value in the PBCallInfo structure. The value has been passed in from the calling function.
PBCallInfo ci;
LPCSTR myPBNIObj = NULL;
IPB_Value* pArg0 = ci->pArgs->GetAt(0);
if (!pArg0->IsNull())
{
    pbstring t = pArg0->GetString();
    if (t != NULL)
        myPBNIObj = session->GetString(t);
}

---

**GetCount**

**Description**

Gets the number of arguments in an instance of PBCallInfo.

**Syntax**

GetCount()

**Return value**

pbint.

**Examples**

This example uses GetCount in a FOR loop used to process different argument types:

```c
int i;
for (i=0; i < ci->pArgs->GetCount(); i++)
{
    pbuint ArgsType;

    if( ci -> pArgs -> GetAt(i) -> IsArray())
    {
        pArguments[i].array_val =
            ci -> pArgs -> GetAt(i) -> GetArray();
        continue;
    }

    if( ci -> pArgs -> GetAt(i) -> IsObject())
    {
        if (ci -> pArgs -> GetAt(i) -> IsNull())
            pArguments[i].obj_val=0;
        else
            pArguments[i].obj_val =
                ci -> pArgs -> GetAt(i) -> GetObject();
        continue;
    }
    ...
```
**IPB_Session interface**

**Description**

The IPB_Session interface is used to interoperate with PowerBuilder. IPB_Session is an abstract interface. It defines methods for accessing PowerScript data, calling PowerScript functions, catching and throwing PowerScript exceptions, and setting a marshaler to convert PowerBuilder data formats to the user’s communication protocol.

**Methods**

The following table lists method by purpose. Full descriptions in alphabetic order follow the table.

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Session management</td>
<td>Release</td>
<td>Releases this IPB_Session. The IPB_Session object becomes invalid after the call.</td>
</tr>
<tr>
<td>Calling functions and events and passing arguments</td>
<td>Add&lt;type&gt;Argument</td>
<td>Adds an argument in a variable argument PowerBuilder call.</td>
</tr>
<tr>
<td></td>
<td>FreeCallInfo</td>
<td>Frees memory allocated by InitCallInfo</td>
</tr>
<tr>
<td></td>
<td>InitCallInfo</td>
<td>Initializes the PBCallInfo structure.</td>
</tr>
<tr>
<td></td>
<td>InvokeClassFunction</td>
<td>Invokes system or user global functions.</td>
</tr>
<tr>
<td></td>
<td>InvokeObjectFunction</td>
<td>Invokes a class member function.</td>
</tr>
<tr>
<td></td>
<td>TriggerEvent</td>
<td>Triggers a PowerBuilder event.</td>
</tr>
<tr>
<td>Referencing objects</td>
<td>FindGroup</td>
<td>Searches for a group with a given name and group type in the current library list.</td>
</tr>
<tr>
<td></td>
<td>FindClass</td>
<td>Searches for a class with a given name within a given group.</td>
</tr>
<tr>
<td></td>
<td>GetClass</td>
<td>Returns the class handle of a PowerBuilder object.</td>
</tr>
<tr>
<td></td>
<td>GetClassName</td>
<td>Returns the name of a class in lowercase.</td>
</tr>
<tr>
<td></td>
<td>GetCurrGroup</td>
<td>Returns the name of the current group.</td>
</tr>
<tr>
<td></td>
<td>GetSuperClass</td>
<td>Returns the base class of a class, if any.</td>
</tr>
<tr>
<td></td>
<td>GetSystemClass</td>
<td>Returns the system class handle of a PowerBuilder object.</td>
</tr>
<tr>
<td></td>
<td>GetSystemFunctionsClass</td>
<td>Returns the class that contains all the system global functions.</td>
</tr>
<tr>
<td></td>
<td>IsAutoInstantiate</td>
<td>Returns true if the specified class is an autoinstantiated class; otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>NewObject</td>
<td>Creates a new object of the specified type.</td>
</tr>
<tr>
<td>Accessing fields</td>
<td>GetFieldID</td>
<td>Gets the internal ID of a class instance variable.</td>
</tr>
<tr>
<td></td>
<td>GetFieldType</td>
<td>Gets the datatype of a class instance variable.</td>
</tr>
<tr>
<td></td>
<td>Get&lt;type&gt;Field</td>
<td>Gets a pointer to the instance variable data for a specified variable.</td>
</tr>
<tr>
<td>Purpose</td>
<td>Method</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td></td>
<td>IsFieldArray</td>
<td>Returns true if the field contains an array, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>IsFieldNull</td>
<td>Returns true if the field contains a null value array, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>IsFieldObject</td>
<td>Returns true if the field contains a pbobject, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>Set&lt;type&gt;Field</td>
<td>A set of functions for setting the value of an instance field of an object.</td>
</tr>
<tr>
<td></td>
<td>SetFieldToNull</td>
<td>Sets the value of a field to null.</td>
</tr>
<tr>
<td>Accessing functions</td>
<td>GetMethodID</td>
<td>Returns the ID of the requested function.</td>
</tr>
<tr>
<td></td>
<td>GetMethodIDByEventID</td>
<td>Returns the ID of the function that has a given predefined PowerBuilder event ID.</td>
</tr>
<tr>
<td>Accessing global variables</td>
<td>GetGlobalVarID</td>
<td>Returns the name of a global variable.</td>
</tr>
<tr>
<td></td>
<td>GetGlobalVarType</td>
<td>Returns the datatype of a global variable.</td>
</tr>
<tr>
<td></td>
<td>Get&lt;type&gt;GlobalVar</td>
<td>Returns the value of a global variable of a specific datatype.</td>
</tr>
<tr>
<td></td>
<td>Set&lt;type&gt;GlobalVar</td>
<td>Sets the value of a global variable of a specific datatype.</td>
</tr>
<tr>
<td></td>
<td>IsGlobalVarArray</td>
<td>Returns true if the global variable contains an array, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>IsGlobalVarNull</td>
<td>Returns true if the global variable contains a null value, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>IsGlobalVarObject</td>
<td>Returns true if the global variable contains a pbobject, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>SetGlobalVarToNull</td>
<td>Sets the value of a shared variable to null.</td>
</tr>
<tr>
<td>Accessing shared variables</td>
<td>GetSharedVarID</td>
<td>Returns the name of a shared variable.</td>
</tr>
<tr>
<td></td>
<td>GetSharedVarType</td>
<td>Returns the datatype of a shared variable.</td>
</tr>
<tr>
<td></td>
<td>Get&lt;type&gt;SharedVar</td>
<td>Returns the value of a shared variable of a specific datatype.</td>
</tr>
<tr>
<td></td>
<td>Set&lt;type&gt;SharedVar</td>
<td>Sets the value of a shared variable of a specific datatype.</td>
</tr>
<tr>
<td></td>
<td>IsSharedVarArray</td>
<td>Returns true if the shared variable contains an array, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>IsSharedVarNull</td>
<td>Returns true if the shared variable contains a null value, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>IsSharedVarObject</td>
<td>Returns true if the shared variable contains a pbobject, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>SetSharedVarToNull</td>
<td>Sets the value of a shared variable to null.</td>
</tr>
</tbody>
</table>
# IPB_Session interface

<table>
<thead>
<tr>
<th>Purpose</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accessing native classes</td>
<td>IsNativeObject</td>
<td>Determines whether a pbobject is an instance of a native class.</td>
</tr>
<tr>
<td></td>
<td>GetNativeInterface</td>
<td>Gets a pointer to the interface of a native class.</td>
</tr>
<tr>
<td>Creating arrays</td>
<td>NewBoundedSimpleArray</td>
<td>Creates a bounded simple data array.</td>
</tr>
<tr>
<td></td>
<td>NewUnboundedSimpleArray</td>
<td>Creates an unbounded simple data array.</td>
</tr>
<tr>
<td></td>
<td>NewBoundedObjectArray</td>
<td>Creates a bounded PowerBuilder object or structure array.</td>
</tr>
<tr>
<td></td>
<td>NewUnboundedObjectArray</td>
<td>Creates an unbounded PowerBuilder object or structure data array.</td>
</tr>
<tr>
<td>Accessing arrays</td>
<td>Get&lt;type&gt;ArrayItem</td>
<td>Returns the value of an array item of a specific datatype.</td>
</tr>
<tr>
<td></td>
<td>GetArrayInfo</td>
<td>Gets information about an array.</td>
</tr>
<tr>
<td></td>
<td>GetArrayLength</td>
<td>Returns the length of an array.</td>
</tr>
<tr>
<td></td>
<td>IsArrayItemNull</td>
<td>Returns true if the array item contains an array, otherwise returns false.</td>
</tr>
<tr>
<td></td>
<td>ReleaseArrayInfo</td>
<td>Releases memory returned by GetArrayInfo.</td>
</tr>
<tr>
<td></td>
<td>Set&lt;type&gt;ArrayItem</td>
<td>Sets the value of an array item of a specific datatype.</td>
</tr>
<tr>
<td></td>
<td>SetArrayItemToNull</td>
<td>Sets the value of an array item to null.</td>
</tr>
<tr>
<td>Accessing strings</td>
<td>GetStringLength</td>
<td>Returns the length of a string in bytes without the terminator.</td>
</tr>
<tr>
<td></td>
<td>GetString</td>
<td>Returns a pointer to the string passed in as an argument.</td>
</tr>
<tr>
<td></td>
<td>NewString</td>
<td>Creates a new string.</td>
</tr>
<tr>
<td></td>
<td>SetString</td>
<td>Frees an existing string and assigns a new string value to it.</td>
</tr>
<tr>
<td>Accessing binary large objects</td>
<td>GetBlob</td>
<td>Returns a pointer to the data buffer for a blob.</td>
</tr>
<tr>
<td></td>
<td>GetBlobLength</td>
<td>Returns the length in bytes of blob data in a buffer.</td>
</tr>
<tr>
<td></td>
<td>NewBlob</td>
<td>Creates a new blob and duplicates a buffer for the new blob data.</td>
</tr>
<tr>
<td></td>
<td>SetBlob</td>
<td>Destroys the existing data in a blob and copies data into it from a buffer.</td>
</tr>
<tr>
<td>Accessing decimal numbers</td>
<td>GetDecimal</td>
<td>Converts decimal data in a pbdec object to a string.</td>
</tr>
<tr>
<td></td>
<td>NewDecimal</td>
<td>Allocates resources for a new decimal data object.</td>
</tr>
<tr>
<td></td>
<td>SetDecimal</td>
<td>Converts a string to a decimal.</td>
</tr>
<tr>
<td>Accessing date and time values</td>
<td>NewDate</td>
<td>Creates a new pbdate data object.</td>
</tr>
<tr>
<td></td>
<td>SetDate</td>
<td>Resets the value of the specified pbdate object.</td>
</tr>
</tbody>
</table>
Add\textit{type}Argument

Description

Adds an argument of a specific type in a variable argument PowerBuilder call.

\begin{description}
\item[Note] This feature is new in this beta release and is still under development. It has not yet undergone full internal testing, and users are not expected to test the feature in this beta.
\end{description}

Syntax

\begin{verbatim}
AddBlobArgument ( PBCallInfo *ci, pbblob value, pbboolean IsNull )
AddBoolArgument ( PBCallInfo *ci, pbboolean value, pbboolean IsNull )
AddCharArgument ( PBCallInfo *ci, pbchar value, pbboolean IsNull )
AddDateArgument ( PBCallInfo *ci, pbdate value, pbboolean IsNull )
AddDateTimeArgument ( PBCallInfo *ci, pbdatetime value, pbboolean IsNull )
AddDecArgument ( PBCallInfo *ci, pbdec value, pbboolean IsNull )
AddDoubleArgument ( PBCallInfo *ci, pbdouble value, pbboolean IsNull )
AddIntArgument ( PBCallInfo *ci, pbint value, pbboolean IsNull )
\end{verbatim}
AddLongArgument (PBCallInfo *ci, pblong value, pbboolean IsNull)
AddLongLongArgument (PBCallInfo *ci, pbllong value, pbboolean IsNull)
AddObjectArgument (PBCallInfo *ci, pbobject value, pbboolean IsNull)
AddRealArgument (PBCallInfo *ci, pbreal value, pbboolean IsNull)
AddStringArgument (PBCallInfo *ci, pbstring value, pbboolean IsNull)
AddStringArgument (PBCallInfo *ci, LPCTSTR value, pbboolean IsNull)
AddTimeArgument (PBCallInfo *ci, pbtime value, pbboolean IsNull)
AddUintArgument (PBCallInfo *ci, pbuint value, pbboolean IsNull)
AddUlongArgument (PBCallInfo *ci, pbulong value, pbboolean IsNull)

Return value
PBXRESULT. PBX_OK on success.

Examples
pbboolean IsNull;
pbint newarg;
...
    session->AddIntArgument (ci, newarg, IsNull)

Usage
This call is used in variable argument PowerBuilder calls, such as
datawindow.retrieve(arg). The value returned by ci->pArgs->GetCount() will
increase by one after the call.

AddGlobalRef

Description
Adds a global reference to the specified PowerBuilder object.

Syntax
AddGlobalRef (pbobject obj)

Return value
pbclass or NULL on error.

Examples
void MyPBNIClass::reference()
{
    d_session->AddGlobalRef(d_pbobject);
}
void MyPBNIClass::unreference()
{
    if(d_pbobject != NULL)
        d_session->RemoveGlobalRef(d_pbobject);
}

See also
RemoveGlobalRef

AddLocalRef
Description
Adds a local reference to the specified PowerBuilder object.
Syntax
AddLocalRef (pbobject obj)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A valid PowerBuilder object handle</td>
</tr>
</tbody>
</table>

Return value
pbclass or NULL on error.
Examples
void MyPBNIClass::reference()
{
    d_session->AddLocalRef(d_pbobject);
}

void MyPBNIClass::unreference()
{
    if(d_pbobject != NULL)
        d_session->RemoveLocalRef(d_pbobject);
}

See also
PopLocalFrame
PushLocalFrame
RemoveLocalRef

ClearException
Description
Clears the current PowerBuilder exception object.
Syntax
ClearException ()
Return value
None.
Usage
HasExceptionThrown returns false after a call to ClearException. If no exception has been thrown, this call has no effect.
See also
GetException
IPB_Session interface

HasExceptionThrown
ThrowException

FindClass
Description
Searches for a class with a given name within a given group.
Syntax
FindClass(pbgroup group, LPCTSTR name)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The handle of the group in which the class resides</td>
</tr>
<tr>
<td>name</td>
<td>The class name in lowercase</td>
</tr>
</tbody>
</table>

Return value
pbclass or NULL on failure.
Examples

```c
    group = session->FindGroup("f_getrow", 
                             pbgroup_function);
    if ( group==NULL )
        return;
    cls = session->FindClass(group, "f_getrow");
    if ( cls==NULL )
        return;
```

Usage
This function searches for a PowerBuilder class with the given name in the given group. For example, in a window definition w_1, w_1 is a group, and w_1 and controls contained in it are all classes of group w_1.

FindGroup
Description
Searches for a group with a given name and group type in the current library list.
Syntax
FindGroup(LPCTSTR name, pbgroup_type type)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>The group name in lowercase</td>
</tr>
<tr>
<td>type</td>
<td>An enumerated type defined in pbgroup_type</td>
</tr>
</tbody>
</table>

Return value
pbgroup or NULL on failure.
Examples

```c
    group = session->FindGroup("user_exception", 
                              pbgroup_userobject);
    if ( group==NULL )
        return;
```
PBNI Reference

```cpp
cls = session->FindClass(group, "user_exception")
```

### FreeCallInfo

**Description**
Frees memory allocated by InitCallInfo.

**Syntax**
```
FreeCallInfo(PBCallInfo *ci)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ci</code></td>
<td>A pointer to the preallocated PBCallInfo structure</td>
</tr>
</tbody>
</table>

**Return value**
None.

**Examples**
```
Session->InvokeObjectFunction(myobj, mid, ci);

ret_val=ci.returnValue->GetInt();
Session->FreeCallInfo(ci);
delete ci;
return ret_val;
```

**Usage**
This function frees memory allocated by InitCallInfo, but does not free the structure `ci` itself.

**See also**
InitCallInfo

### Get<type>ArrayItem

**Description**
Gets the value of an array item of a specified type.

**Syntax**
```
GetBlobArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetBoolArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetCharArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetDateArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetDateTimeArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetDecArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetDoubleArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetIntArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetLongArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetLongLongArrayItem( pbarray array, pblonglong dim[], pbboolean& IsNull )
GetObjectArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetRealArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
```
**IPB_Session interface**

GetStringArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetTimeArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetUintArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )
GetUlongArrayItem( pbarray array, pblong dim[], pbboolean& IsNull )

**Argument | Description**
--- | ---
array | A valid pbarray structure.
dim | The dimension of the array item to be obtained
IsNull | Indicates whether the array item is null

**Return value**
A predefined PBNI datatype that corresponds to the PowerBuilder datatype in the function name.

**Examples**

```c
pbobject pPBObject = NULL;
pbboolean bIsNull = 0;
pblong dim[1];

dim[0] = pb1 + 1;
pPBObject = session->GetObjectArrayItem(array, dim, bIsNull);
```

**See also**
GetArrayInfo
NewUnboundedObjectArray
NewUnboundedSimpleArray
ReleaseArrayInfo

**Get<type>Field**

**Description**
A set of functions for getting the value of an instance field of an object.

**Syntax**
GetArrayField( pbobject obj, pbfieldID fid, pbboolean& isNull )
GetBlobField( pbobject obj, pbfieldID fid, pbboolean& isNull )
GetBoolField( pbobject obj, pbfieldID fid, pbboolean& isNull )
GetCharField( pbobject obj, pbfieldID fid, pbboolean& isNull )
GetDateField( pbobject obj, pbfieldID fid, pbboolean& isNull )
GetDateTimeField( pbobject obj, pbfieldID fid, pbboolean& isNull )
GetDecField( pbobject obj, pbfieldID fid, pbboolean& isNull )
GetDoubleField( pbobject obj, pbfieldID fid, pbboolean& isNull )
GetIntField( pbobject obj, pbfieldID fid, pbboolean& isNull )
GetLongField( pbobject obj, pbfieldID fid, pbboolean& isNull )
GetLongLongField( pbobject obj, pbfieldID fid, pbboolean& isNull )

310
 PBNI Reference

GetObjectField ( pbobject obj, pbfieldID fid, pbboolean&isNull )
GetStringField ( pbobject obj, pbfieldID fid, pbboolean&isNull )
GetTimeField ( pbobject obj, pbfieldID fid, pbit value )
GetUintField ( pbobject obj, pbfieldID fid, pbboolean&isNull )
GetUlongField ( pbobject obj, pbfieldID fid, pbboolean&isNull )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>The handle of the object whose field is to be accessed</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the specified object</td>
</tr>
<tr>
<td>isNull</td>
<td>Indicates whether the field is null</td>
</tr>
</tbody>
</table>

Return value: A predefined PBNI datatype that corresponds to the PowerBuilder datatype in the function name.

Examples:
```c
pbboolean isNull;
pbstring pstr =
    session->GetStringField(proxy, fid, isNull);
if (pstr != NULL)
{
    myclass = session->GetString(pstr);
    // process myclass
}
```

See also: Set<type>Field

**Get<type>GlobalVar**

Description: A set of functions for getting the value of a global variable of a specific datatype.

**Note** This feature is new in this beta release and is still under development. It has not yet undergone full internal testing, and users are not expected to test the feature in this beta.

Syntax:
```c
GetArrayGlobalVar ( pbfieldID fid, pbboolean&isNull )
GetBlobGlobalVar ( pbfieldID fid, pbboolean&isNull )
GetBoolGlobalVar ( pbfieldID fid, pbboolean&isNull )
GetCharGlobalVar ( pbfieldID fid, pbboolean&isNull )
```
IPB_Session interface

GetDateTimeGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetIntGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetLongGlobalVar( pbfieldID fid, pbboolean& isNull )
GetLongLongGlobalVar( pbfieldID fid, pbboolean& isNull )
GetObjectGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetRealGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetStringGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetTimeGlobalVar ( pbfieldID fid, pbint value )
GetUintGlobalVar ( pbfieldID fid, pbboolean& isNull )
GetUlongGlobalVar ( pbfieldID fid, pbboolean& isNull )

Return value
A predefined PBNI datatype that corresponds to the PowerBuilder datatype in the function name.

Examples
fid = session -> GetGlobalVarID("l_gvar");
l_val = session -> GetLongGlobalVar(fid, isNull);
session -> SetLongGlobalVar(fid, l_val + 1);

See also
Set<type>GlobalVar

Get<type>SharedVar

Description
A set of functions for getting the value of a shared variable of a specific datatype.

Note This feature is new in this beta release and is still under development. It has not yet undergone full internal testing, and users are not expected to test the feature in this beta.

Syntax
GetArraySharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetBlobSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetBoolSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetCharSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetDateSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetDateTimeSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetDecSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetDoubleSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetIntSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetLongSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetLongLongSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetObjectSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetRealSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetStringSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetTimeSharedVar ( pbgroup group, pbfieldID fid, pbint value )
GetUintSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )
GetUlongSharedVar ( pbgroup group, pbfieldID fid, pbboolean& isNull )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The group whose shared variable is to be accessed</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the shared variable</td>
</tr>
<tr>
<td>isNull</td>
<td>Indicates whether the variable is null</td>
</tr>
</tbody>
</table>

Return value
A predefined PBNI datatype that corresponds to the PowerBuilder datatype in the function name.

Examples
```cpp
curGroup = session -> GetCurrGroup();
fid = session -> GetSharedVarID(curGroup, "i_svar");
if (fid == 0xffff)
{
    MessageBox(NULL, "Illegal fid!", "default", MB_OK);
    return;
}

i_val = session->GetIntSharedVar(curGroup, fid, isNull);
session->SetIntSharedVar(curGroup, fid, i_val+1);
```

See also
Set<type>SharedVar

GetArrayInfo
Description
Gets information about an array.
Syntax
GetArrayInfo(pbarray array)
IPB_Session interface

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid array handle</td>
</tr>
</tbody>
</table>

Return value: PBArraryInfo*.

Examples:
```
if ( !(ci->pArgs->GetAt(0)->IsNull()) )
{
    array = ci->pArgs->GetAt(0)->GetArray();
    pArrayInfo = session->GetArrayInfo (array);
    pArrayItemCount = session->GetArrayLength(array);
}
else
{
    // NULL array
    pArrayItemCount = 0;
}
```

Usage: If the array is an unbounded array, the bounds information in PBArraryInfo is undetermined. The returned PBArraryInfo must be freed later by ReleaseArrayInfo.

See also:
- NewBoundedObjectArray
- NewBoundedSimpleArray
- ReleaseArrayInfo
- Set<type>ArrayItem

### GetArrayLength

**Description**
Gets the length of an array.

**Syntax**
```
GetArrayLength(parray array)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid array handle</td>
</tr>
</tbody>
</table>

**Return value**: pblong.

**Examples**:
```
if ( !(ci->pArgs->GetAt(0)->IsNull()) )
{
    array = ci->pArgs->GetAt(0)->GetArray();
    pArrayInfo = session->GetArrayInfo (array);
    pArrayItemCount = session->GetArrayLength(array);
}
else
{
// NULL array
pArrayItemCount = 0;
}

See also
NewUnboundedObjectArray
NewUnboundedSimpleArray
ReleaseArrayInfo
Set<type>ArrayItem

GetBlob
Description
Returns a pointer to the data buffer for a blob.
Syntax
GetBlob(pbblob bin)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>A pointer to the source buffer</td>
</tr>
</tbody>
</table>

Return value
void*.

Examples
PBCallInfo* ci = new PBCallInfo;
pbblob ret_val;
pblong bloblen;

    ret_val = ci.returnValue->GetBlob();
bloblen = Session->GetBlobLength(ret_val);
    ret_val = Session->NewBlob
        (Session->GetBlob(ret_val), bloblen);

See also
GetBlobLength
NewBlob
SetBlob

GetBlobLength
Description
Returns the length in bytes of blob data in a buffer.
Syntax
GetBlobLength (pbblob bin)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bin</td>
<td>A pointer to the source buffer</td>
</tr>
</tbody>
</table>

Return value
pblong.

Examples
PBCallInfo* ci = new PBCallInfo;
pbblob ret_val;
pblong bloblen;

ret_val = ci.returnValue->GetBlob();
bloblen = Session->GetBlobLength(ret_val);
ret_val = Session->NewBlob
  (Session->GetBlob(ret_val), bloblen);

See also
GetBlob
NewBlob
SetBlob

GetClass
Description
Returns the class handle of a PowerBuilder object.
Syntax
GetClass (pobject obj)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A valid PowerBuilder object handle</td>
</tr>
</tbody>
</table>

Return value
pbclass or NULL on error.

Examples
BOOL CALLBACK CFontEnumerator::EnumFontProc
{
  LPLOGFONT lpfl,
  LPNEWTEXTMETRIC lpntm,
  DWORD FontType,
  LPVOID userData
}
{
  UserData* ud = (UserData*)userData;
  pbclass clz = ud->session->GetClass(ud->object);
  pbmethodID mid = ud->session->GetMethodID
    (clz, "onnewfont", PBRT_EVENT, "IS");

  PBCallInfo ci;
  ud->session->InitCallInfo(clz, mid, &ci);

  pbstring str = ud->session->NewString
    (lpfl->lffFaceName);
  ci.pArgs->GetAt(0)->SetString(str);
  ud->session->TriggerEvent(ud->object, mid, &ci);
  pbint ret = ci.returnValue->GetInt();
}
ud->session->FreeCallInfo(&ci);

return ret == 1 ? TRUE : FALSE;
}

### GetClassName

**Description**  
Returns the name of a class in lowercase.

**Syntax**  
GetClassName(pbclass cls)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>A valid class handle</td>
</tr>
</tbody>
</table>

**Return value**  
LPCTSTR.

**Examples**  
LPCTSTR NEW_Object::GetPBObjectClassName  
{  
  IPB_Session* session  
  {  
    pbclass pbclass1 = NULL;  
    pbclass1 = session->GetClass((pbobject)m_pbobject);  
    return session->GetClassName (pbclass1);  
  }
}

### GetCurrGroup

**Description**  
Gets the name of the current group.

**Syntax**  
GetCurrGroup( )

**Return value**  
pbgroup or NULL on failure.

**Examples**  
curGroup = session -> GetCurrGroup();  
fid = session -> GetSharedVarID(curGroup,"i_svar");  
if (fid == 0xffff)  
{  
  MessageBox(NULL, "Illegal fid!", "default", MB_OK);  
  return;  
}  
i_val=session -> GetIntSharedVar(curGroup,fid,isNull);
GetDecimal

Description
Converts decimal data in a pbdec object to a string.

Syntax
GetDecimal(pbdec dec, LPCTSTR dest_buf)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dec</td>
<td>The decimal data object to be converted to a string.</td>
</tr>
<tr>
<td>dest_buf</td>
<td>The destination buffer where the string is stored. The buffer must be preallocated with a length greater than 20 bytes.</td>
</tr>
</tbody>
</table>

Return value
PBXRESULT. PBX_OK for success.

Examples
```
case pbvalue_dec:
    if (ci->pArgs->GetAt(i)->IsNull())
    {
        pArguments[i].dec_val = Session->NewDecimal();
        Session->SetDecimal(pArguments[i].dec_val, "1.0");
    }
    else
        pArguments[i].dec_val = 
            ci->pArgs->GetAt(i)->GetDecimal();
    break;
```

See also
NewDecimal
SetDecimal

GetException

Description
Gets the current thrown exception object.

Syntax
GetException ()

Return value
pbobject.

Examples
```
pbclass cls;
pobject ex;
...
ex = session->GetException();
session->ClearException();
cls = session->GetClass(ex);
```
**GetFieldID**

Description: Gets the internal ID of a class instance variable.

Syntax: `GetFieldID(pbclass cls, LPCTSTR fieldName)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cls</code></td>
<td>The class in which the field resides</td>
</tr>
<tr>
<td><code>fieldName</code></td>
<td>The instance member name, in lowercase</td>
</tr>
</tbody>
</table>

Return value: `pbfieldID` or `0xffff` if a field ID cannot be found.

Examples:
```
pbfieldID fid = session->GetFieldID(cls, fieldName);
if(fid == kUndefinedFieldID)
    {
        ci->returnValue->SetLong(kFailedToCreateClass);
        return PBX_E_INVALID_ARGUMENT;
    }

pbboolean isNull;
pbstring pstr = session->GetStringField(proxy, fid, isNull);
```

**GetFieldType**

Description: Gets the datatype of a field declared by a class.

Syntax: `GetFieldType(pbclass cls, pbfieldID fid)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cls</code></td>
<td>The class that defines the field</td>
</tr>
<tr>
<td><code>fid</code></td>
<td>The internal ID of the class instance variable</td>
</tr>
</tbody>
</table>

Return value: `pbint`. A simple datatype defined in the list of `pbvalue_type` enumerated types, such as `pbvalue_int`.

**GetGlobalVarID**

Description: Returns the internal ID of a global variable.

Syntax: `GetGlobalVarID(LPCTSTR name)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>name</code></td>
<td>The name of the global variable in lowercase</td>
</tr>
</tbody>
</table>
IPB_Session interface

Return value
pbfieldID or NULL on failure.

Examples
fid = session -> GetGlobalVarID("l_gvar");
1_val = session -> GetLongGlobalVar(fid, isNull);
session -> SetLongGlobalVar(fid, 1_val + 1);

GetGlobalVarType

Description
Gets the datatype of a global variable.

Syntax
GetGlobalVarType(pbfieldID fid)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fid</td>
<td>The internal ID of the class instance variable</td>
</tr>
</tbody>
</table>

Return value
pbuint. A simple datatype defined in the list of pbvalue_type enumerated types.

GetMethodID

Description
Returns the ID of the requested method.

Syntax
GetMethodID(pbclass cls, LPCTSTR methodName, PBRoutineType rt,
LPCTSTR signature, pbboolean publicOnly)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>pbclass containing the method.</td>
</tr>
<tr>
<td>methodName</td>
<td>The string name of the method in lowercase.</td>
</tr>
<tr>
<td>rt</td>
<td>Type of the method: PBRT_FUNCTION for function or PBRT_EVENT for event.</td>
</tr>
<tr>
<td>signature</td>
<td>Internal signature of the PowerBuilder function, used to identify polymorphic methods in one class. Obtained with the pbsig90 tool. If the signature is a null string (&quot;&quot;), the first method found with the name methodName is returned.</td>
</tr>
<tr>
<td>publicOnly</td>
<td>A boolean that determines whether only public methods are searched (True) or all methods are searched (False). The default is True.</td>
</tr>
</tbody>
</table>

Return value
pbMethodID of the method or kUndefinedMethodID on error.

Examples
pbclass cls;
pbmethodID mid;
cls = session-> GetClass(dwobj);
mid = session->GetMethodID(cls, "retrieve");
PBNI Reference

PBRT_FUNCTION, "LAV");

GetMethodIDByEventID
Description
Returns the ID of the method that has a given predefined PowerBuilder event ID.
Syntax
GetMethodIDByEventID(pbclass cls, LPCTSTR eventId)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>pbclass containing the method</td>
</tr>
<tr>
<td>eventId</td>
<td>A PowerBuilder predefined event string, such as pbnm_bnclicked</td>
</tr>
</tbody>
</table>

Return value
pbMethodID of the method or kUndefinedMethodID on error.

GetNativeInterface
Description
Gets a pointer to the interface of a native class.
Syntax
GetNativeInterface(pbobject obj)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A valid object handle</td>
</tr>
</tbody>
</table>

Return value
IPBX_UserObject

Examples
NewObject* New_Object::GetNewObjectFromPBObject
(IPB_Session* session, pbobject pPBObject)
{
    pbboolean pbbool = 0;
    New_Object* pNew_ObjectRet = NULL;

    pbbool = session->IsNativeObject(pPBObject);

    if (pbbool == 1)
    {
        if (IsADerivedObject(GetPBObjectClassName(session, pPBObject)))
        {
            IPBX_NonVisualObject* pIPBX_NonVisualObject = NULL;

            pIPBX_NonVisualObject = (IPBX_NonVisualObject*)
            (session->GetNativeInterface(pPBObject));
        }
pNew_ObjectRet = dynamic_cast<New_Object*>(pIPBX_NonVisualObject);

return pNew_ObjectRet;

Usage
Use this method in conjunction with IsNativeObject to obtain a direct reference to the IPBX_UserObject associated with a native class in the same PowerBuilder extension so that the class and its methods can be accessed directly.

See also IsNativeObject

GetSharedVarID
Description
Returns the internal ID of a shared variable.

Syntax
GetSharedVarID(pbgroup group, LPCTSTR fieldname)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The group to which the shared variable belongs</td>
</tr>
<tr>
<td>fieldname</td>
<td>The name of the field that contains the shared variable, in lowercase</td>
</tr>
</tbody>
</table>

Return value pbfieldID. Retruns 0xffff if the ID cannot be found.

Examples
curGroup = session -> GetCurrGroup();
fid = session -> GetSharedVarID(curGroup,"i_svar");
if (fid == 0xffff)
{
    MessageBox(NULL, "Illegal fid!", "default", MB_OK);
    return;
}
i_val=session -> GetIntSharedVar(curGroup,fid,isNull);
session -> SetIntSharedVar(curGroup,fid,i_val+1);

GetSharedVarType
Description
Gets the datatype of the specified shared variable.

Syntax
GetSharedVarType ( pbgroup group, pbfieldID fid )
getString

**Description**
Returns a pointer to the string passed in as an argument.

**Syntax**
```
GetString (pbstring* string)
```

**Argument** | **Description**
--- | ---
string | A pointer to a pbstring

**Return value**
LPCTSTR.

**Examples**
```
LPCTSTR classPath = NULL;
IPB_Value* pArg0 = ci->pArgs->GetAt(0);
if (!pArg0->IsNull())
{
    pbstring t = pArg0->GetString();
    if (t != NULL)
        classPath = session->GetString(t);
}
```

**See also**
GetStringLength
NewString
SetString

**GetStringLength**

**Description**
Returns the length of a string in bytes without the terminator.

**Syntax**
```
GetStringLength (pbstring* string)
```

**Argument** | **Description**
--- | ---
string | A pointer to the pbstring whose length is to be determined

**Return value**
pblong.

**Examples**
```
LPCTSTR csp;
```
pblong long_val, outstr_val;
csp = session->GetString( outstr_val );
long_val = session->GetStringLength( outstr_val );

See also
GetString
NewString
SetString

GetSuperClass
Description
Returns the ancestor class of the specified class, if any.
Syntax
GetSuperClass(pbclass cls)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>A valid class handle for the descendent class</td>
</tr>
</tbody>
</table>

Return value
pbclass or 0 if the class has no ancestor.
Examples
pbclass cls, cls_parent;
LPCSTR clsname;

cls = Session->GetClass(
(ci->pArgs->GetAt(0)->GetObject());
cls_parent = Session->GetSuperClass(cls);
clsname = Session->GetClassName(cls_parent);

GetSystemClass
Description
Returns the system class handle of a PowerBuilder object.
Syntax
GetSystemClass (LPCTSTR className)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>className</td>
<td>The name of a PowerBuilder predefined class. The name must be in lowercase</td>
</tr>
</tbody>
</table>

Return value
pbclass or NULL on error.
Examples
pbclass pbclass_oleobject = NULL;
pbobject pbobject_oleobject = NULL;
pbclass_oleobject = session->
GetSystemClass ("oleobject");
if (pbclass_oleobject) {
    pbobject_oleobject = session->NewObject (pbclass_oleobject);
}

if (pbobject_oleobject) {
    PBCallInfo pbcallinfo;
    pbmethodID PBMethodID = 0;

    PBMethodID = session->GetMethodID
    (
        (pbclass)pbclass_oleobject,
        (LPCTSTR)TEXT("ConnectToNewObject"),
        (PBRoutineType)PBRT_FUNCTION,
        (LPCTSTR)TEXT("IXS")
    );
    ...

**GetSystemFunctionsClass**

Description: Returns the class that contains all the system global functions.

Syntax: GetSystemFunctionsClass()

Return value: pbclass or NULL on error.

Examples:
```
cls = session->GetSystemFunctionsClass();
mid = session->GetMethodID
    (cls, "double", PBRT_FUNCTION, "DA");
session->InitCallInfo(cls, mid, ci);
    ci->pArgs -> GetAt(0) -> SetString(mystr);
    session -> InvokeClassFunction(cls, mid, ci);
```

**HasExceptionThrown**

Description: Checks for the existence of an exception that has been thrown but not cleared.

Syntax: HasExceptionThrown()

Return value: pbboolean. Returns true if a PowerBuilder exception has been thrown but not cleared.

Examples:
```
try {
```
IPB_Session interface

```c
session->InvokeObjectFunction(pbobj, mid, &ci);
// Was PB exception thrown?
if (session->HasExceptionThrown())
{
    // Handle PB exception
    session->ClearException();
}
```

See also

ClearException
GetException
ThrowException

InitCallInfo

Description

Initializes the PBCallInfo structure.

Syntax

```c
InitCallInfo(pbclass cls, pbmethodID mid, PBCallInfo *ci)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>The pbclass containing the method</td>
</tr>
<tr>
<td>mid</td>
<td>The pbMethodID returned by GetMethodID</td>
</tr>
<tr>
<td>ci</td>
<td>A pointer to a preallocated PBCallInfo structure</td>
</tr>
</tbody>
</table>

Return value

PBXRESULT. Returns PBX_OK on success, and PBX_E_INVALID_ARGUMENT on failure.

Examples

```c
UserData* ud = (UserData*)userData;
pbclass clz = ud->session->GetClass(ud->object);
pbmethodID mid = ud->session->GetMethodID
               (clz, "onnewfont", PBRT_EVENT, "IS");
PBCallInfo ci;

    ud->session->InitCallInfo(clz, mid, &ci);}
```

Usage

On return, this function allocates enough spaces for the arguments, and then initializes the arguments and return value. You must set appropriate values in the PBCallInfo structure. Note that the structure itself must have been allocated before the call.

See also

FreeCallInfo
InvokeClassFunction

Description
Invokes system or user global functions.

Syntax
InvokeClassFunction(pbclass cls, pbmethodID mid, PBCallInfo *ci)

Argument | Description
--- | ---
cls | The class that contains the global function. If this is a system function, cls is obtained with GetSystemFunctionsClass; otherwise, it is obtained with FindGroup and FindClass, with the function name as the group/class name.

mid | The pbMethodID returned by GetMethodID.

ci | A pointer to a preallocated PBCallInfo structure.

Return value
PBXRESULT. Returns PBX_OK for success, or one of the following for failure:
- PBX_E_INVALID_ARGUMENT
- PBX_E_INVOKE_METHOD_INACCESSIBLE
- PBX_E_INVOKE_WRONG_NUM_ARGS
- PBX_E_INVOKE_REFARG_ERROR
- PBX_E_INVOKE_METHOD_AMBIGUOUS
- PBX_E_INVOKE_FAILURE

Examples
```c
pbclass cls;
pbmethodID mid;
PBCallInfo* ci = new PBCallInfo;
cls=session->GetSystemFunctionsClass();
mid=session->GetMethodID(cls, "double", PBRT_FUNCTION, "DA");
session->InitCallInfo(cls, mid, ci);
    ci.pArgs->GetAt(0)->SetString(mystr);
    session->InvokeClassFunction(cls, mid, ci);
...```

Usage
On return, this function allocates enough spaces for the arguments, and then initializes arguments and return value. You must set appropriate values in the PBCallInfo structure. Note that the structure itself must have been allocated before the call.

See also
InvokeObjectFunction
IPB_Session interface

InvokeObjectFunction

Description: Invokes a class member function.

Syntax: `InvokeObjectFunction(pbobject obj, pbmethodID mid, PBCallInfo *ci)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj</code></td>
<td>The pbobject containing the method</td>
</tr>
<tr>
<td><code>mid</code></td>
<td>The pbMethodID returned by GetMethodID</td>
</tr>
<tr>
<td><code>ci</code></td>
<td>A pointer to a preallocated PBCallInfo structure</td>
</tr>
</tbody>
</table>

Return value: PBXRESULT. Returns PBX_OK for success, or one of the following for failure:

- PBX_E_INVALID_ARGUMENT
- PBX_E_INVOKE_METHOD_INACCESSIBLE
- PBX_E_INVOKE_WRONG_NUM_ARGS
- PBX_E_INVOKE_REFARG_ERROR
- PBX_E_INVOKE_METHOD_AMBIGUOUS
- PBX_E_INVOKE_FAILURE
- PBX_E_INVOKE_FAILURE

Examples:

```c
pbclass cls;
pbmethodID mid;
PBCallInfo* ci = new PBCallInfo;
pbint ret_val;

cls = session->GetClass(dwobj);
mid = session->GetMethodID
    (cls, "Update", PBRT_FUNCTION, "I");
session->InitCallInfo(cls, mid, ci);

session->InvokeObjectFunction(dwobj, mid, ci);

ret_val = ci.returnValue->GetInt();
session->FreeCallInfo(ci);
delete ci;
return ret_val;
```

See also: InvokeClassFunction

IsArrayItemNull

Description: Returns true if the array item contains a null value, otherwise returns false.

Syntax: `IsArrayItemNull( pbarray array, pblong dim[ ] )`
Returns true if the specified class is an autoinstantiated class; otherwise returns false.

Syntax

IsAutoInstantiate(pbclass)

Return value pbboolean.

IsFieldArray

Returns true if the field of the specified object is an array, otherwise returns false.

Syntax

IsFieldArray(pbclass cls, pbfield fid)

Return value pbboolean.

Examples

fid = session->GetFieldID(cls, "arr2_val");
if (session->IsFieldArray(cls, fid))
{
    arr_val=session->GetArrayField(myobj, fid, isNull);
    ...
}
**IPB_Session interface**

**IsFieldNull**
Description: Returns true if the field of the specified object is a null value, otherwise returns false.

Syntax: `IsFieldNull(pbobject obj, pbfield fid)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A valid object handle for the object whose field is to be accessed</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the specified object</td>
</tr>
</tbody>
</table>

Return value: `pbboolean`.

**IsFieldObject**
Description: Returns true if the field of the specified object is an object, otherwise returns false.

Syntax: `IsFieldObject(pbclass cls, pbfield fid)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>A valid class handle for the class whose field is to be accessed</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the specified object</td>
</tr>
</tbody>
</table>

Return value: `pbboolean`.

**IsGlobalVarArray**
Description: Returns true if the global variable contains an array, otherwise returns false.

Syntax: `IsGlobalVarArray(pbfield fid)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fid</td>
<td>The field ID of the global variable</td>
</tr>
</tbody>
</table>

Return value: `pbboolean`.

**Examples**
```java
fid = session -> GetGlobalVarID("arr2_gvar");
if (session -> IsGlobalVarArray(fid))
{
    arr_val=session -> GetArrayGlobalVar(fid, isNull);
    ...
}
```
### IsGlobalVarNull

**Description**
Returns true if the global variable contains a null value, otherwise returns false.

**Syntax**
```
IsGlobalVarNull( pbfield fid)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fid</td>
<td>The field ID of the global variable</td>
</tr>
</tbody>
</table>

**Return value**
pbboolean.

### IsGlobalVarObject

**Description**
Returns true if the global variable contains an object, otherwise returns false.

**Syntax**
```
IsGlobalVarObject( pbfield fid)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fid</td>
<td>The field ID of the global variable</td>
</tr>
</tbody>
</table>

**Return value**
pbboolean.

### IsNativeObject

**Description**
Determines whether a pbobject is an instance of a native class.

**Syntax**
```
IsNativeObject(pbobject obj)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A valid object handle</td>
</tr>
</tbody>
</table>

**Return value**
Boolean

**Examples**
```cpp
pbboolean pbbool = 0;
pbbool = session->IsNativeObject(pPBObject);
if (pbbool == 1)
{
    if (IsADerivedObject(GetPBObjectClassName (session, pPBObject)))
    {
        IPBX_NonVisualObject* pIPBX_NonVisualObject = NULL;
pIPBX_NonVisualObject = (IPBX_NonVisualObject*) (session->GetNativeInterface(pPBObject));
        // perform special processing
    }
}
IPB_Session interface

Usage
Use this method in conjunction with GetNativeInterface to obtain a direct reference to the IPBX_UserObject associated with a native class in the same PowerBuilder extension, so that the class and its methods can be accessed directly.

See also
GetNativeInterface

IsSharedVarArray
Description
Returns true if the shared variable contains an array, otherwise returns false.

Syntax
IsSharedVarArray(pbgp group, pbfield fid)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The group whose shared variable is to be accessed</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the shared variable</td>
</tr>
</tbody>
</table>

Return value
pbboolean.

IsSharedVarNull
Description
Returns true if the shared variable contains a null value, otherwise returns false.

Syntax
IsSharedVarNull(pbgp group, pbfield fid)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The group whose shared variable is to be accessed</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the shared variable</td>
</tr>
</tbody>
</table>

Return value
pbboolean.

IsSharedVarObject
Description
Returns true if the shared variable contains an object, otherwise returns false.

Syntax
IsSharedVarObject(pbgp group, pbfield fid)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>group</td>
<td>The group whose shared variable is to be accessed</td>
</tr>
<tr>
<td>fid</td>
<td>The field ID of the shared variable</td>
</tr>
</tbody>
</table>

Return value
pbboolean.
NewBlob

Description
Creates a new blob and duplicates a buffer for the new blob data.

Syntax
NewBlob (const void* bin, pblong len)

Argument | Description
--- | ---
bin | A void pointer that points to the source buffer
len | The length in bytes of the data in the buffer

Return value
pbblob.

Examples
if (ci->pArgs->GetAt(i)->IsNull())
    pArguments[i].blob_val = Session->NewBlob("null", 4);
else
    pArguments[i].blob_val = ci->pArgs->GetAt(i)->GetBlob();
...

Usage
The buffer containing the new blob data is freed when PopLocalFrame is called.

See also
SetBlob

NewBoundedObjectArray

Description
Creates a bounded PowerBuilder object or structure array.

Syntax
NewBoundedObjectArray(pbclass cls, pbuint dimension, PBArrayInfo::ArrayBound* bounds)

Argument | Description
--- | ---
cls | A valid class handle of the type of PowerBuilder object or structure array to be created
dimension | A number greater than one that indicates the dimension of the array to be created
bounds | An array containing the upper and lower boundaries of the array to be created

Return value
pbbarray or NULL on failure.

Examples
int size;
pbarray pbin_a;
PBAArrayInfo* ai;
PBXRESULT ret;
pbclass cls;
pbgroup group;
IPB_Session interface

```c
size = sizeof(PBArrayInfo) +
      sizeof(PBArrayInfo::ArrayBound);
ai = (PBArrayInfo*)malloc(size);
ai-> bounds[0].upperBound=2;
ai-> bounds[0].lowerBound=1;
ai-> bounds[1].upperBound=2;
ai-> bounds[1].lowerBound=1;
ai-> numDimensions=2;

// Create new array pbin_a
  group = session-> FindGroup("w_main", pbgroup_window);
  if (group==NULL)
      return;
  cls = session->FindClass(group, "commandbutton");
  if( cls==NULL)
      return;
  pbin_a = session->NewBoundedObjectArray(cls,
      ai-> numDimensions, ai-> bounds);
```

**NewBoundedSimpleArray**

Description: Creates a bounded simple data array.

Syntax: ```NewBoundedSimpleArray(pbuint type, pbuint dimension,
PBArrayInfo::ArrayBound* bounds)```  

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>type</code></td>
<td>An enumerated variable of type <code>pbvalue_</code> indicating the type of simple unbounded array to be created</td>
</tr>
<tr>
<td><code>dimension</code></td>
<td>A number greater than one that indicates the dimension of the array to be created</td>
</tr>
<tr>
<td><code>bounds</code></td>
<td>An array containing the upper and lower boundaries of the array to be created</td>
</tr>
</tbody>
</table>

Return value: `pbarray` or `NULL` on failure.

**NewDate**

Description: Creates a new `pbdate` data object.

Syntax: ```NewDate()```
Return value  pbdate.

Examples  
if (ci->pArgs->GetAt(0)->IsNull())
{
    pArguments[i].date_val = Session->NewDate();
    Session->SetDate(pArguments[i].date_val, 1900,1,1); // Date: 1900-01-01
    isNull[i]=true;
}
else
{
    pArguments[i].date_val = ci->pArgs->GetAt(i)->GetDate();
    isNull[i]=false;
}

Usage  The initial value is 1900-1-1.
See also  SetDate
          SplitDate

NewDateTime

Description  Creates a new pbdatetime data object.

Syntax  NewDateTime()

Return value  pbdatetime.

Examples  
if (ci->pArgs->GetAt(i)->IsNull())
{
    pArguments[i].datetime_val=Session->NewDateTime();
    Session->SetDateTime(pArguments[i].datetime_val, 1900, 1 , 1, 1, 1, 1); // Datetime:
    // 1900-01-01 01:01:01
}
else
{
    pArguments[i].datetime_val =
    ci->pArgs->GetAt(i)->GetDateTime();
}

Usage  The initial value is 0:0:0.0.
See also  SetDateTime
          SplitDateTime
**NewDecimal**

Description

Allocates resources for a new decimal data object.

Syntax

```
NewDecimal()
```

Return value

pbdec or NULL on failure.

Examples

```
if (ci->pArgs->GetAt(i)->IsNull())
{
    pArguments[i].dec_val=Session->NewDecimal();
    Session->SetDecimal(pArguments[i].dec_val,"1.0");
}
else
    pArguments[i].dec_val =
        ci->pArgs->GetAt(i)->GetDecimal();
```

See also

SetDecimal

---

**NewObject**

Description

Creates a new object of the specified type.

Syntax

```
NewObject(pbclass cls)
```

Return value

pbobject of the given class or structure.

Examples

```
pbclass cls;
pbobject ex;
pbgroup group;

   group = session->FindGroup
         ("user_exception", pbgroup_userobject);
   if (group==NULL)
       return;
   cls = session->FindClass(group, "user_exception");
   if (group==NULL)
       return;
   ex = session->NewObject(cls);
```

Usage

The returned object’s life cycle is restricted to the current frame unless AddGlobalRef is called on the object.
NewString
Description Creates a new string.
Syntax NewString(LPCTSTR)
Return value pbstring.
Examples
    pbclass cls;
    cls = session->GetSystemFunctionsClass();
    if( cls == NULL )
    {
        ret_val = session->NewString("null");
        return ret_val;
    }
Usage The returned string is destroyed when PopLocalFrame is called.
See also SetString

NewTime
Description Creates a new pbtime data object.
Syntax NewTime()
Return value pbtime.
Examples
    Session->SplitTime(ci.returnValue->GetTime(), &hh, &mm, &ss);
    ret_val = Session->NewTime();
    Session->SetTime(ret_val, hh, mm, ss);
Usage The initial value is 1900-1-1 0:0:0.0.
See also SetTime
    SplitTime

NewUnboundedObjectArray
Description Creates an unbounded PowerBuilder object or structure data array.
Syntax NewUnboundedObjectArray(pbclass cls)
<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cls</td>
<td>A valid class handle of the type of PowerBuilder object or structure array to be created</td>
</tr>
</tbody>
</table>
IPB_Session interface

Return value
pbarray or NULL on failure.

Usage
An unbounded array can have only one dimension, so no dimension information is needed.

See also
GetArrayInfo
NewUnboundedSimpleArray
ReleaseArrayInfo
Set<type>ArrayItem

NewUnboundedSimpleArray

Description
Creates an unbounded simple data array.

Syntax
NewUnboundedSimpleArray(pbuint type)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>An enumerated variable of type pbvalue_* indicating the type of simple unbounded array to be created</td>
</tr>
</tbody>
</table>

Return value
pbarray or NULL on failure.

Examples
```c
pbin_a = session->NewUnboundedSimpleTypeArray(pbvalue_int);
for ( i=1; i <= 5; i++ )
{
    dim[0] = i;
    pval->SetInt(session, in_a[i-1] );
    ret = session -> SetArrayItem(pbin_a, dim, pval);
}
```

Usage
An unbounded array can have only one dimension, so no dimension information is needed.

See also
GetArrayInfo
NewUnboundedObjectArray
ReleaseArrayInfo
Set<type>ArrayItem
PopLocalFrame
Description
Pops the current local reference frame from the current native method stack frame. All local references to the objects added in that local frame are removed. All the pbobject, pbstring, and pbdecimal variables created by calling NewDecimal, NewObject, or NewString in the current frame are destroyed automatically.
Syntax
PopLocalFrame()
Return value
None.
See also
AddLocalRef
PushLocalFrame
RemoveLocalRef

PushLocalFrame
Description
Pushes a local reference frame onto the current native method stack frame. A local frame is analogous to a scope in C++.
Syntax
PushLocalFrame()
Return value
None.
See also
PopLocalFrame
RemoveLocalRef

Release
Description
Releases the current IPB_Session. The IPB_Session object becomes invalid after the call.
Syntax
Release()
Return value
None.
Examples
if (pIPB_ObjectFactory)
{
    pIPB_ObjectFactory->Release();
    pIPB_ObjectFactory = NULL;
}
**ReleaseArrayInfo**

**Description**
Release memory returned by GetArrayInfo.

**Syntax**
ReleaseArrayInfo(PBArrayInfo* pbarrayinfo)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbarrayinfo</td>
<td>A valid PBArrayInfo handle</td>
</tr>
</tbody>
</table>

**Return value**
PBXRESULT. PBX_OK for success.

**Examples**
PBArryInfo* ai;
...
session->ReleaseArrayInfo(ai);

**Usage**
If the array is an unbounded array, the bounds information in PBArrayInfo is undetermined.

**See also**
GetArrayInfo
NewUnboundedObjectArray
NewUnboundedSimpleArray
Set<type>ArrayItem

---

**RemoveGlobalRef**

**Description**
Removes a global reference to the specified PowerBuilder object.

**Syntax**
RemoveGlobalRef (pbobject obj)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A valid PowerBuilder object handle</td>
</tr>
</tbody>
</table>

**Return value**
None.

**Examples**
void MyPBNIClass::reference()
{  
    d_session->AddGlobalRef(d_pbobject);
}

void MyPBNIClass::unreference()
{  
    if(d_pbobject != NULL)  
        d_session -> RemoveGlobalRef(d_pbobject);
}

**See also**
AddGlobalRef
RemoveLocalRef

Description
Removes a local reference to the specified PowerBuilder object.

Syntax
RemoveLocalRef (pobject obj)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>obj</td>
<td>A valid PowerBuilder object handle</td>
</tr>
</tbody>
</table>

Return value
None.

See also
AddLocalRef
PopLocalFrame
PushLocalFrame

Set<type>ArrayItem

Description
Assigns a value to an array item of a specific type.

Syntax
SetBlobArrayItem ( pbarray array, pblong dim[], pbblob value )
SetBoolArrayItem ( pbarray array, pblong dim[], pbboolean value )
SetCharArrayItem ( pbarray array, pblong dim[], pbchar value )
SetDateArrayItem ( pbarray array, pblong dim[], pbdate value )
SetDateTimeArrayItem ( pbarray array, pblong dim[], pbdatetime value )
SetDecArrayItem ( pbarray array, pblong dim[], pbdec value )
SetDoubleArrayItem ( pbarray array, pblong dim[], pbdouble value )
SetIntArrayItem ( pbarray array, pblong dim[], pbint value )
SetLongArrayItem ( pbarray array, pblong dim[], pblong value )
SetLongLongArrayItem ( pbarray array, pblonglong dim[], pblong value )
SetObjectArrayItem ( pbarray array, pblong dim[], pbobject obj )
SetRealArrayItem ( pbarray array, pblong dim[], pbreal value )
SetStringArrayItem ( pbarray array, pblong dim[], pbstring value )
SetStringArrayItem ( pbarray array, pblong dim[], LPCTSTR value )
SetTimeArrayItem ( pbarray array, pblong dim[], pbtime value )
SetUintArrayItem ( pbarray array, pblong dim[], pbuint value )
SetUlongArrayItem ( pbarray array, pblong dim[], pbulong value )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>array</td>
<td>A valid pbarray handle.</td>
</tr>
</tbody>
</table>
IPB_Session interface

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>A pblong array to hold indexes of each dimension. The number of dimensions must equal the number of dimensions of the array.</td>
</tr>
<tr>
<td>value</td>
<td>The value to be set.</td>
</tr>
</tbody>
</table>

Return value
PBXRESULT. PBX_OK for success.
If the index exceeds the bounds of a bounded array, returns
PBX_E_ARRAY_INDEX_OUTOF_BOUNDS.
If the data passed in does not match the data type of the array, returns
PBX_E_MISMATCHED_DATA_TYPE.

Examples
```
case pbvalue_long:
    {
        pblong pValue;

        for (int i = 1; i <= size; i++)
            {
                dim[0] = i;
                pValue = carr[i - 1];
                d_session->SetLongArrayItem(d_pbarray, dim, pValue);
            }
    }
break;
```

Usage
This function assigns the IPB_Value pointed to by the value argument to the
array item in the same way that the IPB_Value Set<type> function sets a value.

See also
Get<type>ArrayItem
GetArrayInfo
NewUnboundedObjectArray
NewUnboundedSimpleArray
ReleaseArrayInfo

Set<type>Field
Description
A set of functions for setting a new value in an instance field of an object.

Syntax
```
SetArrayField ( pobject obj, pbfieldID fid, pbarray value )
SetBlobField ( pobject obj, pbfieldID fid, pbblob value )
SetBoolField ( pobject obj, pbfieldID fid, pbboolean value )
SetCharField ( pobject obj, pbfieldID fid, pbchar value )
```
SetDateField ( pbobject obj, pbfieldID fid, pbdate value )
SetDateTimeField ( pbobject obj, pbfieldID fid, pbdatetime value )
SetDecField ( pbobject obj, pbfieldID fid, pbdec value )
SetDoubleField ( pbobject obj, pbfieldID fid, pbdouble value )
SetIntField ( pbobject obj, pbfieldID fid, pbint value )
SetLongField ( pbobject obj, pbfieldID fid, pblong value )
SetLongLongField ( pbobject obj, pbfieldID fid, pblonglong value )
SetObjectField ( pbobject obj, pbfieldID fid, pbobject value )
SetRealField ( pbobject obj, pbfieldID fid, pbreal value )
SetStringField ( pbobject obj, pbfieldID fid, pbstring value )
SetStringField ( pbobject obj, pbfieldID fid, LPCTSTR value )
SetTimeField ( pbobject obj, pbfieldID fid, pbtime value )
SetUintField ( pbobject obj, pbfieldID fid, pbuint value )
SetUlongField ( pbobject obj, pbfieldID fid, pbulong value )

Argument | Description
---|---
obj | The handle of the object whose field is to be accessed
fid | The field ID of the specified object
value | The value to be set

Return value
PBX_RESULT.

Examples
pbstring str = session->NewString(d_message.c_str());
if (str != NULL)
    session->SetStringField(d_pbobj, d_fidMsg, str);

See also
Get&type>Field

Set&type>GlobalVar

Description
A set of functions for setting the value of a global variable of a specific datatype.

Note
This feature is new in this beta release and is still under development. It has not yet undergone full internal testing, and users are not expected to test the feature in this beta.

Syntax
SetArrayGlobalVar ( pbfieldID fid, parray value )
IPB_Session interface

SetBlobGlobalVar ( pbfieldID fid, pbblob value )
SetBoolGlobalVar ( pbfieldID fid, pbboolean value )
SetCharGlobalVar ( pbfieldID fid, pbchar value )
SetDateGlobalVar ( pbfieldID fid, pbdate value )
SetDateTimeGlobalVar ( pbfieldID fid, pbdatetime value )
SetDecGlobalVar ( pbfieldID fid, pbdec value )
SetDoubleGlobalVar ( pbfieldID fid, pbdouble value )
SetIntGlobalVar ( pbfieldID fid, pbint value )
SetLongGlobalVar ( pbfieldID fid, pblong value )
SetLongLongGlobalVar ( pbfieldID fid, pblonglong value )
SetObjectGlobalVar ( pbfieldID fid, pbobject value )
SetRealGlobalVar ( pbfieldID fid, pbreal value )
SetStringGlobalVar ( pbfieldID fid, pbstring value )
SetStringGlobalVar ( pbfieldID fid, LPCTSTR value )
SetTimeGlobalVar ( pbfieldID fid, pbtime value )
SetUintGlobalVar ( pbfieldID fid, pbuint value )
SetUlongGlobalVar ( pbfieldID fid, pbulong value )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>fid</td>
<td>The field ID of the global variable</td>
</tr>
<tr>
<td>value</td>
<td>The value to be set</td>
</tr>
</tbody>
</table>

Return value: PBX_RESULT.

Examples:

```cpp
fid = session -> GetGlobalVarID("1_gvar");
l_val = session -> GetLongGlobalVar(fid, isNull);
session -> SetLongGlobalVar(fid, l_val + 1);
```

See also: Get<type>GlobalVar

**Set<type>SharedVar**

Description: A set of functions for setting the value of a shared variable of a specific datatype.

**Note** This feature is new in this beta release and is still under development. It has not yet undergone full internal testing, and users are not expected to test the feature in this beta.
**Syntax**

SetArraySharedVar ( pbgroup group, pbfieldID fid, pbarray value )
SetBlobSharedVar ( pbgroup group, pbfieldID fid, pbblob value )
SetBoolSharedVar ( pbgroup group, pbfieldID fid, pbboolean value )
SetCharSharedVar ( pbgroup group, pbfieldID fid, pbchar value )
SetDateSharedVar ( pbgroup group, pbfieldID fid, pbdate value )
SetDateTimeSharedVar ( pbgroup group, pbfieldID fid, pbdatetime value )
SetDecSharedVar ( pbgroup group, pbfieldID fid, pbdec value )
SetDoubleSharedVar ( pbgroup group, pbfieldID fid, pbdouble value )
SetIntSharedVar ( pbgroup group, pbfieldID fid, pbint value )
SetLongSharedVar( pbgroup group, pbfieldID fid, pblong value )
SetLongLongSharedVar( pbgroup group, pbfieldID fid, pblonglong value )
SetObjectSharedVar ( pbgroup group, pbfieldID fid, pbobject value )
SetRealSharedVar ( pbgroup group, pbfieldID fid, pbreall value )
SetStringSharedVar ( pbgroup group, pbfieldID fid, pbsstring value )
SetStringSharedVar ( pbgroup group, pbfieldID fid, LPCTSTR value )
SetTimeSharedVar ( pbgroup group, pbfieldID fid, pbtime value )
SetUintSharedVar ( pbgroup group, pbfieldID fid, pbuint value )
SetUlongSharedVar ( pbgroup group, pbfieldID fid, pbulong value )

**Return value**

PBX_RESULT.

**See also**

Get<type>SharedVar

---

**SetArrayItemToNull**

**Description**

Sets the value of an array item to a null value.

**Syntax**

SetArrayItemToNull( pbarray array, pblong dim[] )

**Argument** | **Description**
--- | ---
array | A valid pbarray structure
dim | The dimension of the array item to be set to null

**Return value**

pbboolean.
SetBlob

Description
Destroys the existing data in a blob and copies data into it from a buffer.

Syntax
SetBlob (pbblob blb, const void* bin, pblong len)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>blb</td>
<td>A valid pbblob object whose value is to be reset</td>
</tr>
<tr>
<td>bin</td>
<td>A pointer to the source buffer</td>
</tr>
<tr>
<td>len</td>
<td>The length in bytes of the data in the buffer</td>
</tr>
</tbody>
</table>

Return value
PBXRESULT. Returns PBX_OK for success or PBX_E_INVALID_ARGUMENT if the new blob value is invalid. Otherwise returns PBX_E_OUTOF_MEMORY.

Usage
A deep copy is performed. The existing value is destroyed first, then the contents of the bin argument are copied into a new value.

See also
NewBlob

SetDate

Description
Resets the value of the specified pbdate object.

Syntax
SetDate (pbdate date, pbint year, pbint month, pbint day)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>The pbdate object to be reset</td>
</tr>
<tr>
<td>year</td>
<td>A year in the range -9999 to 9999</td>
</tr>
<tr>
<td>month</td>
<td>A month in the range 1 to 12</td>
</tr>
<tr>
<td>day</td>
<td>A day in the range 1 to 31</td>
</tr>
</tbody>
</table>

Return value
PBX_RESULT. PBX_OK for success or PBX_E_INVALID_ARGUMENT if the new date is invalid.

Examples
This example sets the date to March 12, 1938:

    session->SetDate(date_val, 1938, 3, 12);

Usage
If the parameters are invalid, the date is reset to 1900-1-1.

See also
NewDate
SplitDate
SetDateTime

Description: Resets the value of the specified pbdatetime object.

Syntax:

\[
\text{SetDate (pbdatetime } dt, \text{ pbint year, pbint month, pbint day, pbint hour, pbint minute, pbdouble second)}
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( dt )</td>
<td>The pbdatetime object to be reset</td>
</tr>
<tr>
<td>( year )</td>
<td>A year in the range -9999 to 9999</td>
</tr>
<tr>
<td>( month )</td>
<td>A month in the range 1 to 12</td>
</tr>
<tr>
<td>( day )</td>
<td>A day in the range 1 to 31</td>
</tr>
<tr>
<td>( hour )</td>
<td>An hour in the range 0 to 23</td>
</tr>
<tr>
<td>( minute )</td>
<td>A minute in the range 0 to 59</td>
</tr>
<tr>
<td>( second )</td>
<td>A second in the range 0 to 59.32767</td>
</tr>
</tbody>
</table>

Return value: PBX_RESULT. PBX_OK for success or PBX_E_INVALID_ARGUMENT if the new datetime is invalid.

Examples:

This example sets the datetime value to August 19, 1982 at 10:30:45.10:

```
session->SetDate(date_val, 1982, 8, 19, 10, 30, 45.1);
```

Usage: If the parameters are invalid, the datetime value is reset to 1900-1-1 0:0:0.0.

See also:

NewDateTime
SplitDateTime

SetDecimal

Description: Converts a string to a decimal.

Syntax:

\[
\text{SetDecimal(pbedc dec, LPCTSTR dec_str)}
\]

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( dec )</td>
<td>The decimal data object to be created</td>
</tr>
<tr>
<td>( dec_str )</td>
<td>The string containing decimal data to be converted</td>
</tr>
</tbody>
</table>

Return value: PBXRESULT. PBX_OK for success.

Examples:

```
session->SetDecimal(dec_val,"3.4");
```

Usage: If the string contains invalid data, the decimal value is set to 0.0.

See also:

NewDecimal
IPB_Session interface

**SetFieldToNull**
Description Sets the value of the specified field to null.
Syntax `SetFieldToNull(pbobject obj, pbfield fid)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>obj</code></td>
<td>A valid object handle</td>
</tr>
<tr>
<td><code>fid</code></td>
<td>The field ID of the specified object</td>
</tr>
</tbody>
</table>

Return value None.

**SetGlobalVarToNull**
Description Sets the value of the specified global variable to null.
Syntax `SetGlobalVarToNull(pbobject obj, pbfield fid)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>fid</code></td>
<td>The field ID of the global variable</td>
</tr>
</tbody>
</table>

Return value None.

**SetSharedVarToNull**
Description Sets the value of the specified shared variable to null.
Syntax `SetSharedVarToNull(pbgroup group, pbfield fid)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>group</code></td>
<td>The group to which the shared variable belongs</td>
</tr>
<tr>
<td><code>fid</code></td>
<td>The field ID of the shared variable</td>
</tr>
</tbody>
</table>

Return value None.

**SetString**
Description Frees an existing string and assigns a new string value to it.
Syntax `SetString (pbstring string, LPCTSTR src)`
Return value

PBXRESULT. Returns PBX_OK for success or PBX_E_INVALID_ARGUMENT if the new string value is invalid. Otherwise returns PBX_E_OUTOF_MEMORY.

Examples

This example uses the IPB_Session SetString method to set the ret_val string to the return value in the PBCallInfo structure. It also uses the IPB_Value SetString method to set values in PBCallInfo:

```cpp
def foobar():
    pbclass cls;
    pbmethodID mid;
    PBCallInfo* ci = new PBCallInfo;
    pbstring ret_val;
    LPCSTR pStr;

    cls = Session -> GetClass(myobj);
    if (isAny)
        mid = Session-> GetMethodID(cls, "uf_any_byvalue",
                                    PBRT_FUNCTION, "AAAAA");
    else
        mid = Session-> GetMethodID(cls, "uf_string_byvalue",
                                    PBRT_FUNCTION, "SSSSS");
    Session-> InitCallInfo(cls, mid, ci);

    ci-> pArgs -> GetAt(0) -> SetString(s_low);
    ci-> pArgs -> GetAt(1) -> SetString(s_mid);
    ci-> pArgs -> GetAt(2) -> SetString(s_high);
    pStr = Session -> GetString(s_null);
    if (pStr != 0)
        { if (strcmp(pStr, "null") == 0 )
            ci-> pArgs -> GetAt(3) -> SetToNull();
            else
                ci-> pArgs -> GetAt(3) -> SetString(s_null);
        }
    Session -> InvokeObjectFunction(myobj, mid, ci);
    ret_val = Session -> NewString("\n");
    Session -> SetString(ret_val, Session->GetString
                          (ci->returnValue->GetString()));
    Session -> FreeCallInfo(ci);
    delete ci;
    return ret_val;
```
IPB_Session interface

Usage
A deep copy is performed. The existing value is destroyed first, then the contents of the src argument are copied into a new value.

See also
NewString

SetTime

Description
Resets the value of the specified pbtime object.

Syntax
SetTime (pbtime time, pbint hour, pbint minute, pbdouble second)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>The pbtime object to be reset</td>
</tr>
<tr>
<td>hour</td>
<td>An hour in the range 0 to 23</td>
</tr>
<tr>
<td>minute</td>
<td>A minute in the range 0 to 59</td>
</tr>
<tr>
<td>second</td>
<td>A second in the range 0 to 59.32767</td>
</tr>
</tbody>
</table>

Return value
PBX_RESULT. PBX_OK for success or PBX_E_INVALID_ARGUMENT if the new time is invalid.

Examples
```c
if (ci->pArgs->GetAt(i)->IsNull())
{
    pArguments[i].time_val=Session->NewTime();
    Session->SetTime(pArguments[i].time_val, 1, 1, 1);
    // Time: 01:01:01
}
else
{
    pArguments[i].time_val =
        ci->pArgs->GetAt(i)->GetTime();
}
```

Usage
If the parameters are invalid, the time is reset to 0:0:0.0.

See also
NewTime
SplitTime

SplitDate

Description
Splits the specified pbdate object into a year, month, and day.

Syntax
SplitDate (pbdate date, pbint *year, pbint *month, pbint *day)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>The pbdate object to be split</td>
</tr>
</tbody>
</table>
SplitDateTime
Description
Splits the specified pbdatetime object into a year, month, day, hour, minute, and second.
Syntax
SplitDateTime(pbdatetime dt, pbigint *year, pbigint *month, pbigint *day, pbigint *hour, pbigint *minute, pbdouble *second)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dt</td>
<td>The pbdatetime object to be split</td>
</tr>
<tr>
<td>year</td>
<td>A year in the range -9999 to 9999</td>
</tr>
<tr>
<td>month</td>
<td>A month in the range 1 to 12</td>
</tr>
<tr>
<td>day</td>
<td>A day in the range 1 to 31</td>
</tr>
<tr>
<td>hour</td>
<td>An hour in the range 0 to 23</td>
</tr>
<tr>
<td>minute</td>
<td>A minute in the range 0 to 59</td>
</tr>
<tr>
<td>second</td>
<td>A second in the range 0 to 59.32767</td>
</tr>
</tbody>
</table>

Return value
PBXRESULT. PBX_OK for success.

See also
NewDateTime
SetDateTime

SplitTime
Description
Splits the specified time object into an hour, minute, and second.
Syntax
SplitTime(pbstime time, pbigint *hour, pbigint *minute, pbdouble *second)
IPB_Session interface

### SplitTime

**Argument** | **Description**  
--- | ---  
`time` | The `pbtime` object to be split  
`hour` | An hour in the range 0 to 23  
`minute` | A minute in the range 0 to 59  
`second` | A second in the range 0 to 59.32767  

**Return value**  
PBX_RESULT. PBX_OK for success.

**Examples**

```c
Session->SplitTime(ci.returnValue->GetTime(), &hh, &mm, &ss);
ret_val = Session->NewTime();
Session->SetTime(ret_val, hh, mm, ss);
```

**See also**  
NewTime  
SetTime

### ThrowException

**Description**  
Throws a PowerBuilder exception or inherited exception, and replaces the existing exception if there is one.

**Syntax**  
`ThrowException (pbobject ex)`

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ex</code></td>
<td>The exception to be thrown. The exception must first be created with <code>NewObject</code>.</td>
</tr>
</tbody>
</table>

**Return value**  
None.

**Examples**

```c
pbobject ex;
...
session->ThrowException(ex);
```

**See also**  
ClearException  
GetException  
HasExceptionThrown

### TriggerEvent

**Description**  
Triggers a PowerBuilder event.

**Syntax**  
`TriggerEvent(pbobject obj, pbmethodID mid, PBCallInfo *ci)`
Return value

PBXRESULT. Returns PBX_OK for success, or one of the following for failure:

- PBX_E_INVALID_ARGUMENT
- PBX_E_INVOKE_METHOD_INACCESSABLE
- PBX_E_INVOKE_WRONG_NUM_ARGS
- PBX_E_INVOKE_REFARG_ERROR
- PBX_E_INVOKE_METHOD_AMBIGUOUS
- PBX_E_INVOKE_FAILURE
- PBX_E_INVOKE_FAILURE

Examples

// Trigger clicked event on DataWindow

cls = session->GetClass(dwobj);
mid = session->GetMethodID
    (cls, "clicked", PBRT_EVENT, "LIILCdwobject.");
session->InitCallInfo(cls, mid, ci);

session->TriggerEvent(dwobj, mid, ci);
...

IPB_Value interface

Description

The IPB_Arguments and IPB_Value interfaces are used to pass values between the PowerBuilder VM and PowerBuilder extension modules. Through the IPB_Value interface, you can access information about each variable, including its type, null flag, access privileges, array or simple type, and reference type.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetType</td>
<td>Returns the datatype of a single data item or array</td>
</tr>
<tr>
<td>Get&lt;type&gt;</td>
<td>Set of functions returning the datatype of the IPB_Value instance</td>
</tr>
<tr>
<td>IsArray</td>
<td>Returns true if the IPB_Value instance contains an array, otherwise returns false</td>
</tr>
</tbody>
</table>
### IPB_Value interface

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IsByRef</td>
<td>Returns true if the IPB_Value instance is passed by reference</td>
</tr>
<tr>
<td>IsNull</td>
<td>Returns true if the IPB_Value instance contains a null value, otherwise returns false</td>
</tr>
<tr>
<td>IsObject</td>
<td>Returns true if the IPB_Value instance contains an object or object array, otherwise returns false</td>
</tr>
<tr>
<td>SetToNull</td>
<td>Used to set the data contained in the IPB_Value instance to null so that data can be reset.</td>
</tr>
<tr>
<td>Set&lt;type&gt;</td>
<td>Set of functions that set the datatype of the IPB_Value instance</td>
</tr>
</tbody>
</table>

### Get<type>

**Description**
Set of functions returning the datatype of the IPB_Value instance.

**Syntax**
- GetArray ( )
- GetBlob( )
- GetBool ( )
- GetChar ( )
- GetDate ( )
- GetDateTime( )
- GetDecimal ( )
- GetDouble ( )
-GetInt ( )
-GetLong( )
-GetLongLong( )
-GetObject ( )
-GetReal( )
-GetString ( )
-GetTime( )
-GetUint( )
-GetUlong ( )

**Return value**
A predefined PBNI datatype that corresponds to the PowerBuilder datatype in the function name.
Examples

```c
short year, month, day;
TCHAR buffer[40];
pbdate date = val->GetDate();

params.session->SplitDate(date, &year, &month, &day);
sprintf(buffer, "%4d-%02d-%02d", year, month, day);
d_string = buffer;
return 0;
```

Usage

If IPB_Value contains a null value, or if you are trying to get a specific datatype from an IPB_Value instance of another datatype, the data retrieved is undetermined. If the datatype is string, blob, decimal, time, date, datetime, array, or object, the return value points to the same address pointed to by IPB_Value, so changing either the variable that holds the return value or the value of the IPB_Value instance will affect the other.

**GetType**

Description

Returns the datatype of a single data item or array.

Syntax

```c
GetType()
```

Return value

pbint

Examples

```c
ArgsType = ci->pArgs->GetAt(i)->GetType();
switch (ArgsType)
{
    case pbvalue_int:
        if (ci->pArgs->GetAt(i)->IsNull())
            pArguments[i].int_val=1;
        else
            pArguments[i].int_val =
                ci->pArgs->GetAt(i)->GetInt();
        break;
    ...
```

Usage

If the IPB_Value instance contains an object or structure, GetType returns the class ID of the data. Otherwise it returns a simple datatype defined in the list of pbvalue enumerated types.
**IPB_Value interface**

**IsArray**

Description: Returns true if the IPB_Value instance contains an array, otherwise returns false.

Syntax: `IsArray()`

Return value: `pbboolean`

Examples:
```c
if(ci->pArgs->GetAt(i)->IsArray())
{
    pArguments[i].array_val =
    ci->pArgs->GetAt(i)->GetArray();
    continue;
}
```

**IsByRef**

Description: Returns true if the IPB_Value instance contains a by reference argument, otherwise returns false.

Syntax: `IsByRef()`

Return value: `pbboolean`

Examples:
```c
if(ci->pArgs->GetAt(i)->IsByRef())
...
```

**IsNull**

Description: Returns true if the IPB_Value instance contains a null value, otherwise returns false.

Syntax: `IsNull()`

Return value: `pbboolean`

Examples:
```c
if(ci->pArgs->GetAt(i)->IsObject())
{
    if (ci->pArgs->GetAt(i)->IsNull())
        pArguments[i].obj_val=0;
    else
        pArguments[i].obj_val =
        ci->pArgs->GetAt(i)->GetObject();
    continue;
}
...
```
IsObject
Description Returns true if the IPB_Value instance contains an object or object array, otherwise returns false.
Syntax IsObject( )
Return value pbboolean
Examples
   if( ci->pArgs->GetAt(i)->IsObject() )
   {
      if (ci->pArgs->GetAt(i)->IsNull())
         pArguments[i].obj_val = 0;
      else
         pArguments[i].obj_val =
            ci->pArgs->GetAt(i)->GetObject();
      continue;
   }
   ...
### IPB_Value interface

SetUlong (pbulong ulong)

**Return value**

A predefined PBNI datatype that corresponds to the PowerBuilder datatype in the function name.

**Examples**

This example uses the IPB_Value SetString method to set values in PBCallInfo. It also uses the IPB_Session SetString method to set the ret_val string to the return value in the PBCallInfo structure:

```cpp
pbclass cls;
pbmethodID mid;
PBCallInfo* ci = new PBCallInfo;
pbstring ret_val;
LPCTSTR pStr;

cls = Session -> GetClass(myobj);
if (isAny)
    mid = Session -> GetMethodID(cls, "uf_any_byvalue", PBRT_FUNCTION, "AAAAA");
else
    mid = Session -> GetMethodID(cls, "uf_string_byvalue", PBRT_FUNCTION, "SSSSS");
Session -> InitCallInfo(cls, mid, ci);

ci -> pArgs -> GetAt(0) -> SetString(s_low);
ci -> pArgs -> GetAt(1) -> SetString(s_mid);
ci -> pArgs -> GetAt(2) -> SetString(s_high);
pStr = Session -> GetString(s_null);
if (pStr != 0)
    {
        if (strcmp(pStr, "null") == 0 )
            ci -> pArgs -> GetAt(3) -> SetToNull();
        else
            ci -> pArgs -> GetAt(3) -> SetString(s_null);
    }
Session -> InvokeObjectFunction(myobj, mid, ci);
ret_val = Session -> NewString("");
Session -> SetString(ret_val, Session->GetString((ci->returnValue->GetString())));
Session -> FreeCallInfo(ci);
delete ci;
return ret_val;
```

**Usage**

These functions automatically set the value of IPB_Value to not null and throw an exception if the datatype to be set does not match the existing datatype. If the datatype is string or blob, a deep copy is performed. The existing value is destroyed first, then the contents of the argument are copied into a new value.
**SetToNull**

**Description**  
Used to set the data contained in the IPB_Value instance to null so that data can be reset.

**Syntax**  
SetActive()

**Return value**  
None.

**Examples**  
```c
      case pbvalue_blob:
        pStr=(LPCTSTR)Session->GetBlob(retVal.blob_val);
        if (strncmp(pStr, "null", 4)==0 )
          ci->returnValue->SetActive();
        else
          ci->returnValue->SetBlob(retVal.blob_val);
        break;
```

---

**IPB_VM struct**

**Description**  
The IPB_VM struct is used to load PowerBuilder applications in third-party applications and interoperate with the PowerBuilder virtual machine (PBVM). It has three functions:

- CreateSession
- RunApplication

---

**CreateSession**

**Description**  
Creates an IPB_Session object which will be used to call PowerBuilder functions.

**Syntax**  
```c
      CreateSession(LPCTSTR applicationName, LPCTSTR* libraryList, pbuint numLibs, IPB_Session** session)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>applicationName</td>
<td>The name of the current application object in lowercase</td>
</tr>
<tr>
<td>libraryList</td>
<td>The library list of the PowerBuilder application that contains the objects and functions to be called</td>
</tr>
<tr>
<td>numLibs</td>
<td>The number of libraries in the library list</td>
</tr>
<tr>
<td>session</td>
<td>A pointer to IPB_Session*, which will return the current IPB_Session pointer after the call</td>
</tr>
</tbody>
</table>
Return value
PBXRESULT. PBX_OK for success.

Examples
IPB_Session* session;
IPB_VM* vm = NULL;
fstream out;
ifstream in;
PBXRESULT ret;

HINSTANCE hinst=LoadLibrary("pbvm90.dll");
if ( hinst== NULL) return 0;

out<< "Loaded PowerBuilder VM successfully!"<<endl;

P_PB_GetVM getvm = (P_PB_GetVM)GetProcAddress
  (hinst, "PB_GetVM");
if (getvm == NULL) return 0;

getvm(&vm);
if (vm == NULL) return 0;

static const char *liblist[] =
{
  "mydemo.pbl"
};

ret= vm->CreateSession("mydemo", liblist, 1, &session);
if (ret != PBX_OK)
{
  out << "Create session failed." << endl;
  return 0;
}
out << "Create session succeeded!" <<endl;

See also
RunApplication

RunApplication
Description
Runs the application specified.

Syntax
RunApplication(LPCTSTR applicationName, LPCTSTR* libraryList, pbuint numLibs, LPCSTR commandLine, IPB_Session** session)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>applicationName</td>
<td>The name of the application object to be run, in lowercase</td>
</tr>
<tr>
<td>libraryList</td>
<td>The library list of the application</td>
</tr>
</tbody>
</table>

360
Return value
PBXRESULT. PBX_OK for success.

See also
CreateSession

PBArraryInfo struct

Description
PBArraryInfo is a C++ struct used to hold information about arrays.

Properties

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>numLibs</td>
<td></td>
<td>The number of libraries in the library list</td>
</tr>
<tr>
<td>commandLine</td>
<td></td>
<td>Parameters to be passed to the application object</td>
</tr>
<tr>
<td>session</td>
<td>IPB_Session*</td>
<td>A pointer to IPB_Session*, which will return the current IPB_Session pointer after the call</td>
</tr>
</tbody>
</table>

Argument | Description
---|---------------------------------------------------
numLibs | The number of libraries in the library list
commandLine | Parameters to be passed to the application object
session | A pointer to IPB_Session*, which will return the current IPB_Session pointer after the call
PBCallInfo class

Description
PBCallInfo is a C++ class used to hold arguments and return type information in function calls between PBNI and PowerBuilder.

<table>
<thead>
<tr>
<th>Member</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pArgs</td>
<td>IPB_Arguments*</td>
<td>Interface used to access arguments</td>
</tr>
<tr>
<td>returnValue</td>
<td>IPB_Value</td>
<td>Holds return data after the call</td>
</tr>
<tr>
<td>returnClass</td>
<td>pbclass</td>
<td>Holds return class after the call</td>
</tr>
</tbody>
</table>

PBArrayAccessor template class

Description
There are two versions of the PBArrayAccessor template class. The first version is used to access the items in an array of a standard type. The second version is used to access items in a string array. The standard types are defined as ValueTypes in pbtraits.h and are pbint, pbuint, pblong, pblonglong, pbulong, pbboolean, pbreal, pbdouble, pbdec, pbdate, pbtime, pbdatetime, pbchar, pbblob, and pbstring.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetAt</td>
<td>Gets the array item at the specified dimension</td>
</tr>
<tr>
<td>SetAt</td>
<td>Sets the array item at the specified dimension</td>
</tr>
</tbody>
</table>

GetAt

Description
Gets the array item at the specified dimension.

Syntax
GetAt(pblong dim[])

Return value
ValueType (defined in pbtraits.h).

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>The dimension of the array item to be obtained</td>
</tr>
</tbody>
</table>

Examples
See SetAt.
See also

**IsNull**

**Description**

Returns true if the array item contains a null value, otherwise returns false.

**Syntax**

IsNull(pblong dim[ ])

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>The dimension of the array item to be tested</td>
</tr>
</tbody>
</table>

**Return value**

pbboolean.

**Examples**

---

**SetAt**

**Description**

Sets the array item at the specified dimension.

**Syntax**

For arrays of a specified ValueType:

SetAt(pblong dim[ ], ValueType v)

For string arrays:

SetAt(pblong dim[ ], LPCTSTR string)

SetAt(pblong dim[ ], pbstring string)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>The dimension of the array item to be set</td>
</tr>
<tr>
<td>v</td>
<td>A ValueType defined in pbtraits.h</td>
</tr>
<tr>
<td>string</td>
<td>A string of type pbstring or LPCTSTR</td>
</tr>
</tbody>
</table>

**Return value**

None.

**Examples**

This example shows the use of GetAt and SetAt in arrays of a type specified by a ValueType:

```cpp
template < typename T, pbvalue_type I>
void ArrayCreator<T, I>::f_unbounded_simple_array(
    IPB_Session* session,
    ifstream in,
    ofstream out,
    LPCSTR data_type)
{
    parray out_array;
```
int i;
pblong dim[4], itemcount1, itemcount2;

T *iarg, oarg;

in >> itemcount1;
iarg = new T[itemcount1];

// Create unbounded integer array
{
    PBUnboundedArrayCreator<I> ac(session);
    out_array = ac.GetArray();
    PBArraryAccessor<I> aa(session, out_array);
    for(i=0; i<itemcount1; i++)
        in >> iarg[i];
    for (i=0; i<itemcount1; i++)
    {
        dim[0]=i+1;
        aa.SetAt(dim, iarg[i]);
    }
    itemcount2 = session->GetArrayItemCount(out_array);
    out <<"The array item count is " << itemcount2 << endl;
    for (i=0; i<itemcount2; i++)
    {
        dim[0]=i+1;
        oarg=aa.GetAt(dim);
        if (oarg != iarg[i])
            out << "*** ERROR"<< endl;
        else
            out << oarg << " ";
    }
    delete []iarg;
    out << endl;
    return;
}

See also               GetAt

**SetToNull**

Description              Sets the value of the specified array item to null.

Syntax                   SetToNull(pblong dim[])
PBObjectArrayAccessor class

Description
The PBObjectArrayAccessor class is used to access the items in an object array.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetAt</td>
<td>Gets the array item at the specified dimension</td>
</tr>
<tr>
<td>SetAt</td>
<td>Sets the array item at the specified dimension</td>
</tr>
</tbody>
</table>

GetAt

Description
Gets the array item at the specified dimension.

Syntax
GetAt(pblong dim[])

Return value
pboobject.

Examples
This example shows the use of GetAt in an object array:

```c
PBObjectArrayAccessor aa(session, *array_val);
for (i=0;i<itemcount2;i++)
{
    dim[0] = i+1;
    oarg = aa.GetAt(dim);
    cls = session->GetClass(oarg);
    if( cls == NULL )
        return;
    fid = session->GetFieldID(cls, "text");
    if ( fid == 0xffff)
```
 PBObjectArrayAccessor class

```c
return;
    fid_pv = session->GetFieldAddress(oarg,fid);
    mystr = fid_pv->GetString();
    ostr_a[i] = session->GetString(mystr);
}
```

See also SetAt

**SetAt**

**Description**
Sets the array item at the specified dimension.

**Syntax**
```
SetAt(pblong dim[], pbobject obj)
```

**Argument** | **Description**
---|---
`dim` | The dimension of the array item to be set
`obj` | A valid object handle

**Return value**
None.

**Examples**
This example shows the use of SetAt in an object array:

```c
PBObjectArrayAccessor aa(session,*array_val);
for (i=0;i<itemcount1;i++)
{
    cls = session->FindClass(group,sp[i]);
    if( cls == NULL )
        return;
    iarg = session->NewObject(cls);
    session->ReferenceObject(iarg);
    dim[0] = i+1;
    aa.SetAt(dim, iarg);
    fid = session->GetFieldID(cls, "text");
    if ( fid == 0xffff )
        return;
    fid_pv = session->GetFieldAddress(iarg, fid);
    mystr = fid_pv->GetString();
    istr_a[i] = session->GetString(mystr);
}
```

See also GetAt
PBBoundedArrayCreator template class

Description
There are two versions of the PBBoundedArrayCreator template class. The first version is used to create a bounded array of a standard type. The second version is used to create a bounded array of strings. The standard types are defined as ValueTypes in pbtraits.h and are pbint, pbuint, pblong, pblonglong, pbulong, pbboolean, pbreal, pbdouble, pbdec, pbdate, pbtime, pbdatetime, pbchar, pbblob, and pbstring.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetArray</td>
<td>Gets the array created</td>
</tr>
<tr>
<td>SetAt</td>
<td>Sets the array item at the specified dimension</td>
</tr>
</tbody>
</table>

GetArray

Description
Gets the array created.

Syntax
GetArray()

Return value
pbarray.

Examples

```cpp
LPCTSTR *ostr_a;
char **sp;
int i;
pbarray out_array;
arrayBounds* bounds;
pbuint dim1, dim2, current_dim;
pblong itemcount1, itemcount2;
PBXRESULT ret;
PBArraryInfo* ai;
pbstring *iarg, *oarg;
typedef PBBoundedArrayCreator<pbvalue_string>
  BoundedStringArrayCreator;

in >> dim1;
// allocate memory for pointer bounds
bounds = (arrayBounds*)malloc(dim1*sizeof
  (PBArraryInfo::ArrayBound));
bounds = new arrayBounds[dim1];
// read in lowerbound and upperbound for each dimension
// and calculate the array item count
```
PBBoundedArrayCreator template class

```cpp
itemcount1 = 1;
for (i=0; i<dim1; i++)
{
    in >> bounds[i].lowerBound >> bounds[i].upperBound;
    itemcount1 = itemcount1*
    (bounds[i].upperBound - bounds[i].lowerBound +1);
}
sp = new char*[itemcount1];
osstr_a = new LPCTSTR[itemcount1];
iarg = new pbstring[itemcount1];
// Read in array items
for (i=0; i<itemcount1; i++)
{
    sp[i] = new char[20];
in >> sp[i];
iarg[i]= session->NewString(sp[i]);
}
// create bounded simple array and set iarg[i] to it
{
    BoundedStringArrayCreator ac(session, dim1, bounds);
current_dim = 1;
    BoundedArrayItem<pbstring, pbvalue_string, BoundedStringArrayCreator>::f_set_arrayitem
    (session, ac, dim1, bounds, iarg, current_dim);
    BoundedArrayItem<pbstring, pbvalue_string, BoundedStringArrayCreator>::array_itemcount = 0;
    out_array = ac.GetArray();
}
```

See also

SetAt

**SetAt**

**Description** Sets the array item at the specified dimension.

**Syntax**

For arrays of a specified ValueType:

SetAt(pblong *dim[], ValueType v)

For string arrays:

SetAt(pblong *dim[], LPCTSTR string)
SetAt(pblong *dim[], pbstring string)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>The dimension of the array item to be set</td>
</tr>
</tbody>
</table>
Return value

None.

Examples

This example shows the use of SetAt in arrays of a type specified by a ValueType:

```cpp
void BoundedArrayItem<T,I,C>::f_set_arrayitem(IPB_Session* session, C& ac, pblong dimensions, arrayBounds* bounds, T* iarg, int current_dim)
{
    int i;
    if (current_dim > dimensions)
        return;
    for(i= bounds[current_dim-1].lowerBound; i<= bounds[current_dim-1].upperBound; i++)
    {
        if (current_dim == dimensions)
        {
            dim[current_dim-1]= i;
            ac.SetAt(dim,iarg[array_itemcount]);
            array_itemcount++;
        }
        else
        {
            dim[current_dim-1]= i;
            BoundedArrayItem<T,I,C>::f_set_arrayitem(session, ac, dimensions, bounds, iarg, current_dim+1);
        }
    }
}
```

See also

GetArray
**PBBoundedObjectArrayCreator class**

**Description**
The PBBoundedObjectArrayCreator class is used to create an object array.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetArray</td>
<td>Gets the array created</td>
</tr>
<tr>
<td>SetAt</td>
<td>Sets the array item at the specified dimension</td>
</tr>
</tbody>
</table>

**GetArray**

**Description**
Gets the array created.

**Syntax**
GetArray()

**Return value**
pbarray.

**Examples**
```
PBBoundedObjectArrayCreator<pbvalue_string>
ac(session);
for (i=0;i<itemcount1;i++)
{
    ac.SetAt(i+1,iarg[i]);
}
out_array = ac.GetArray();
```

**See also**
SetAt

**SetAt**

**Description**
Sets the array item at the specified dimension.

**Syntax**
For arrays of a specified ValueType:
```
SetAt(pblong dim[], ValueType v)
```

For string arrays:
```
SetAt(pblong dim[], LPCTSTR string)
SetAt(pblong dim[], pbstring string)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>The dimension of the array item to be set</td>
</tr>
<tr>
<td>v</td>
<td>A ValueType defined in pbtraits.h</td>
</tr>
</tbody>
</table>
PBUnboundedArrayCreator template class

Description
There are two versions of the PBUnboundedArrayCreator template class. The first version is used to create a bounded array of a standard type. The second version is used to create a bounded array of strings. The standard types are defined as ValueTypes in `pbtraits.h` and are `pbint`, `pbuint`, `pblong`, `pblonglong`, `pbulong`, `pbboolean`, `pbreal`, `pbdouble`, `pbdec`, `pbdate`, `pbtime`, `pbdatetime`, `pbchar`, `pbblob`, and `pbstring`.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetArray</td>
<td>Gets the array created</td>
</tr>
<tr>
<td>SetAt</td>
<td>Sets the array item at the specified dimension</td>
</tr>
</tbody>
</table>

GetArray

Description
Gets the array created.

Syntax
GetArray()

Return value
`pbarray`

Examples
```
PBUnboundedArrayCreator<pbvalue_string> ac(session);
for (i=0; i<itemcount1; i++)
{
    ac.SetAt(i+1,iarg[i]);
}
out_array = ac.GetArray();
```

See also
SetAt
**PBUndedObjectArrayCreator class**

**SetAt**

**Description**
Sets the array item at the specified position.

**Syntax**
For arrays of a specified ValueType:
```
SetAt(pblong pos, ValueType v)
```
For string arrays:
```
SetAt(pblong pos, LPCTSTR string)
SetAt(pblong pos, pbstring string)
```

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>A pblong identifying a position in the array</td>
</tr>
<tr>
<td>v</td>
<td>A ValueType defined in pbtraits.h</td>
</tr>
<tr>
<td>string</td>
<td>A string of type pbstring or LPCTSTR</td>
</tr>
</tbody>
</table>

**Return value**
None.

**Examples**
This example shows the use of SetAt in arrays of a type specified by a ValueType:
```
PBUndedArrayCreator<I> ac(session);
in >> iarg[i];
for (i=0; i<itemcount1; i++)
{
    ac.SetAt(i+1, iarg[i]);
}
out_array = ac.GetArray();
```

**See also**
GetArray

**PBUndedObjectArrayCreator class**

**Description**
The PBUndedObjectArrayCreator class is used to create an object array.

**Methods**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetArray</td>
<td>Gets the array created</td>
</tr>
<tr>
<td>SetAt</td>
<td>Sets the array item at the specified dimension</td>
</tr>
</tbody>
</table>
GetArray
Description
Gets the array created.
Syntax
GetArray( )
Return value
pbarray.
See also
SetAt

SetAt
Description
Sets the array item at the specified dimension.
Syntax
For arrays of a specified ValueType:
   SetAt( pblong pos, ValueType v )
For string arrays:
   SetAt( pblong pos, LPCTSTR string )
   SetAt( pblong pos, pbstring string )

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pos</td>
<td>A pblong identifying a position in the array</td>
</tr>
<tr>
<td>v</td>
<td>A ValueType defined in pbtraits.h</td>
</tr>
<tr>
<td>string</td>
<td>A string of type pbstring or LPCTSTR</td>
</tr>
</tbody>
</table>

Return value
None.
See also
GetArray

IPBX_Marshaler interface
Description
The IPBX_Marshaler interface is used to invoke remote methods and convert PowerBuilder data formats to the user’s communication protocol. A marshaler extension is a PowerBuilder extension that acts as the bridge between PowerBuilder and other components, such as EJBs, Java classes, CORBA objects, Web services, and so on.

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Destroy</td>
<td>Destroys an instance of an object inherited from the IPBX_Marshaler struct</td>
</tr>
</tbody>
</table>
**IPBX_Marshaler interface**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetModuleHandle</td>
<td>Returns the handle of the DLL that contains the native class</td>
</tr>
<tr>
<td>InvokeRemoteMethod</td>
<td>Use the InvokeRemoteMethod function in PowerBuilder marshaler native classes to call remote methods</td>
</tr>
</tbody>
</table>

**Destroy**

**Description**
Use the Destroy function to destroy instances of objects inherited from the IPBX_Marshaler struct.

**Syntax**
Destroy( )

**Return value**
None.

**Examples**
This code destroys the current instance of the SampleMarshaler struct:
```cpp
void SampleMarshaler::Destroy()
{
    delete this;
}
```

**Usage**
You must implement this function in the marshaler native class after creating an instance of a marshaler struct and invoking remote methods.

**See also**
GetModuleHandle
InvokeRemoteMethod

**GetModuleHandle**

**Description**
Use the GetModuleHandle function to return the handle of the DLL that contains the native class. This function is required to enable the PowerBuilder VM to determine which DLLs can be unloaded.

**Syntax**
GetModuleHandle( )

**Return value**
pbulong

**Examples**
This code in the implementation of a marshaler class returns the handle of the DLL:
```cpp
extern pbulong thisModuleHandle;
pulong SampleMarshaler::GetModuleHandle()
{
```
return thisModuleHandle;
}

The handle is set in the main module:

pbulong thisModuleHandle = 0;

BOOL APIENTRY DllMain( HANDLE hModule,
    DWORD ul_reason_for_call,
    LPVOID lpReserved
)
{
    thisModuleHandle = (pbulong)hModule;

    switch (ul_reason_for_call)
    {
    case DLL_PROCESS_ATTACH:
    case DLL_THREAD_ATTACH:
    case DLL_THREAD_DETACH:
    case DLL_PROCESS_DETACH:
        break;
    }
    return TRUE;
}

Usage
You must implement this function in the marshaler native class.

See also
Destroy
InvokeRemoteMethod

InvokeRemoteMethod

Description
Use the InvokeRemoteMethod function in PowerBuilder marshaler native classes to call remote methods.

Syntax
InvokeRemoteMethod(IPB_Session *session, pbproxyobject obj, LPCTSTR methodDesc, PBCallInfo *ci)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>session</td>
<td>This IPB session.</td>
</tr>
<tr>
<td>obj</td>
<td>The proxy object for the remote object.</td>
</tr>
<tr>
<td>methodDesc</td>
<td>An arbitrary string stored as an alias name for the remote method in the proxy. For example, function int foo(int a) alias &quot;This is a method in remote BizTalk&quot;.</td>
</tr>
<tr>
<td>ci</td>
<td>The parameters and return value setting for the call.</td>
</tr>
</tbody>
</table>
**IPBX_Marshaler interface**

### Return value
PBXRESULT PBX_OK if the call succeeded.

### Examples
This example shows a header file for a sample marshaler class:

```cpp
#include "sampleinclude.h"
#include <pbext.h>

class SampleMarshaler : public IPBX_Marshaler
{
private:
    string d_mystring;
    long d_mylong;

private:
    void myMethod(string arg1);

public:
    SampleMarshaler(
            string myString,
            long mylong
    );
    ~SampleMarshaler();

    virtual PBXRESULT InvokeRemoteMethod
    (
            IPB_Session* session,
            pbproxyObject obj,
            LPCSTR methodDesc,
            PBCallInfo* ci
    );

    virtual pbulong GetModuleHandle();
    virtual void Destroy();
};
```

The associated C++ implementation file contains code like this:

```cpp
PBXRESULT SampleMarshaler::InvokeRemoteMethod
(
    IPB_Session* session,
    pbproxyObject obj,
    LPCSTR methodDesc,
    PBCallInfo* ci
)
{
    // method invocation
}
```
Usage

You must implement this function in the marshaler native class.

See also

Destroy
GetModuleHandle

IPBX_NonVisualObject struct

Description

The IPBX_NonVisualObject structure inherits from IPBX_UserObject and is the
direct ancestor class of the PowerBuilder native classes.

IPBX_UserObject struct

Description

The IPBX_UserObject is the ancestor class of the PowerBuilder native classes.
It has one function: Destroy.

Destroy

Description

Use the Destroy function to destroy the current instance of a PowerBuilder
native class that inherits from IPBX_UserObject.

Syntax

Destroy( )

Return value

None.

Examples

void MyPBNIClass::Destroy()
{
    delete this;
}

Usage

You must implement this function in the native class after creating an instance
of the class and invoking remote methods.

See also

Invoke

Invoke

Description

Use the Invoke function in PowerBuilder objects to call methods in
PowerBuilder native classes.
IPBX_UserObject struct

Syntax
 Invoke(IPB_Session * session, pbobject obj, pbmethodID mid, PBCallInfo *ci)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>session</td>
<td>This IPB session</td>
</tr>
<tr>
<td>obj</td>
<td>The PowerBuilder extension object to be invoked</td>
</tr>
<tr>
<td>mid</td>
<td>The pbMethodID returned by GetMethodID</td>
</tr>
<tr>
<td>ci</td>
<td>The parameters and return value setting for the call</td>
</tr>
</tbody>
</table>

Return value
PBXRESULT.PBX_OK for success.

Examples
In this example, the function invoked depends on the value—0, 1, or 2—of the method ID returned from the GetMethodID function:

```c
PBXRESULT PBNIExt::Invoke
{
    IPB_Session  *session,
    pbobject     obj,
    pbmethodID   mid,
    PBCallInfo   *ci
}
{
    PBXRESULT result = PBX_OK;

    switch (mid)
    {
        case mFuncA:
            result = FuncA(session, obj, ci);
            break;

        case mFuncB:
            result = FuncB(session, obj, ci);
            break;

        case mFuncC:
            result = FuncC(session, obj, ci);
            break;

        default:
            result = PBX_E_INVOKE_FAILURE;
            break;
    }

    return PBX_OK;
}
```

See also
GetMethodID
Exported methods

Description
The following methods must be implemented in the PowerBuilder extension module:

PBX_CreateNonVisualObject
PBX_GetDescription

PBX_CreateNonVisualObject

Description
Used in a PowerBuilder script to create a new instance of a PowerBuilder extension object.

Syntax
PBX_CreateNonVisualObject(IPB_Session* pbsession, pbobject pbobj, LPCTSTR xtraName, IPBX_NonVisualObject **obj)

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pbsession</td>
<td>This IPB session</td>
</tr>
<tr>
<td>pbobj</td>
<td>The name of a pbobject corresponding to the PowerBuilder extension object to be created</td>
</tr>
<tr>
<td>xtraname</td>
<td>The name of the PowerBuilder native class in lowercase</td>
</tr>
<tr>
<td>obj</td>
<td>The PowerBuilder extension object to be created</td>
</tr>
</tbody>
</table>

Return value
PBXRESULT. PBX_OK for success.

Examples
In this example the extension contains several classes. The object created depends on the string value of the class name passed in.

```c
PBXEXPORT PBXRESULT PBXCALL PBX_CreateNonVisualObject
{
    IPB_Session* pbsession,
    pbobject pbobj,
    LPCTSTR xtraName,
    IPBX_NonVisualObject **obj
}
{
    PBXRESULT result = PBX_OK;

    string cn(className);
    if (cn.compare("class_a") == 0)
    {
        *obj = new class_a(pobj);
    }
    else if (cn.compare("class_b") == 0)
    {
```
Exported methods

*obj = new class_b(pbobj);
}  
else if (cn.compare("class_c") == 0)  
{  
  *obj = new class_b(pbobj);
else  
{  
  *obj = NULL;
  result = PBX_E_NO_SUCH_CLASS;
}  

return PBX_OK;
};

Usage
You must implement this function in every PowerBuilder extension module. The function is exported from the extension module and is called from PowerScript when the CREATE statement is used to create a new PowerBuilder extension object.

PBX_GetDescription
Description
Passes a description of all the classes and functions in the PowerBuilder extension module to PowerBuilder.

Syntax
PBX_GetDescription ( )

Return value
LPCTSTR containing the description of the module.

Examples
This extension module contains three classes:

```
PXBEXPORT LPCTSTR PBXCALL PBX_GetDescription()  
{  
  static const TCHAR desc[] = {
    "class class_a from nonvisualobject
    "function long meth1(string classpath)
    "function string meth2()
    "end class

    "class class_b from nonvisualobject
    "subroutine sbrt1()
    "subroutine sbrt2()
    "function long func1()
    "end class

    "class class_c from nonvisualobject
    "end class"
};
```
return desc;
}

Usage
You must implement this function in every PowerBuilder extension module. The function is exported from the PowerBuilder extension module and is used by PowerBuilder to display the prototype of each class.

Method exported by PowerBuilder VM
The following method is exported by the PowerBuilder VM:

PB_GetVM

PB_GetVM
Description
Passes the IPB_VM interface to the user.

Syntax
PB_GetVM (IPB_VM** vm)

Examples
This example loads the PowerBuilder VM and calls the f_getrowcount function on the nvo_dw custom class user object.

```cpp
#include <pbext.h>
#include <iostream.h>
typedef PBXEXPORT PBXRESULT (*P_PB_GetVM)(IPB_VM** vm);

class LibraryLoader
{
public:
    LibraryLoader(LPCSTR libname)
    {
        d_hinst = LoadLibrary(libname);
    }

~LibraryLoader()
    {
        FreeLibrary(d_hinst);
    }

    operator HINSTANCE()
    {
        return d_hinst;
    }
```
```cpp
private:
    HINSTANCE d_hinst;
};

int main()
{
    int int_rowcount;
    PBXRESULT ret;
    LibraryLoader loader("pbvm90.dll");
    if ((HINSTANCE)loader == NULL) return 0;

    HINSTANCE loader(((HINSTANCE)loader, "PB_GetVM");
    if (getvm == NULL) return 0;

    IPB_VM* vm = NULL;
    getvm(&vm);
    if (vm == NULL) return 0;

    static const char *liblist[] =
    {
        "load_pbvm.pbl"
    };

    IPB_Session* session = NULL;
    ret = vm->CreateSession
        ("load_pbvm", liblist, 1, &session);
    if (ret != PBX_OK)
    {
        cout << " Create session failure!" << endl;
        return 0;
    }
    return 1;
}
```

Usage
To load the PowerBuilder VM and run a PowerBuilder application in a
third-party server or application, you first create an IPB_VM object using the
PB_GetVM function. Then create an IPB_Session object within IPB_VM,
using the application’s name and library list as arguments.
Two tools are provided with the PBNI SDK:

- `pbsig90` gets the internal signature of a PowerBuilder function from a PBL name.
- `pbx2pdb90` generates a PBD from a DLL.

When you install PowerBuilder, these tools are installed in the SDK subdirectory of your PowerBuilder 9.0 directory.

---

**pbsig90**

**Description**  
The PowerBuilder function signature is the internal signature of a PowerBuilder function that is used to identify polymorphism functions in a class. The `pbsig90` tool obtains these function signatures from a PBL.

**Note** You can also obtain the signature by selecting the function in the System Tree and selecting Properties from its pop-up menu.

**Syntax**  
`pbsig90 pbl_name`

**Examples**  
This command extracts function signatures from one of the PBLs in the Code Examples sample application:

```
pbsig90 pbexamw1.pbl
```

Here is some of the output from the previous command:

```plaintext
PB Object Name: w_date_sort
   public subroutine of_sort (string as_Column,
                              string as_Order)
   /* QSS */

PB Object Name: w_date_window
   public function boolean of_is_leap_year
      (integer ai_year)
   /* BI */
```
public subroutine of_days ()
    /* Q */

PB Object Name: w_dde_server
public subroutine check_hotlink (checkbox status, string data, string item)
    /* QC_checkbox.SS */

PB Object Name: w_dir_tree
public function integer
   _wfCollapse_rows (datawindow adw_datawindow, long al_startrow)
    /* IC_datawindow.L */
public function long of_recurse_dir_list (string as_path, long ai_parent)
    /* LSL */
public function string of_build_dw_tree (long al_handle)
    /* SL */

This example illustrates the use of a letter code to represent a PowerBuilder system class or a custom class. Consider this function:

    function integer of_get_all_sales_orders (Ref s_sales_order astr_order[], date adt_date, integer ai_direction)

For this function, the pbsig90 tool returns the following string. The first argument is an unbounded array of type s_sales_order and is passed by reference:

    /* IRCs_sales_order.[]YI */

Usage

The pbsig90 tool generates a string that represents the declaration and signature of all the functions and events in the PBL, including argument types, return types, and passing style. Each function and event is followed by a commented string. You pass the commented string, for example QSS in the first comment in the example above, as the last argument to the GetMethodID method.

The following table shows how the output from pbsig90 maps to primitive datatypes.

<table>
<thead>
<tr>
<th>Output</th>
<th>Datatype</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>No type (subroutine)</td>
</tr>
<tr>
<td>I</td>
<td>integer</td>
</tr>
<tr>
<td>L</td>
<td>long</td>
</tr>
<tr>
<td>K</td>
<td>longlong</td>
</tr>
</tbody>
</table>
The passing style is indicated by a prefix on the type.

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Pass by reference</td>
</tr>
<tr>
<td>X</td>
<td>Pass read only</td>
</tr>
<tr>
<td>V</td>
<td>Pass by value</td>
</tr>
</tbody>
</table>

For example, the following output indicates that the function returns an integer and has a single integer argument passed by value:

```
/* IVI */
```

**PowerBuilder arrays**  
PowerBuilder arrays are indicated with a pair of square brackets `[ ]` as a suffix. For bounded arrays, the brackets enclose the bounds.

```
/* IRCdatastore.RS[]SS */
```

**PowerBuilder system or custom class**  
Additional letter codes represent a PowerBuilder system class or a custom class. The letter `C` followed by the name of a PowerBuilder object or enumerated class and a period (`.name`) represents an argument or return value of that type.
### pbx2pbd90

**Description**

The `pbx2pbd90` tool generates a PBD from a PowerBuilder extension DLL. The generated PBD must be added to the library list of any PowerBuilder application target that uses the objects and methods in the PowerBuilder extension.

**Syntax**

```
pbx2pbd90 [+] des.pbd src1.dll [ src2.dll src3.dll ...srcn.dll ]
```

**Examples**

This example generates a new PBD `test.pbd` from `test.dll`. The input and output files are in the current directory:

```
pbx2pbd90 test.pbd test.dll
```

This example appends generated information from `C:\myproject\src.dll` to `C:\mypbds\des.pbd` (if `des.pbd` does not exist, it is created):

```
pbx2pbd90 + C:\mypbds\des.pbd C:\myproject\src.dll
```

This example generates a new PBD `D:\pbds\test.pbd` from all the DLL files in the `C:\myproject` directory:

```
pbx2pbd90 D:\pbds\test.pbd C:\myproject\*.dll
```

This example generates PBD information from all the DLL files in the `C:\temp` and `D:\temp` directories and appends the information to the existing generated PBD file `D:\pbds\test.pbd`:

```
pbx2pbd90 + D:\pbds\test.pbd c:\temp\*.dll d:\temp\*.dll
```

**Usage**

If you specify an absolute path for the DLL file when you generate the PBD, the PowerBuilder application searches for the DLL only in the specified path.

If you do not specify the path for the DLL file, the PowerBuilder application searches the system path for the DLL.