WebSphere® Partner Agreement Manager

Script Developer's Guide

Version 2 Release 2

Note: Before using this information and the product it supports, read the information in *Notices* on page 177.

◢ C O N T E N T S

TABLE OF CONTENTS

WELCOME TO THE SCRIPT DEVELOPER'S GUIDE

This document describes WebSphere® Partner Agreement Manager and explains how to create and use scripts (in VBScript and JavaScript) to automate public and private business processes.

To create your own scripts, follow these general steps:

- *Introducing process scripting* on page 1 will help you gain a basic understanding of Partner Agreement Manager scripts, the Script Editor, Manager, and Tester.

- *Getting started with scripts* on page 11 provides a tutorial to show you how to use the tools to add scripts to a private process that is part of a public process.

- *Creating scripts* on page 25 includes information on how you can create and test scripts. It also includes how to use the Script Editor and Script Manager.

- *Using script methods* on page 39 explains how to use scripts and procedures.

- *Script method reference* on page 73 documents all the procedures available for your scripts.

Who should use this information

This guide is for those who need to automate public and private business processes using script language.

Related information

For additional information see the following:

- The readme.htm file. This file may contain information that became available after this book was published. Before installation, the readme.htm file is located in the root directory of the product CD-ROM. After installation, the readme.htm file is located in the root directory of the Partner Agreement Manager installation.

- The StartHere.htm file. This file contains links to the Partner Agreement Manager readme.htm file and *Partner Agreement Manager Installation Guide*. Before installation, the StartHere.htm file is located in the root directory of the product CD-ROM. After installation, the StartHere.htm file is located in the root directory of the Partner Agreement Manager installation.

- The *Partner Agreement Manager Installation Guide*, form number GC34-5964-02, which describes how to install Partner Agreement Manager.

- The *Partner Agreement Manager Administrator's Guide*, form number BIAAAB02, which describes how to set up, configure, and administer Partner Agreement Manager after you install it.

- The *Partner Agreement Manager User's Guide*, form number BIAAAC02, which describes how to start a Partner Agreement Manager session, design public and private processes, define element definition sets, create business objects, and manage process distribution.

- The *Partner Agreement Manager Adapter Developer's Guide*, form number BIAAAD02, which describes how to develop and administer adapters using the Partner Agreement Manager Adapter Development Environment.

- The *Partner Agreement Manager External API Guide*, form number BIAAAF02, which describes principles behind the Partner Agreement Manager External API. See also, the Javadoc for the External API, which is installed in the Partner Agreement Manager Docs folder.

- The *Partner Agreement Manager Adapters for MQSeries User's Guide*, form number BIAAAG02, which describes how to install, configure, and run the Partner Agreement Manager Adapters for MQSeries.
- The *Partner Agreement View User's Guide*, form number GC34-5965-02, which describes how to install, configure, and use Partner Agreement View.

SUMMARY OF CHANGES

This edition includes these changes since the previous, second, edition:

- *External APIs.* Partner Agreement Manager 2.2 provides added flexibility to external applications through additional APIs. These APIs allow third-party applications to take advantage of the Partner Agreement Manager partner management and process engine through programmatic access. The API is distributed as a set of Java classes that the external application can import. Communication between the API classes and the Process Server is through RMI, but in the future can be swapped out for HTTP or SOAP. Specifically, APIs have been added to the following functional areas:

  - Session Service API
  - Admin Service API
  - Document Service API
  - Partner Service API
  - Adapter Service API
  - Process Service API

- *LDAP Support.* Partner Agreement Manager 2.2 provides centralized user authentication and administration through an LDAP directory. Partner Agreement Manager can retrieve user information—such as name, e-mail address, phone, and fax—stored in an LDAP directory. Updating this information is done in a single place, through the LDAP management tool. Users are authenticated through the same directory, giving them single-sign-on capabilities across enterprise applications.

- *Double-byte character sets (DBCS) and National Language Support (NLS).* Double-byte character sets are now supported in Partner Agreement Manager 2.2. Double-byte and multibyte data can be transferred and operated on in business objects and adapters. NLS lets Partner Agreement Manager display user interface text in other languages.

- *Improved XML Support.* The Partner Agreement Manager 2.2 engine fundamentally changes the way it interacts with business objects by replacing proprietary parsers with a third-party parser. This simplifies support of DTD 1.0 and the support of XML Schemas when the standard is finalized.

  The Business Object and Script API have been extended with new classes and methods. The new classes and methods let you work with business objects as W3C Documents.

- *Adapter Asynchronous Callback.* An additional Adapter API allows adapters to be more efficient with long-running adapter operations. The Asynchronous Callback method tells the Adapter Server that an operation will be long-running, that system resources should be freed while the adapter waits for a response from the end system, and that another method will be called when the response arrives. The Asynchronous Callback method frees the adapter developer from using the request-retry method that makes the Adapter Server responsible for polling the end system for the response.

- *Script API Changes.* The script API now provides access to the PartnerGroupContext and the Public and Private Process Contexts. Through these contexts, you can get information such as partner group binding, a reference to the process, inputs to the process (which contain a reference to the sender, the ID of the sending node, and the variable name), and unique node and loop IDs.

- *Certificate Support.* Partner Agreement Manager 2.2 is able to request and import certificates from certificate authorities like VeriSign. This lets organizations use their existing certificate, or request a new one if their partners do not accept self-signed certificates. Partner Agreement Manager 1.1 supported only self-signed certificates.

- *Outbound Proxy Support.* Partner Agreement Manager 2.2 channels that use HTTP communication can work with outbound proxies that use authentication. Outbound proxy authentication is used within *internal* networks to ensure that only people and applications that are authenticated may communicate with an *external* network. Authentication in the outbound proxy is done with a standard user name and password combination. You can turn on the outbound proxy feature after installation. Thereafter, all outbound HTTP communication will use the same user name and password combination for the proxy.

NOTE: Note that this feature is only used by channels using HTTP communication; it does not apply to channels that use the built-in Partner Agreement Manager proxy.

1

INTRODUCING PROCESS SCRIPTING

Read this chapter as an introduction to creating and using scripts with WebSphere Partner Agreement Manager (PAM) processes.

Sections in this chapter include:

- *How you can use scripts* on page 2.
- *Where you can use scripts* on page 2.
- *The Script Editor, Manager, and Tester* on page 3.
- *Terms used in this guide* on page 5.
- *What you should know before using this guide* on page 10.

How you can use scripts

To better automate business processes with Partner Agreement Manager, you can create scripts in private processes. You can write your script code in Microsoft Visual Basic Scripting Edition (VBScript) or JavaScript. Partner Agreement Manager also provides methods through PAM script extensions.

In private processes, you can use scripts to:

- Create business object instances. (First, from a Process window, you define a context business object variable of a particular business object type; then, from within a script, you instantiate the business object.)
- Populate business object fields with data.
- Perform calculations.
- Manipulate context variables.
- Select a path in a branch or loop of a private process. For example, if a script action is followed by a branch, you could use a script to determine which branch path to take.
- Print debugging messages during testing phases.

Where you can use scripts

To add a script to a private process, you can add a script action, which has the sole purpose of holding a script. You can also attach a script to other types of actions. This is equivalent to adding a script action immediately after the action; the script runs after the action completes.

Tip: A script action is the preferred method for adding scripts to a private process (rather than attaching a script to another type of action). This makes it easier to follow and debug your processes.

In addition to a script action, you can also add a script to these action types:

- *Notification action*. The script runs after the notification is sent. For example, a script can set a field in a business object that indicates that a notification was sent.
- *Approval action*. The script runs after a response is received on the approval request, or, when the action times out. For example, you can use a script to escalate the approval to a manager if the action times out.

- *Mapping action*. The script runs after the map is performed. For example, a script can check the validity of an output business object.

- *Timer action*. The script runs after the time interval elapses. For example, a script can set the system time in a business object field when the timer expires.

- *Extension action*. The script runs after an adapter extension operation completes or times out. For example, a script can check the output result of the extension action and set a path in a branch accordingly.

- *Subprocess action*. The script runs after the public subprocess completes or times out. For example, a script can check the status returned by a subprocess and set the path.

If there is a system error that prevents the associated action from completing, the script does not run. If there is an error in the script, the script terminates at that error, unless your script contains error handling logic. See *Handling run-time errors and exceptions* on page 70 for more information.

You cannot add a script to these action types:

- *Output object action*. Binds a private process business object to a public process path. It does not make sense to add a script after this action, because changes made to the private process business object would not be reflected in the public process.

- *Termination action*. Forces a process to terminate in a controlled manner. Because this action terminates the process, no further scripts or actions can be executed after it.

The Script Editor, Manager, and Tester

Partner Agreement Manager provides easy access to a Script Editor from a private process action. In addition, it lets you store and manage reusable script code in the PAM database through the Script Manager. You can also test run scripts in a Script Tester before you run them within a process.

To access a Script Editor, double-click an action, and create or edit the script in the Script panel of the Properties dialog box. Alternatively, you can right-click an action and choose Script Editor to create or edit a script in a Script Editor window.

You can type the script in VBScript or JavaScript and, while in the Script Editor, have Partner Agreement Manager check the syntax.

NOTE: The syntax of JavaScript on UNIX has some slight differences from the syntax of JavaScript on Windows NT. All of the examples in this book have been tested on Windows NT. See the JavaScript documentation for your platform for more on JavaScript syntax.

In the Script panel, you can access the Script Manager. The Script Manager lets you save scripts in the PAM database—independently of a process—so you can reuse them as needed. You can view, create, edit, and delete scripts in the Script Manager.

Here is an example of a simple script as it appears in the Script panel:

This is same script written in JavaScript.

Choose VBScript or JavaScript.

Type the script here.

Checks VBScript or JavaScript syntax.

Inserts a Script Manager script at the cursor position.

Opens the Script Manager.

Adds the script to the Script Manager.

Script Properties

General Script

Script Type:
JavaScript

Script:
```
main () {
  createBO("MyBOVar");
  MyBOVar.setData ("MyField", "MyData");
  println ("MyScript has run.");
}
```
Insert Script... Save Script... Check Script Scri

OK

Script Properties

General Script

Script Type:
VBScript

Script:
```
sub main
  createBO("MyBOVar")
  MyBOVar.setData "MyField", "MyData"
  println ("MyScript has run.")
end sub
```
Insert Script... Save Script... Check Script Script Manager...

OK Cancel

To test a script, right-click an action containing a script in the Private Process window, then choose Test Script. In the Script Tester window, you can run the script, view existing context variables, add temporary context variables, and add temporary data to context variables. This lets you try different scenarios to test the effectiveness of a script. You can also export and import context variable data for reuse in the Script Tester.

Terms used in this guide

Before learning about the Partner Agreement Manager script extensions you need to understand Partner Agreement Manager terminology. Many of these terms were introduced in the *Partner Agreement Manager User's Guide*, which you should consult for more complete information. Following is a summary of the terms relevant to this guide.

In addition, the next chapter, *Getting started with scripts*, is a tutorial that provides a quick review of the Partner Agreement Manager workflow and how scripts fit into this workflow.

Processes, steps, and actions

Partner Agreement Manager lets you define public and private processes that support business-to-business integration:

- A *public process* defines the flow of information between business partners. It has individual *steps*; each step is owned by one partner.
- A *private process* defines the detailed *actions* of a public process step for one partner.

Both partners approve a public process before using it. However, partners can activate new versions of their own private processes at any time.

Here is the simple public process used in the tutorial. It has two steps.

A public process step that passes a business object to the next step in the public process.
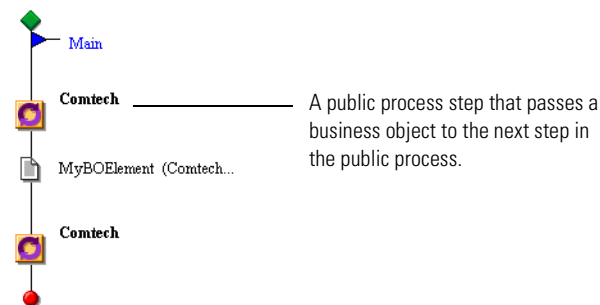
Next is a simple private process with two actions. It is bound to the first step of the public process. In other words, it's the private actions that make up the public step.

Script
Action 1 —————————— A private process action that contains a script.

Output Object
Action 2 —————————— A private process action that binds a business object to the public process.
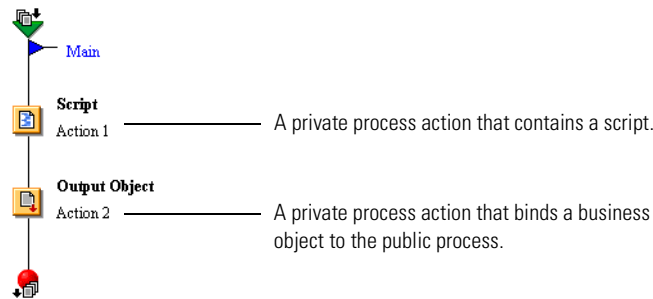
Business objects and elements

A *business object* is a container for data. For example, a business object might contain the data for a purchase order. You can exchange a business object between partners, or use it as temporary storage for one partner, within one or more steps. The tutorial illustrates an example of the second case.

Business objects are made of one or more *elements*. An element can be a field or a group. A *field* holds a single piece of data, such as a status. A *group* is a collection of related groups, fields, or both. A group specifies whether the elements within it are mandatory or optional, repeatable or single. Mandatory fields are required, while optional fields can be left blank; a field can occur multiple times or be allowed to occur a single time only. For example, an address group can contain fields for street, city, state, and country, with street and city being required and state and country being optional.

A *business object type* specifies the elements that make up a business object. A business object is an instance of a particular business object type, and the instance is stored in a context business object variable (more on context variables follows). A *message* is a business object and its public process transmission properties.

To create a business object type, first you create a hierarchical *element definition set*. Then you designate one of the elements as the top-level element of the business object type. The business object type is referred to by the name of this top-level element. You can have one business object type per element set, and the business object type must have at least one field.

For example, the following Element Definition Set window shows an element definition set called MyElementSet. The element MyBOElement could be designated as the top-level element for a MyBOElement business object type. This element contains one field, called MyField.
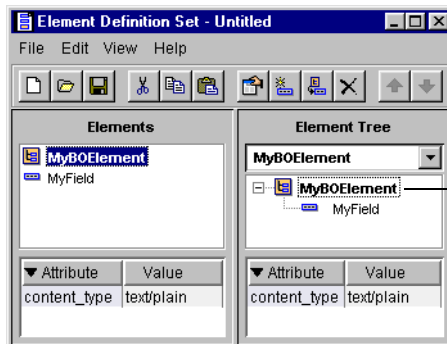
The Element Tree shows the relationship between elements and fields.

A public process step can receive and pass on one or more business objects. The private process corresponding to a step can instantiate and populate business objects through scripts, mapping actions, and adapters. A business object is considered valid when all mandatory fields (except those within an unpopulated optional group) have non-null values.

The private process defines what business objects are expected from the public step; the business objects must be of the type specified in the public process. A business object must be valid before a private process can pass it to a public process.

For example, for the public process shown in *Processes, steps, and actions*, the output object action must specify a business object of the type MyBOElement.

NOTE: An empty string ("") as a data value is equivalent to null. So setting a mandatory field to an empty string does not make it valid. A string containing one or more spaces (" ") is not null, and is considered to be data.

Paths

Branches and loops with separate *paths* can follow a step or action. These paths are identified by a path name. You can set paths in private processes by using scripts. Different paths specify different potential outcomes.

Here is a private process with a branch containing two paths: Yes and No.

Path1 is the Yes result path; Path2 is the No result path.
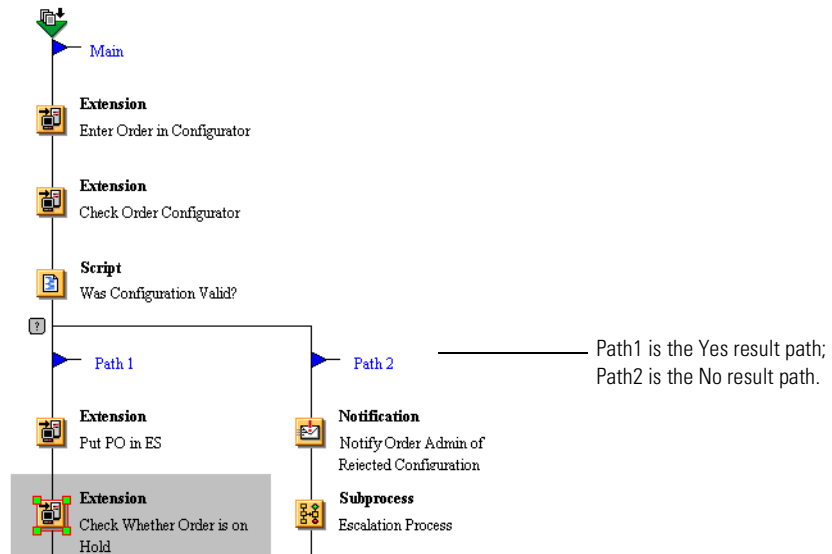
Context variables

Context variables store information shared by multiple steps or actions in a public or private process, including the input and output of a process. The context (the scope) of the variable is the process where it was declared. In addition, a context variable is visible only to the partner that created it. If you define a context variable in a public process, all private processes owned by that partner within that public process can use it; if you define a context variable in a private process, then only that private process has access to it.

There are two types of context variables you can create:

- *Context business object variables* store instances of business objects. The variable is defined to be of a particular business object type. When you refer to a business object instance in a script, you refer to it by its context variable name. Partner Agreement Manager automatically creates context variables for business objects that are input to a private process; it names them I1, I2, and so on. (I is short for "Input.")

- *Context variants* hold data values, such as "approved" or "33.95", as strings. You can use them to store the input for actions, to set flags (such as the time-out flag for an approval action), to move information within scripts, or to store the results of an approval action. Partner Agreement Manager always creates an ActiveInputSet context variant for each private process; this is useful when you can have more than one set of input business objects that activate a private process.

In the Process windows, you can use the Variables menu to add and delete context variables.

The following private process has an ActiveInputSet context variant and a context business object variable called MyBOVar, as shown at the bottom of the window:

The variants and business object variables appear here.
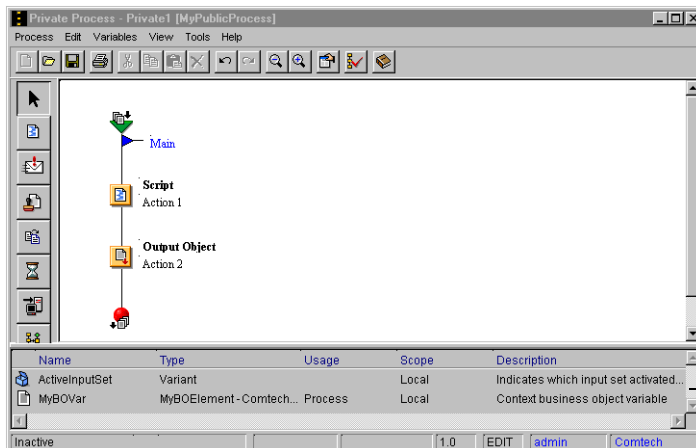
Methods

In this guide, we refer to Partner Agreement Manager Script extension *methods*. A method is a set of statements that performs a task, and may or may not take arguments (constants, variables, or expressions).

In VBScript terms, a *procedure* may be a sub-procedure that does not return a value, or a function procedure that does return a value. In JavaScript terms, a method is assigned to an API object.

For simplicity, VBScript procedures and functions, as well as JavaScript methods, are all called methods in this guide.

What you should know before using this guide

This guide assumes that you understand the basic concepts of scripting and know at least one scripting language, either VBScript or JavaScript. It does not teach you how to create scripts in VBScript or JavaScript; you need to learn these languages on your own.

You should also be familiar with creating processes with Partner Agreement Manager, as described in the *Partner Agreement Manager User's Guide*. You should understand the material in the user's guide before attempting to create scripts as described in this guide.

2

Getting started with scripts

This chapter provides a tutorial that illustrates how to use scripts in a private process, which is associated with a public process. It will also help you understand scripts in relation to public process steps, private process actions, and variables within processes.

In this tutorial, you perform these general steps:

This tutorial provides a general overview of how scripts fit into the overall Partner Agreement Manager workflow. In addition, when you are ready to test a script, you may want to create simple processes like the processes you will create here. This way you can test your script, including any PAM methods, before activating it in a real-life situation.

In this chapter, you create a public process that exchanges a business object between two steps. In the first step, you create a private process that uses a script to instantiate a business object and add data to it. You then bind the business object to the public process. In the second step, you create a private process, then use the input business object within a script. For simplicity, your organization will own both public process steps.

This tutorial should take less than an hour to complete. For best results, work through the entire tutorial in one session. Before you begin, make sure you have installed Partner Agreement Manager on the computer you will be using.

For more complete information on using Partner Agreement Manager to create and run processes, see the *Partner Agreement Manager User's Guide.*

The rest of the sections in this chapter correspond to tutorial steps. Now you can start the tutorial.

Run Partner Agreement Manager

Start Partner Agreement Manager in the normal manner, so the window appears:
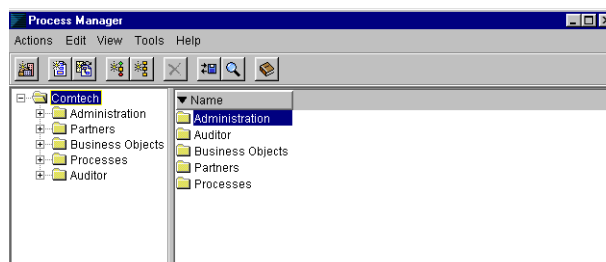
NOTE: The windows shown in this chapter have Comtech as the partner name. The partner name for your organization will be different. In the steps that follow, replace *MyOrg* with your partner name.

Define a business object type

Business objects contain information used by processes. In this section, you define an element definition set for a business object type that you want to use in a simple public process. When you are finished defining them, the Element Definition Set window will look like this:
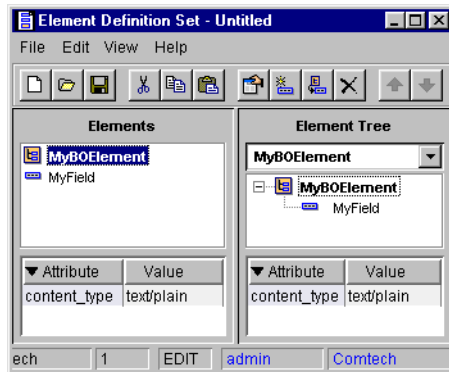
The element definition set is called MyElementSet. After creating the set, you will define MyBOElement to be the top-level group element of a simple business object type (which is also called MyBOElement). MyBOElement has one subordinate field element, called MyField. You will reference the names of these elements in the scripts you create.

To create the element definition set and business object type:

1 In the Process Manager window, click the New Element Definition Set button in the toolbar.

2 In the Element Definition Set window, click the New Element button.

3 In the New Element dialog box, define the business object elements:

   A. Type MyBOElement in the Element Name field.

   B. In the Content panel, select Group.

   C. Click Add.

   D. In the Type field, type MyField.

   E. Click OK.

4 In the Element Definition Set window, select MyBOElement in the Elements list, then choose Create Business Object Type from the File menu.

5 In the Save Element Definition Set dialog box, type MyElementSet, then click OK.

6 In the Create Business Object Type dialog box, select MyBOElement and click OK.

7 In the Business Object Type dialog box, click Preview to view the business object type. Click Close, then OK.

8 Choose Freeze Definition from the File menu.

9 Choose Exit from the File menu.

DEFINE A PUBLIC PROCESS WITH TWO STEPS

Now you will define a public process with two steps, which pass a business object (of the type MyBOElement) from the first step to the second, so when you are finished it looks like this:
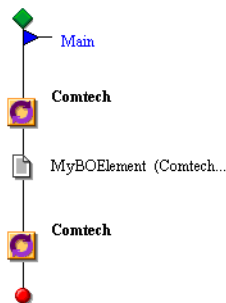
To define the public process:

1 In the Process Manager window, click the New Public Process button in the toolbar.

2 In the Public Process window, to place a simple step, click the Simple Step tool, then click the turquoise circle.

3 Right-click the step, then choose Properties.

4 In the Step Properties dialog box, click the Partner tab, select Partner, and choose your organization name. Click OK.

5 In the Public Process window, right-click the message and choose Properties.

   A message is a business object and its public process properties.

6 In the Message Properties dialog box, click the Contents tab, then select Business Object Type and choose MyBOElement. Click OK.

7 In the Public Process window, use the Simple Step tool to place another step after the first.

8 Right-click the new step, then choose Properties.

9 In the Step Properties dialog box, click the Partner tab, select Partner, and choose your organization name. Click OK.

10 Right-click the message and choose Properties.

11 In the Message Properties dialog box, click the Contents tab, then select None. Click OK.

   The public process is terminated.

12 To make sure the process meets Partner Agreement Manager requirements, choose Tools > Verify, then click Close.

   If you get errors or warnings, correct your mistake.

13 Choose Save As from the Process menu, name the process MyPublicProcess, then click OK.

Define the private process for the first step

Next you will define a private process, so that when you are finished it looks like this:
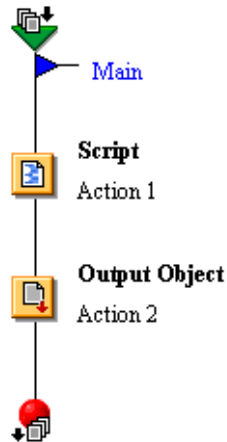
In this private process, the script action creates the business object instance. The output object action outputs the business object to the public process, so the next step can use it. The private process must output a business object of the same type as you specified in the public process.

To create the private process:

1 In the Public Process window, right-click the first step and choose Private Process.

2 In the Private Process window, to place a script action, click the Script tool, then click the turquoise circle that appears.

3 To place another action, click the Output Object tool, then click the turquoise circle that appears after the first action.

The private process now contains a script action and an output object action.

CREATE AND VIEW CONTEXT VARIABLES, INPUT, AND OUTPUT

You are now ready to create the context business object variable for the business object. In addition, in this section you will view the business object type in the catalog, and view the input and output of the private process and the output object action. After you define the variable, called MyBOVar, the variable list at the bottom of the window will look like this:
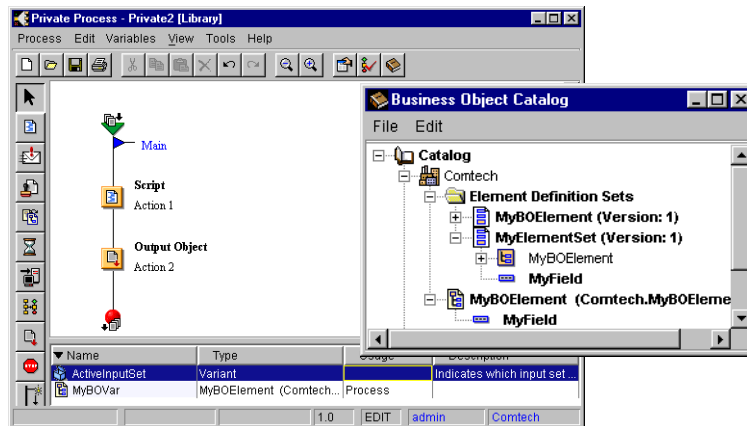
To create and view context variables:

1 To see the context variables in this process, choose Variables from the View menu.

2 To add a context business object variable, choose New Business Object from the Variables menu.

3 In the New Business Object Variables dialog box, type MyBOVar in the Name field. In the Type field, choose MyBOElement. In the Description field, type Business object variable. Then click OK.

4 To view the structure of your business object, choose Business Object Catalog from the View menu.

5 In the Business Object Catalog, expand the tree (click the +) for your organization until you see the MyBOElement business object type and MyField, which is contained by it. You can also view the element definition set you created.

6 Double-click the first item in the private process to view the input to this process, as defined in the public process. Then click OK.

7 Double-click the termination of the private process to view the expected output of this process, as defined in the public process. Then click OK.

8 To set the output of this process, right-click the output object action, choose Properties, click the Action tab, and choose MyBOVar in the Variable field. Click OK.

IMPORTANT: You must not skip this step. If you do, you will get output object errors when you verify your process.

CREATE A SCRIPT FOR THE FIRST ACTION

Now you add a script to the script action. Here is the script you will create:

This is a script written in JavaScript.

This is same script written in VBScript.
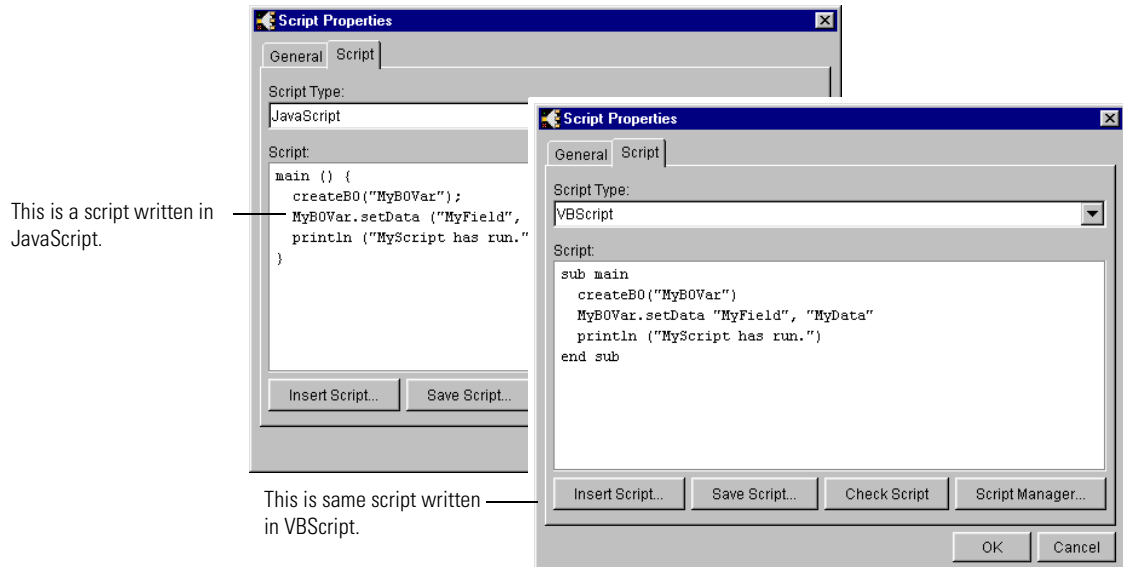
All scripts must have a **main** method, because this is the Partner Agreement Manager entry point. In this script, the **createBO** method creates a business object instance, and the **setData** method sets the value of MyField to the string MyData. Then the **println** method prints a message. If you ran Partner Agreement Manager as a service, **println** output goes into a log file. The log file is called **PAM.log** and it is created in the **Partners\Partner***nnn* directory. If you ran Partner Agreement Manager from a shortcut, **println** output is displayed in the server's console and also goes to the log file.

To create the VBScript in the script action:

1 Right-click the script action and choose Properties.

> **TIP:** Alternatively, you could choose Script Editor and enter the script from there.

2 In the Script Properties dialog box, click the Script tab, and choose a Script Type of VBScript.

3 Type this script in the Script field:

```
sub main
createBO("MyBOVar")
MyBOVar.setData "MyField", "MyData"
println("MyScript1 has run.")
end sub
```

4 Click Check Script to check VBScript syntax. Then click OK.

5 In the Script Properties dialog box, click OK.

6 Choose Verify from the Tools menu, then click Close.

If you get errors or warnings, correct your mistake. If you get output object errors, you probably did not click OK after setting the properties of the output object action.

7 Choose Save from the Process menu.

8 Choose Exit from the Process menu.

Define the private process for the second step

When you are finished with this section, you will have created a private process, including a script, as shown here:
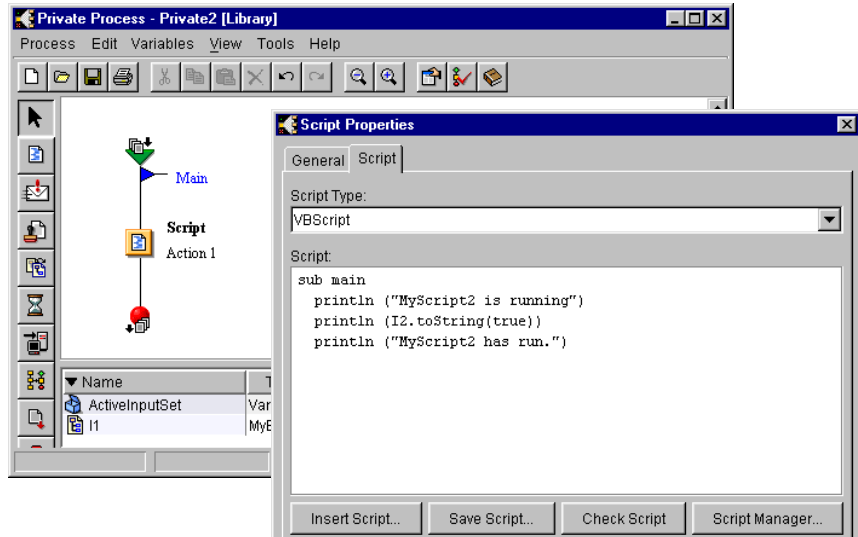
The business object you created in the first step is input to this private process, and is called I1 ("I" for "Input" followed by the number one). The script prints information about the business object. Although this script is not very useful in itself, it helps to illustrate how you can pass a business object between steps and use it in scripts.

To create the private process for the second step:

1 In the Public Process window, right-click the second step and choose Private Process.

2 In the Private Process window, to place a script action, click the Script tool, then click the turquoise circle that appears.

3 Double-click the first item in the private process to view the input to this process.

InputSet1 is the value of the ActiveInputSet variable. Your business object, which was created in the first private process, is input to this private process.

4 To see the context variables in this process, choose Variables from the View menu.

I1 is a reference to the input business object you created in the first step.

5 Right-click the script action, choose Properties, and add the following VBScript script.

```
sub main
println("MyScript2 is running.")
println(I1.toString(True))
println("MyScript2 has run.")
end sub
```

The **toString** method provides descriptive information about the I1 business object, and the **println** method prints the information to the server log file and perhaps the console window.

6 Click Check Script. Then click OK.

7 In the Script Properties dialog box, click OK.

8 Choose Verify from the Tools menu, then click Close.

   If you get errors or warnings, correct your mistake.

9 Double-click the termination of the private process to view the expected output of this process. Then click Cancel.

   No business object output is defined in the public process.

10 Choose Save from the Process menu.

11 Choose Exit from the Process menu.

12 In the Public Process window, choose Exit from the Process menu.

DISTRIBUTE AND TEST THE PUBLIC PROCESS

Next, run the public process:
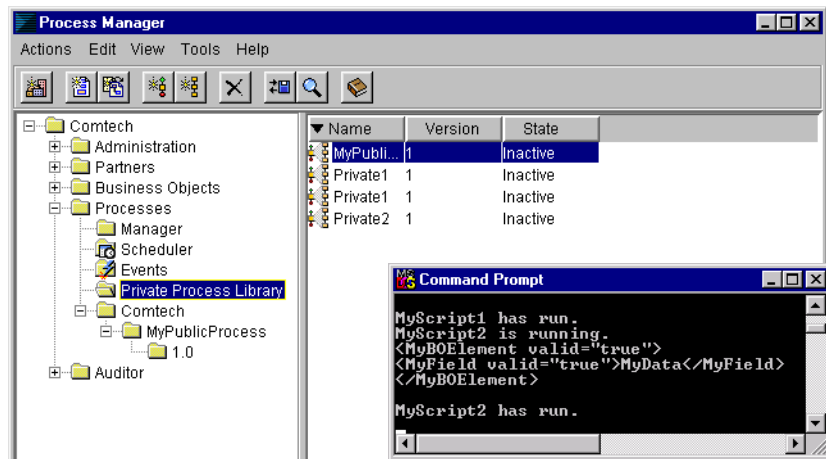
To distribute and test the public process:

1 In the left panel of the Process Manager window, navigate to your new public process in the following folder hierarchy: *MyOrg* > Processes > *MyOrg* > MyPublicProcess > 1.0.

2 While the 1.0 folder is selected in the left pane, right-click MyPublicProcess in the right pane and choose Process Distribution Manager.

3 In the Process Distribution Manager dialog box, select Distribution for the Process State, and click Apply.

4 Select Test Installation and click Apply.

5 Click Close.

6 In the right pane of the Process Manager window, right-click MyPublicProcess and choose Start.

7 In the Start Process dialog box, click OK.

8 Look at the messages in the server log file and perhaps the console window.

Modify a script, activate it, and run it

In this section, you make a slight modification to the first script, activate it in the public process, then rerun it:
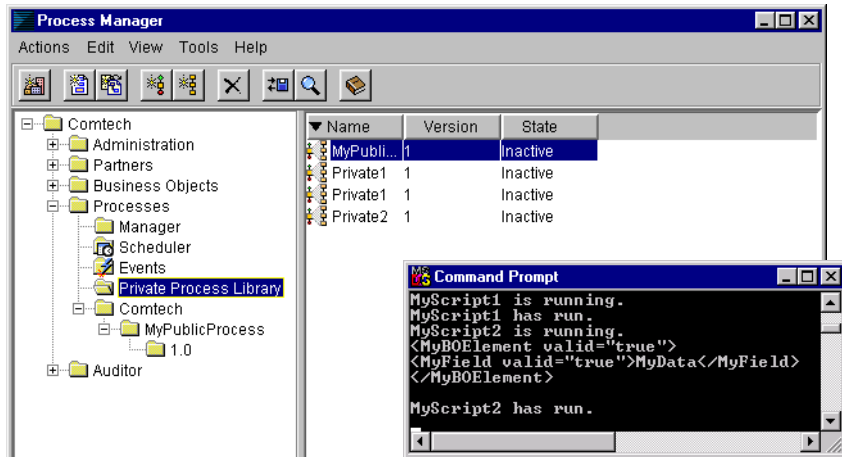
To modify the script and run it:

1 In the right pane of the Process Manager window, right-click Private1 and choose Open.

2 Choose Process > Save As New Version so you can edit it.

3 Right-click the first action and choose Properties.

4 Under main, add this line:

```
println("MyScript1 is running.")
```

5 Check VBScript syntax, then close the Script Properties dialog box.

6 Choose Tools > Verify, then click Close.

If you get errors or warnings, correct your mistake.

7 Choose Exit from the Process menu and save changes.

8 In the Public Process window, right-click the first step, then choose Activate Private Process.

9 In the Activate Private Process dialog box, select the most recent version (version 2), then click OK.

10 Choose Exit from the Process menu and save changes.

11 In the right pane of the Process Manager window, right-click MyPublicProcess and choose Start.

12 In the Start Process dialog box, click OK.

13 Look at the messages in the server log files and perhaps the console window.

EXIT PARTNER AGREEMENT MANAGER

Optionally, you can delete the items you created for this tutorial.

To delete the processes:

1 While the 1.0 folder is selected, right-click MyPublicProcess and choose Process Distribution Manager.

2 In the Process Distribution Manager dialog box, select Deactivation, select Deactivate Immediately, click Apply, and click Close.

3 Right-click MyPublicProcess and choose Delete. Also delete all private processes.

To exit Partner Agreement Manager:

▶ If you are finished using Partner Agreement Manager, choose Exit from the Actions menu.

CREATING SCRIPTS

This chapter gives you the information you need to create and test your own scripts. It explains how to use the Script Editor, Script Manager, and Script Tester.

Sections in this chapter include:

- *Before you create a script* on page 26.
- *Using the Script Editor and Script Manager* on page 28.
- *Testing scripts* on page 34.
- *Using the Script Tester* on page 34.

Before you create a script

You create scripts from the Partner Agreement Manager Private Process window. You can add a script to a private process that is bound to a public process, or add a script to a private process stored in the Private Process Library. To run a script, you can run a public process or test run the script in the Script Tester, as described at the end of this chapter.

Before you create a script, you first must have a private process you can add it to. In addition, you need to define or determine the names of business object types and element names, context business object variables, context variants, and private process paths you want to work with in scripts. The following sections describe the general steps Partner Agreement Manager users might follow before creating scripts.

Note that *MyOrg* refers to your organization name, while *Partner* refers to a business partner.

Adding a script to a private process bound to a public process

In general, Partner Agreement Manager users follow these steps before adding scripts to a process.

1 Create the business object types that you want to use to exchange information in the public and private processes.

You can create a new type, use an existing type, or receive a read-only type from a partner. You need to freeze the element definition set where the type is defined before you can run a public process that uses it.

To create a type, first you create an element definition set, then specify one element to be the top-level element of a business object type. For example, in the Process Manager window, click the New Element Definition Set button in the toolbar, define the element definition set in the Element Definition Set window, save the set, then choose Create Business Object Type from the File menu.

To view a type, in the left pane of the Process Manager window, select the folder in this hierarchy: *MyOrg* > Business Objects > *MyOrg* or *Partner*; in the right pane, double-click the type to view information about it. Or, to view a type from a Process window, choose Business Object Catalog from the View menu, and expand the tree until you reach the type you want to look at.

If the element definition set has been frozen, you need to first create a new copy or version of it before you can edit it, and define a new business object type (remember to update references to the type in your process). You can only edit types owned by your organization.

2 Create a public process, including steps, business objects, and public context variables.

You can create a new public process, open an existing public process, or receive a read-only public process from a partner. Before you can run a public process, it must be distributed and installed.

To create a public process, in the Process Manager window, click the New Public Process toolbar button. The Public Process window appears.

To view a public process, in the left pane of the Process Manager window, select the folder in the following folder hierarchy: *MyOrg* > Processes > *MyOrg* or *Partner* > *Public-Process* > *Version*; right-click the process in the right pane and choose Open.

If the public process has been distributed and installed, you need to create a new version of it before you can edit it, and define a new business object type, then go through the distribution and installation process again before you can run it.

3 Create private processes, including actions and private context variables, for the steps owned by your organization.

You can create a new private process or open an existing private process that is bound to a step owned by your organization. You cannot view or edit the private processes owned by another organization.

To create or view a private process bound to a particular step in a public process, in the Public Process window, right-click the step (owned by your organization) and choose Private Process. The Private Process window appears.

Or, in the left pane of the Process Manager window, select the folder in the following folder hierarchy: *MyOrg* > Processes > *MyOrg* > *Public-Process* > *Version*; right-click the process in the right pane and choose Open.

If the private process has been activated, you need to create a new version of it before you can edit it, then activate the new version to run it.

4 Create scripts. See *Adding a script to a private process in the Private Process Library*, next.

Adding a script to a private process in the Private Process Library

You can add scripts to private processes that are outside of a public process. These private processes are stored in the Private Process Library. To run the private process, you can insert it into a step in a public process or test run the script in the Script Tester, as described at the end of this chapter.

In general, Process Manager users follow these steps to create a process for the Private Process Library:

1 Create the business object types you want to use to exchange information in the private process.

You can create a new type, use an existing type, or receive a read-only type from a partner. See step 1 in the previous section for more information.

2 Create a private process, including actions and private context variables.

To create a process, in the Process Manager window, click the New Private Process button.

To view an existing private process that is not bound to a public process, in the left pane of the Process Manager window, navigate to the public process in the following folder hierarchy: *MyOrg* > Processes > Private Process Library > *Private-Process* > *Version*; right-click the process in the right pane and choose Open.

The Private Process window appears.

3 Create scripts, as described in the following section.

Using the Script Editor and Script Manager

You can access a Script Editor and the Script Manager from a private process action. Scripts in JavaScript and VBScript are treated the same way.

You can create, view, and edit scripts from a Script Editor in a Properties dialog box:

The same script in JavaScript.

Choose VBScript or JavaScript.

Type the script here.

Checks VBScript or JavaScript syntax.

Adds the script to the Script Manager.

Inserts a Script Manager script at the cursor position.

Opens the Script Manager.

Or from a Script Editor window:

Undoes or redoes the last operation.

The same script in JavaScript.

Choose VBScript or JavaScript.

Type the script here.
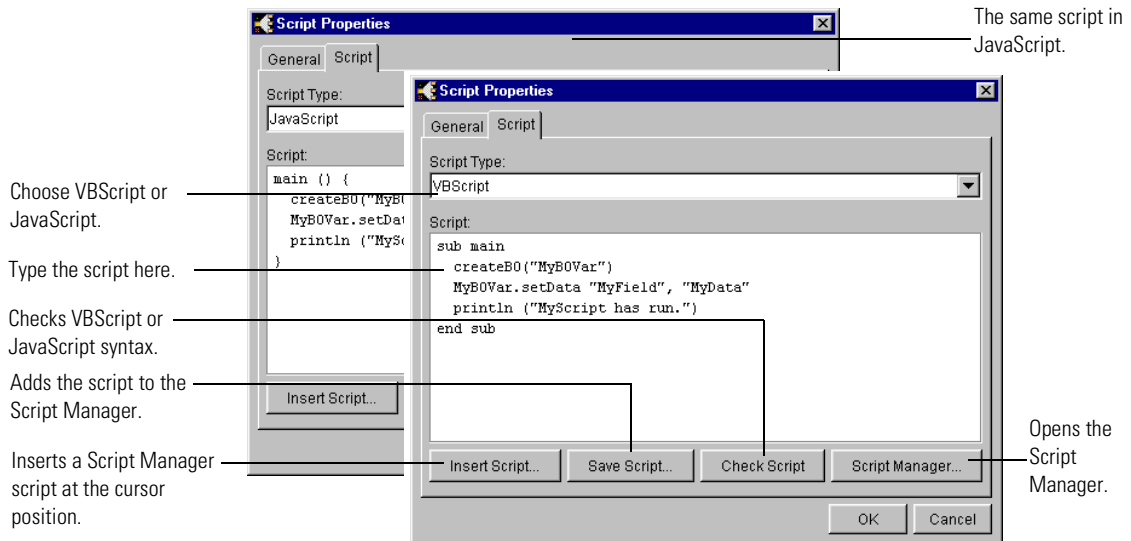
In this guide, both are referred to as the "Script Editor." The Script panel lets you access the Script Manager. The Script Editor window lets you go to a particular line number, update the script in the private process while keeping the Script Editor open, undo and redo operations, select all, and disable scripts so they don't run.

When you write a script in a Script Editor and save it with a process, that script becomes part of the process. When you write a script in a Script Editor and save it in the Script Manager (by clicking the Save Script button), that script is stored in the PAM database, independently of whether you save it in a process or not. You can then insert the script into a Script Editor whenever you need it.

You can add, edit, and view scripts in the Script Manager by clicking Script Manager in the Script panel:

JavaScript scripts.

Choose VBScript or JavaScript.

Select a script from the list of scripts in the Script Manager.

Lets you create a new script to add to the Script Manager.

Removes the selected script from the Script Manager.

Lets you edit the selected script in the Script Manager.
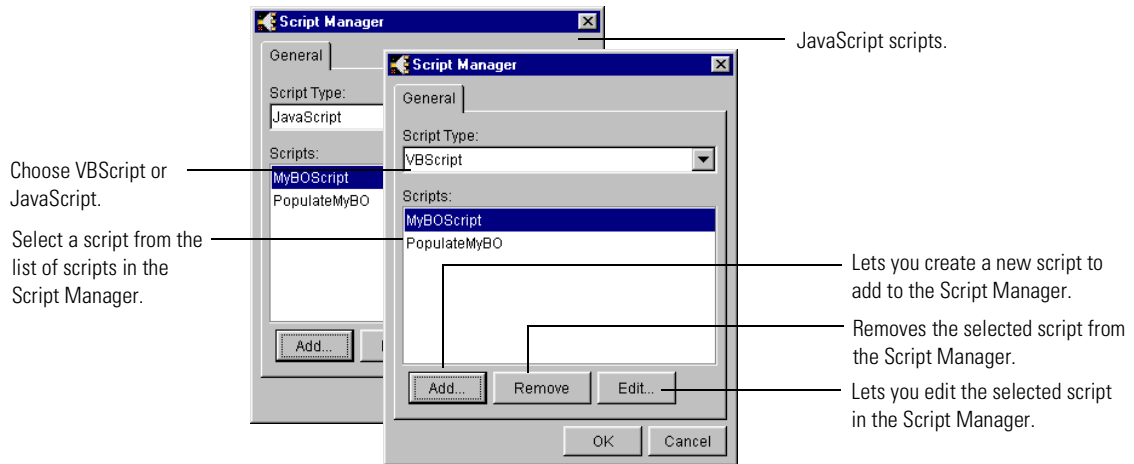
Opening a Script Editor

You can create, view, and edit scripts from a Properties dialog box or from a Script Editor window, as described here.

To open the Script panel in a Properties dialog box:

1 In the Private Process window, double-click the action you want to add a script to.

Alternatively, you can select the action then:

> **A.** Right-click and choose Properties.
>
> **B.** Click the Properties button on the palette.
>
> **C.** Choose Item Properties from the Edit menu.

Remember that you can add a script to a script action, or to another type of action that lets you add a script that runs after the action completes. For example, you can add a script to a notification action, which runs the script after the notification action completes.

2 In the dialog box, click the Script tab.

The Script panel appears.

3 In the Script Type field, choose VBScript or JavaScript.

> **WARNING:** If you switch from VBScript or JavaScript to None, your script code is removed from the panel. To retrieve your code, you would need to click Cancel or not save the private process. You can switch between VBScript and JavaScript, however, and your script remains in the panel.

You are now ready to type your script. When you are finished, click OK to add the script to the action, then save the process. You can click Save Script to save the script outside of a process, in the Script Manager.

To open the Script Editor window:

1 In the Private Process window, select the action, then right-click and choose Script Editor, or choose Script Editor from the Edit menu.

2 In the Script Type field, choose VBScript or JavaScript.

Choose None to disable the script (it will not run).

> **NOTE:** Unlike the Script panel, you can switch from VBScript or JavaScript to None without losing your script code. Remember, if you open the script in the Script panel while the type is None, the script code disappears.

You are now ready to type your script. When you are finished, choose Update Process from the Script menu to add the script to the action, then save the process.

Copying and pasting script code

The Script Editor and Manager support the standard copy and paste operations, including Control-X, Control-C, and Control-V, Cut, Copy, and Paste menu items in the Edit menu, and Cut, Copy, and Paste buttons in the toolbar. Remember that you can only have one Script Editor open at a time, so you may have to open a script, copy it, then open another script to paste into.

Editing scripts in the Script Editor window

The Script Editor window lets you perform some edit operations not available from the Script panel.

To	Do this
Undo an operation.	Click the Undo toolbar button or choose Undo from the Edit menu.
Redo an operation.	Click the Redo toolbar button or choose Redo from the Edit menu.
Select the entire script.	Choose Select All from the Edit menu.
Go to a particular line.	Choose Goto Line from the Edit menu, type the line number, and click OK.
View a line or column number.	Click a line and look at the status bar at the bottom of the window.

Inserting a script from the Script Manager

Follow these steps:

1 In the Script panel, place your cursor at the location where you want to insert the script, then click Insert Script.

2 In the Insert Script dialog box, select a script and click OK.

The script appears at the insertion point.

Viewing, adding, editing, and deleting scripts in the Script Manager

Follow these steps:

1 In the Script panel, click Script Manager.

2 In the Script Manager dialog box, choose a Script Type.

3 Perform the operation you want:

- To add a script, click Add.
  In the dialog box, type the script, then click OK.

- To view or edit a script, select a script and click Edit.
  In the dialog box, edit the script, then click OK.

- To delete a script, select a script and click Remove.

4 In the Script Manager dialog box, click OK to save the operations you performed, or click Cancel to cancel them.

The operation is performed immediately, independent of whether you click Cancel in the Script panel.

Checking VBScript and JavaScript syntax

Partner Agreement Manager can check that the syntax in your VBScript and JavaScript code is correct. Note that this utility does not check that the PAM method syntax is correct.

To check VBScript or JavaScript syntax:

- While the script is displayed in the Script panel, click Check Script.

- While the script is displayed in a Script Editor window, choose Verify from the Tools menu or click the Verify Script toolbar button.

Saving a script

You can save a script in a process, in the Script Manager, or both.

To save a script in a private process:

1 In the Script panel, click OK. Or, in the Script Editor window, choose Update Process from the Script menu, or click the Close box and click Yes to update.

2 Save the private process.

To save a script in the Script Manager, which stores it in the PAM database:

1 In the Script panel, click Save Script.

2 Type the script name, then click OK.

You can now view, edit, or delete the script in the Script Manager by clicking Script Manager in the Script panel.

Testing scripts

You can test your scripts in several ways to make sure they are correct before deploying them.

During design time, you can check VBScript and JavaScript syntax. While the script is displayed in a Script Editor, click Check Script or choose Verify from the Tools menu.

Next, you can run the scripts to make sure they work as planned. To test run scripts, use the Script Tester, described in the next section. To run scripts in a process, you must run a public process that contains the private process with the script.

Before testing a script in a public process used by other partners, you may want to test the scripts in your own test public and private processes, similar to the tutorial in *Getting started with scripts* on page 11.

A fast way to create populated business objects for your test processes is to use a mapping action. The mapping action lets you quickly instantiate and populate a business object with default values that can be handled by subsequent steps and actions.

When you are convinced that a private process works as planned, you can activate the private process in an approved public process.

To create a new version of an activated process so you can edit it, display the process in the Private Process window, then choose Save As New Version from the Process menu. To activate a new version, right-click the public process step, choose Activate Private Process, select the new version, and click OK.

Using the Script Tester

The Script Tester lets you test run scripts associated with a private process. You can edit and run a script without saving changes to it, you can try different data sets to make sure your script handles data as planned, and you can export and import test data to XML files so you don't have to reenter it.

The Script Tester looks similar to the Script Editor window. In addition, at the bottom of the window, it displays all public and private context variables the script can use, and the path if set by the setPath method:
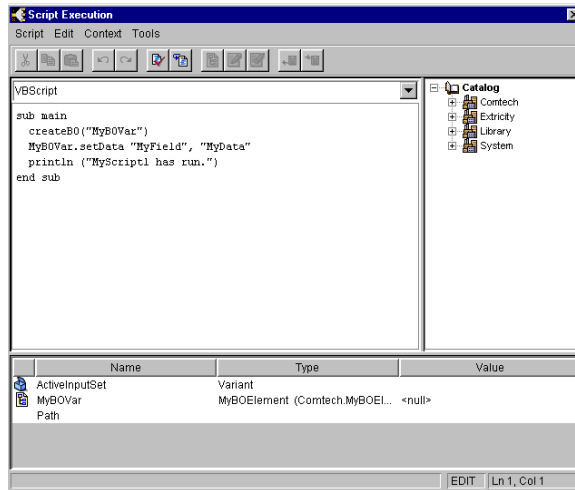
Opening the Script Tester

To display a script in the Script Tester, follow these steps:

1 In the Private Process window, display the private process containing the script.

2 Right-click the action containing the script, then choose Test Script. Or select the action and choose Test Script from the Tools menu.

The script appears in the Script Execution window.

Running a script

To run a script from the Script Execution window:

1 Add data that your script needs to execute and, if needed, edit the script.

2 Choose Execute from the Tools menu. Or click the Execute toolbar button.

If the script executes successfully, you receive an Execution Complete dialog box.

If there is an error, an error dialog box appears. For some errors, after clicking OK the line containing the error is highlighted.

Editing and updating a script

You can edit a script as you would in the Script Editor window, and test run it as many times as needed without saving the changes.

If you want to save the changes you made to a script while you were in the Script Tester, do this:

1 In the Script Tester, choose Update Script from the Script menu.

   The changes will now appear when you open a Script Editor.

2 Save the script in the private process, the Script Manager, or both, as described in *Saving a script* on page 33.

Checking VBScript and JavaScript syntax

The Script Tester can check that the syntax in your VBScript and JavaScript code is correct. Note that this utility does not check that the PAM method syntax is correct.

To check VBScript or JavaScript syntax:

▶ While the script is displayed in a Script Execution window, choose Verify from the Tools menu or click the Verify toolbar button.

Adding data to an existing context variable

You can add the data your script needs to run, so it mimics the activity of the public and private processes.

Warning: When you close the Script Tester, you lose the values of all context variables. You can export business object data to an XML file to preserve the data, as described in a following section.

To add data to a context business object variable:

1 At the bottom of the Script Execution window, double-click the context business object variable. Or select it and click the Edit toolbar button or choose Edit from the Context menu.

2 In the BO Instance Editor, enter and edit data.

   The display is similar to the Element Tree view in the Element Definition Set Editor. You can add data as you would for a map; see the *Partner Agreement Manager User's Guide* for more information.

To add a value to a context variant:

1 At the bottom of the Script Execution window, double-click the context variant. Or select it and click the Edit toolbar button or choose Edit from the Context menu.

2 In the Edit Variant dialog box, type the value, then click OK.

Creating an empty business object instance

To create a business object instance, as if you called createBO in a previous script (the data is null):

▶ Select the context business object variable, then choose Create from the Context menu.

The menu item is only available if the Value field contains <null>.

Clearing test data

To remove all data in context variables:

▶ In the Script Execution window, choose Clear from the Context menu to clear the selected item or Clear All from the Context menu to clear all test data.

WARNING: If you want to preserve business object data, you need to export the data to an XML file before clearing it.

Resetting test data

You can reset test data back to the state it was before the script *last ran*. For example, if you find an error in your script, you can fix the error then run your script with the original test data (as it was before you ran the script with the error).

To reset test data:

▶ In the Script Execution window, choose Reset from the Context menu to reset the selected item or Reset All from the Context menu to reset all data.

WARNING: If you want to preserve business object data, you need to export the data to an XML file before clearing it.

Exporting test data

You might want to export business object data after you entered it, so you can reuse it after you close the Script Tester. In addition, if a script populated a business object and you want to use this data with a following script, you can export the data and then import it for the other script, or examine the test data later.

To export business object data to an XML file:

1 In the Script Execution window, select the context business object variable, then choose Export from the Context menu. Or click the Export toolbar button.

2 In the Business Object Instance Editor, choose Export from the Instance menu. Or click the Export toolbar button.

Importing test data

You can import business object data from an XML file created with the Export command, as described in the previous section. You can only import data of the same business object type.

To import business object data from an XML file:

1 In the Script Execution window, select the context business object variable, then choose Import from the Context menu. Or click the Import toolbar button. Or, in the Business Object Instance Editor, Import from the Instance menu. Or click the Import toolbar button.

2 In the Load Business Object dialog box, select the XML file and click Open.

The XML file must be the same type as the context business object variable you selected.

Closing the Script Tester

To close the Script Tester:

1 In the Script Execution window, choose Exit from the Script menu.

2 Click the Close button.

4

USING SCRIPT METHODS

Read this chapter for information on how to use Partner Agreement Manager methods in your scripts.

Sections in this chapter include:

- *Creating the script entry point* on page 40.
- *Working with context variants* on page 41.
- *Instantiating and accessing a business object in a script* on page 46.
- *Accessing elements and element sequences in a business object* on page 49.
- *Working with group and field elements* on page 53.
- *Working with element sequences* on page 63.
- *Setting the path in a private process* on page 67.
- *Printing a message to the console and log file* on page 69.
- *Handling run-time errors and exceptions* on page 70.

For a reference that describes each method in alphabetical order, see *Script method reference* on page 73. The reference chapter also describes the sample business object that is used in many of the examples in this book.

While this chapter does briefly describe how to use some features of PAM windows, you should consult the *Partner Agreement Manager User's Guide* for the most complete information.

NOTE: The syntax of JavaScript on UNIX has some slight differences from the syntax of JavaScript on Windows NT. All of the examples here have been tested on Windows NT. See the JavaScript documentation for your platform for more on JavaScript syntax.

CREATING THE SCRIPT ENTRY POINT

The script entry point is, as the name suggests, the point at which the script is entered and execution begins. The entry point for Partner Agreement Manager in your script code is a method called main.

For example, in VBScript:

```
sub main
  setVar "foo", "3"
end sub
```

In JavaScript:

```
function main () {
  setVar ("foo", "3");
}
```

In a script, you can define methods outside of the main method that are called in main. For example, myFunction is such a method.

In VBScript:

```
sub main
dim x, y
  y = -1
  x = myFunction(y)
end sub

function myFunction(number)
  myFunction = abs(number)
end function
```

In JavaScript:

```
function main () {
```

```
var x, y;
  y = -1;
  x = myFunction(y);
}

function myFunction(number) {
  myFunction = abs(number);
}
```

See *Handling run-time errors and exceptions* on page 70 for another example.

Working with context variants

A context variant is a type of context variable that holds a data value, such as "approved" or "33.95". You can define it at the public process level or private process level, which determines its scope. In scripts, you can get and set the values of context variants.

Partner Agreement Manager automatically creates an ActiveInputSet context variant for each private process. The ActiveInputSet is useful if there could be more than one set of business objects received by the private process. When you have very complex branch logic, you could have a variety of input sets.

For example, if a private process has two input context business object variables called I1 and I2, and the public process input branch logic is XOR, then either I1 or I2 will not be null. If I1 contains data, the ActiveInputSet variant will have the value InputSet1; if I2 contains data, the value is InputSet2. You can use the ActiveInputSet context variant to determine which path provided the private process with a business object. (Alternatively, instead of checking this variant, you could use the isBONull method to determine whether I1 or I2 was null. See *Determining if a business object has been instantiated* on page 48 for more information.)

Viewing existing context variants

Partner Agreement Manager lets you easily determine the names of context variants you want to use in scripts.

To view the context variables defined in the private process:

▶ In the Private Process window, choose Variables from the View menu.

The context variables defined in that private process appear at the bottom of the window.

To view the context variables defined in a public process (and owned by your organization):

▶ In the Public Process window, choose Variables from the View menu.

The context variables appear at the bottom of the window.

To view the context variables input to the private process, including a description of the input sets identified by the ActiveInputSet variable:

▶ In the Private Process window, double-click the input icon (green triangle) at the beginning of the process.

The context variables input to that private process appear in a dialog box.

CREATING A CONTEXT VARIANT

You can create a context variant in the Public or Private Process window, which determines its scope. If the context of a variant is public, all public process steps *owned by your organization* in that public process can use it; if private, just actions in that private process have access to it.

To create a context variant:

▶ From within the Public or Private Process window, choose New Variant from the Variables menu.

GETTING A VALUE FROM A CONTEXT VARIANT

Use the **getVar** method to get the value of a context variant. It returns a string containing the value.

In VBScript and JavaScript:

getVar(*context-variant*)

In VBScript:

```
sub main
  dim varX
  varX = getVar("approval_response")
end sub
```

In JavaScript:

```
main () {
  var varX;
  varX = getVar("approval_response");
}
```

VBScript and JavaScript can implicitly convert strings to other data types. However, sometimes you need to explicitly convert the getVar return value from a string to another data type. For example, if you try to compare two strings that contain numbers, a comparison of the string values is unreliable.

This VBScript example implicitly converts varX to a number because it's being compared to a number:

```
sub main
  dim varX
  varX = getVar("counter")
  if varX > 5 then
  ...
end sub
```

If you want to compare numeric values in two strings, you need to do explicit conversions to numeric values. The following example uses correct syntax, but can provide the wrong result if you want to compare numbers.

In VBScript:

```
varX = getVar("qty-avail")
varY = getVar("qty-requested")
if varX < varY then            ' This is incorrect because it
                               ' performs an alphanumeric
                               ' comparison.
...
```

The comparison should have been written like this:

```
if cint(varX) < cint(varY) then
```

Or:

```
if varX - varY < 0 then
```

The same example in JavaScript, with correct syntax but incorrect result:

```
varX = getVar("qty_avail");
varY = getVar("qty_requested");
```

```
// Note this comparison is incorrect
if (varX && varY) {
  println ("false");
  VerifyResult ("pass", "pass", "getVar_alphanumericcompare");
} else {
  println ("true");
  VerifyResult ("pass", "fail", "getVar_alphanumericcompare");
}
```

The comparison should have been written like this:

```
if (parseInt(varX) && parseInt(varY)) {
...
```

Storing a value in a context variant

Use the **setVar** method to set the value of a context variant.

In VBScript:

setVar *context-variant*, *value*

In JavaScript:

setVar (*context-variant*, *value*);

value is a string or a variable referring to a string.

In VBScript:

```
sub main
  setVar "approval_response", "yes"
end sub
```

Or:

```
sub main
  dim varY
  varY = "yes"
  setVar "approval_response", varY
end sub
```

NOTE: In VBScript, if a process has more than one parameter and can return a value, and you're writing code that uses the return value, you must use parentheses in the call.

In JavaScript:

```
setVar ("approval_response", "yes");
```

USING THE ACTIVEINPUTSET CONTEXT VARIANT

The ActiveInputSet context variant is useful when you could have different sets of business objects input to a private process, and you want to process them uniquely. You can view a description of what objects are input for each ActiveInputSet value by double-clicking the input icon (green triangle) at the top of the private process in the Private Process window.

The following example sets the path depending on the input business object.

In VBScript:

```
sub main
  set priv_context = getPrivateProcessContext ()
  if getVar("ActiveInputSet") = "InputSet1" then
      priv_context.setPath("Deal with I1")
  else
      setPath("Deal with I2")
  end if
end sub
```

In JavaScript:

```
function main () {
  priv_context = getPrivateProcessContext ();
  if getVar(("ActiveInputSet") == "InputSet1") {
      priv_context.setPath("Deal with I1");
  } else {
      priv_context.setPath("Deal with I2");
  }
}
```

See *Setting the path in a private process* on page 67 for more information on the setPath method.

Instantiating and accessing a business object in a script

A business object is a container for the data that partners exchange, or a temporary holder for complex information within one or more steps for a single partner. For example, a business object might contain the information for a purchase order.

A business object type specifies what elements are in a business object, and their hierarchy. A business object instance is stored in a context business object variable, which is defined to be of a particular business object type. You use the context business object variable to reference the business object in a script.

A context business object variable can be defined as part of a public or private process. If the context of a variable is public, all private process actions *owned by your organization* in that public process can use it; if private, just actions in that private process have access to it. Normally, you declare context business object variables at the private process level. However, the inputs and outputs of subprocesses must be declared at the public level. Also, you can use a public context variable to store data across multiple private processes without making it visible to partners. (Remember that an output object action sends data to a partner.)

Before you can manipulate a business object in a script, you must have a context business object variable, either created by the PAM user or Partner Agreement Manager:

- PAM users can create context business object variables in the Public or Private Process window.

- Partner Agreement Manager automatically creates private-level context variables for business objects that are input to a private process; it names them I1, I2, and so on. (I is short for "Input.") For example, if two paths converge at a public step, the private process for that step automatically has two context business object variables: I1 and I2.

Also, each business object needs to be instantiated:

- You can create a business object instance with the createBO method. It stores the business object in a context business object variable.

- If the business object was input to the private process (for example, I1), it has been instantiated.

- The business object could have already been instantiated in a previous action. You can test if a context business object variable already refers to a business object instance by using the isBONull method.

WARNING: You should not call createBO on context business object variables that already have data, or you could lose the data.

VIEWING AND CREATING A BUSINESS OBJECT TYPE

Context business object variables are defined to be of a particular business object type. Remember that the business objects output by a private process must be of the same type as the business objects output by the corresponding public process step. When you declare a context business object variable in a private process, you should make sure the type matches that expected by the public process.

To create a business object type:

1 From the Process Manager window, click the New Element Definition Set button in the toolbar, define the element definition set in the Element Definition Set window.

2 Choose Create Business Object Type from the File menu.

To view a type:

1 In the left pane of the Process Manager window, select the folder in this hierarchy: *MyOrg* > Business Objects > *MyOrg* or *Partner*; in the right pane, double-click the type to view information about it.

2 From a Process window, choose Business Object Catalog from the View menu, and expand the tree until you reach the type you want to look at.

CREATING A CONTEXT BUSINESS OBJECT VARIABLE

To create a context business object variable for use in a script, you must do the following:

1 Create a business object type, or determine the name of an existing type you want to use.

See the previous section for more information.

2 Add a context business object variable of this business object type.

To create a new context business object variable, open the public or private process in a Process window, then choose New Business Object from the Variables menu.

If the context of a variable is public, all public process steps *owned by your organization* in that public process can use it; if private, just actions in that private process have access to it.

You can now use the variable in scripts. You can pass the context business object variable to the createBO method to create a business object instance.

INSTANTIATING A BUSINESS OBJECT

Business objects must be instantiated before you can use them in scripts. To instantiate a business object, call createBO, passing it a context business object variable.

In both VBScript and JavaScript:

createBO(*context-business-object-variable*)

For example, in VBScript:

```
sub main
  createBO("po")
  po.setData "po_number", "24567"
end sub
```

Or, in JavaScript:

```
createBO ("po");
po.setData ("po_number), '24567");
```

If you call this method on a context business object variable containing data, the data may be cleared, so you should only call this method on context business object variable that does not yet refer to a business object instance.

DETERMINING IF A BUSINESS OBJECT HAS BEEN INSTANTIATED

You can use the isBONull method to determine if a business object has already been instantiated (so you don't have to call the createBO method).

In both VBScript and JavaScript:

isBONull(*context-business-object-variable*)

The method returns Boolean true if the context business object variable hasn't already been created, or false if it has; in the latter case, you do not want to call the `createBO` method.

In VBScript:

```
if (isBONull("po")) then
  createBO("po")
end if
```

In JavaScript:

```
if (isBONull("po")) {
  createBO("po");
}
```

ACCESSING ELEMENTS AND ELEMENT SEQUENCES IN A BUSINESS OBJECT

A business object is made of fields and groups. Both fields and groups are *elements*: fields are elements that contain data and groups are elements that contain other subordinate elements. A context business object variable is a reference to the top-level element of a business object type, and is usually a group.

An *element sequence* is a collection of consecutive elements of the same element type within a business object and at the same level in the hierarchy.

After a business object has been instantiated, you can access and manipulate the elements and sequences it contains. When you want to use a method to perform an operation, you can identify an element or element sequence in any of these ways:

- To specify the top-level element of a business object, use a context business object variable.

- To specify an element or sequence under the top-level element, you can get an element reference by using the getElement, getElementSequence, getElementAt, newElement, or NewElementAt method.
- To specify a subordinate element, use a context business object variable or an element reference, and a *tag path* to the subordinate element, relative to the context variable or element reference. A *tag path* string specifies the hierarchy from a higher-level element to a lower-level element. The name of an element, specified in the business object type, is called its *tag name*.

Getting a reference to an element or element sequence

All of the methods listed in this section return an element or element sequence reference that you can use with other PAM methods. You do not have to get an element or sequence reference. Instead, you can specify a context business object variable and tag path, as described in the following section.

Note: The top-level element of a business object is referenced by the context business object variable. So you would not need to get a reference to a top-level element by using the following methods.

Remember that for VBScript, whenever you get an element or element sequence reference and store it in a variable, you need to use the Set method. When you assign a value to a context variant, you do not use the Set method.

Getting an element

Use the getElement method to get a reference to a group or field identified by a tag path string (tag paths are described in the following section on page 52). In both VBScript and JavaScript:

element.getElement(*tag-path*)

The tag path can specify an element that is not repeatable, or specify an element in an element sequence by its index number.

For example, after the following lines run, the shipping_address variable holds a reference to the ship_to group. po_line is a repeatable group in the context business object variable called po; the reference would be to the ship_to group in the first po_line element of the sequence. You could then manipulate the ship_to group by using the shipping_address variable.

In VBScript:

```
dim shipping_address
set shipping_address = po.getElement("po_line[0]/ship_to")
```

In JavaScript:

```
var shipping_address;
shipping_address = po.getElement("po_line[0]/ship_to");
```

Getting an Element Sequence

Use the **getElementSequence** method to get a reference to an entire element sequence, which is a collection of consecutive "sibling" elements (repeatable) of the same element type. In both VBScript and JavaScript:

element.getElementSequence(*tag-path*)

This example creates a reference to the po_line element sequence, stored in the lines variable.

In VBScript:

```
set lines = po.getElementSequence("po_line")
```

In JavaScript:

```
lines = po.getElementSequence("po_line");
```

Getting an element at a specific sequence position

The **getElementAt** method returns a reference to the group or field at the specified position in an element sequence. In both VBScript and JavaScript:

element-sequence.getElementAt(*index*)

This example gets the element at the position i.

In VBScript:

```
set single_line =
po.getElementSequence("po_line").getElementAt(i)
```

Note that the previous statement is equivalent to the following getElement statement ("&i&" adds the value of i to the tag path string).

```
set single_line = po.getElement("po_line["&i&"]")
```

In JavaScript:

```
single_line = po.getElementSequence("po_line").getElementAt(i);
```

Note that the previous statement is equivalent to the following getElement statement ("+i+" adds the value of i to the tag path string).

```
single_line = po.getElement("po_line["+i+"]");
```

The `newElement` and `newElementAt` methods, which add an element to an element sequence, also return an element object reference. See *Adding an element to a sequence* on page 64 for more information.

SPECIFYING TAG PATHS TO ELEMENTS AND SEQUENCES

A tag path string specifies an element or element sequence. The tag path is relative to the element you are calling the method on. Tag paths are made of element names, as specified in the business object type; element names are case-sensitive.

SPECIFYING AN ELEMENT

If the tag path string is empty (`""`), represented by quotes with no spaces between, it specifies the element itself. Otherwise, it specifies a subordinate element, with each element identifier delimited by a slash (/). The path is relative to the element you are calling the method on. For example, if you are calling a method on the context business object variable po, and a group under it is summary_info that has a field comments, the tag path to the field comments would be summary_info/comments.

In this VBScript statement:

```
set elemComments = po.getElement("summary_info/comments")
```

Or, in JavaScript:

```
elemComments = po.getElement("summary_info/comments");
```

SPECIFYING AN ELEMENT IN AN ELEMENT SEQUENCE

For groups and fields that may repeat, you can refer to a particular element in an element sequence by using an index number. Element sequences are indexed starting at zero (0), so four occurrences of an element will be indexed 0, 1, 2, 3. For example, po_line[0] would be the first po_line element and po_line[1] would be the second.

In this VBScript statement:

```
set elemShip_to = po.getElement("po_line[0]/ship_to")
```

Or, in JavaScript:

```
elemShip_to = po.getElement("po_line[0]/ship_to");
```

This statement gets a reference to the ship_to element contained by the first po_line element in the context business object variable called po.

Specifying an entire element sequence

For repeatable groups and fields, you can specify an entire element sequence by not specifying an index number.

In VBScript:

```
set eseqLines = po.getElementSequence("po_line")
```

In JavaScript:

```
eseqLines = po.getElementSequence("po_line");
```

Working with group and field elements

Partner Agreement Manager script extensions provide many methods to access, modify the data, and get information on the groups and fields of a business object. You can use the methods in this section to manipulate an element, which could be:

- a nonrepeatable group or field.
- a group or field that is part of an element sequence.
- a business object (specified by the context business object variable name).

Note: An empty string ("") as a data value is equivalent to null. A string containing one or more spaces (" ") is not null, and is considered to be data.

Getting data from a field

Use the **getData** method to get data in a field; it returns a string representation of the data contained in the field. You can optionally supply a tag path string.

In both VBScript and JavaScript:

field.getData()
element.getData(*tag-path*)

tag-path specifies the path to the field, relative to *element*.

The following code returns the value of the item_code field in the first po_line group of the po business object; po_line is an element sequence, and this statement gets the item_code value in the first element in the sequence. The item variant holds a string.

In VBScript:

```
dim item
item = po.getData("po_line[0]/item_code")
```

In JavaScript:

```
var item ;
item = po.getData("po_line[0]/item_code");
```

Or you could write the same example as:

In VBScript:

```
dim item
item = po.getElement("po_line[0]/item_code").getData()
```

In JavaScript:

```
var item;
item = po.getElement("po_line[0]/item_code").getData();
```

Adding data to a field

Use the **setData** method to set data in a field. Any existing data is overwritten. An exception is thrown if a user tries to set invalid data; for example, trying to set an ANY or EMPTY element (XML), or trying to change fixed attribute data.

In VBScript:

field.setData *value*
element.setData *tag-path*, *value*

In JavaScript:

field.setData(*value*);
element.setData(*tag-path*, *value*);

value must be a string. *tag-path* specifies the path to the field, relative to *element.*

The following example sets the po_number field in the po business object to the string 24567.

In VBScript:

```
po_num = "24567"
po.setData "po_number", po_num
```

In JavaScript:

```
po_num = '24567";
po.setData ("po_number", po_num);
```

Or you could write the same example as:

In VBScript:

```
po_num = "24567"
po.getElement("po_number").setData po_num
```

In JavaScript:

```
po_num = "24567";
po.getElement("po_number").setData (po_num);
```

Copying data into an element

Use the copyIn method to copy data from an element to another element, either within the same business object or between business objects. The elements copied from (and to) must be of the same structure. The elements:

- Do not need to have the same name.
- Must have identical child elements or one should be a subset of the copied to element.
- Attributes must be exactly the same.

You can copy smaller groups into larger groups, but not the reverse. The elements in the larger group that are not in the smaller group will be initialized.

Subordinate element sequences are also copied, such that their lengths will match that of the sequences copied from. The sequence copied to is removed and replaced by the copy.

In VBScript:

element2.copyIn *element1*
element.copyIn *tag-path*, *element1*

In JavaScript:

element2.copyIn(*element1*);
element.copyIn(*tag-path*, *element1*);

Where *element1* is the element to copy from and *element2* is the element to copy to. *tag-path* is the path to the element to copy to, relative to *element.*

NOTE: The following generic statement does not copy a value but creates an element reference. You should use the copyIn method to copy data.

In VBScript:

```
set elemGroup1 = bo.getElement("Group")
```

In JavaScript:

```
elemGroup1 = bo.getElement ("Group");
```

The following example copies the content of the first element in the po_line element sequence into the second element of the po_line element sequence.

In VBScript:

```
set line1 = po.getElement("po_line[0]")
set line2 = po.getElement("po_line[1]")
line2.copyIn line1
```

Or you could write the code as:

```
po.copyIn "po_line[1]", po.getElement("po_line[0]")
```

In JavaScript:

```
line1 = po.getElement("po_line[0]");
line2 = po.getElement("po_line[1]");
line2.copyIn line1;
```

Or you could write the code as:

```
po.copyIn ("po_line[1]", po.getElement("po_line[0]"));
```

CLEARING DATA FROM AN ELEMENT

Use the clearAll method to remove all data contained in an element. If the element is a group, all subordinate fields are cleared. The length of any element sequences within this element are set to zero (0). The clearData method also removes all data, but does not change the length of element sequences.

In VBScript:

element.clearAll
element.clearData

In JavaScript:

element.clearAll();
element.clearData();

When you call clearAll on a repeatable group, if there are subordinate element sequences that have been populated, the length of the element sequences become zero (0); however, the sequence length of the group you called the method on stays the same.

The following statement clears the entire po business object.

In VBScript:

```
po.clearAll
```

In JavaScript:

```
po.clearAll();
```

The next statement does the same, but the length of the element sequences do not change.

In VBScript:

```
po.clearData
```

In JavaScript:

```
po.clearData();
```

Both of the following statements clear the elements in the first po_line group in the po business object, but one clears lengths and the other doesn't.

In VBScript:

```
po.getElement("po_line[0]").clearAll
po.getElement("po_line[0]").clearData
```

In JavaScript:

```
po.getElement("po_line[0]").clearAll();
po.getElement("po_line[0]").clearData();
```

GETTING THE NAME OF AN ELEMENT

Use the **getTagName** method to get the name of an element. This can also be thought of as the element type name (as specified in the business object type). In both VBScript and JavaScript:

element.getTagName()

You can use this method to write your own generic methods. The following subroutine takes a field and prints its name followed by its data value.

In VBScript:

```
sub printDataValue(el)
  println(el.getTagName()&" : "&el.getData())
end sub
```

In JavaScript:

```
function printDataValue(el) {
```

```
    println(el.getTagName() +" : "+el.getData());
}
```

Checking if an element is a group or field

The isField method checks whether an element is a group or field. This method is useful when you want to use code that can operate on different business object types, for example. In both VBScript and JavaScript:

element.isField()

The following example supports the processing of two different business object types: one type has ship_to as a group with an address field within it, while the other type has ship_to as a field holding address data. The following piece of code is reusable for both cases.

In VBScript:
```
function getShippingAddress(ship_to)
  if (ship_to.isField()) then
     getShippingAddress = ship_to.getData()
  else
     getShippingAddress = ship_to.getData("address")
  end if
end function
```

In JavaScript:
```
function getShippingAddress(ship_to) {
  if (ship_to.isField()) {
     getShippingAddress = ship_to.getData();
  } else {
     getShippingAddress = ship_to.getData("address");
  }
}
```

Checking if an element contains data

The hasData method checks whether a field or attribute contains data (is not null) or if any field in a group contains data. If you supply an element sequence, all elements in the sequence are checked for data. In both VBScript and JavaScript:

element.hasData()
element-sequence.hasData()

It returns Boolean true if it contains data.

The following example checks if there are any elements in the design_drawing element sequence in the po business object.

In VBScript:

```
if (po.getElementSequence("design_drawing").hasData() = False) then
  println("There are no design drawings associated with this PO"& _
      po.getData("po_number"))
end if
```

Note: The underscore [_] means the statement was continued on the next line.

In JavaScript:

```
if (po.getElementSequence("design_drawing").hasData() == False) {
  println("There is no design drawing associated with this PO " +
      po.getData("po_number"))
}
```

The hasData method is often used with the isValid method, as described in the next section.

Checking if an element is valid

The isValid method determines the validity of an element and its attributes based on how it was defined in the business object type:

- If the element you're checking is a group, isValid determines whether all mandatory fields it contains have data; if a subordinate optional group contains data, it also checks whether all mandatory fields in the optional group have data. If a subordinate optional group does not have data in it, it is ignored when determining the validity of the element.
- If the element is a field, isValid checks whether it contains data or not.

isValid does not consider whether the element you're calling the method on was defined as optional or mandatory when it determines validity. (Remember that the parent element defines whether an element is optional or mandatory.) For fields, it only checks whether the field has data or not; for groups, isValid checks the elements it contains for validity based on whether they are optional or mandatory.

Here is the syntax for both VBScript and JavaScript:

element.isValid()

isValid returns Boolean true if the element is a field containing data or is a group whose subordinate elements are valid (or are optional and contain no data), or returns false if the element is a null field or is a group with a subordinate element that requires data but is null.

In this example, if the summary_info group has data in it but it is not valid, it prints an error message for the user.

In VBScript:

```
dim summary = po.getElement("summary_info")
if (summary.hasData()) then
  if (summary.isValid() = False) then
     println("All mandatory fields of summary must be filled
in")
  end if
end if
```

In JavaScript:

```
  if (po.getElement("summary_info").hasData()) {
     if (po.getElement("summary_info").isValid() == False) {
        println("All mandatory fields of summary " +
           "must be filled in")
     }
  }
```

See *isValid method* on page 145 for more complex examples.

Getting descriptive information about an element

The toString method returns a string describing the validity and content of an element and any elements subordinate to it. All mandatory subordinate elements appear in the description; optional subordinate elements without any data do not appear. This helps you see which elements must have data for this element to be valid. You can use the println method to display the value returned by toString. Here is the syntax in both VBScript and JavaScript:

element.toString(*include-data*)

If *include-data* is Boolean true, toString includes any data values in the description; if false, it does not include the data values, which makes the description shorter.

This example prints the description string, including the data values, for the po business object.

In VBScript:
```
println(po.toString(True))
```

In JavaScript:
```
println(po.toString(True));
```

The output might look like this (note that the top-level Purchase_Order element is not valid because the mandatory field po_date has no value):

```
<Purchase_Order valid="false">
<po_number valid="true">123</po_number>
<po_date valid="false"></po_date>
<supplier_id valid="true">99999</supplier_id>
<po_line valid="true">
<item_code valid="true">2222</item_code>
<qty valid="true">55</qty>
<expected_ship_date valid="true">9.9.99</expected_ship_date>
<summary_info valid="true">
<comments valid="true">first line item</comments>
</summary_infor>
</po_line>
</Purchase_Order>
```

Working with element sequences

An element sequence is a collection of consecutive elements of the same element type. The following methods operate on an element sequence; in the call, you use a reference to an element sequence that was returned by the getElementSequence method. Remember that the methods described in the previous section operate on an element, which can be part of a sequence.

Checking if an element sequence contains data

When operating on an element sequence, the hasData method checks whether any field in an element sequence contains data. In both VBScript and JavaScript:

*element-sequence.*hasData()

It returns Boolean true if it contains data.

The following example checks if there are any design_drawing elements in the po business object.

In VBScript:
```
if (po.getElementSequence("design_drawing").hasData() = False) then
  println("There are no design drawings associated with this PO"& _
      po.getData("po_number"))
end if
```

In JavaScript:
```
if (po.getElementSequence("design_drawing").hasData() == False) {
  println("There is no design drawing associated with this PO " +
        po.getData("po_number"))
  }
```

Determining how many elements are in a sequence

The length method returns the number of elements in an element sequence. This is useful for setting boundary values to loop through all the elements in an element sequence. In both VBScript and JavaScript:

*element-sequence.*length()

The following example loops through all the po_line elements and prints the data in each po_line in string form.

In VBScript:

```
dim lines, nLines
set lines = po.getElementSequence("po_line")
nLines = lines.length()
for i=0 to nLines - 1
  println(lines.getElementAt(i).toString(true))
next
```

In JavaScript:

```
var lines, nLines;
lines = po.getElementSequence("po_line");
nLines = lines.length();
for (i=0; i==nLines - 1; i++) {
  println(lines.getElementAt(i).toString(true));
}
```

ADDING AN ELEMENT TO A SEQUENCE

The newElement method adds a new element to the end of a sequence and returns a reference to the newly created element. The index of the new element will be *length* - 1 (where *length* refers to the value after the method is called). In both VBScript and JavaScript:

element-sequence.newElement()

The newElementAt method inserts a new element at the specified position in the sequence and returns a reference to the newly created element. It adds 1 to the index of the element currently in that position (if any) and any following elements, so they are "shifted to the right." Valid index values are 0 to *length*. So, as with the newElement method, you can use this method to add to the end of the sequence. Here is the syntax in both VBScript and JavaScript:

element-sequence.newElementAt(*index*)

The following example creates a new po_line element at the end of the sequence and copies into it the values of the element before it in the sequence.

In VBScript:

```
set lines = po.getElementSequence("po_line")
```

```
nlines = lines.length()
last_line_index = nlines - 1
set new_line = lines.newElement()
if (last_line_index >= 0) then
  new_line.copyIn(lines.getElementAt(last_line_index))
end if
```

In JavaScript:

```
lines = po.getElementSequence("po_line");
nlines = lines.length();
last_line_index = nlines - 1;
new_line = lines.newElement();
if (last_line_index >= 0) {
  new_line.copyIn(lines.getElementAt(last_line_index));
}
```

The next example adds a new po_ line element at the beginning of the sequence.

In VBScript:

```
set line = po.getElementSequence("po_line").newElementAt(0)
```

In JavaScript:

```
line = po.getElementSequence("po_line").newElementAt(0);
```

DELETING AN ELEMENT FROM A SEQUENCE

The removeAll method removes all elements in a sequence. Any data contained in any of the elements is lost. The length of this element sequence becomes 0.

The removeElementAt method removes the element (and its data) at the specified position in a sequence. The indices of elements at greater index values are reduced by 1 (they are "shifted to the left"). The length of the sequence is reduced by 1.

In VBScript:

element-sequence.removeAll
element-sequence.removeElementAt *index*

In JavaScript:

element-sequence.removeAll();
element-sequence.removeElementAt(*index*);

The following example removes all the po_line elements from the po business object.

In VBScript:

```
po.getElementSequence("po_line").removeAll
```

In JavaScript:

```
po.getElementSequence("po_line").removeAll();
```

The next example loops through all po_line elements and removes all of the invalid ones. Note that removeElementAt will move the remaining elements one index down in the sequence. So you should perform this operation starting at the end of the sequence so your index value is always valid.

In VBScript:

```
dim line
dim lines

set lines = po.getElementSequence("po_line")
for i = lines.length() - 1 to 0 step -1
  set line = lines.getElementAt(i)
  if not line.isValid() then
      lines.removeElementAt(i)
  end if
next
```

In JavaScript:

```
var line;
var lines;

lines = po.getElementSequence("po_line");
for (i = lines.length() - 1; i >= 0; i--) {
  line = lines.getElementAt(i);
  if (!line.isValid()) {
      lines.removeElementAt(i);
  }
}
```

Setting the path in a private process

You can use the **setPath** method to set the path in a private process. You must make a setPath call before each branch in the process flow. (The path directly follows the action that contains the script.)

In VBScript:

PrivateProcessContext.setPath *private-process-path-name*

In JavaScript:

PrivateProcessContext.setPath(*private-process-path-name*);

You must type the context-sensitive path name exactly as it appears in the Private Process window. If you do not provide a path name for a loop, the default path is taken; this is the main, straight-line path. For branches, you must always provide a path.

Remember that public process paths are determined by private-process output business objects only.

Following is an example of using the **setPath** method to set the path based on the customer name.

In VBScript:

```
sub main
  dim customerName
  dim priv_context
  set priv_context = getPrivateProcessContext ()
  customerName = getVar("customer_name")
  if (customerName = "Acme") then
      priv_context.setPath("Approved")
  else
      priv_context.setPath("NotApproved")
  end if
end sub
```

Next is an example of using the **setPath** method in the loop block of a private process to cycle through the loop five times.

In VBScript:

```
sub main
```

```
  dim counter
  dim priv_context
  set priv_context = getPrivateProcessContext ()
  counter = getVar("loop_counter")
  if counter >= 5 then
      priv_context.setPath("Main")
  else
      counter = counter + 1
      setVar "loop_counter", counter
      priv_context.setPath("Loop")
  end if
end sub
```

In JavaScript:

```
var counter;
var priv_context;
priv_context = getPrivateProcessContext ();
counter = getVar("loop_counter");
if (counter >= 5) {
  priv_context.setPath("Main");
} else {
  priv_context.setPath("Loop");
}
```

The following script tests whether an approval action timed out before the user could respond. Assume that a context variant called TimerFlag is set by the action.

In VBScript:

```
sub main
  set priv_context = getPrivateProcessContext ()
  timer_flag = getVar("TimerFlag")
  if timer_flag = "TIMEOUT" then
      priv_context.setPath("Escalate")
  else
      priv_context.setPath("Check Approval Response")
  end if
end sub
```

In JavaScript:

```
var timer_flag;
priv_context = getPrivateProcessContext ();
```

```
timer_flag = getVar("TimerFlag");
if (timer_flag += "TIMEOUT") {
  priv_context.setPath("Escalate");
} else {
  priv_context.setPath("Check Approval Response");
}
```

The next script tests the response of the approval question for an approval action. Assume that a context variant ApprovalFlag is set by the action.

In VBScript:

```
sub main
  set priv_context = getPrivateProcessContext ()
  app_flag = getVar("ApprovalFlag")
  if app_flag then
      priv_context.setPath("Allocation Quantity OK")
  else
      priv_context.setPath("Request Order Refinement")
  end if
end sub
```

In JavaScript:

```
priv_context = getPrivateProcessContext ();
app_flag = getVar("ApprovalFlag");
if (app_flag) {
  priv_context.setPath("Allocation Quantity OK");
} else {
  priv_context.setPath("Request Order Refinement");
}
```

Printing a message to the console and log file

You can use the println method to print a message to the server's console window and log file. This is useful for debugging, because you can track the progress of your script and display values of context variables and business object fields, for example. The syntax in both JavaScript and VBScript:

println(*string*)

Here is a simple example in VBScript:

```
sub main
```

```
    println("A script")
end sub
```

The same example in JavaScript:

```
function main () {
  println ("A script");
}
```

For additional examples, see *Getting descriptive information about an element* on page 62 and *Determining how many elements are in a sequence* on page 63.

If you run Partner Agreement Manager as a service, println output goes into a log file. The log file is called PAM.log and it is created in the Partners\Partner*nnn* directory. If you run Partner Agreement Manager from a shortcut, println output is displayed in the server's console and also goes to the log file.

Handling run-time errors and exceptions

If you get an error in a script, the script terminates, unless you handle the error. Some of the methods return error codes that you can handle in your script code:

Method	Run-time error/exception
copyIn	ElementTypeException, InvalidQueryException, IndexOutOfBoundsException
getData	ElementTypeException, InvalidQueryException, IndexOutOfBoundsException
getElement	InvalidQueryException, IndexOutOfBoundsException
getElementAt	IndexOutOfBoundsException
getElementSequence	InvalidQueryException, IndexOutOfBoundsException
getElementAt	IndexOutOfBoundsException
newElementAt	IndexOutOfBoundsException
setAttr	InvalidDataException
setData	ElementTypeException, InvalidDataException, InvalidQueryException, IndexOutOfBoundsException

Here are descriptions of these errors:

Run-time error/exception	Description
ElementTypeException	The element types do not match. For the getData and setData methods, you called the method on a group element instead of a field element. For the copyIn method, you tried to copy to an element with a different tag name, for example, from po_data to po_number.
IndexOutOfBoundsException	When specifying an element in a sequence, you provided an invalid index value, such as a number greater than *length* - 1 for that element sequence.
InvalidDataException	The data that you are trying to set in the element or attribute is not valid for the definition of the element or attribute. For example, you will get this exception when trying to set data to an ANY or EMPTY element type.
InvalidQueryException	The tag path you supplied is invalid (if the index value was invalid, you would get IndexOutOfBoundsException instead). For example, you can get this error if you typed a wrong name, such as PO_Data instead of po_data.

Following is an example of handling errors by using VBScript inline error handling. The errCheck function, defined below, checks whether a run-time error has occurred. It returns true if the error is encountered; false if it has not. The errCheck function is called in the main method. See VBScript documentation for more information on error handling.

```
rtErr = "NONE"

sub main()
  dim value
  dim priv_context

  ' Get the private process context
  set priv_context = getPrivateProcessContext ()

  ' In the case of an error, continue execution.
  On Error Resume Next
```

```
                ' Get a field value.
              value = po.getData("po_line[0]/ship_to")
              ' Handle run-time errors generated by the previous statement.
              if errCheck() then
                  if InStr(rtErr, "InvalidQueryException") <> 0 then
                      priv_context.setPath "LoopQuery"
                  elseif InStr(rtErr, "ElementTypeException") <> 0 then
                      priv_context.setPath "ExitBranch"
                  elseif InStr(rtErr, "IndexOutOfBoundsException") <> 0 then
                      priv_context.setPath "IndexBranch"
                  end if
              else
                  println("getData call successful")
              end if
            end sub


        ' ErrCheck checks whether a run-time error occurred.
        ' It returns False if the no run-time error is encountered,
        ' True otherwise.
        function errCheck()
          if Err.Number <> 0 then
              ' an error has occurred
              errCheck = True
              println("===***===???? Error check-" & _
                  "  Error number: " & Err.Number & ", " & _
                  Err.Description & " has occurred in " _
                  & Err.Source)
              rtErr = Err.Description
          else
              errCheck = False
          end if
          Err.Clear
        end function
```

5

Script method reference

This chapter is a reference for the Partner Agreement Manager script extension methods—available for use with VBScript or JavaScript—that let you manipulate business objects and context variants. Use the methods to develop scripts that are tailored to Partner Agreement Manager features.

Sections in this chapter include:

What methods are available?

In addition to your own custom scripts, Partner Agreement Manager provides script extension methods for common process needs. Script extensions can be used to access the public, private, and partner group contexts.

Script extensions for all contexts

The following script extensions can be used in both the public and private contexts:

- *General script extension methods* (for example, setVar or println)
- *BusinessObject methods*
- *Element methods*
- *ElementSequence methods*

Script extensions for specific contexts

The following script extensions can be used only in specific contexts. Each interface name defines the applicable context.

- *PublicProcessContext methods*
- *PartnerGroupContext methods*
- *PublicProcessNodeInput methods*
- *PrivateProcessContext methods*

General script extension methods

This is a general grouping of the most commonly used script extension methods. The following table provides a summary:

Use this	To do this
createBO method	Create a business object instance stored in a context business object variable. After you call this method, you can add data to the business object. You should not call this method on a context business object variable that already contains a business object instance, because any data may be cleared.
getPrivateProcessContext method	Retrieve information about the private process in which the action is executing.
getPublicProcessContext method	Retrieve information about the public process that activated the private process in which the action is executing.
getVar method	Get a value from a context variant. It returns a string containing the value.
isBONull method	Test if a context business object variable contains a business object instance. The method returns Boolean false if it does; in this case, you do not want to call the createBO method on the context business object variable.
main method	Create the entry point for the Partner Agreement Manager script. Each script must have a main method.
println method	Print a message to the console. This is useful for debugging.
setPath method	Deprecated. Use the setPath method that uses the PrivateProcessContext instead.
setVar method	Store a value in a context variant.

BusinessObject methods

The BusinessObject methods let you access information about the type and root element of the business object. In addition, they provide methods that read and write the business object to a character stream. This is useful when creating or writing a business object to, or from, an XML flat file.

Following is a summary of the BusinessObject methods that are available:

Use this	To do this
deepClone method	Return a new business object containing the same type information and data content as this business object.
fromXMLString method	Initialize the content of this business object from the specified string.
getDocument method	Return a reference to this business object via the w3c XML DOM API.
getRootElement method	Get the root element of the business object content.
getTypeID method	Get the type identifier for this business object.
toXMLString method	Return the content of this business object in a string.
validate method	Check the validity of an element in this business object.

Element methods

The Element methods let you access and manipulate the content of a business object instance. Both fields and groups are elements:

- Fields are elements that can contain data.

- Groups are elements that contain other elements.

Following is a summary of the Element methods that are available:

Use this	To do this
clearAll method	Remove all data contained in an element. If the element is a group, all subordinate fields are cleared. The length of any element sequences within this element are set to zero (0).
clearData method	Remove all data contained in an element. If the element is a group, all subordinate fields are cleared. Any element sequences in this element keep their current length.
copyIn method	Copy data from an element to another element of the same business object type, either within the same business object or between business objects. The elements copied from and to must be of the same element type: you can copy a group into a group, a field into a field, and they must have the same name (as specified in the business object type) and have the same hierarchy of subordinate elements. Subordinate element sequences are copied; the element sequence length does not have to be the same between elements.
getAttr method	Retrieve an attribute value by name.
getAttrDefault method	Retrieve the default attribute value by name.
getBusinessObject method	Get the business object which "owns" this element instance.
getData method	Return a string representation of the data contained in a field.
getElement method	Get a reference to the group or field identified by a tag path string. The tag path can specify an element in a sequence by its index number.

Use this	To do this
getElementSequence method	Get a reference to an element sequence. An element sequence is a collection of consecutive "sibling" elements of the same element type. The group or field is specified as repeatable in the group that contains it.
getTagName method	Get the name of this element. This can also be thought of as the element type name.
hasData method	Check whether a group or field contains data (is not null).
isField method	Check whether an element is a field or group. This method is useful when you want to use code that can manipulate different business object types, for example.
isValid method	Determine the validity of an element based on the content of the business object type.
removeAttr method	Remove an attribute by name.
removeAttrs method	Remove all attributes of the element.
removeElement method	Remove the element specified by the tag path, from the business object.
setAttr method	Add a new attribute.
setData method	Set the data contained in a field element.
toString method	Return a string describing the validity and content of an element and any elements subordinate to it that contain data. Mandatory subordinate elements appear in the description string; optional subordinate elements without any data do not appear. This helps you see which elements must have data for this element to be valid. You can use the println method to display the value returned by toString.

ELEMENTSEQUENCE METHODS

The ElementSequence methods let you manipulate an element sequence: a collection of consecutive Element objects of the same element type. Element sequences are indexed starting at zero (0), for example, four occurrences of an element (the sequence length is 4) will be indexed 0, 1, 2, 3. Valid index values are zero to the length of the sequence minus one (0 to *length* - 1). Following is a summary of the ElementSequence methods that are available:

Use this	To do this
getElementAt method	Get the group or field at the specified position in this element sequence.
getTagName method	Get the tag name of this element sequence.
hasData method	Check whether any element in an element sequence contains data.
length method	Return the number of elements in this element sequence. This is useful for setting boundary values to loop through all the elements in an element sequence.
newElement method	Add a new element to the end of this sequence and return a reference to the newly created element.
newElementAt method	Insert a new element at the specified position in the sequence and return a reference to the newly created element. Adds 1 to the index of the element currently in that position (if any) and any following elements, so they are "shifted to the right." As with the newElement method, you can use this method to add to the end of the sequence.
removeAll method	Remove all elements in a sequence. Any data contained in any of the elements is deleted. The length of this element sequence becomes 0.
removeElementAt method	Remove an element (and its data) at the specified position in a sequence. The indexes of elements at greater index values are reduced by 1 (they are "shifted to the left"). The length of the sequence is reduced by 1.

PublicProcessContext methods

The PublicProcessContext methods let you get information about the public process. Following is a summary of the PublicProcessContext methods that are available:

Use this	To do this
getInputs method	Get the inputs to this public process node.
getLoopID method	Get the public process loop ID. If the public process is not in a loop, return a string length of 0.
getNodeTypeID method	Get the type ID of the public process node.
getPartnerGroupContext method	Retrieve the context object that can be used to access partner group information in the context of this public process.
getProcessRef method	Get the reference to the public process.
getProcessTypeRef method	Get the reference to the public process type.
isProductionProcess method	Determine if the public process is in Production mode or Test mode.

PartnerGroupContext methods

The PartnerGroupContext methods let you get information about the PartnerGroup. Following is a summary of the PartnerGroupContext methods that are available:

Use this	To do this
getBinding method	Get the binding for the specified partner group.
getGroupRefs method	Get the list of references to the groups that are included in this public process.

Note: The partner group context is specific to the public process.

PublicProcessNodeInput methods

The PublicProcessNodeInput methods let you get information on the input to the node of the public process. Following is a summary of the PublicProessNodeInput methods that are available:

Use this	To do this
getSenderNodeTypeID method	Get the ID of the sending node.
getSenderRef method	Get the reference to the sending partner.
getVariableName method	Get the name of the input variable containing this input.

PrivateProcessContext methods

The PrivateProcessContext methods let you get information about the private process and set the private process path. Following is a summary of the PrivateProcessContext methods that are available:

Use this	To do this
getLoopID method	Get the private process loop ID. If the private process is not in a loop, return a string length of 0.
getNodeTypeID method	Get the type ID of the private process node.
getPath method	Get the currently selected path. If no path has been selected using the PrivateProcessContext.setPath call, return null.
getPathNames method	Get the list of valid paths that can be taken.
getProcessRef method	Get the reference to the private process.
getProcessTypeRef method	Get the reference to the private process type.
setPath method	Set the path to be taken when this node completes.

Alphabetical reference

This section is an alphabetical reference to the methods.

Parameters and variables used in the syntax specifications

The following parameters and variables are used in the syntax specifications:

Parameter or Variable	Description
attribute	Name of an attribute.
bo	Name of a context business object variable (a type of context variable) that Partner Agreement Manager created or the PAM user defined from the Public or Private Process window. It is case-sensitive and must be defined in a process that the script executes in.
context-variant	Name of a context variant (a type of context variable) that Partner Agreement Manager created or the PAM user defined from the Public or Private Process window. It is case-sensitive and must be defined in a process that the script executes in.
element	An element reference returned by a getElement, getElementAt, newElement, or newElementAt method (*element* could be the name of the variable that stores the reference). Or a *bo* that stores a business object instance. An element can be a group or field.
element-sequence	An element sequence reference returned by the getElementSequence method. *element-sequence* can be the name of the variable that stores the reference.
field	*element* that stores a reference to a field only.
index	Number zero (0) or greater, referring to a position in an element sequence. Members of an element sequence are numbered starting with zero (0).
string	Variant containing character data.
tag-path	Tag path string specifying the path to an element or element sequence, relative to the element you are calling the method on. See *Specifying tag paths to elements and sequences* on page 52 for more information.

Parameter or Variable	Description
use_external_ID	External definition ID of the business object.
xml_string	The input XML string corresponding to an instance of this type of business object.

NOTE: An empty string ("") as a data value is equivalent to null. A string containing one or more spaces (" ") is not null, and is considered data.

THE BUSINESS OBJECT TYPE USED IN THE EXAMPLES

Many of the examples in the following reference pages act on a business object of the Purchase_Order business object type with the following fields and groups (groups are bold for readability). The context business object variable, of type Purchase_Order, is po.

Group or field name	Type
Purchase_Order	The top-level element of the business object (here a group)
po_number	Mandatory Single Field
po_date	Mandatory Single Field
supplier_id	Mandatory Single Field
po_line	Mandatory Repeatable Group
item_code	Mandatory Single Field
qty	Mandatory Single Field
supplier_item_code	Optional Single Field
expected_ship_date	Mandatory Single Field
rate	Optional Single Field
ship_to	Optional Single Group
address	Mandatory Single Field
attention	Optional Single Field
phone	Optional Single Field
preferred_carrier	Optional Single Field

Group or field name	Type
summary_info	Mandatory Single Group
comments	Mandatory Single Field
comment_by	Optional Single Field
design_drawing	Optional Repeatable Group
location	Optional Single Field
drawing	Mandatory Single Field

clearAll method

Removes all data contained in this element. If the element is a group, all subordinate fields are cleared. The length of any element sequences within this element are set to zero (0). (Note that the clearData method also removes all data, but does not change the length of element sequences.)

When you call clearAll on a repeatable group, if there are subordinate element sequences that have been populated, the length of the element sequences become zero (0); however, the length of the group you called the method on stays the same. If you call the method on a repeatable field, the length of the field becomes zero (0).

VBScript syntax

element.clearAll

JavaScript syntax

element.clearAll();

Variable

element is a reference to a business object, or group or field contained by a business object.

Return value

None

Example

This statement clears the entire po business object. If there are element sequences that have been populated (for repeatable elements po_line or design_drawing), the length of the element sequence becomes zero (0).

VBScript:
```
po.clearAll
```

JavaScript:
```
po.clearAll();
```

This statement clears the first po_line group in the po business object.

VBScript:

```
po.getElement("po_line[0]").clearAll
```

JavaScript:

```
po.getElement("po_line[0]").clearAll();
```

See also

createBO, clearData, getElement, removeAll, removeElementAt

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

clearData method

Removes all data contained in this element. If the element is a group, all subordinate fields are cleared. Any element sequences in this element keep their current length. (Note that the clearAll method also removes all data, but changes the length of element sequences to zero [0].)

VBScript syntax

element.clearData

JavaScript syntax

element.clearData();

Variable

element is a reference to a business object, or group or field contained by a business object.

Return value

None

Examples

This statement clears the entire po business object. If there are element sequences within this object (for repeatable elements po_line or design_drawing), the length of the element sequence stays the same.

VBScript:

```
po.clearData
```

JavaScript:

```
po.clearData();
```

This statement clears the first po_line group in the po business object:

VBScript:

```
po.getElement("po_line[0]").clearData
```

JavaScript:

```
po.getElement("po_line[0]").clearData();
```

createBO, clearAll, getElement, getElementAt, removeAll, removeElementAt

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

copyIn method

Copies an element to another element of the same business object type, either within the same business object or between business objects. The elements copied from (and to) must be of the same structure. The elements:

- Do not need to have the same name.
- Must have identical child elements or one should be a subset of the copied to element.
- Attributes must be exactly the same.

You can copy smaller groups into larger groups, but not the reverse. The elements in the larger group that are not in the smaller group will be initialized.

Subordinate element sequences are also copied, such that their lengths will match that of the sequences copied from. The sequence copied to is removed and replaced by the copy.

VBScript syntax

element2.copyIn *element1*
element.copyIn *tag-path*, *element1*

JavaScript syntax

element2.copyIn(*element1*);
element.copyIn(*tag-path*, *element1*);

Parameters and variables

element1 is a reference to an element to copy from.
element2 is a reference to an element to copy to.
tag-path specifies the element to copy to, relative to *element*, which is a reference to a business object, group, or field.

Return value

None

Run-time errors/exceptions

InvalidQueryException indicates that the tag path you supplied is invalid (if the index value was invalid, you would get IndexOutOfBoundsException instead). For example, you can get this error if you typed a wrong name.

IndexOutOfBoundsException indicates that, when specifying an element in a sequence, you provided an invalid index value, such as a number greater than *length* - 1 for that element sequence.

EXAMPLE

This example copies the ship_to group in the first element of the po_line element sequence to the ship_to group of the second po_line element sequence:

VBScript:

```
set line1_ship_to = po.getElement("po_line[0]/ship_to")
set line2_ship_to = po.getElement("po_line[1]/ship_to")
line2_ship_to.copyIn line1_ship_to
```

JavaScript:

```
line1_ship_to = po.getElement("po_line[0]/ship_to");
line2_ship_to = po.getElement("po_line[1]/ship_to");
line2_ship_to.copyIn (line1_ship_to);
```

Or it could be written as:

VBScript:

```
po.copyIn "po_line[1]/ship_to", po.getElement("po_line[0]/
ship_to")
```

JavaScript:

```
po.copyIn ("po_line[1]/ship_to", po.getElement("po_line[0]/
ship_to"));
```

SEE ALSO

createBO, getData, getElement, getElementAt, setData

createBO method

Creates a business object instance. The instance is stored in a context business object variable, which you define in the Public or Private Process window of Partner Agreement Manager. The context business object variable is defined to be of a particular business object type.

Warning: You should not call createBO on business objects that already have data, or you could lose data. You can check if a business object has been instantiated by using the isBONull method.

VBScript syntax

createBO(*bo*)

JavaScript syntax

createBO(*bo*);

Parameter

bo is a case-sensitive name of a context business object variable in a process that the script executes in.

Return value

Boolean; true if the operation completed successfully, false if it did not

Example

This example creates a business object instance and stores it in the context business object variable called po. Then it sets the value of the po_number field.

VBScript:
```
sub main
  createBO("po")
  po.setData "po_number", "24567"
end sub
```

JavaScript:
```
createBO("po");
po.setData ("po_number", "24567");
```

The following example uses the return value as well:

VBScript:

```
sub main
  if createBO("po") then
      po.setData "po_number", "24567"
  else
      println("Could not create BO")
  end if
end sub
```

JavaScript:

```
if (createBO("po")) {
    po.setData ("po_number", "24567");
  } else {
      println("Could not create BO");
  }
```

SEE ALSO

main

To see a list of general methods, see *General script extension methods* on page 75.

deepClone method

Returns a new business object containing the same type information and data content as this business object. This is a deep clone, the returned business object will share no mutable objects with this business object.

VBScript syntax

bo.deepClone()

JavaScript syntax

bo.deepClone();

Return value

bo, the clone.

Example

This example returns a clone of the po business object.

VBScript:

```
po.getBusinessObject().deepClone()
```

JavaScript:

```
po.getBusinessObject().deepClone();
```

See also

getBusinessObject, getElement, getElementSequence

To see a list of all the methods that are related to business objects, see *BusinessObject methods* on page 76.

fromXMLString method

Initializes the content of this business object from the specified XML string.

VBScript syntax

bo.fromXMLString *xml_string*

JavaScript syntax

bo.fromXMLString(*xml_string*);

Variable

xml_string, the input XML string corresponding to an instance of this type of business object.

Return value

Returns the business object as a string.

Run-time errors/exceptions

Throws IOException

Example

This example takes the po business object, in XML format, and returns it in a string format.

VBScript:

```
po.getBusinessObject().fromXMLString po_xml
```

JavaScript:

```
po.getBusinessObject().fromXMLString(po_xml);
```

See also

toXMLString

To see a list of all the methods that are related to business objects, see *BusinessObject methods* on page 76.

getAttr method

Retrieves an element's attribute value by name.

VBScript syntax
element.getAttr *attribute*

JavaScript syntax
element.getAttr(*attribute*);

Variable
attribute is the name of the attribute to be retrieved.

Return value
The attribute value as a string, or the empty string if that attribute does not have a specified or default value.

Run-time errors/exceptions
Throws InvalidQueryException

Example
This example returns the item_code attribute value.

VBScript:
```
po.getAttr "item_code"
```

JavaScript:
```
po.getAttr("item_code");
```

See also
setAttr, getAttrDefault

To see a list of all the methods that are related to elements, see *Element methods*

getAttrDefault method

Retrieves an attribute's default value by name.

VBScript syntax

element.getAttrDefault *attribute*

JavaScript syntax

element.getAttrDefault(*attribute*);

Variable

attribute is the name of the attribute.

Return value

The attribute default value as a string, or null if that attribute does not have a default value.

Run-time errors/exceptions

Throws InvalidQueryException

Example

This example returns the item_code attribute default value.

VBScript:

```
po.getAttrDefault "item_code"
```

JavaScript:

```
po.getAttrDefault("item_code");
```

See also

getAttr, setAttr

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

getBinding method

Retrieve the binding for the specified partner group. If the group is not valid or if the binding is not yet set, this method returns null.

VBScript Syntax

PartnerGroupContext.getBinding(*group_ref*)

JavaScript Syntax

PartnerGroupContext.getBinding(*group_ref*);

Parameter and variable

group_ref is the reference to the group. It is a string and is returned by getGroupRefs.

Return value

The binding for the specified group

Examples

These examples print the binding for the group "gi".

VBScript:

```
set pub_context = getPublicProcessContext()
set group_context = pub_context.getPartnerGroupContest()
println (group_context.getBinding("gi"))
```

JavaScript:

```
pub_context = getPublicProcessContext();
group_context = pub_context.getPartnerGroupContest();
println (group_context.getBinding("gi"));
```

See also

getPartnerGroupContext, getGroupRefs

To see a list of the related partner group context methods, see

getBusinessObject method

Gets the business object that "owns" this element instance. This element is a member of the returned business object's element content tree.

NOTE: This method works only on the root element of a business object's content tree. All other elements in the tree will return null.

VBScript syntax

root-element.getBusinessObject()

JavaScript syntax

root-element.getBusinessObject();

Variable

None

Return value

The business object that owns this element. If not the root element, returns null.

Example

This example gets the po business object.

VBScript:

```
po.getBusinessObject()
```

JavaScript:

```
po.getBusinessObject();
```

See also

getAttr, getElement, getElementSequence

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

getData method

Returns a string representation of the data contained in the specified field.

VBScript syntax

field.getData()
element.getData(*tag-path*)

JavaScript syntax

field.getData();
element.getData(*tag-path*);

Parameter and variables

field is a reference to a field to get data from.
tag-path is the path to the field or attribute, relative to *element*, which is a
reference to a business object, group, or field.

Return value

string representation of data

Run-time errors/exceptions

ElementTypeException indicates that the element types do not match. You
called the method on a group element instead of a field element.

InvalidQueryException indicates that the tag path you supplied is invalid (if
the index value was invalid, you would get IndexOutOfBoundsException
instead). For example, you can get this error if you typed a wrong name.

IndexOutOfBoundsException indicates that, when specifying an element in
a sequence, you provided an invalid index value, such as a number greater
than *length* - 1 for that element sequence.

Example

This VBScript code returns the value of the item_code field in the first
po_line group. The item variant holds the returned string.

```
dim item
item = po.getData("po_line[0]/item_code")
```

Or the code could be written as:

```
var item;
item = po.getData("po_line[0]/item_code");
```

SEE ALSO

createBO, copyIn, getElement, getElementAt, setData

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

GETDOCUMENT METHOD

Returns a reference to this business object via the W3C XML Document Object Model (DOM) interface.

VBSCRIPT SYNTAX
bo.getDocument()

JAVASCRIPT SYNTAX
bo.getDocument();

RETURN VALUE
org.w3c.dom.Document

EXAMPLE
This example returns a reference to the po business object.

VBScript:
```
po.getBusinessObject().getDocument()
```

JavaScript:
```
po.getBusinessObject().getDocument();
```

SEE ALSO
getRootElement, getTypeID

To see a list of all the methods that are related to business objects, see *BusinessObject methods* on page 76.

getElement method

Gets a reference to the group or field identified by the tag path string. Besides a non repeatable element, the tag path can specify an element in an element sequence by using an index number.

Although a business object is an element, a business object is referenced by its context business object variable, defined in the Partner Agreement Manager process that the script runs in. So you do not use this method on a business object, but only on the elements and sequences it contains.

VBScript syntax

element.getElement(*tag-path*)

JavaScript syntax

element.getElement(*tag-path*);

Parameter and variable

tag-path is the path to the group or field, relative to *element*, which is a reference to a business object, group, or field.

Return value

a reference to an element identified by *tag-path*

Run-time errors/exceptions

InvalidQueryException indicates that the tag path you supplied is invalid (if the index value was invalid, you would get IndexOutOfBoundsException instead). For example, you can get this error if you typed a wrong name.

IndexOutOfBoundsException indicates that, when specifying an element in a sequence, you provided an invalid index value, such as a number greater than *length* - 1 for that element sequence.

Example

In this example, the ship_to group is assigned to the shipping_address variable:

VBScript:

```
dim shipping_address
set shipping_address = po.getElement("po_line[0]/ship_to")
```

JavaScript:

```
var shipping_address;
shipping_address = po.getElement("po_line[0]/ship_to");
```

See also

createBO, getElementAt, getElementSequence

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

getElementAt method

Gets a reference to the group or field at the specified position in an element sequence.

VBScript syntax

element-sequence.getElementAt(*index*)

JavaScript syntax

element-sequence.getElementAt(*index*);

Parameter and variable

element-sequence is a reference returned by the getElementSequence method. *index* is the index value of the element (remember indexes start at zero [0]).

Return value

a reference to an element at this *index*

Run-time errors/exceptions

IndexOutOfBoundsException indicates that, when specifying an element in a sequence, you provided an invalid index value, such as a number greater than *length* - 1 for that element sequence.

Example

This example gets the element at the position i.

VBScript:

```
set single_line =
po.getElementSequence("po_line").getElementAt(i)
```

This statement is equivalent to the previous statement ("&i&" adds the value i to the tag path string):

```
set single_line = po.getElement("po_line["&i&"]")
```

JavaScript:

```
single_line = po.getElementSequence("po_line").getElementAt(i);
```

This statement is equivalent to the previous statement (`"+i+"` adds the value i to the tag path string):

```
single_line = po.getElement("po_line["+i+"]");
```

See also

createBO, getElement, getElementSequence

To see a list of all the methods that are related to element sequences, see *ElementSequence methods* on page 79.

getElementSequence method

Gets a reference to an element sequence. An element sequence is a collection of consecutive "sibling" elements of the same element type. The group or field must be specified as repeatable by its parent group. You need a reference to an element sequence before you can manipulate it with other methods.

VBScript syntax

element.getElementSequence(*tag-path*)

JavaScript syntax

element.getElementSequence(*tag-path*);

Parameter and variable

tag-path is the path to the element sequence, relative to *element*, which is a reference to a business object, group, or field.

Return value

a reference to an element sequence specified by *tag-path*

Run-time errors/exceptions

InvalidQueryException indicates that the tag path you supplied is invalid (if the index value was invalid, you would get IndexOutOfBoundsException instead). For example, you can get this error if you typed a wrong name.

IndexOutOfBoundsException indicates that, when specifying an element in a sequence, you provided an invalid index value, such as a number greater than *length* - 1 for that element sequence.

Example

This example assigns the po_line element sequence to the eseqLines variable (Purchase_Order defines po_line as repeatable).

VBScript:
```
set eseqLines = po.getElementSequence("po_line")
```

JavaScript:
```
eseqLines = po.getElementSequence("po_line");
```

See also

createBO, getElement, getElementAt

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

getGroupRefs method

Retrieve a list of references to the groups that are included in the public process. If there are no groups, an empty iterator is returned.

VBScript Syntax

PartnerGroupContext.getGroupRefs()

JavaScript Syntax

PartnerGroupContext.getGroupRefs();

Return value

Iterator of reference(s) to the partner group(s).

Examples

These examples print the list of references to the groups included in the public process.

VBScript:

```
set partner_group_context = getPartnerGroupContext()
set iter = partner_group_context.getGroupRefs()
has_next = iter.hasNext()
while has_next
  println (iter.next())
  has_next = iter.hasNext()
wend
```

JavaScript:

```
partner_group_context = getPartnerGroupContext();
iter = partner_group_context.getGroupRefs();
has_next = iter.hasNext();
while (has_next) {
  println (iter.next());
  has_next = iter.hasNext();
}
```

See also

getPartnerGroupContext, getBinding

To see a list of the related partner group context methods, see
PartnerGroupContext methods on page 80.

GETINPUTS METHOD

Retrieve the inputs to this public process node. If there are no inputs (i.e., this is the first node in the process) this method returns an empty iterator.

VBSCRIPT SYNTAX

PublicProcessContext.getInputs()

JAVASCRIPT SYNTAX

PublicProcessContext.getInputs();

RETURN VALUE

Iterator of PublicProcessNodeInput objects

EXAMPLES

These examples get and print the inputs to a public process node.

VBScript:

```
set iter = pub_context.getInputs()
has_next = iter.hasNext()
while has_next
  set message = iter.next()
  println (message.getSenderRef())
  println (message.getSenderNodetypeID())
rem Name of private process context var that points to the
rem business object contained by this msg
  println (message.getvariableName())
  has_next = iter.hasNext()
wend
```

JavaScript:

```
iter = pub_context.getInputs();
has_next = iter.hasNext();
while (has_next) {
  message = iter.next();
  println (message.getSenderRef());
  println (message.getSenderNodetypeID());
  /* Name of private process context var that points to the
     business object contained by this msg) */
```

```
    println (message.getvariableName());
    has_next = iter.hasNext();
}
```

See also

getPublicProcessContext

To see a list of all the related public process context methods, see
PublicProcessContext methods on page 80.

getLoopID method

(private process)

Retrieve the private process loop ID. If the current node is not inside a loop, this method returns a string of length 0. This method can be used to generate a unique ID.

VBScript syntax

PrivateProcessContext.getLoopID()

JavaScript Syntax

PrivateProcessContext.getLoopID();

Return value

The loop ID, or a string of length 0 if the current node isn't inside a loop.

Examples

These examples get the Loop ID.

VBScript:

```
set priv_context = getPrivateProcessContext()
println (priv_context.getLoopID())
```

JavaScript:

```
priv_context = getPrivateProcessContext();
println (priv_context.getLoopID());
```

See also

getPrivateProcessContext, getNodeTypeID (private process)

To see a list of all the related private process context methods, see *PrivateProcessContext methods* on page 81.

getLoopID method

(public process)

Retrieve the public process loop ID. If the current node is not inside a loop, will return a string of length 0.

VBScript syntax
PublicProcessContext.getLoopID()

JavaScript syntax
PublicProcessContext.getLoopID();

Return value
The String form of the loop ID

Examples
These examples get the loop ID.

VBScript:
```
set pub_context = getPublicProcessContext()
println (pub_context.getLoopID())
```

JavaScript:
```
pub_context = getPublicProcessContext();
println (pub_context.getLoopID());
```

See also
getPublicProcessContext, getNodeTypeID (public process)

To see a list of all the related public process context methods, see *PublicProcessContext methods* on page 80.

getNodeTypeID method

(private process)

Retrieve the type ID of the private process node.

VBScript syntax

PrivateProcessContext.getNodeTypeID()

JavaScript syntax

PrivateProcessContext.getNodeTypeID();

Return value

The ID of the private process type's node.

Examples

These examples get the node type ID.

VBScript:

```
set priv_context = getPrivateProcessContext()
println (priv_context.getNodeTypeID())
```

JavaScript:

```
priv_context = getPrivateProcessContext();
println (priv_context.getNodeTypeID());
```

See also

getPrivateProcessContext, getLoopID

To see a list of all the related private process context methods, see
PrivateProcessContext methods on page 81.

getNodeTypeID method

(public process)

Retrieve the type ID of the public process node.

VBScript syntax

PublicProcessContext.getNodeTypeID()

JavaScript syntax

PublicProcessContext.getNodeTypeID();

Return value

The String form of the public process type node ID

Examples

These examples get the node type ID.

VBScript:

```
set pub_context = getPublicProcessContext()
println (pub_context.getNodeTypeID())
```

JavaScript:

```
pub_context = getPublicProcessContext();
println (pub_context.getNodeTypeID());
```

See also

getPublicProcessContext, getLoopID (public process)

To see a list of all the related public process context methods, see
PublicProcessContext methods on page 80.

getPartnerGroupContext method

Retrieve the context object that can be used to access and manipulate partner group information in the context of this public process.

VBScript syntax

*PublicProcessContext.*getPartnerGroupContext()

JavaScript syntax

*PublicProcessContext.*getPartnerGroupContext();

Return value:

The partner group context

Examples

These examples get the partner group context.

VBScript:
```
set group_context = pub_context.getPartnerGroupContext()
set iter = group_context.getGroupRefs
has_next = iter.has_next
while has_next
  set group_ref = iter.next()
  println (group_ref)
  println (group_context.getBinding(group_ref))
  has_next =iter.hasNext()
wend
```

JavaScript:
```
group_context = pub_context.getPartnerGroupContext();
set iter = group_context.getGroupRefs();
has_next = iter.has_next();
while (has_next) {
  group_ref = iter.next();
  println (group_ref);
  println (group_context.getBinding(group_ref));
  has_next = iter.hasNext();
}
```

getPublicProcessContext getPrivateProcessContext

To see a list of all the related public process context methods, see *PublicProcessContext methods* on page 80.

To see a list of the related partner group context methods, see *PartnerGroupContext methods* on page 80.

getPath method

(private process)

Retrieve the currently selected path. If no path has been selected using the setPath() call, this method returns null.

VBScript syntax

PrivateProcessContext.getPath()

JavaScript syntax

PrivateProcessContext.getPath();

Return value

The currently selected path.

Examples

These examples get the currently selected path.

VBScript:

```
set priv_context = getPrivateProcessContext()
println (priv_context.getPath())
```

JavaScript:

```
priv_context = getPrivateProcessContext();
println (priv_context.getPath());
```

See also

getPathNames, setPath (private process)

To see a list of all the related private process context methods, see *PrivateProcessContext methods* on page 81.

getPathNames method

(private process)

Retrieve the list of valid paths that can be taken. If this node is not an XOR-SPLIT or a WHILE node (or step), this method returns an empty iterator.

VBScript syntax

PrivateProcessContext.getPathNames()

JavaScript syntax

PrivateProcessContext.getPathNames();

Return value

Iterator of String path names. If this node is not a branch, the iterator returned is empty.

Examples

These examples print the valid paths.

VBScript:

```
set priv_context = getPrivateProcessContext()
set iter = priv_context.getPathNames()
has_next = iter.hasNext()
while has_next
  println (iter.next())
  has_next = iter.hasNext()
wend
```

JavaScript:

```
priv_context = getPrivateProcessContext();
iter = priv_context.getPathNames();
has_next = iter.hasNext();
while (has_next) {
  println (iter.next());
  has_next = iter.hasNext();
}
```

getPrivateProcessContext, setPath (private process), getPath

To see a list of all the related private process context methods, see *PrivateProcessContext methods* on page 81.

getPrivateProcessContext method

This is the private process context object that is exposed to private process actions. This object contains information about the private process in which the action is executing.

VBScript Syntax

getPrivateProcessContext ()

JavaScript Syntax

getPrivateProcessContext ();

Return value

The private process context object.

Examples

These examples get the private process context.

VBScript:

```
set priv_context = getPrivateProcessContext()
```

JavaScript:

```
priv_context = getPrivateProcessContext();
```

See also

getPublicProcessContext getPartnerGroupContext

To see a list of general methods, see *General script extension methods* on page 75.

getProcessRef method

(private process)

Retrieve the reference to the private process. The reference is the string form of the private process ID.

VBScript syntax

PrivateProcessContext.getProcessRef( )

JavaScript syntax

PrivateProcessContext.getProcessRef( );

Return value

The reference to the private process

Examples

These examples print the process ref for the private process context.

VBScript:
```
set priv_context = getPrivateProcessContext()
println (priv_context.getProcessRef)
```

JavaScript:
```
priv_context = getPrivateProcessContext();
println (priv_context.getProcessRef());
```

See also

getPrivateProcessContext, getProcessTypeRef

To see a list of all the related private process context methods, see *PrivateProcessContext methods* on page 81.

getProcessRef method

(public process)

Retrieve reference to the public process. The reference is the string form of the public process ID.

This can be used to create the PublicProcessRef object used in the External API. See the *Partner Agreement Manager External API Guide* for more information on the External API.

VBScript syntax

PublicProcessContext.getProcessRef()

JavaScript syntax

PublicProcessContext.getProcessRef();

Return value

The reference to the public process

Examples

These examples print the public process ID.

VBScript:

```
set pub_context = getPublicProcessContext()
println(pub_context.getProcessRef())
```

JavaScript:

```
pub_context = getPublicProcessContext();
println(pub_context.getProcessRef());
```

See also

getPublicProcessContext, getProcessTypeRef (public process)

To see a list of all the related public process context methods, see *PublicProcessContext methods* on page 80.

getProcessTypeRef method

(private process)

Retrieve the reference to the private process type. The reference is the string form of the private process type ID.

VBScript syntax

PrivateProcessContext.getProcessTypeRef()

JavaScript syntax

PrivateProcessContext.getProcessTypeRef();

Return value

The reference to the private process type

Examples

These examples print the process type ref.

VBScript:
```
set priv_context = getPrivateProcessContext()
println (priv_context.getProcessTypeRef)
```

JavaScript:
```
priv_context = getPrivateProcessContext();
println (priv_context.getProcessTypeRef());
```

See also

getPrivateProcessContext, getProcessRef

To see a list of all the related private process context methods, see *PrivateProcessContext methods* on page 81.

getProcessTypeRef method

(public process)

Retrieve the reference to the public process type. The reference is the string form of the public process type ID.

VBScript syntax

PublicProcessContext.getProcessTypeRef()

JavaScript syntax

PublicProcessContext.getProcessTypeRef();

Return value

The reference to the public process type

Examples

These examples get the process type ref for a public process.

VBScript:

```
set pub_context = getPublicProcessContext()
println (pub_context.getProcessTypeRef())
```

JavaScript:

```
pub_context = getPublicProcessContext();
println (pub_context.getProcessTypeRef ());
```

See also

getPublicProcessContext, getProcessRef (public process)

To see a list of all the related public process context methods, see *PublicProcessContext methods* on page 80.

To see a list of all the private process context methods, see *PrivateProcessContext methods* on page 81.

getPublicProcessContext method

Retrieves the public process context object that is exposed to private process actions. This object contains information about the public process that activated the private process in which the action is executing.

VBScript syntax

getPublicProcessContext()

JavaScript syntax

getPublicProcessContext();

Return value

The reference to the public process.

Examples

These examples get the public process context.

VBScript:

```
set pub_context = getPublicProcessContext()
```

JavaScript:

```
pub_context = getPublicProcessContext();
```

See also

getPrivateProcessContext getPartnerGroupContext

To see a list of general methods, see *General script extension methods* on page 75.

To see a list of all the public process context methods, see *PublicProcessContext methods* on page 80.

To see a list of all the private process context methods, see *PrivateProcessContext methods* on page 81.

getRootElement method

Gets the root element from this business object's content.

VBScript syntax

bo.getRootElement()

JavaScript syntax

bo.getRootElement();

Return value

element, the root element.

Example

The following example returns the root element for the po business object:

VBScript:
```
po.getBusinessObject().getRootElement()
```

JavaScript:
```
po.getBusinessObject().getRootElement();
```

See also

getDocument, getTypeID

To see a list of all the methods that are related to business objects, see
BusinessObject methods on page 76.

getSenderNodeTypeID method

Retrieve the ID of the sending node.

VBScript syntax

PublicProcessNodeInput.getSenderNodeTypeID()

JavaScript syntax

PublicProcessNodeInput.getSenderNodeTypeID();

Return value

The String form of the ID of the sending node

Examples

These examples retrieve the ID of the sending node.

VBScript:

```
set iter = pub_context.getInputs()
has_next = iter.hasNext()
while has_next
  set message = iter.next()
  println (message.getSenderRef())
  println (message.getSenderNodetypeID())
rem Name of private process context var that points to
rem the business object contained by this msg
  println (message.getvariableName())
  has_next = iter.hasNext()
wend
```

JavaScript:

```
iter = pub_context.getInputs();
has_next = iter.hasNext();
while (has_next) {
  message = iter.next();
  println (message.getSenderRef());
  println (message.getSenderNodetypeID());
  /* Name of private process context var that points to BO
     contained by this msg */
  println (message.getvariableName());
  has_next = iter.hasNext();
}
```

See also

To see a list of all the related public process node input methods, see *PublicProcessNodeInput methods* on page 81.

getSenderRef method

Retrieve the reference to the sending partner. The reference is String form of the Partner ID of the sending partner.

VBScript syntax

PublicProcessNodeInput.getSenderRef()

JavaScript syntax

PublicProcessNodeInput.getSenderRef();

Return value

The reference to the sending partner

Examples

These examples get the sender ref for each input.

VBScript:

```
set iter = pub_context.getInputs()
has_next = iter.hasNext()
while has_next
  set message = iter.next()
  println (message.getSenderRef())
  println (message.getSenderNodetypeID())
rem Name of private process context var that points to
rem the business object contained by this msg
  println (message.getvariableName())
  has_next = iter.hasNext()
wend
```

JavaScript:

```
iter = pub_context.getInputs();
has_next = iter.hasNext();
while (has_next) {
  message = iter.next();
  println (message.getSenderRef());
  println (message.getSenderNodetypeID());
  /* Name of private process context var that points to BO
     contained by this msg) */
```

```
    println (message.getvariableName());
    has_next = iter.hasNext();
}
```

See also

To see a list of all the related public process node input methods, see
PublicProcessNodeInput methods on page 81.

getTagName method

Gets the name of this element or element sequence. This can also be thought of as the element type name.

VBScript syntax

element.getTagName()
element-sequence.getTagName()

JavaScript syntax

element.getTagName();
element-sequence.getTagName();

Variable

element is a reference to a business object, or group, or field contained by a business object.
element-sequence is a reference returned by the getElementSequence method.

Return value

String value of the tag name (type name) for this element or element sequence.

Example

This example prints "Purchase_Order".

VBScript:
```
println(po.getTagName())
```

JavaScript:
```
println(po.getTagName());
```

This example prints "po_line".

VBScript:
```
println(po.getElementSequence("po_line").getTagName())
```

JavaScript:
```
println(po.getElementSequence("po_line").getTagName());
```

The following subroutine takes a field and prints its name followed by its data value.

VBScript:
```
sub printDataValue(el)
  println(el.getTagName()&" : "&el.getData())
end sub
```

JavaScript:
```
function printDataValue(el) {
  println(el.getTagName() +" : "+el.getData());
}
```

SEE ALSO

createBO, getElement, getElementAt

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

To see a list of all the methods that are related to element sequences, see *ElementSequence methods* on page 79.

getTypeID method

Gets the type identifier from this business object's content.

VBScript syntax
bo.getTypeID()

JavaScript syntax
bo.getTypeID();

Return value
The type identifier for this business object.

Example
The following example returns the type ID for the po business object:

VBScript:
```
po.getBusinessObject().getTypeID()
```

JavaScript:
```
po.getBusinessObject().getTypeID();
```

See also
getDocument, getRootElement

To see a list of all the methods that are related to business objects, see
BusinessObject methods on page 76.

getVar method

Gets the value of a context variant.

VBScript syntax

getVar(*context-variant*)

JavaScript syntax

getVar(*context-variant*);

Parameter

context-variant is a type of context variable defined in the Partner Agreement Manager process that the script runs in.

Return value

string

Examples

This example uses a context variant to determine which path to follow. It increments the value of the context variant until it is greater than or equal to 10, at which point the Main path is taken instead of the Loop path.

VBScript:

```
sub main
  set priv_context = getPrivateProcessContext()
  counter = getVar("loop_counter")
  if counter >= 10 then
      priv_context.setPath("Main")
  else
      counter = counter + 1
      setVar "loop_counter", counter
      priv_context.setPath("Loop")
  end if
end sub
```

Note that the string counter is automatically converted to an integer, as specified in the VBScript language. However, if you want to compare two context variants, it is recommended that you use CInt or CDbl (for example, "9" > "10" but cint ("9") < cint("10")).

JavaScript:

```
var counter;
priv_context = getPrivateProcessContext();
counter = getVar("loop_counter");
if (counter >= 10) {
  priv_context.setPath("Main");
} else {
  priv_context.setPath("Loop");
}
```

SEE ALSO

setVar, main

To see a list of general methods, see *General script extension methods* on page 75.

getVariableName method

Retrieve the name of the input variable containing this input. The actual business object is contained in the variable whose name is returned by this method.

VBScript syntax

PublicProcessNodeInput.getVariableName()

JavaScript syntax

PublicProcessNodeInput.getVariableName();

Return value

The input variable name

Examples

These examples get the variable name for each input to the public process.

VBScript:

```
set iter = pub_context.getInputs()
has_next = iter.hasNext()
while has_next
  set message = iter.next()
  println (message.getSenderRef())
  println (message.getSenderNodetypeID())
rem Name of private process context var that points to
rem the business object contained by this msg
  println (message.getVariableName())
  has_next = iter.hasNext()
wend
```

JavaScript:

```
iter = pub_context.getInputs();
has_next = iter.hasNext();
while (has_next) {
  message = iter.next();
  println (message.getSenderRef());
  println (message.getSenderNodetypeID());
  /* Name of private process context var that points to BO
     contained by this msg) */
```

```
    println (message.getVariableName());
    has_next = iter.hasNext();
}
```

See also

getInputs

To see a list of all the related public process node input methods, see *PublicProcessNodeInput methods* on page 81.

hasData method

Checks whether an element or element sequence contains data:

- If the element is a field, the method returns true if the field contains data (is not null).
- If the element is a group, the method returns true if any subordinate field contains data.
- For an element sequence, the method returns true if any element in the sequence contains data.

Note: An empty string ("") as a data value is equivalent to null. A string containing one or more spaces (" ") is not null, and is considered to be data.

VBScript syntax

element.hasData( )
element-sequence.hasData( )

JavaScript syntax

element.hasData( );
element-sequence.hasData( );

Variable

element is a reference to a business object, or group or field contained by a business object.
element-sequence is a reference returned by the getElementSequence method.

Return value

Boolean; true if this *element* or *element-sequence* contains data in any field, false if it contains no data

Examples

The following example checks if there are any design_drawing elements in the po business object. Note that the underscore (_) means the statement was carried to the next line.

VBScript:

```
if (po.getElementSequence("design_drawing").hasData() = False) then
  println("There is no design drawing associated with this PO " & _
      po.getData("po_number"))
end if
```

JavaScript:

```
if (po.getElementSequence("design_drawing").hasData() == False) {
  println("There is no design drawing associated with this PO " +
      po.getData("po_number"));
}
```

In this example, if the summary_info field has data in it but it is not valid, it prints an error message for the user:

VBScript:

```
dim summary
summary = po.getElement("summary_info")
if (summary.hasData()) then
  if (summary.isValid() = False) then
      println("All mandatory fields of summary must be filled in")
  end if
end if
```

JavaScript:

```
if (po.getElement("summary_info").hasData()) {
  if (po.getElement("summary_info").isValid() == False) {
      println("All mandatory fields of summary " +
              "must be filled in");
  }
}
```

SEE ALSO

createBO, getElement, getElementAt, getElementSequence

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

To see a list of all the methods that are related to element sequences, see *ElementSequence methods* on page 79.

isBONull method

Determines if a business object has already been instantiated (so you don't have to call the createBO method).

VBScript syntax

isBONull(*bo*)

JavaScript syntax

isBONull(*bo*);

Parameter

bo is a context business object variable defined in the Partner Agreement Manager process that the script runs in.

Return value

Boolean; true if this business object has not been created, false if it has

Example

This example ensures that you do not create the po object twice:

VBScript:

```
if (isBONull("po")) then
  createBO("po")
end if
```

JavaScript:

```
if (isBONull("po")) {
  createBO("po");
}
```

See also

createBO

To see a list of general methods, see *General script extension methods* on page 75.

isField method

Checks whether this element is a field or group. This is useful when you want to use code that can manipulate different business object types, for example.

VBScript syntax

element.isField()

JavaScript syntax

element.isField();

Variable

element is a reference to a business object, or group or field contained by a business object.

Return value

Boolean; true if the element is a field, false if the element is a group

Example

The following example supports two different business object types: one type has ship_to as a group with an address field within it, while the other type has ship_to as a field holding address data. This code is reusable for both cases.

VBScript:

```
function getShippingAddress(ship_to)
  if (ship_to.isField()) then
     getShippingAddress = ship_to.getData()
  else
     getShippingAddress = ship_to.getData("address")
  end if
end function
```

JavaScript:

```
function getShippingAddress(ship_to) {
  var result;
  if (ship_to.isField()) {
      result = ship_to.getData();
  } else {
      result = ship_to.getData("address");
  }
return (result);
}
```

SEE ALSO

createBO, getElement, getElementAt

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

isProductionProcess method

Determines if the public process is in production mode or test mode.

VBScript syntax

PublicProcessContext.isProductionProcess

JavaScript syntax

PublicProcessContext.isProductionProcess();

Return value:

True if the process is executing in Production mode, false if it is not.

Examples

These examples print whether the process is in Production mode.

VBScript:

```
set pub_context = getPublicProcessContext()
println(pub_context.isProductionProcess())
```

JavaScript:

```
pub_context = getPublicProcessContext();
println(pub_context.isProductionProcess());
```

See also

getPublicProcessContext

To see a list of all the related public process context methods, see

isValid method

Determines the validity of an element based on the definition of the business object type:

- If the element you're checking is a group, isValid determines whether all mandatory fields it contains have data; if a subordinate optional group contains data, it also checks whether all mandatory fields in the optional group have data. If a subordinate optional group does not have data in it, it is ignored when determining the validity of the element. For subordinate element sequences, each element is checked individually for validity.

- If the element is a field, isValid checks whether it contains data or not.

isValid does not consider whether the element you called isValid on was optional or mandatory when it determines validity. For fields, it only checks whether the field has data or not; for groups, isValid checks the elements it contains for validity based on whether it was optional or mandatory.

Note: An empty string ("") as a data value is equivalent to null. A string containing one or more spaces (" ") is not null, and is considered to be data.

VBScript syntax

element.isValid()

JavaScript syntax

element.isValid();

Variable

element is a reference to a business object, or group or field contained by a business object.

Return value

Boolean; true if *element* is a field containing data or is a group whose subordinate elements are valid (or are optional and contain no data), false if *element* is a null field or is a group with a subordinate element that requires data but is null

Examples

The following table shows the isValid return values when you supply a Purchase_Order business object with these *element* values:

Field or group supplied to isValid	Type	Value	isValid return
Purchase_Order	A group element; the root of the business object		false
po_number	Mandatory Single Field	123	true
po_date	Mandatory Single Field	null	false
supplier_id	Mandatory Single Field	null	false
po_line	Mandatory Repeatable Group		true
item_code	Mandatory Single Field	123	true
qty	Mandatory Single Field	123	true
supplier_item_code	Optional Single Field	null	false
expected_ship_date	Mandatory Single Field	9.9.99	true
rate	Optional Single Field	1.23	true
ship_to	Optional Single Group		false
address	Mandatory Single Field	null	false
attention	Optional Single Field	null	false
phone	Optional Single Field	null	false
preferred_carrier	Optional Single Field	null	false
summary_info	Mandatory Single Group		true
comments	Mandatory Single Field	abc	true
comment_by	Optional Single Field	null	false
design_drawing	Optional Repeatable Group		true
location	Optional Single Field	null	false
drawing	Mandatory Single Field	123	true

Following is an example of checking the validity of the top-level element, Purchase_Order, referred to by the context business object variable called po. It prints an error message if the po business object is not valid (_ is the line continuation character).

VBScript:

```
if (po.isValid() = False) then
  println("The Purchase_Order object has unfilled mandatory
elements " _
  & po.toString(true))
end if
```

JavaScript:

```
if (po.isValid() == False) {
  println("The Purchase_Order object has unfilled mandatory
elements " +
  & po.toString(true));
}
```

The po business object is valid in these circumstances:

- The mandatory fields po_number, po_date, and supplier_id have data (a string that is not null).
- There are one or more valid po_line group elements.
- The mandatory field in the mandatory group summary_info has data.
- If the optional repeatable group design_drawing has data, the mandatory field called drawing must have data. If there are multiple elements in the sequence, they must all be valid.
- All mandatory fields directly subordinate to the mandatory repeatable group po_line must have data. If the optional single group ship_to has data, then the mandatory field in ship_to must have data. If po_line is a sequence with multiple elements, they must all be valid.

If a po_line element in a sequence has been created, but has null data values, it makes the business object invalid. The following code removes an element in a sequence that has null data values.

VBScript:

```
if (po.getElement("po_line["&i&"]").hasData() = False) then
  po.getElementSequence("po_line").removeElementAt(i)
end if
```

Note that i is the index value and "&i&" puts the value of i in a tag path string.

JavaScript:

```
if (po.getElement("po_line["+i+"]").hasData() == False) {
  po.getElementSequence("po_line").removeElementAt(i);
}
```

Note that i is the index value and "+i+" puts the value of i in a tag path string.

The next example checks a single po_line element for validity.

VBScript:

```
set line1 = po.getElement("po_line[0]")
if (line1.isValid() = False) then
  ' If line element is invalid, detemine what required fields
  ' do not have data.
  if (line1.getElement("item_code").hasData() = False) then
     println("the item_code field requires data")
  end if
  if (line1.getElement("qty").hasData() = False) then
     println("the qty field requires data")
  end if
  if (line1.getElement("expected_ship_date").hasData() = False)
then
     println("the expected_ship_date field requires data")
  end if
  set shipto = line1.getElement("ship_to")
  if (shipto.hasData()) then
     ' If the optional group ship_to contains any data, then
the
     ' mandatory field address must contain data.
     if (shipto.isValid() = False) then
        println("the address field of the ship_to element
requires data")
     end if
  end if
end if
```

JavaScript:

```
  line1 = po.getElement("po_line[0]");
  if (line1.isValid() == False) {
     // if line element is invalid, determine what required
     // fields do not have data.
     if (line1.getElement("item_code").hasData() == False) {
```

```
        println("the item_code field requires data");
    }
    if (line1.getElement("qty").hasData() == False) {
        println("the qty field requires data");
    }
    if (line1.getElement("expected_ship_date").hasData() ==
False) {
        println("the expected_ship_date field requires data");
    }
    ship_to = line1.getElement("ship_to");
    if (ship_to.hasData()) {
        // if the optional group ship_to contains any data,
then the
        // mandatory field address must contain data.
        if(ship_to.isValid() == False) {
            println("the address field of the ship_to element
requires data")
        }
```

The po_line element is valid in these circumstances:

- The mandatory fields item_code, qty, and expected_ship_date must have data (a string that is not null).
- The optional fields supplier_item_code and rate may have data.
- If the optional, single group ship_to has data in any field, then the mandatory field called address must have data.

It is useful to use the isValid method in conjunction with the hasData method whenever operating on an optional element. In the example, the ship_to field is checked for data before it is checked for validity.

Remember that whether an element is optional or mandatory is determined by whether the parent group sets the element as optional or mandatory. This means that if you call isValid on the ship_to group, the mandatory field it contains must have data for it to be valid.

The following example checks the summary_info field for validity. The comments field is required to have data, while the comments_by field can optionally have data.

VBScript:

```
set summary = po.getElement("summary_info")
if (summary.isValid() = False) then
  println("The comments field is mandatory and needs to have
data.")
end if
```

JavaScript:

```
if (po.getElement("summary_info").hasData()) {
  if (po.getElement("summary_info").isValid() == False) {
    println("All mandatory fields of summary " +
            "must be filled in");
  }
}
```

The following example checks if the optional, repeatable design_drawing element has data.

VBScript:

```
set des_drawing = po.getElement("design_drawing[0]");
if (des_drawing.isValid() = False) then
  println("The drawing field is mandatory and needs to have
data.")
end if
```

JavaScript:

```
des_drawing = po.getElement("design_drawing[0]");
if (des_drawing.isValid() == False) {
  println("The drawing field is mandatory and needs to have
data.");
}
```

Note that the first line will return an error if the length of the sequence is zero (0). So you should check the length first.

SEE ALSO

createBO, hasData

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

length method

Returns the number of elements in this element sequence. This is useful for setting boundary values to loop through all the elements in an element sequence.

VBScript syntax

element-sequence.length()

JavaScript syntax

element-sequence.length();

Variable

element-sequence is a reference returned by the getElementSequence method.

Return value

integer specifying the number of elements in the sequence

Example

This example loops through all the po_line elements and prints the data in each po_line in string form.

VBScript:

```
dim lines
set lines = po.getElementSequence("po_lines")
nLines = lines.length()
for i=0 to nLines - 1
  println(lines.getElementAt(i).toString(true))
next
```

JavaScript:

```
var lines, nlines;
lines = po.getElementSequence("po_line");
nLines = lines.length();
for (i=0; i<= nLines - 1; i++) {
  println(lines.getElementAt(i).toString(true));
}
```

See also

createBO, getElementSequence

To see a list of all the methods that are related to element sequences, see *ElementSequence methods* on page 79.

MAIN METHOD

The script entry point. Private process script execution starts in the main method.

VBScript syntax

sub main
code
end sub

JavaScript syntax

function main () {
code
}

Example

VBScript:

```
sub main
  setVar "foo", "3"
end sub
```

JavaScript:

```
function main () {
  setVar ("foo", "3");
}
```

See also

createBO, setPath

To see a list of general methods, see *General script extension methods* on page 75.

newElement method

Adds a new element to the end of this sequence and returns a reference to the newly created element. The index of the new element will be *length* - 1.

VBScript syntax

element-sequence.newElement()

JavaScript syntax

element-sequence.newElement();

Variable

element-sequence is a reference returned by getElementSequence.

Return value

the new, empty *element*

Example

This example creates a new po_line element at the end of the sequence and copies into it the value of the element before it in the sequence.

VBScript:

```
set lines = po.getElementSequence("po_line")
nlines = lines.length()
last_line_index = nlines - 1
set new_line = lines.newElement()
if (last_line_index >= 0) then
  new_line.copyIn(lines.getElementAt(last_line_index))
end if
```

JavaScript:

```
lines = po.getElementSequence("po_line");
nlines = lines.length();
last_line_index = nlines - 1;
new_line = lines.newElement();
if (last_line_index >= 0) {
  new_line.copyIn(lines.getElementAt(last_line_index));
}
```

See also

createBO, getElementSequence

To see a list of all the methods that are related to element sequences, see

newElementAt method

Inserts a new element at the specified position in the sequence and returns a reference to the newly created element. Adds 1 to the index of the element currently in that position (if any) and any following elements, so they are "shifted to the right." Valid index values are 0 to *length*. If the index is *length*, an element is added to the end of the sequence.

VBScript syntax

element-sequence.newElementAt(*index*)

JavaScript syntax

element-sequence.newElementAt(*index*);

Variable

element-sequence is a reference returned by the getElementSequence method. *index* specifies the element in the sequence (remember indexes start at zero [0]).

Return value

the new, empty *element* in the sequence

Run-time errors/exceptions

IndexOutOfBoundsException indicates that, when specifying an element in a sequence, you provided an invalid index value, such as a number greater than *length* for that element sequence.

Example

This example adds a new po_line element at the beginning of the sequence.

VBScript:

```
set line = po.getElementSequence("po_line").newElementAt(0)
```

JavaScript:

```
line = po.getElementSequence("po_line").newElementAt(0);
```

See also

createBO, getElementSequence

To see a list of all the methods that are related to element sequences, see *ElementSequence methods* on page 79.

PRINTLN METHOD

Prints a message to the server's console, a log file, or both. This is useful for debugging.

If you run Partner Agreement Manager as a service, println output goes into a log file. The log file is called PAM.log and is created in the Partners\Partner*nnn* directory. If you run Partner Agreement Manager from a shortcut, println output is displayed on the console and in the log file.

VBScript syntax

println(*string*)

JavaScript syntax

println (*string*);

Variable

string is the string you want to print.

Return value

true

Examples

This example prints the description string, including the data values.

VBScript:

```
println(po.toString(True))
```

JavaScript:

```
println(po.toString(True));
```

This example prints "*tag-name* has *x* characters".

VBScript:

```
sub printStringLength(e)
  println(e.getTagName() & " has " & len(e.getData()) & "
characters")
end sub
```

JavaScript:

```
function printStringLength(e) {
  println(e.getTagName() + " has " + e.getData().length + "
characters");
}
```

See also

toString

To see a list of general methods, see *General script extension methods* on page 75.

removeAll method

Removes all elements in this sequence. Any data contained in any of the elements is deleted. The length of this element sequence becomes 0.

VBScript syntax

element-sequence.removeAll

JavaScript syntax

element-sequence.removeAll();

Variable

element-sequence is a reference returned by the getElementSequence method.

Return value

None

Example

The following example removes all the po_line elements from the po business object

VBScript:

```
po.getElementSequence("po_line").removeAll
```

JavaScript:

```
po.getElementSequence("po_line").removeAll;
```

See also

createBO, clearAll, clearData, getElementSequence, removeElementAt

To see a list of all the methods that are related to element sequences, see *ElementSequence methods* on page 79.

removeAttr method

Removes an attribute by name.

VBScript syntax
element.removeAttr *attribute*

JavaScript syntax
element.removeAttr(*attribute*);

Variable
attribute is the name of the attribute to remove.

Return value
None

Run-time errors/exceptions
Throws InvalidQueryException

Example
This example removes the item_code attribute from the po element.

VBScript:
```
po.removeAttr "item_code"
```

JavaScript:
```
po.removeAttr("item_code");
```

See also
removeAttrs, removeElement

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

removeAttrs method

Removes all attributes associated with the specified element, leaving the empty element in place.

VBScript syntax
element.removeAttrs()

JavaScript syntax
element.removeAttrs();

Return value
None

Example
This example removes all attributes from the po element.

VBScript:
```
po.removeAttrs()
```

JavaScript:
```
po.removeAttrs();
```

See also
removeAttr, removeElement

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

removeElement method

Removes the element specified by the tag path from the business object.

VBScript syntax

removeElement *tag-path*

JavaScript syntax

removeElement(*tag-path*);

Variable

tag-path to the element.

Return value

None

Run-time errors/exceptions

Throws InvalidQueryException—If the tag path is invalid or if it refers to an element sequence instead of an element;
IndexOutOfBoundsException—which indicates that, when specifying an element in a sequence, you provided an invalid index value, such as a number greater than length - 1 for that element sequence.

Example

This example removes ship_to element from the po business object.

VBScript:

```
po.removeElement("po_line[0]/ship_to")
```

JavaScript:

```
po.removeElement("po_line[0]/ship_to");
```

See also

removeAttr, removeAttrs

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

removeElementAt method

Removes the element at the specified position in this sequence, and its data. The indices of elements at greater index values are reduced by 1 (they are "shifted to the left"). After the call successfully completes, the length of the sequence is reduced by 1.

VBScript syntax

element-sequence.removeElementAt *index*

JavaScript syntax

element-sequence.removeElementAt(*index*);

Variable

element-sequence is a reference returned by the getElementSequence method. *index* specifies the element in the sequence (remember indexes start at zero [0]).

Return value

None

Run-time errors/exceptions

IndexOutOfBoundsException indicates that, when specifying an element in a sequence, you provided an invalid index value, such as a number greater than *length* - 1 for that element sequence.

Example

The following example loops through all po_line elements and removes all of the invalid ones. Note that removeElementAt will move the remaining elements one index down in the sequence. So you should perform this operation starting at the end of the sequence so your index value is always valid.

VBScript:

```
dim line
dim lines

set lines = po.getElementSequence("po_line")
for i = lines.length() - 1 to 0 step -1
  set line = lines.getElementAt(i)
```

```
  if not line.isValid() then
      lines.removeElementAt(i)
  end if
next
```

JavaScript:

```
var line;
var lines;

lines = po.getElementSequence("po_line");
for (i = lines.length() - 1; i>= 0; i--) {
   line = lines.getElementAt(i);
  if (!line.isValid()) {
      lines.removeElementAt(i);
  }
}
```

See also

createBO, clearAll, clearData, getElementSequence, removeAll

To see a list of all the methods that are related to element sequences, see
ElementSequence methods on page 79.

setAttr method

Adds a new attribute to an element. If an attribute with that name is already present in the element, its value is changed to be that of the value parameter. This value is a simple string; it is not parsed as it is being set.

VBScript syntax

element.setAttr *attribute, value*

JavaScript syntax

element.setAttr(*attribute, value*);

Variable

attribute is the name of the attribute to create or alter.
value is the attribute value to set in string format.

Return value

None

Run-time errors/exceptions

Throws InvalidQueryException, InvalidDataException

Example

This example sets the item_code attribute to the value of the string in item_value.

VBScript:

```
po.setAttr "item_code","item_value"
```

JavaScript:

```
po.setAttr("item_code","item_value");
```

See also

getAttr, getAttrDefault

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

setData method

Sets the data contained in this field element.

NOTE: An empty string ("") as a data value is equivalent to null. A string containing one or more spaces (" ") is not null, and is considered to be data.

VBScript syntax

field.setData *value*
element.setData *tag-path*, *value*

JavaScript syntax

field.setData(*value*);
element.setData(*tag-path*, *value*);

Parameters and variables

value is a string containing the data value that you want the field to have.
field is a reference to the field you want to set.
tag-path specifies the path to a field, relative to *element*, which is a reference to a business object, group, or field.

Return value

None

Run-time errors/exceptions

ElementTypeException indicates that the element types do not match. You called the method on a group element instead of a field element.

InvalidQueryException indicates that the tag path you supplied is invalid (if the index value was invalid, you would get IndexOutOfBoundsException instead). For example, you can get this error if you typed a wrong name.

IndexOutOfBoundsException indicates that, when specifying an element in a sequence, you provided an invalid index value, such as a number greater than *length* - 1 for that element sequence.

Example

This example sets the po_number field to the string 24567.

VBScript

```
po_num = "24567"
po.setData "po_number", po_num
```

This is equivalent to:

```
po_num = "24567"
po.getElement("po_number").setData po_num
```

JavaScript:

```
po_num = "24567";
po.setData ("po_number", po_num);
```

This is equivalent to:

```
po_num = "24567";
po.getElement("po_number").setData (po_num);
```

SEE ALSO

createBO, getElement, getElementAt

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

setPath method

Sets the private process path. Remember that public process paths are determined by output business objects only.

Important: This method is being deprecated in favor of the setPath method that uses the PrivateProcessContext, described on page 170.

To see a list of general methods, see *General script extension methods* on page 75.

setPath method

(private process)

Set the path to be taken when this node completes. If the specified path is not valid, this method returns false. Otherwise, this method returns true. This call is ignored if the node is not an XOR-SPLIT or a WHILE node.

Important: Use this method rather then the setPath described on page 169. That setPath method is being deprecated.

VBScript syntax
PrivateProcessContext.setPath(*path*)

JavaScript syntax
PrivateProcessContext.setPath(*path*);

Parameter
path is the name of the path to set. This is a string.

Return value
True if the path is valid. This method will not work in the script tester.

Examples
These examples set the path.

VBScript:
```
set priv_context = getPrivateProcessContext()
priv_context.setPath("Pathname")
```

JavaScript:
```
priv_context = getPrivateProcessContext();
priv_context.setPath("Pathname");
```

See also
getPrivateProcessContext, getPath

To see a list of all the related private process context methods, see *PrivateProcessContext methods* on page 81.

setVar method

Sets the value of a context variant.

VBScript syntax

setVar *context-variant*, *value*

Note: If a process has more than one parameter, can return a value, and your code uses the return value, you must use parentheses in the call.

JavaScript syntax

setVar (*context-variant*, *value*);

Parameter

value can be a variant, constant, or literal string. *context-variant* is a type of context variable defined in the Partner Agreement Manager Process window.

Return value

Boolean; true if the variable exists, false if it does not

Example

This example sets the order_id to 29.

VBScript:
```
sub main
  setVar "order_id", "29"
end sub
```

JavaScript:
```
setVar ("order_id", "29");
```

See also

getVar

To see a list of general methods, see *General script extension methods* on page 75.

toString method

Returns a string describing the validity and content of this element and any elements subordinate to it that contain data. Mandatory subordinate elements always appear in the description; optional subordinate elements without any data do not appear. This helps you see which elements must have data for this element to be valid. You can use the println method to display the value returned by toString.

VBScript syntax

element.toString(*include-data*)

JavaScript syntax

element.toString(*include-data*);

Parameter and variable

If the Boolean *include-data* is true, toString includes the data in the description; if false, it does not include the data values, which makes the description shorter.
element is a reference to a business object, or group or field contained by a business object.

Return value

string describing the element

Example

This example prints the description string, including the data values.

VBScript:

```
println(po.toString(True))
```

JavaScript:

```
println(po.toString(True));
```

The output might look like this (note that the top-level Purchase_Order element is not valid because the mandatory field po_date has no value):

```
<Purchase_Order valid="false">
<po_number valid="true">123</po_number>
<po_date valid="false"></po_date>
<supplier_id valid="true">99999</supplier_id>
```

```
<po_line valid="true">
<item_code valid="true">2222</item_code>
<qty valid="true">55</qty>
<expected_ship_date valid="true">9.9.99</expected_ship_date>
<summary_info valid="true">
<comments valid="true">first line item</comments>
</summary_infor>
</po_line>
</Purchase_Order>
```

This example loops through all the po_line elements and prints the data in each po_line in string form.

VBScript:

```
nLines = po.getElementSequence("po_line").length()
for i=0 to nLines - 1
  println(po.getElement("po_line["&i&"]").toString(true))
next
```

JavaScript:

```
nLines = po.getElementSequence("po_line").length();
for (i=0; i== nLines - 1; i++) {
  println(po.getElement("po_line["&i&"]").toString(true));
}
```

See also

println

To see a list of all the methods that are related to elements, see *Element methods* on page 77.

toXMLString method

Returns the content of this business object in a string. The returned string will contain an XML declaration and a DOCTYPE entity. The value of SYSTEM ID string used in the DOCTYPE may be either the PAM business object definition ID (BODefID) or an external ID.

VBScript syntax

bo.toXMLString *use_external_ID*

JavaScript syntax

bo.toXMLString(*use_external_ID*);

Variable

use_external_ID, if true, the external definition id, if available, should be used as the DOCTYPE SYSTEM ID. If false, if the PAM BODefID value should be used as the SYSTEM ID.

Return value

Returns the content of this business object in an XML string.

Example

This example takes the po business object, in a string format, and returns it in an XML string format.

VBScript:

```
po.getBusinessObject().toXMLString true
```

JavaScript:

```
po.getBusinessObject().toXMLString(true);
```

See also

fromXMLString

To see a list of all the methods that are related to business objects, see *BusinessObject methods* on page 76.

VALIDATE METHOD

Checks the validity of an element within this business object. You can either validate an immediate element's content or an entire element subtree.

You can also check whether elements that specify PCDATA in their content model contain non-empty text data. This is a higher standard than the 1.0 XML specification for PCDATA, which does not require one or more characters of text.

If the validity check passes, a null string is returned. If the validity check fails, the string contains the first validity error encountered.

VBScript syntax

bo.validate (*element, deep, require_nonempty_pcdata)*

JavaScript syntax

bo.validate (*element, deep, require_nonempty_pcdata);*

Variable

element, the element within the business object.
deep, set to true if the element's entire subtree should be checked for validity, false if only the element's immediate content should be checked.
require_nonempty_pcdata, set to true if elements declared to contain PCDATA must contain text of length greater than 0. Set to false if no check for PCDATA length should be performed.

Return value

Returns a string describing the validity error, or null if the business object validity check succeeds.

Examples

This example checks the po_line and its entire subtree to ensure that no elements are empty.

VBScript:
```
po.getBusinessObject().validate(po_line,true,true)
```

JavaScript:
```
po.getBusinessObject().validate(po_line,true,true);
```

This example checks the po_line (and not its subtree) to ensure that no elements are empty.

VBScript:

```
po.getBusinessObject().validate(po_line,false,true)
```

JavaScript:

```
po.getBusinessObject().validate(po_line,false,true);
```

This example checks the po_line for validity.

VBScript:

```
po.getBusinessObject().validate(po_line,false,false)
```

JavaScript:

```
po.getBusinessObject().validate(po_line,false,false);
```

SEE ALSO

isValid

To see a list of all the methods that are related to business objects, see *BusinessObject methods* on page 76.

A

NOTICES

This information was developed for products and services offered in the United States. IBM may not offer the products, services, or features discussed in this information in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this information. The furnishing of this information does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the information. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this information at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you. Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM United Kingdom Laboratories,
Mail Point 151,
Hursley Park,
Winchester,
Hampshire,
England
SO21 2JN.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

AIX
DB2
IBM
MQSeries
SupportPac
WebSphere

Pentium is a registered trademark of Intel Corporation in the United States and/or other countries.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and/or other countries.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, and service names may be trademarks or service marks of others.

GLOSSARY

action—a task performed as part of a private process. A private process action is the equivalent of a step in a public process. See the following terms in this glossary for more information about the action types you can include in a private process:

- approval action
- extension action
- mapping action
- notification action
- output object action
- script action
- subprocess action
- termination action
- timer action

See also *private process*.

adapter—the software bridge between Partner Agreement Manager processes and specific end-system and business-application interfaces. Adapters manage interactions between business applications and the Adapter Server. They allow private processes to interact with external business applications while a process is running, and they allow PAM to start public processes based on events that occur in external business applications. See also *adapter implementation*, *adapter instance*, *adapter type*.

adapter implementation—the implementation declaration for an adapter type. It specifies the name and location of the Java source file that defines the application logic used to communicate with a specific end system through that end system's interface. The application logic is specified in the form of properties. See also *adapter*, *adapter instance*, *adapter type*.

adapter instance—an instance of an adapter implementation. The adapter instance is used in a private process extension action and provides the specific values to be used for the properties declared in the adapter implementation. See also *adapter*, *adapter implementation*, *adapter type, extension action*.

adapter type—a definition that is stored in XML format and specifies the adapter's properties as well as the operations and events it supports. A single adapter type can have multiple implementations, and each implementation can have multiple instances. See also *adapter*, *adapter implementation*, *adapter instance*.

approval action—a private process action that you use to ask for a response from a user before letting the process continue to run. You can use an approval action, for example, to ask for an OK when a purchase order exceeds a predetermined amount. See also *private process*.

business object—a message transmitted as part of a public process. Business objects take the form of purchase orders, acknowledgments, requests for clarification, and so on. See also *business object type*.

business object type—a definition that determines the types of information a message can contain. It has three properties: the top-level element in its element definition set, its key field, and whether instances of it return audit information for non-repudiation purposes. The name of the business object type is the name of the element you select as its top-level element. See also *business object*, *element definition set*, *non-repudiation*.

business object variable—one of the two types of variables used in Partner Agreement Manager to store information within a process. Business object variables create an instance of a business object type. They can be used to store, for example, the outputs from extension actions, the inputs for map actions, or the inputs and outputs for subprocesses. See also *business object*, *business object type*, *extension action*, *variant variable*.

CA—see *certificate authority*.

certificate—a security document that binds a public encryption key to an entity (an individual or organization) known as the principal. The security document (a digital certificate) is signed by another entity known as the issuer. A digital certificate for which both the principal and issuer are the same entity is known as a self-signed certificate. A certificate for which the principal and issuer are different entities is issued by a certificate authority (CA) like VeriSign and is known as a CA-issued (or third-party-signed) certificate. Partner Agreement Manager supports both self-signed and CA-issued certificates. PAM also supports the binding of certificates to be used for signature authentication, message encryption, and SSL authentication for channels other than Partner Agreement Manager. See also *certificate authority*, *SSL.*

certificate authority—a trusted third-party organization or company that issues digital certificates used to create digital signatures and public-private key pairs. The role of the certificate authority, or CA, is to authenticate the entities (individuals or organizations) involved in electronic transactions. CAs are a critical component in data security and electronic commerce because they guarantee that the two parties exchanging information are really who they claim to be. See also *certificate.*

channel—a communications mechanism that encapsulates all the processing information needed to send messages to a partner's system, as well as to translate data received from a partner into Partner Agreement Manager messages. PAM provides channels for RosettaNet, EDI, cXML, and other systems and protocols. See also *message.*

digital certificate—see *certificate.*

DTD—Document Type Definition. A type of file associated with SGML and XML documents that defines how the formatting tags should be interpreted by the application presenting the document. In Partner Agreement Manager, a DTD file contains the complete description of a business object type's element definition set. See also *business object*, *business object type*, *element definition set.*

element definition set—a collection of data fields (or elements) or groups of data fields that defines the structure and meaning of a business object type. See also *business object*, *business object type*.

encryption certificate—see *certificate.*

event—a piece of information that comes into Partner Agreement Manager as a message from another source (an enterprise system or business application, for example) and triggers a public process. See also *message.*

event push—a method that uses the HTTP POST mechanism to push events into Partner Agreement Manager as a way to trigger processes. A port on the Process Server is set to listen for events in the form of HTTP POST messages. When a message is detected, PAM uses the information in the message to generate an event. See also *event*.

extended enterprise—a business model under which companies that work together as partners function as efficiently as a single organization through the implementation of automated communication technologies.

extension action—a private process action that communicates via an adapter with an external application that is registered with Partner Agreement Manager. You can use an extension action, for example, to launch a spreadsheet application, perform calculations, and update the enterprise system, or to get information from an enterprise system or listen for an event in the enterprise system. See also *adapter*, *private process*.

LDAP—Lightweight Directory Access Protocol. LDAP provides a standard method for accessing information from a central directory. After user authentication is set up in the LDAP directory, applications that use the LDAP protocol can retrieve the information from that directory. An authenticated user can log in to any application that supports the LDAP protocol with the same user name and password.

linked certificate—see *certificate*.

map—a Java Script or VBScript that inserts data into fields in an output business object type generated by a private process. The map specifies which fields in the output business object type receive data, and it identifies the information source.

map method—a reusable logical block of code that inserts data into a particular type of element or element sequence in a business object type. Within a map method, you can write the expressions that map individual input and output fields in the sequence. Or you can create a submap and drag input fields to output fields and have Partner Agreement Manager create the appropriate mapping expressions. See also *map*, *submap*.

mapping action—a private process action that you use to call a map. The map specifies the fields in a business object type that will receive data extracted from another source. You use a mapping action when you want to extract data from one business object type and insert it in a different business object type. For example, you use a mapping action to transform a purchase order generated by your inventory system into a sales order in a format that your partner expects. See also *map*, *private process*.

message—a structured communication used to pass information and control to another partner in a public process. The action in the process passes to the partner who receives the message. The content of a message is determined by its business object type. A message can be transmitted via synchronous or asynchronous methods, as determined by its communication service type. See *business object type*.

non-repudiation—a business object security feature that authenticates instances of a business object type and maintains an audit record to verify that they were received by the intended recipient. For business object instances that you receive, Partner Agreement Manager authenticates each instance and maintains an audit record to verify that the instance actually originated with the stated originator. If you disable auditing for a business object type, non-repudiation support is disabled for all messages that contain instances of that business object type.

notification action—a private process action that you use to send an e-mail, fax, or pager message to addressees that you specify. You use a notification action to inform someone inside or outside your organization that an event has occurred. For example, you can use a notification action to alert the order entry department when a purchase order arrives from a customer. See also *private process*.

output object action—a private process action that you use to bind a business object to the expected output object and path in a public process. You use an output object action at the point in a private process when you are ready to send a business object to the associated public process. This is typically the last action in the private process. See also *private process*.

partner group—a group of partners that perform the same role in a process at different times. Instead of duplicating a public process and substituting a different partner name, you can set up a partner group for the public process and then designate a specific partner as the participant when you start an instance of the process. For example, you might design a generic purchasing process that works equally well with any of your suppliers and then designate the appropriate partner when you start the process.

partner profile—information that identifies an organization, specifies a contact person in that organization, lists the communication services the organization supports, and defines the organization's security profile. When partners agree to participate in a public process, they must exchange profile information as a way to ensure authenticity before they can proceed.

PIP—Partner Interface Process. RosettaNet PIPs are specialized system-to-system XML-based dialogs that define business processes between supply-chain partners and provide models and documents for the implementation of e-commerce standards. Each PIP includes a technical specification based on the RosettaNet Implementation Framework (RNIF), a message guideline document with a PIP-specific version of the business dictionary, and an XML message guideline document. See also *RosettaNet*.

post method—the last block of code that is executed when a mapping action runs. Its only parameter is the output business object. You use the post method when you need to perform post-processing on the output business object. For example, you might use the post method to set the value of a summary field based on the number of line items in the output business object, or to examine a range of dates in a repeated group, extract the most recent date, and post that date in a header field. See also *mapping action*, *pre method*.

pre method—the first block of code that is executed when a mapping action runs. The pre method's parameters are the map inputs. You use the pre method to access a map's inputs and set global variables based on their content. See also *mapping action*, *post method*.

private process—a task or set of tasks that business partners participating in a public process perform at points where they need to take action internally. Partners participating in a public process must implement a private process for each public process step that they own. A private process begins with input from the public process and ends with output that feeds back into the public process. The input can be the receipt of a business object from a partner, or it can be a triggering event from an internal system. The output is the business object that transfers control back to the public process. See also *action*, *process*, *public process*.

private process action—see *action*.

process—the flow of actions and the exchange of business information between partners in an extended enterprise. A process operates on two levels, public and private. See *extended enterprise*, *private process*, *public process*.

public process—the step-by-step flow of messages, events, and actions between two or more business partners. Public processes are set up by agreement between partners, and each step in a public process has a private process associated with it. A public process is developed by one partner, and all the partners who participate in it must review and approve it before it can be implemented. The partner who designs a public process is its owner. See also *private process*, *process*.

RosettaNet—a consortium of major information technology, electronic components, and semiconductor manufacturing companies that is working to create and implement industry-wide, open e-business process standards. See also *PIP*.

script action—a private process action that consists of a script written in VBScript or JavaScript and is designed to manipulate information or set up conditional actions based on input. You use a script to establish decision-making criteria for branches or loops, to set variables, or to calculate values that are used elsewhere in the private process. See also *private process.*

security certificate—see *certificate.*

self-signed certificate—see *certificate.*

signature certificate—see *certificate.*

SSL—Secure Sockets Layer. The SSL protocol is a security protocol that provides for communications privacy and reliability over the Internet. The protocol allows client/server applications to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery.

submap—a secondary level map that is called by a map method to insert data into an output element other than the top-level element. See *map*, *map method.*

subprocess action—a private process action you use to call an existing public process. You can call any public process in which your organization owns the first partner action. For example, you can use a subprocess to get a quote approved by a third-party supplier before responding to a customer. See also *private process.*

termination action—a private process action that you use to stop a process at a predetermined point for a reason that you specify. You can use a termination action to deal with errors in data that might prevent a process from completing successfully. For example, you might want to stop a process in cases where an enterprise system passes incomplete or corrupted information to it. See also *private process.*

third-party-signed certificate—another name for a CA-issued certificate. See *certificate.*

timer action—a private process action that you use to insert a pause. You can use a timer action to specify the period of time you want to elapse before the next action in the process starts. See also *private process.*

variant variable—single field variables. Variant variables store text strings—the type of information contained in a single field element. You can use variant variables to store the input for actions, to set flags (such as the time-out flag for an approval action), to move information within scripts, or to store the results of an approval action. See also *business object variable.*

INDEX