PEX: Developing Productions with Java and .NET

Version 2020.3
2021-02-04

InterSystems Corporation 1 Memorial Drive Cambridge MA 02142 www.intersystems.com

PEX: Developing Productions with Java and .NET
InterSystems IRIS Data Platform Version 2020.3 2021-02-04

For Support questions about any InterSystems products, contact:

InterSystems Worldwide Response Center (WRC)
Tel: +1-617-621-0700
Tel: +44 (0) 844 854 2917
Email: support@InterSystems.com

Table of Contents

1

Introduction

The Production EXtension (PEX) framework provides you with a choice of implementation languages when you are developing interoperability productions. Interoperability productions enable you to integrate systems with different message formats and communication protocols. If you are not familiar with interoperability productions, see "Introduction to Productions."

In this release, PEX supports Java and the .NET languages. PEX provides flexible connections between business services, processes, and operations that are implemented in PEX-supported languages or in InterSystems ObjectScript. In addition, you can use PEX to develop, inbound and outbound adapters. The PEX framework allows you to create an entire production in Java or .NET or to create a production that has a mix of Java, .NET, or ObjectScript components. Once integrated, the production components written in Java and .NET are called at runtime and use the PEX framework to send messages to other components in the production.

Some reasons to use the PEX framework include:

- Developing a new production and you choose to implement it entirely in Java or .NET.
- Creating new adapters for protocols using available Java or .NET libraries.
- Using available Java or .NET libraries to perform complex analysis or calculations.
- Implementing persistent messaging and long-running business processes without using ObjectScript in your application's technology stack.

You can create the following production components using the PEX language support:

- Inbound Adapter
- Outbound Adapter
- Business Service
- Business Process
- Business Operation

In addition to developing the code for the production components, you will typically define, configure, and monitor the production using the Management Portal. For example, you'll use the Management Portal to create the production, configure the business services, processes, and operations, start and stop the production, and trace persistent messages running through the production.

Note: PEX replaces the Java Business Hosts feature. For information on migrating existing Java Business Hosts productions to use PEX, see the community article Migrate from Java Business Host to PEX. Java Business Hosts was deprecated in release 2019.3 and has been removed from this release.

1.1 Basic Workflow

Implementing Java and .NET components into an interoperability production consists of three basic steps:

1. In you favorite IDE, write the business host or adapter in Java or .NET and compile the code.
2. In the Management Portal, add a built-in PEX component that points to the new Java or .NET class. To integrate a custom Java or .NET business host into a production, use the Management Portal to open the production and add a built-in PEX business host to the production that acts as a proxy between the custom business host and the rest of the production. This built-in PEX business host has a setting that specifies the Java or .NET class of the custom business host. Integrating an adapter into the production also leverages a built-in PEX class as a proxy between the business host and the adapter written in Java or .NET.
3. From the Management Portal or InterSystems Terminal, manually start the Java or .NET gateway. The Java gateway require Jar files containing the PEX framework in the classpath. The PEX framework requires the Java Gateway and/or the .NET Gateway be running on the port that is defined in the settings of the built-in PEX components. For more details, see Running the Object Gateways for Java and .NET.

In Java, the PEX Java library, `com.intersystems.enslib.pex`, includes superclasses for each type of production component. For example, `com.intersystems.enslib.pex.BusinessOperation` is the superclass for business operations written in Java. Implementing a business host or adapter in Java begins by creating the appropriate subclass and overriding its methods.

In .NET, the PEX library `InterSystems.EnsLib.PEX` includes superclasses for each type of production component. For example, `InterSystems.EnsLib.PEX.BusinessProcess` is the superclass for business processes written in .NET. Implementing a business host or adapter in .NET begins by creating the appropriate subclass and overriding its methods.

In the interoperability production model, business processes are persistent objects throughout the processing of a message. Consequently, they retain the same value from initialization, processing a message, and receiving the response. . Although the Java and .NET business process objects are not persistent, the PEX framework provides the persistence for these classes. When you define a business process in Java or .NET, you define what variables will retain their values across invocations by defining them with the Persistent attribute. To specify the Persistent attribute, in Java specify `@Persistent` before defining the variable and in .NET, specify `[Persistent]` before defining the variable

1.2 Environmental Considerations

You can use InterSystems IRIS Interoperability only within an interoperability-enabled namespace that has a specific web application. When you create classes, you should avoid using reserved package names. The following subsections give the details.

1.2.1 Production-enabled Namespaces

An *interoperability-enabled namespace* is a namespace that has global mappings, routine mappings, and package mappings that make the classes, data, and menus that support productions available to it. For general information on mappings, see "Configuring Namespaces." (You can use the information in that section to see the actual mappings in any interoperability-enabled namespace; the details may vary from release to release, but no work is necessary on your part.)

The system-provided namespaces that are created when you install InterSystems IRIS are not interoperability-enabled, except, on the community edition, the USER namespace is an interoperability-enabled namespace. Any new namespace

that you create is by default interoperability-enabled. If you clear the **Enable namespace for interoperability productions** check box when creating a namespace, InterSystems IRIS creates the namespace with productions disabled.

Important: All system-provided namespaces are overwritten upon reinstallation or upgrade. For this reason, InterSystems recommends that customers always work in a new namespace that you create. For information on creating a new namespace, see "Configuring Data."

1.2.2 Web Application Requirement

Also, you can use a production in a namespace only if that namespace has an associated web application that is named `/csp/`*namespace*, where namespace is the namespace name. (This is the default web application name for a namespace.) For information on defining web applications, see "Applications."

1.2.3 Reserved Package Names

In any interoperability-enabled namespace, avoid using the following package names: Ens, EnsLib, EnsPortal, or CSPX. These packages are completely replaced during the upgrade process. If you define classes in these packages, you would need to export the classes before upgrading and then import them after upgrading.

Also, InterSystems recommends that you avoid using any package names that *start* with Ens (case-sensitive). There are two reasons for this recommendation:

- When you compile classes in packages with names that start with Ens, the compiler writes the generated routines into the ENSLIB system database. (The compiler does this because all routines with names that start with Ens are mapped to that database.) This means that when you upgrade the instance, thus replacing the ENSLIB database, the upgrade removes the generated routines, leaving only the class definitions. At this point, in order to use the classes, it is necessary to recompile them.

  In contrast, when you upgrade the instance, it is not necessary to recompile classes in packages with names that do not start with Ens.

- If you define classes in packages with names that start with Ens, they are available in all interoperability-enabled namespaces, which may or may not be desirable. One consequence is that it is not possible to have two classes with the same name and different contents in different interoperability-enabled namespaces, if the package name starts with Ens.

2

About Business Hosts and Adapters

This chapter discusses information that applies to all business hosts and adapters written in Java or .NET. For implementation details about a specific production component, refer to that component's chapter.

2.1 General Notes about Methods

As you override methods to implement a production component, keep the following in mind:

- Each production component must override all abstract methods. For information about which methods are abstract, see "PEX API Reference."
- While native interoperability methods use one input argument and one output argument and return a status, the corresponding PEX methods take one input argument and return the output argument as a return value.
- All error handling for PEX methods are done with exceptions.
- For native interoperability methods that don't require persistent objects as input and output arguments, the corresponding PEX methods can also use arbitrary objects as arguments and return values. PEX utilizes forward proxy and reverse proxy of the Gateway to map the arbitrary object appropriately.
- For native interoperability methods that require persistent objects as arguments, such as the methods that send messages to other processes, the corresponding PEX methods can use PEX Messages as arguments and return values. Examples of such methods are SendRequestSync, SendRequestAsync, OnRequest, OnResponse and OnMessage.
- When overriding callback methods, you should not change the formal spec of the methods even if you have customized the message class. The argument types should remain as objects.

2.2 Setting Runtime Variables

The Java and .NET code of a custom production component can contain properties that will be set at runtime based on values defined in the Management Portal. At runtime, the values specified in the **remoteSettings** field of the built-in PEX component are assigned to the corresponding Java or .NET component. Values are defined in the format: `property=value`. More than one property/value pair can be specified as long as each pair is on its own line. Do not include any spaces on the line.

For example, suppose your inbound adapter needs a username and password at runtime. In your code, you create public string variables called `username` and `pw`. Then, in the Management Portal, you can define the following in the **remoteSettings** field of the business service that uses the adapter:

```
username=Employee1
pw=MySecret
```

To set the values in the XML defining the production, you can set the variables using thee `%remoteSettings` field. Note that if a component includes an adapter, you must set the variable for the host and the adapter.

```
<Production Name="Demo.PEX.SimpleFileAccess"
...
  <Item Name="Demo.PEX.WriteDataToFileAdapterOperation" Category=""
    <Setting Target="Host" Name="%remoteSettings">outPath=C:\Practice\Data\Out\</Setting>
    <Setting Target="Adapter" Name="%remoteSettings">outPath=C:\Practice\Data\Out\</Setting>
  </Item>
</Production>
```

To access these setting in your PEX code, create a public member in your class. The PEX framework sets the value to the setting value. For example, in Java:

```
package demo;
public class CustomOutboundAdapter extends com.intersystems.enslib.pex.OutboundAdapter {
   public String outPath;
   public String username;
```

And in .NET C#:

```
namespace demo
{
  public class CustomOperationWithAdapter : BusinessOperation
  {
    public string outPath;
    public string pw;
...
```

2.3 Messaging

Within the PEX framework, most messages sent *between business hosts* are objects instantiated from a subclasses of `com.intersystems.enslib.pex.Message` in Java and `InterSystems.EnsLib.PEX.Message` in .NET. You simply add properties to your subclass, and then pass instantiated objects of the subclass using methods like `SendRequestAsync()` and `SendRequestSync()`. Within InterSystems IRIS, the Java or .NET message object corresponds to an object of class `EnsLib.PEX.Message`, which makes the message persistent and dynamic. By manipulating the object of type `EnsLib.PEX.Message`, you can reference any property in the Java or .NET object. Internally, the Java or .NET object is represented as JSON as it passes between business hosts, so viewing the messages in the Management Portal are displayed in JSON format.

Though you can use other message objects, they must still be persistent if they are being passed between business hosts. To pass an object to a built-in ObjectScript component, you use the type IRISObject and map it to the persistent object expected by that component. For example, if you are sending a message to the EnsLib.PubSub.PubSubOperation, you would map the IRISObject to EnsLib.PubSub.Request. Trying to pass a non-persistent object as a message between business hosts results in a runtime error.

Objects sent from an inbound adapter to a business service are arbitrary and do not need to be persistent.

3

Inbound Adapters

Business services use inbound adapters to receive specific types of input data. You can write a custom inbound adapter that is used by an ObjectScript business service, or it can be used by a business service that is also written in a PEX-supported language. For general information related to all production components written in Java or .NET, see About Business Hosts and Adapters.

3.1 Java and .NET Classes

An inbound adapter written in Java extends `com.intersystems.enslib.pex.InboundAdapter` .

An inbound adapter written in .NET extends `InterSystems.EnsLib.PEX.InboundAdapter` .

3.2 Using the Inbound Adapter

Typically, the `OnTask()` method performs the main function of the inbound adapter. At runtime, the `OnTask()` method is called at the interval defined in the settings of the business service that is using the adapter. From within `OnTask()`, call `BusinessHost.ProcessInput()` to dispatch to the associated business service. This `ProcessInput()` method takes an arbitrary object as input and receives an arbitrary object as a return value. The input object becomes the input argument of the business service's `OnProcessInput()` method, and the return object comes from the output of the business service's `OnProcessInput()` method. These arbitrary objects do not need to be persistent within InterSystems IRIS.

In the Management Portal, a business service has a setting, **AdapterClassname**, that specifies the class of the inbound adapter associated with the business service. When the inbound adapter is written in Java or .NET, the business service's **AdapterClassname** setting must be `EnsLib.PEX.InboundAdapter`. In addition, the **remoteClassname** setting must be set to the name of the Java or .NET class of the inbound adapter.

4

Outbound Adapters

Business operations use outbound adapters to send out specific types of data from the production. You can write a custom outbound adapter that is used by an ObjectScript business operation, or it can be used by a business operation that is also written in Java or .NET. For general information related to all production components written in Java or .NET, see About Business Hosts and Adapters.

4.1 Java and .NET Classes

An outbound adapter written in Java extends `com.intersystems.enslib.pex.OutboundAdapter` .

An outbound adapter written in .NET extends `InterSystems.EnsLib.PEX.OutboundAdapter` .

4.2 Using the Outbound Adapter

Within the outbound adapter, you can create all the methods you need to successfully send out data from the production. Each of these methods can be called from the business operation associated with the adapter. The business operation calls these methods by using the OutboundAdapter.Invoke() method.

In the Management Portal, a business operation has a setting, **AdapterClassname**, that specifies the class of the outbound adapter being used. When the outbound adapter is written in Java or .NET, the business service's **AdapterClassname** setting must be `EnsLib.PEX.OutboundAdapter`. In addition, the **remoteClassname** setting must be set to the name of the Java class of the outbound adapter.

5

Business Services

Business services connect with external systems and receive messages from them through an inbound adapter. For general information related to all production components written in Java, see About Business Hosts and Adapters.

5.1 Java and .NET Classes

A business service written in Java extends `com.intersystems.enslib.pex.BusinessService`.

A business service written in .NET extends `InterSystems.EnsLib.PEX.BusinessService`.

5.2 Using the Business Service

The `OnProcessInput()` method takes an arbitrary object from the `ProcessInput()` method of the business service's adapter, and returns an arbitrary object. These arbitrary objects do not need to be persistent.

For information about sending messages between the business service and another business host, see Messaging.

When you want to integrate the business service into the production:

1. Open the production in the Management Portal and add a business service based on the PEX class `EnsLib.PEX.BusinessService`.
2. Define the **remoteClassname** setting of the new PEX business service as the Java or .NET class you created for the custom business service.
3. Use the **gatewayHost** and **gatewayPort** settings of the PEX business service to define the values of the Object Gateway.

For example, suppose you have a business service written in a Java class called `MyJavaBusinessService`. To incorporate this custom business service into the production, you go to the Management Portal and add a business service based on the PEX class `EnsLib.PEX.BusinessService`. Once added, edit the **RemoteClassname** setting of the PEX business service to point to the Java classname `MyJavaBusinessService`.

6

Business Processes

Business processes allow you to define business logic, including routing and message transformation. Business processes receive messages from other business hosts in the production for processing.

For general information related to all production components written in Java, see About Business Hosts and Adapters.

6.1 Java and .NET Classes

A business service written in Java extends `com.intersystems.enslib.pex.BusinessProcess`.

A business service written in .NET extends `InterSystems.EnsLib.PEX.BusinessProcess`.

For information about sending messages between the business process and another business host, see Messaging.

6.1.1 Persistent Properties

Within InterSystems IRIS, business processes are persistent objects. For the lifespan of a Business Process, properties of the Business Process are stored and accessible during each callbacks. Within the PEX framework, you can save properties of the business process class saved by using `@Persistent` annotation in Java or the `[Persistent]` notation in .NET. Within InterSystems IRIS, persistent properties are saved in the corresponding instance of the business process. Persistent properties are restored before each callback and are saved after each callback. The @Persistent annotation only works for String, primitive types and their boxed types. The following is an example of using the @Persistent annotation:

```
@Persistent
public integer runningTotal = 0;

@Persistent
public string CurrentName = "temp";
```

6.2 Using the Business Process

When you want to integrate the business process into the production:

1. Open the production in the Management Portal and add a business service based on the PEX class `EnsLib.PEX.BusinessProcess`.
2. Define the **remoteClassname** setting of the new PEX business process as the Java class you created for the custom business process.

3. Use the **gatewayHost** and **gatewayPort** settings of the PEX business process to define the values of the Java Gateway.

For example, suppose you have a business process written in a Java class called `MyJavaBusinessProcess`. To incorporate this custom business process into the production, you go to the Management Portal and add a business process based on the PEX class `EnsLib.PEX.BusinessProcess`. Once added, edit the **RemoteClassname** setting of the PEX business process to point to the Java classname `MyJavaBusinessProcess`.

7

Business Operations

Business operations connect with external systems and send messages to them via an outbound adapter.

For general information related to all production components written in Java or .NET, see About Business Hosts and Adapters.

7.1 Java and .NET Classes

A business operation written in Java extends `com.intersystems.enslib.BusinessOperation`.

A business operation written in .NET extends `InterSystems.EnsLib.PEX.BusinessOperation`.

For information about sending messages between the business operation and another business host, see Messaging.

7.2 Using the Business Operation

At runtime, the `OnMessage()` method is called when the business operation receives a message from another business host. From within this method, the business operation can call any of the methods defined in the outbound adapter associated with the business operation.

Parameters for calls from a business operation to an outbound adapter are primitive types only, and do not need to be persistent.

When you want to integrate the business operation into the production:

1. Open the production in the Management Portal and add a business operation based on the PEX class `EnsLib.PEX.BusinessOperation`.
2. Define the **remoteClassname** setting of the new PEX business operation as the Java class you created for the custom business operation.
3. Use the **gatewayHost** and **gatewayPort** settings of the PEX business operation to define the values of the Object Gateway.

For example, suppose you have a business operation written in a Java class called `MyJavaBusinessOperation`. To incorporate this custom business operation into the production, you go to the Management Portal and add a business operation based on the PEX class `EnsLib.PEX.BusinessOperation`. Once added, edit the **RemoteClassname** setting of the PEX business operation to point to the Java classname `MyJavaBusinessOperation`.

8

Running the Object Gateway for Java and .NET

If you are using Java components or .NET components, the PEX framework requires an Object Gateway for Java or for .NET to be running when the production is running. It is recommended that you manually start the Object Gateway from the Management Portal or InterSystems Terminal. You should not add EnsLib.JavaGateway.Service to the production to automatically start the Java gateway.

You can use the Management Portal to define a Java Gateway or a .NET Gateway. Once a gateway is defined, you can use the same page to start and stop the gateway.

To start or stop a Java or .NET gateway, in the Management Portal:

1. Select **System Administration** > **Configuration** > **Connectivity** > **Object Gateways**
2. Select the Java or .NET gateway.
3. Select **Start** to start the gateway and **Stop** to stop the gateway. You can only edit the gateway settings when the gateway is stopped.

8.1 Defining the Java Gateway

To define the Java Gateway:

1. On the Object Gateways page, select **Create New Gateway**.
2. Select Object Gateway for **Java**.
3. Specify a gateway name, such as JavaGateway.
4. Specify the server name/IP address. If the Java Gateway is running locally, the default IP address is 127.0.0.1. You must specify this value you specify in the **Gateway Host** setting for every Java component in the production.
5. Specify the port number that the Java Gateway will use. You must specify this port number in the **Gateway Port** setting for every Java component in the production.
6. The following Jar files must be on the classpath of the Java Gateway used for the PEX production:
   - The PEX framework jar files: `intersystems-jdbc-3.1.0.jar` and `intersystems-gateway-3.1.0.jar`, which are located in the *install-dir*`\dev\java\lib\JDK18\` directory.

- The gson, serialization library jar file, `gson-2.8.5.jar`, which is located in the *install-dir*`\dev\java\lib\gson\` directory.
- Jar files that contain the subclasses you created for each production component. For example, if you wrote an inbound adapter in Java, you might have a jar file `MyJavaAdapter.jar`.

7. You can optionally set the other fields on the page.
8. Select **Save**.

If you are using a dedicated Gateway for your PEX production, it makes sense to add these jar files to the classpath when starting the Gateway. Alternatively, you can specify these jar files for individual PEX components in the Management Portal. Every PEX component in a production has a setting **Gateway Extra CLASSPATH**. You can add the jar file that corresponds to the PEX component in this setting rather than specifying it when you start the Gateway, but don't forget to also add the gson jar file to the classpath.

8.2 Defining the .NET Gateway

To define the .NET Gateway:

1. On the Object Gateways page, select **Create New Gateway**.
2. Select Object Gateway for **.Net**.
3. Specify a gateway name, such as DotNetGateway.
4. Specify the server name/IP address. If the DotNet Gateway is running locally, the default IP address is 127.0.0.1. You must specify this value you specify in the **Gateway Host** setting for every Java component in the production.
5. Specify the port number that the .NET Gateway will use. You must specify this port number in the **Gateway Port** setting for every .NET component in the production.
6. Select the **Execute as 64–bit** check box.
7. On Windows, select **.NET Version** 4.5.
8. You can optionally set the other fields on the page.
9. Select **Save**.

A

PEX API Reference

This reference applies to all PEX-supported languages, Java and the .NET languages.

A.1 BusinessOperation

The BusinessOperation class to provide implementation for a business operation, which sends the message to the external system. This class corresponds to the PEX framework EnsLib.PEX.BusinessOperation class. The EnsLib.PEX.BusinessOperation RemoteClassName property identifies the Java or .NET class with the business operation implementation.

The business operation can optionally use an adapter to handle the outgoing message. Specify the adapter in the Outbound-Adapter property. If the business operation has an adapter, it uses the adapter to send the message to the external system. The adapter can either be a PEX adapter or an ObjectScript adapter.

A.1.1 Adapter Instance Variable

The Adapter instance variable provides access to the outbound adapter. You specify the adapter in the production definition.

A.1.2 OnMessage() Method

The **OnMessage()** message is called when the business operation receives a message from another production component. Typically, the operation will either send the message to the external system or forwards it to a business process or another business operation. If the operation has an adapter, it uses the **Adapter.invoke()** method to call the method on the adapter that sends the message to the external system. If you operation is forwarding the message to another production component, it uses the **SendRequestAsync()** or the **SendRequestSync()** method.

Abstract method: you must implement.

Parameters: (*request*)

- *request* — of type Object, this contains the incoming message for the business operation.

You must implement an OnMessage method with a single parameter of type Object. Within the method you can cast the parameter to the actual type passed by the caller.

Returns: Object

A.1.3 OnInit() Method

The **OnInit()** is called when the component is started. Use **OnInit()** to initialize any structures needed by the component.

Abstract method: you must implement.

Parameters: none

Returns: void

A.1.4 OnTearDown() Method

The **OnTearDown()** method is called before the component is terminated. Use **OnTeardown()** to free any structures.

Abstract method: you must implement.

Parameters: none

Returns: void

A.1.5 SendRequestAsync() Method

The **SendRequestAsync()** method sends the specified message to the target business process or business operation asynchronously.

Parameters: (*target*, *request* [, *description*])

- *target* — a string that specifies the name of the business process or operation to receive the request. The target is the name of the component as specified in the Item Name property in the production definition, not the class name of the component.
- *request* — specifies the message to send to the target. The request can either have a class that is a subclass of Message class or have the IRISObject class. If the target is a built-in ObjectScript component, you should use the IRISObject class. The IRISObject class enables the PEX framework to convert the message to a class supported by the target. Otherwise, you can use a subclass of the Message class.
- *description* — an optional string parameter that sets a description property in the message header.

Returns: void

A.1.6 SendRequestSync() Method

The **SendRequestSync()** method sends the specified message to the target business process or business operation synchronously.

Parameters: (*target*, *request* [,*timeout* [, *description*]])

- *target* — a string that specifies the name of the business process or operation to receive the request. The target is the name of the component as specified in the Item Name property in the production definition, not the class name of the component.
- *request* — specifies the message to send to the target. The request can either have a class that is a subclass of Message class or have the IRISObject class. If the target is a built-in ObjectScript component, you should use the IRISObject class. The IRISObject class enables the PEX framework to convert the message to a class supported by the target. Otherwise, you can use a subclass of the Message class.
- *timeout* — an optional integer that specifies the number of seconds to wait before treating the send request as a failure. The default value is -1, which means wait forever.

- *description* — an optional string parameter that sets a description property in the message header.

A.1.7 LOGINFO(), LOGALERT(), LOGWARNING(), LOGERROR(), and LOGASSERT() Methods

The log methods write to the production log, which can be viewed in the management portal in the component's Log tab. These methods have the same parameter and differ only in the type of the log message:

- LOGINFO() has an info type.
- LOGALERT() has an alert type.
- LOGWARNING() has a warning type.
- LOGERROR() has an error type.
- LOGASSERT() has an assert type.

Parameters: (*message*)

- *message* — a string that is written to the log.

A.2 BusinessProcess

The BusinessProcess classes typically contains most of the logic in a production. A business process can receive messages from a business service, another business process, or a business operation. It can modify the message, convert it to a different format, or route it based on the message contents. The business process can route a message to a business operation or another business process.

A.2.1 Persistent Property

The Persistent property provides persistence of the specified public variable while the business process is handling a incoming message and all responses to the corresponding outgoing messages that it sends. An ObjectScript BusinessProcess instance is active from being initiated to being terminated and preserves property values across different function calls. For Java and .NET business processes, each method is called in a separate instance and the values of public variables are not be preserved unless you specify the Persistent property for them. In Java, specify the Persistent property as `@Persistent` before the declaration of the public variable. In .NET, specify the Persistent property as `[Persistent]` before the declaration of the public variable. The Persistent property only works for String, primitive types and their boxed types.

A.2.2 OnRequest() Method

The OnRequest() method handles requests sent to the business process. A production calls this method whenever an initial request for a specific business process arrives on the appropriate queue and is assigned a job in which to execute.

Abstract method: you must implement.

Parameters: (*request*)

- *request* — an object that contains the request message sent to the business process.

A.2.3 OnResponse() Method

The OnResponse() method handles responses sent to the business process in response to messages that it sent to the target. A production calls this method whenever a response for a specific business process arrives on the appropriate queue and is assigned a job in which to execute. Typically this is a response to an asynchronous request made by the business process where the responseRequired parameter has a true value.

Abstract method: you must implement.

Parameters: (*request*, *response*, *callRequest*, *callResponse*, *completionKey*)

- *request* — an object that contains the initial request message sent to business process.
- *response* — an object that contains the response message that this business process can return to the production component that sent the initial message.
- *callRequest* — an object that contains the request that the business process sent to its target.
- *callResponse* — an object that contains the incoming response.
- *completionKey* — a string that contains the completionKey specified in the completionKey parameter of the outgoing SendAsync() method.

A.2.4 OnComplete() Method

The OnComplete() method is called after the business process has received and handled all responses to requests it has sent to targets.

Abstract method: you must implement.

Parameters: (*request*, *response*)

- *request* — an object that contains the initial request message sent to business process.
- *response* — an object that contains the response message that this business process can return to the production component that sent the initial message.

A.2.5 OnInit() Method

The OnInit() method is called when the component is started.

Abstract method: you must implement.

Parameters: none

Returns: void

A.2.6 OnTearDown() Method

The OnTearDown() method is called before the component is terminated.

Abstract method: you must implement.

Parameters: none

Returns: void

A.2.7 Reply() Method

The Reply() method sends the specified response to the production component that sent the initial request to the business process.

Parameters: (response)

- response — an object that contains the response message.

A.2.8 SendRequestAsync() Method

The SendRequestAsync() method sends the specified message to the target business process or business operation asynchronously.

Parameters: (*target*, *request* [, *responseRequired* [, *completionKey* [, *description*]]])

- *target* — a string that specifies the name of the business process or operation to receive the request. The target is the name of the component as specified in the Item Name property in the production definition, not the class name of the component.
- *request* — specifies the message to send to the target. The request can either have a class that is a subclass of Message class or have the IRISObject class. If the target is a built-in ObjectScript component, you should use the IRISObject class. The IRISObject class enables the PEX framework to convert the message to a class supported by the target. Otherwise, you can use a subclass of the Message class.
- *responseRequired* — a boolean value that specifies if the target must send a response message.
- *completionKey* — a string that will be sent with the response message.
- *description* — an optional string parameter that sets a description property in the message header.

A.2.9 SendRequestSync() Method

The SendRequestSync() method sends the specified message to the target business process or business operation synchronously.

Parameters: (*target*, *request* [,*timeout* [, *description*]])

- *target* — a string that specifies the name of the business process or operation to receive the request. The target is the name of the component as specified in the Item Name property in the production definition, not the class name of the component.
- *request* — specifies the message to send to the target. The request can either have a class that is a subclass of Message class or have the IRISObject class. If the target is a built-in ObjectScript component, you should use the IRISObject class. The IRISObject class enables the PEX framework to convert the message to a class supported by the target. Otherwise, you can use a subclass of the Message class.
- *timeout* — an optional integer that specifies the number of seconds to wait before treating the send request as a failure. The default value is -1, which means wait forever.
- *description* — an optional string parameter that sets a description property in the message header.

A.2.10 SetTimer() Method

The SetTimer() method specifies the maximum time the business process will wait for all responses.

Parameters: (*timeout* [, *completionKey*])

- *timeout* — an integer that specifies a number of seconds or a string that specifies a time period, such as "PT15S", which represents 15 seconds of processor time.
- *completionKey* — a string that will be returned with the response if the maximum time is exceeded.

A.2.11 LOGINFO(), LOGALERT(), LOGWARNING(), LOGERROR(), and LOGASSERT() Methods

The log methods write to the production log, which can be viewed in the management portal in the component's Log tab. These methods have the same parameter and differ only in the type of the log message:

- LOGINFO() has an info type.
- LOGALERT() has an alert type.
- LOGWARNING() has a warning type.
- LOGERROR() has an error type.
- LOGASSERT() has an assert type.

Parameters: (*message*)

- *message* — a string that is written to the log.

A.3 BusinessService

The BusinessService class is responsible for receiving the data from the external system and sending it to business processes or business operations in the production. The business service can use an adapter to access the external system. There are three ways of implementing a business service:

1. Polling business service with an adapter — The production framework at regular intervals calls the adapter's OnTask() method, which sends the incoming data to the the business service ProcessInput() method, which, in turn calls the OnProcessInput method with your code.
2. Polling business service that uses the default adapter — In this case, the framework calls the default adapter's OnTask method with no data. The OnProcessInput() method then performs the role of the adapter and is responsible for accessing the external system and receiving the data.
3. Nonpolling business service — The production framework does not initiate the business service. Instead custom code in either a long-running process or one that is started at regular intervals initiates the business service by calling the Director.CreateBusinessService() method.

A.3.1 Adapter Instance Variable

The Adapter instance variable provides access to the inbound adapter associated with the business service.

A.3.2 OnProcessInput() Method

The OnProcessInput() method receives the message from the inbound adapter via the ProcessInput() method and is responsible for forwarding it to target business processes or operations. If the business service does not specify an adapter, then the default adapter calls the OnProcessInput() method with no message and the business service is responsible for receiving the data from the external system and validating it.

Abstract method: you must implement.

Parameters: (*message*)

- *message* — an object containing the data that the inbound adapter sent to the business service. The message can have any structure agreed upon by the inbound adapter and the business service. The message does not have to be a subclass of Message or IRISObject and is typically not persisted in the database.

A.3.3 OnInit() Method

The OnInit() method is called when the component is started. Use the OnInit() method to initialize any structures needed by the component.

Abstract method: you must implement.

Parameters: none

Returns: void

A.3.4 OnTearDown() Method

The OnTearDown() method is called before the business component is terminated. Use the OnTeardown() method to free any structures.

Abstract method: you must implement.

Parameters: none

Returns: void

A.3.5 ProcessInput() Method

The inbound adapter calls the ProcessInput0 method of the business service and the ProcessInput() method in turn calls the OnProcessInput() method that is your custom code.

Parameters: (*input*)

- *input* — an object with an arbitrary structure by agreement with the inbound adapter. The parameters passed to the ProcessInput() method do not need to be persistent objects.

Returns: object

A.3.6 SendRequestAsync() Method

The SendRequestAsync() method sends the specified message to the target business process or business operation asynchronously.

Parameters: (*target*, *request* [, *description*])

- *target* — a string that specifies the name of the business process or operation to receive the request. The target is the name of the component as specified in the Item Name property in the production definition, not the class name of the component.
- *request* — specifies the message to send to the target. The request can either have a class that is a subclass of Message class or have the IRISObject class. If the target is a built-in ObjectScript component, you should use the IRISObject class. The IRISObject class enables the PEX framework to convert the message to a class supported by the target. Otherwise, you can use a subclass of the Message class.

- *description* — an optional string parameter that sets a description property in the message header.

A.3.7 SendRequestSync() Method

The SendRequestSync() method sends the specified message to the target business process or business operation synchronously.

Parameters: (*target*, *request* [,*timeout* [, *description*]])

- *target* — a string that specifies the name of the business process or operation to receive the request. The target is the name of the component as specified in the Item Name property in the production definition, not the class name of the component.
- *request* — specifies the message to send to the target. The request can either have a class that is a subclass of Message class or have the IRISObject class. If the target is a built-in ObjectScript component, you should use the IRISObject class. The IRISObject class enables the PEX framework to convert the message to a class supported by the target. Otherwise, you can use a subclass of the Message class.
- *timeout* — an optional integer that specifies the number of seconds to wait before treating the send request as a failure. The default value is -1, which means wait forever.
- *description* — an optional string parameter that sets a description property in the message header.

A.3.8 LOGINFO(), LOGALERT(), LOGWARNING(), LOGERROR(), and LOGASSERT() Methods

The log methods write to the production log, which can be viewed in the management portal in the component's Log tab. These methods have the same parameter and differ only in the type of the log message:

- LOGINFO() has an info type.
- LOGALERT() has an alert type.
- LOGWARNING() has a warning type.
- LOGERROR() has an error type.
- LOGASSERT() has an assert type.

Parameters: (*message*)

- *message* — a string that is written to the log.

A.4 Director

The Director class is used for nonpolling business services, that is, business services which are not automatically called by the production framework (through the inbound adapter) at the call interval. Instead these business services are created by a custom application by calling the Director.CreateBusinessService() method.

A.4.1 CreateBusinessService() Method

The CreateBusinessService() method initiates the specified business service.

Parameters: (*connection*, *target*)

- *connection* — an IRISConnection object that specifies the connection to the Object Gateway for Java or .NET.
- *target* — a string that specifies the name of the business service in the production definition.

Return value: *businessService* — the return value contains the Business Service instance that has been created.

A.5 InboundAdapter

he InboundAdapter is responsible for receiving the data from the external system, validating the data, and sending it to the business service by calling the ProcessInput() method.

A.5.1 BusinessHost Instance Variable

The BusinessHost instance variable provides access to the business service associated with the inbound adapter. The adapter calls the ProcessInput() method of the business service.

A.5.2 OnTask() Method

The OnTask() method is called by the production framework at intervals determined by the business service CallInterval property. The OnTask() method is responsible for receiving the data from the external system, validating the data, and sending it in a message to the business service OnProcessInput() method. The message can have any structure agreed upon by the inbound adapter and the business service.

Abstract method: you must implement.

Parameters: none

A.5.3 OnInit() Method

The OnInit() method is called when the component is started. Use the OnInit() method to initialize any structures needed by the component.

Abstract method: you must implement.

Parameters: none

Returns: void

A.5.4 OnTearDown() Method

The OnTearDown() method is called before the business component is terminated. Use the OnTeardown() method to free any structures.

Abstract method: you must implement.

Parameters: none

Returns: void

A.5.5 LOGINFO(), LOGALERT(), LOGWARNING(), LOGERROR(), and LOGASSERT() Methods

The log methods write to the production log, which can be viewed in the management portal in the component's Log tab. These methods have the same parameter and differ only in the type of the log message:

- LOGINFO() has an info type.
- LOGALERT() has an alert type.
- LOGWARNING() has a warning type.
- LOGERROR() has an error type.
- LOGASSERT() has an assert type.

Parameters: (*message*)

- *message* — a string that is written to the log.

A.6 OutboundAdapter

The OutboundAdapter is responsible for sending the data to the external system.

A.6.1 BusinessHost Instance Variable

The BusinessHost instance variable provides access to the BusinessOperation associated with the OutboundAdapter.

A.6.2 Invoke() Method

The Invoke() method allows the BusinessOperation to execute any public method defined in the OutboundAdapter.

Parameters: (*methodname*, *arguments*)

- *methodname* — specifies the name of the method in the outbound adapter to be executed.
- *arguments* — contains the arguments of the specified method.

A.6.3 OnInit() Method

The OnInit() method is called when the component is started. Use the OnInit() method to initialize any structures needed by the component.

Abstract method: you must implement.

Parameters: none

Returns: void

A.6.4 OnTearDown() Method

The OnTearDown() method is called before the business component is terminated. Use the OnTeardown() method to free any structures.

Abstract method: you must implement.

Parameters: none

Returns: void

A.6.5 LOGINFO(), LOGALERT(), LOGWARNING(), LOGERROR(), and LOGASSERT() Methods

The log methods write to the production log, which can be viewed in the management portal in the component's Log tab. These methods have the same parameter and differ only in the type of the log message:

- LOGINFO() has an info type.
- LOGALERT() has an alert type.
- LOGWARNING() has a warning type.
- LOGERROR() has an error type.
- LOGASSERT() has an assert type.

Parameters: (*message*)

- *message* — a string that is written to the log.

A.7 Message

The Message class is the abstract class that is the superclass class for persistent messages sent from one component to another. The Message class has no properties or methods. Users subclass the Message class in order to add proprties. The PEX framework provides the persistence to objects derived from the Message class.