

LISTEN.  
THINK.  
SOLVE.<sup>SM</sup>

INTEGRATED PRODUCTION & PERFORMANCE SUITE



Production Management

# *FactoryTalk*<sup>®</sup> ProductionCentre



**PLANT OPERATIONS**  
**RELEASE 10.4**  
**ACTIVITY DEVELOPERS GUIDE**

PUBLICATION PFACT-RM104A-EN-E

ALLEN-BRADLEY • ROCKWELL SOFTWARE

**Rockwell**  
**Automation**

**Contact Rockwell**

Customer Support Telephone — 1.440.646.3434

Online Support — <http://support.rockwellautomation.com>

**Copyright Notice**

© 2017 Rockwell Automation Technologies, Inc. All rights reserved. Printed in USA.

This document and any accompanying Rockwell Software products are copyrighted by Rockwell Automation Technologies, Inc. Any reproduction and/or distribution without prior written consent from Rockwell Automation Technologies, Inc. is strictly prohibited. Please refer to the license agreement for details.

**Trademark Notices**

FactoryTalk, Rockwell Automation, Rockwell Software, FactoryTalk View, and the Rockwell Software logo are registered trademarks of Rockwell Automation, Inc.

The following logos and products are trademarks of Rockwell Automation, Inc.:

FactoryTalk Shop Operations Server, FactoryTalk ProductionCentre, FactoryTalk View, FactoryTalk Administration Console, FactoryTalk Services Platform, FactoryTalk Live Data, RSAssetSecurity, FactoryTalk Security, FTPC Administrator, Integrate, Operational Data Store, ODS, Plant Operations, Process Designer, Live Transfer, Rockwell Software CPGSuite, Rockwell Software PharmaSuite, and Rockwell Software AutoSuite.

**Other Trademarks**

ActiveX, Microsoft, Microsoft Access, SQL Server, Visual Basic, Visual C++, Visual SourceSafe, Windows, Windows ME, Windows NT, Windows 2000, Windows Server 2003, Windows 2008, and Windows 2012 are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

Adobe, Acrobat, and Reader are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries.

ControlNet is a registered trademark of ControlNet International.

DeviceNet is a trademark of the Open DeviceNet Vendor Association, Inc. (ODVA).

Ethernet is a registered trademark of Digital Equipment Corporation, Intel, and Xerox Corporation.

OLE for Process Control (OPC) is a registered trademark of the OPC Foundation.

Oracle, SQL\*Net, and SQL\*Plus are registered trademarks of Oracle Corporation.

All other trademarks are the property of their respective holders and are hereby acknowledged.

**Warranty**

This product is warranted in accordance with the product license. The product's performance may be affected by system configuration, the application being performed, operator control, maintenance, and other related factors. Rockwell Automation is not responsible for these intervening factors. The instructions in this document do not cover all the details or variations in the equipment, procedure, or process described, nor do they provide directions for meeting every possible contingency during installation, operation, or maintenance. This product's implementation may vary among users.

This document is current as of the time of release of the product; however, the accompanying software may have changed since the release. Rockwell Automation, Inc. reserves the right to change any information contained in this document or the software at anytime without prior notice. It is your responsibility to obtain the most current information available from Rockwell when installing or using this product.

# Table of Contents

<b>Read Me First .....</b>	<b>5</b>
Audience and Expectations .....	5
Organization.....	5
Other Information Sources.....	6
Related Documentation.....	6
Solutions and Technical Support.....	6
 <b>Chapter 1 Activities Overview and Getting Started.....</b>	 <b>7</b>
Activities Overview .....	8
Synchronous versus Asynchronous Activity Execution .....	8
Synchronous Execution.....	9
Asynchronous Execution .....	9
Creating Activities that Support Localization .....	10
Supported Development Environments .....	10
The Application Framework SDK ZIP File.....	10
 <b>Chapter 2 Creating Activities .....</b>	 <b>13</b>
Creating the Project .....	14
Creating the Package and Class .....	16
Default Methods.....	16
Providing the Description .....	18
Defining the Configuration Parameters .....	18
Defining Input Parameters .....	22
Defining Output Parameters .....	23
Creating Custom Events .....	24
Adding Visual and Non-Visual Controls .....	25
GUI versus Non-GUI Activities .....	26

Activities Within Activity Sets .....	26
Using Plant Operations UI Controls .....	27
Writing Scripts for Plant Operations Control Events.....	31
Using Non-Plant Operations Controls .....	33
Providing Business Logic .....	33
Returning a Response Object.....	33
Retrieving Configuration Parameters .....	35
Retrieving Input Parameters .....	35
Setting Output Parameters .....	36
Managing Lifecycles.....	36
 <b>Chapter 3 Testing and Distributing Activities .....</b>	<b>41</b>
Configuring Default Tab Behavior (Optional) .....	42
Testing Activities in Process Designer .....	42
Testing Activities in Shop Operations .....	42
Running an Activity from the IDE .....	43
Adding an Activity to a Form or Event Sheet .....	46
Testing and Debugging an Activity .....	47
Distributing Activities .....	48
Setting Up Member Help for Activities.....	49
Updating Activities.....	50
 <b>Chapter 4 Activities Code Examples .....</b>	<b>53</b>
Non-GUI Activities .....	54
Creating a Basic Hello World Activity .....	54
Creating Activities to Interface with Other Systems .....	55
GUI Activities.....	56
Using Java Swing Controls.....	56
 <b>Index.....</b>	<b>65</b>



# Read Me First

## Audience and Expectations

This book is intended for experienced professionals who understand their company's business needs and the technical terms used in this guide. In addition, we expect the user to be experienced with the following:

- Java development environments
- Multi-threaded programming
- Writing code in Java

This guide assumes that the supporting network equipment and software, including Plant Operations and a development environment, have been installed.

## Organization

This book contains the following chapters:

- **Chapter 1, “Activities Overview and Getting Started”** – Provides introductory information on activities.
- **Chapter 2, “Creating Activities”** – Provides instructions for creating activities.
- **Chapter 3, “Testing and Distributing Activities”** – Provides instructions for testing and distributing activities.
- **Chapter 4, “Activities Code Examples”** – Provides code examples.

## Other Information Sources

In addition to this guide, the following information sources are available.

### Related Documentation

**Table 1** lists other available documents related to activities.

**Table 1 Related Documents**

Title	Purpose
Process Designer and Objects Help located at <a href="http://&lt;serverName&gt;:&lt;portName&gt;/PlantOperations/docs/help/pd/index.htm">http://&lt;serverName&gt;:&lt;portName&gt;/PlantOperations/docs/help/pd/index.htm</a> *	Online help for activities and activity sets.
Plant Operations Release Notes located at <a href="http://&lt;serverName&gt;:&lt;portName&gt;/PlantOperations/docs/mergedProjects/readme/readme.htm">http://&lt;serverName&gt;:&lt;portName&gt;/PlantOperations/docs/mergedProjects/readme/readme.htm</a> *	Release Notes for Plant Operations.
* <serverName>:<portName> is the Plant Operations application server name and HTTP port number.	

### Solutions and Technical Support

The FactoryTalk® ProductionCentre (called FTPC hereafter) online knowledge base contains information and articles related to activities. Contact Rockwell Automation for a user name and password to access the knowledge base.

## Activities Overview and Getting Started

### In this chapter

- ❑ **Activities Overview** 8
- ❑ **Synchronous versus Asynchronous Activity Execution** 8
  - Synchronous Execution 9
  - Asynchronous Execution 9
- ❑ **Creating Activities that Support Localization** 10
- ❑ **Supported Development Environments** 10
- ❑ **The Application Framework SDK ZIP File** 10

## Activities Overview

An activity is a re-usable and configurable component used to create end-user applications. Activities are implemented outside of Process Designer, using Java. (See the *FactoryTalk ProductionCentre Supported Platforms Guide* for the supported Java version.) Each activity is a piece of well-defined functionality (for example, a consumption activity). You can design the activity to be visual (GUI) or non-visual (non-GUI) in the end-user interface. Non-GUI activities can be used with forms or event sheets. GUI activities can only be used on forms. Although a non-GUI activity does not have an end-user interface, you can use it to display dialogs if it is used on a form, but not if it is used on an event sheet. Two types of developers are involved in creating and using activities.

- Activity Developers create activities and then distribute them using the Process Designer activity object.
- Application Developers use and configure those activities in their applications.

---

**NOTE:** In this document, code written by Activity Developers in Java is referred to as code, and code written by Application Developers in Process Designer is referred to as script.

---

An activity should be highly configurable. Ideally, an application will have a small number of highly-configurable activities, rather than a large number of highly-specialized activities. In addition to providing the activity behavior and business logic, the Activity Developer can create custom events and define configuration, input, and output parameters for the Application Developer to use. You will also define the user interface, if applicable.

You can use activities to create activity sets in Process Designer. An activity set is a workflow of activities or other activity sets. You create an activity set using a sequential function chart (SFC) that follows both the IEC and Rockwell Automation standards. See the Process Designer Online Help for more information.

The intended audience for this document is Activity Developers.

## Synchronous versus Asynchronous Activity Execution

An Application Developer can use either a synchronous or asynchronous call to execute an activity. Keep this in mind when you design your activity. The methods that an Application Developer can use to execute an activity are:

- **activityExecute():** Activity code executes directly (synchronous call).



- **activityStart():** Activity code starts a new thread (asynchronous call), and then calls `activityExecute()`. Therefore, an activity's business logic always resides in the `activityExecute()` method.
- **complete():** if the activity is inside an activity set, the activity thread will be unblocked and executed. If the activity is directly on a form, this method will invoke the `activityExecute()` method.
- You can help the Application Developer by providing a recommendation in the activity documentation on which execution is best to use. The main considerations are:
  - Does the activity take a long time to execute?
  - Will the Application Developer want the user to be able to perform other tasks while the activity executes?

If either of the above is true, then you may want to recommend asynchronous execution.

## Synchronous Execution

When an Application Developer executes an activity in a synchronous thread, the activity executes on the same thread as the current application execution, and the current execution stops until the activity call comes back with a response. When an activity is executed this way, the activity state model is not applicable, and activity lifecycle events do not fire. You can define custom events, which are not related to the activity state model, and these events will fire regardless of the execution type. See [“Creating Custom Events” on page 24](#).

## Asynchronous Execution

When an Application Developer executes an activity in an asynchronous thread, the activity executes on its own, new thread and current application execution continues. When an activity is executed this way, the activity state model applies.

The activity state model divides the lifecycle of an activity into a series of states. When an activity enters a new state, an event is fired. Application Developers can write scripts for those events. They can also write scripts for the custom events that you defined. See [“Creating Custom Events” on page 24](#). The Application Developer can also control when the activity should enter certain states by adding methods such as `activityStart()` or `activityHold()` to their script. They cannot control other states, such as an error condition, but they can write a script that should run if the activity enters that state. As the Activity Developer, you can also write code that should run when an activity enters a certain state. See [“Managing Lifecycles” on page 36](#) for important information.

## Creating Activities that Support Localization

You should design and create each activity so that it can easily be adapted to various languages and regions without code changes. You can do this by following these internationalization best practices:

- Never hard-code messages. Use the Plant Operations message object instead. See *Localizing an Application* and *Using the API with Messages* in the Process Designer and Objects Help for more information and code examples.
- Avoid hard-coding GUI component labels. Use the Plant Operations message object instead. See the *Messages* topics in the Process Designer and Objects Help for more information and code examples.
- Make sure culturally-dependent data, such as dates and currencies, appear in formats that conform to the end user's region and language. A variety of Plant Operations functions support number and date internationalization. See *Localizing Numbers and Dates* in the Process Designer and Objects Help for more information and code examples.
- Localize the Plant Operations Response object error messages. See *Localizing the Response Object Messages* in the Process Designer and Objects Help for more information and code examples.

## Supported Development Environments

You can develop and debug your activities in:

- Rational Application Developer (RAD) for WebSphere Software
- Eclipse IDE for Java Developers
- Eclipse IDE for Java EE Developers
- Eclipse Classic

See the *FactoryTalk ProductionCentre Supported Platforms Guide* for the supported versions of RAD and Eclipse. Rockwell Automation recommends that you use RAD. In this document, specific steps are given for RAD and Eclipse environments for functionality such as setting up and testing your application.

---

**IMPORTANT:** You must write your code in Java. .Net is not supported.

---

## The Application Framework SDK ZIP File

To prepare your development environment to create activities, you must download the installable version of Process Designer that is packaged in the Application Framework SDK ZIP file.

1. Log into the FTPC download site. (Contact Customer Support for your customer-specific site.)
2. Download the appropriate Application Framework SDK product. The file name is PC<version>-<build>-ApplicationFrameworkSDK\_<appserver>. ZIP where:
  - <version> is the FTPC version number.
  - <build> is the FTPC build number.
  - <appserver> is the application server for your environment.

---

**NOTE:** Ensure that you download the ZIP file that matches the version, build, and application server of your FTPC installation. Not doing so may result in problems running Process Designer.

---

3. Open the ZIP file you just downloaded and extract the **pcclient** folder to the location of your choice.

---

**IMPORTANT:** If you are using Windows Vista, please note that Windows Vista does not allow executable files to run under the default user “App\Data\LocalLaw” folder. Do not extract these files into that folder.

---

4. If you are using WebSphere, then you must perform the following steps to obtain the IBM JRE:
  - a. Create a folder on your machine to hold the IBM JRE. The folder’s fully-qualified path and name must not contain spaces, and the path should contain as few subdirectories as possible. For example, C:\tools\ibmjre.
  - b. Open the Application Framework SDK JAR file using WinZip or a similar compression utility.
  - c. Locate the AppClient.zip file. To quickly find the file, sort the files by size, largest to smallest, and then you will see it at the top.
  - d. From WinZip, extract AppClient.zip to the folder you created in Step 4a.
  - e. From the folder you created in Step 4a, use WinZip to extract the contents of AppClient.zip.

The IBM JRE is located in <folder>\java. For example, C:\tools\ibmjre\java.



## Creating Activities

### In this chapter

- ❑ **Creating the Project** 14
- ❑ **Creating the Package and Class** 16
  - Default Methods 16
- ❑ **Providing the Description** 18
- ❑ **Defining the Configuration Parameters** 18
- ❑ **Defining Input Parameters** 22
- ❑ **Defining Output Parameters** 23
- ❑ **Creating Custom Events** 24
- ❑ **Adding Visual and Non-Visual Controls** 25
  - GUI versus Non-GUI Activities 26
  - Activities Within Activity Sets 26
  - Using Plant Operations UI Controls 27
  - Writing Scripts for Plant Operations Control Events 31
  - Using Non-Plant Operations Controls 33
- ❑ **Providing Business Logic** 33
  - Returning a Response Object 33
  - Retrieving Configuration Parameters 35
  - Retrieving Input Parameters 35
  - Setting Output Parameters 36
  - Managing Lifecycles 36

You can create an activity with a visual user interface (GUI) or one without (non-GUI). Application Developers can use visual activities on forms only. They can use non-visual activities on both forms and event sheets.

To create an activity, you must complete the following steps.

**Table 2-1 Creating Activities Checklist**

Done?	Step	Page
1.	Create the new project, import the Application Framework SDK jar file, and set the correct JRE.	page 14
2.	Create the activity's package and class. Before you can create the class, you must know if you are creating a GUI or non-GUI activity.	page 16
3.	Provide a description for the activity.	page 18
4.	Define the configuration parameters.	page 18
5.	Define the input parameters.	page 22
6.	Define output parameters.	page 23
7.	Create custom events.	page 24
8.	Add the non-visual and visual controls. If it is a GUI activity, then provide the layout.	page 25
9.	If it is a GUI activity, then you can write code for any of the UI control's events.	page 31
10.	Provide the business logic.	page 33

After you create the activity, you must test it, and then distribute it. See [Chapter 3, “Testing and Distributing Activities”](#) for more information.

## Creating the Project

To start creating an activity, you must create the activity's project. You also need to import the PCClient.userlibraries that you downloaded in [“The Application Framework SDK ZIP File”](#) on [page 10](#) and set the correct JRE. The exact steps you must take will depend on the development environment (IDE) you are using.

The steps in Eclipse are:

1. Select File > New > Java Project.
2. In the Create a Java project dialog:
  - a. Enter the Project name. For example, MyCompanyActivities.
  - b. If you are using WebSphere, then set the JRE to the IBM JRE you extracted in [Step 4](#) of [“The Application Framework SDK ZIP File”](#) on

page 10. If you are using JBoss, then set the default JRE to the one currently supported by FTPC. Please see the *FactoryTalk ProductionCentre Supported Platforms Guide* for the supported version.

- c. Click [Next].
3. In the Java Settings dialog, import the PCCClient library:
  - a. Select the Libraries tab and click [Add Library].
  - b. Select User Library and click [Next].
  - c. Click [User Libraries] and then [Import...].
  - d. Click [Browse] and browse to the location of PCCClient.userlibraries (Step 3 of “The Application Framework SDK ZIP File” on page 10).
  - e. Click either [OK] or [Finish] until you get back to the main Eclipse interface.
  - f. Right-click on your project and select *Import*.
  - g. Under the General folder, select *Existing Projects into Workspace* and click [Next].
  - h. Browse to the folder the contains the PCCClient library. For example, if you are on a JBoss Advanced environment, this folder is called PCCClient\_<build\_number>\_JBossAdv.
  - i. Click [OK].
4. Click [Finish].

The steps in RAD are:

1. Select File > New > Project.
2. In the Select a wizard dialog, select Java Project, and then click [Next].
3. In the Create a Java project dialog, enter the Project name, and then click [Finish].
4. Right-click the project you just created, and then select Properties.
5. In the Properties dialog, select Java Build Path, and then click [Add External JARs].
6. In the JAR Selection dialog, browse for the file that you downloaded in “The Application Framework SDK ZIP File” on page 10, and then click [Open].
7. In the Properties dialog, click [OK].
8. Select Window > Preferences.
9. In the Preferences dialog, select Java > Installed JREs, and then click [Add].
10. In the Add JRE dialog, enter the following, and then click [OK]:
  - JRE type: enter Standard VM.
  - JRE name: if you are using WebSphere, then set to the IBM JRE you extracted in Step 4 of “The Application Framework SDK ZIP File” on

page 10. If you are using BEA Weblogic or JBoss, then set to the supported JRE listed in the *FactoryTalk ProductionCentre Supported Platforms Guide*.

- JRE home directory: browse to the JRE home directory, for example, c:\SUN\_JDK\_<version> or c:\tools\ibmjre\java.

11. In the Preferences dialog, check the JRE you just added, and then click [OK].

## Creating the Package and Class

After you have decided if the activity will be GUI or non-GUI, you can create the activity's package and class. A GUI activity extends the `ActivityControl` class, and a non-GUI activity extends the `Activity` class.

---

**NOTE:** The class name that you define will be what Application Developers see in the Activity Selector dialog in Process Designer when they add an activity to a form or event sheet.

---

The exact steps you must take will depend on the IDE you are using.

The steps in Eclipse and RAD are:

1. Select the project you created in “Creating the Project” on page 14, right-click and select New > Package.
2. In the New Java Package dialog, enter a name for the package. For example, `com.mycompany.services.activities`, and then click [Finish].
3. Select the package, right-click and select New > Class.
4. In the New Java Class dialog, do the following, and then click [Finish]:
  - a. In the Name edit box, enter a name for the class. For example, `MyGUIActivity`.
  - b. In the Superclass edit box:
    - If you are creating a non-GUI activity, then enter `com.rockwell.activity.Activity`.
    - If you are creating a GUI activity, then enter `com.datasweep.compatibility.ui.ActivityControl`.
  - c. Check Constructors from superclass.

The class is created with all of the default methods.

## Default Methods

The following default methods are provided after you create the class:



Table 2-2 Default Activity Methods

Method	Description
activityExecute()	Runs when the Application Developer executes the activity using either activityExecute() or activityStart(). You will write the majority of the activity's business logic here. It must return a Response object.
configurationDescriptors()	Contains the configuration parameters.
configurationItemSet(...)	Runs after the setConfigurationItem() method is called in runtime.
configurationLoaded()	Runs after the setConfiguration() method is called in runtime.
getActivityDescription()	Contains the detailed description of the activity.
getActivityEvents()	Contains the custom events.
getBaseName()	<p>Contains the activity control/container's name. You can provide a value for the name property. This is what Application Developers will see in Process Designer in the Form Explorer and Event Sheet Explorer. If you do not provide a value, then a default name will be provided, similar to other form controls and event sheet containers: <i>activityControl1</i>, <i>esActivity1</i>, and so on. You can use the getBaseName() method to assist you when you name UI controls. See <a href="#">"Using Plant Operations UI Controls" on page 27</a>.</p> <p>The default method is not provided for a GUI activity. To override the default value, you must create the method:</p> <pre>public String getBaseName() {     return "YourActivityName"; }</pre>
inputDescriptors()	Contains the input parameters.
inputItemSet(...)	Runs after the setInputItem() method is called in runtime.
outputDescriptors()	Contains the output parameters.
shutdown()	Runs when the form or event sheet shuts down the activity control it contains. This method is called regardless of whether the activity itself was ever executed.
startup()	Runs when the form or event sheet opens the activity control it contains. This does not mean that the activity was executed, only that the form or event sheet has loaded or started the control. This method is called regardless of whether the activity itself is ever executed.

## Providing the Description

Application developers need to know what your activity's purpose is, so you should provide a helpful description of the activity. Application developers see this description when they import an activity into Process Designer and when they add an activity control to a form or event sheet. Providing a detailed description allows them to see right away if the activity will provide the functionality they require.

To provide the description, add code to the `getActivityDescription()` method. The following example sets the activity's description.

```
public String getActivityDescription() {
    return "This activity retrieves a unit object from " +
        "the database. It has one input parameter, " +
        "serial number, of type String, and one output " +
        "parameter, unit.";
}
```

## Defining the Configuration Parameters

Most activities will be highly configurable, so you must define configuration parameters that the Application Developer will set. Application Developers can set the configuration parameters in Process Designer using the Customizer dialog or during runtime using the `setConfigurationItem()` method. Generally though, configuration parameters will not change during runtime. Use an input parameter instead for a value that will change during runtime. For information on retrieving configuration parameters, see [“Retrieving Configuration Parameters” on page 35](#).

---

**NOTE:** The Application Developer is responsible for providing the configuration parameters either in buildtime using the Customizer dialog or at runtime using the activity's `setConfigurationItem(...)` method.

---

To provide configuration parameters, add code to the `configurationDescriptors()` method. When you define a configuration parameter using the `ItemDescriptor` object, the method takes four arguments:

- the class of the target activity (Class)
- the name of the configuration parameter (String)
- the class of configuration parameter (Class)
- the object associated with the configuration parameter (Object)

---

**NOTE:** You cannot define a configuration parameter as a Plant Operations object. If the activity requires a Plant Operations object, define it as an input parameter instead. See [“Defining Input Parameters” on page 22](#).

---

The following example defines one configuration parameter of type String.

```
public ItemDescriptor[] configurationDescriptors() {  
    return (new ItemDescriptor[] {  
        ItemDescriptor.createItemDescriptor(  
            MyActivityNonGUI.class,  
            "serial_number",  
            String.class,  
            new Object[] {  
                ItemDescriptor.SHORTDESCRIPTION,  
                "The serial number of the object." })),  
    });  
}
```

You can define a configuration parameter that provides Application Developers with a drop-down list of values, instead of text entry, when they set it in Process Designer using the Customizer dialog. For example, if you want to save data as an integer, but you need to provide something more descriptive for the user to select from, then you can use a drop-down list. You must create an object that contains the list of values for the drop-down list, and then you use that object when you create the object associated with the configuration parameter. If you want to use constants, then you need to define those first.

When you create the object that contains the list of values for the drop-down list, the method takes three arguments:

- the text that the user sees in the drop-down list
- the actual value that is passed when the user selects a description in the list
- the name of the enumeration value

The following example defines two configuration parameters. One will use a drop-down list.

```
// The following constants are declared at the
// beginning of the class.
public static final int SALINEVALUES_FUNCTION1 = 1;
public static final int SALINEVALUES_FUNCTION2 = 2;
public static final int SALINEVALUES_FUNCTION3 = 3;

...

public ItemDescriptor[] configurationDescriptors() {

    // Create the list of values for the drop-down list.
    Object enumSaline[] =
    {
        "Slightly saline solution - 1000 to 3000 ppm",
        new Integer(SALINEVALUES_FUNCTION1),
        "MyActivityNonGUI.SALINEVALUES_FUNCTION1",

        "Moderately saline solution - 3000 to 10000 ppm",
        new Integer(SALINEVALUES_FUNCTION2),
        "MyActivityNonGUI.SALINEVALUES_FUNCTION2",

        "Highly saline solution - 10000 to 35000 ppm",
        new Integer(SALINEVALUES_FUNCTION3),
        "MyActivityNonGUI.SALINEVALUES_FUNCTION3",
    };

    // Define the configuration parameters.
    return (new ItemDescriptor[] {

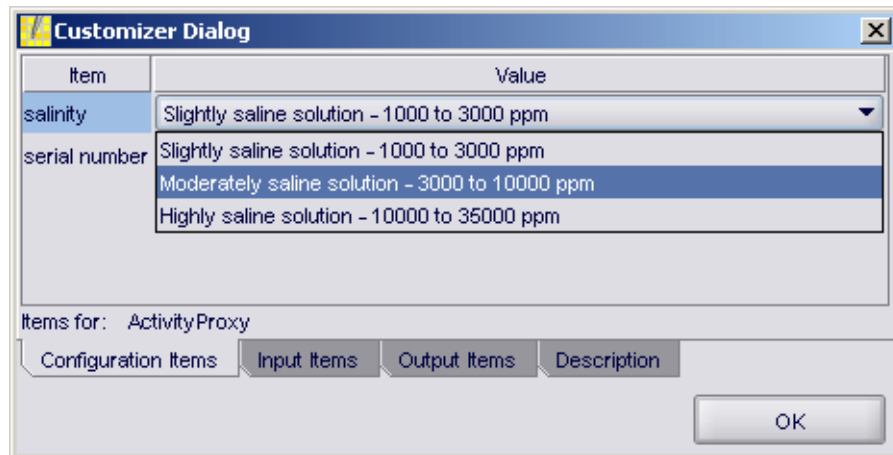
        ItemDescriptor.createItemDescriptor(
            MyActivityNonGUI.class,
            "serial number",
            String.class,
            new Object[] {
                ItemDescriptor.SHORTDESCRIPTION,
                "The serial number of the object."}),
    });
}
```

```

        ItemDescriptor.createItemDescriptor(
            MyActivityNonGUI.class,
            "salinity",
            Integer.class,
            new Object[] {
                ItemDescriptor.ENumerationValues,
                enumSaline,
                ItemDescriptor.ShortDescription,
                "The salinity of the solution."}),
        });
    }
}

```

In Process Designer, Application Developers will see:



You can also set a default value that you want Application Developers to see in the Customizer dialog when it first opens. You must set the default value in the activity's constructor using the `setConfigurationItem(...)` method of the current activity. The following code sets default values for the "serial number" field and the "salinity" drop-down list from the above example.

```

public MyActivityNonGUI() {
    this.setConfigurationItem("serial number",
        "Enter a serial number");
    this.setConfigurationItem("salinity",
        new Integer(SALINEVALUES_FUNCTION1));
}

```

In the Configuration Items tab in the Customizer dialog, Application Developers will see "Enter a serial number" as the default value for the "serial number" item and "Slightly saline solution - 1000 to 3000 ppm" as the default selected for the "salinity" item.

## Defining Input Parameters

Usually, an activity will require data, provided at runtime, to accomplish its functionality. For example, a unit's serial number. Input data can be one of three types:

- simple (for example, a String)
- a Plant Operations object instance
- a container of object instances (for example, a Vector)

---

**NOTE:** The Application Developer is responsible for providing the input parameters at runtime using the activity's `setInputItem(...)` method.

---

For information on retrieving input parameters, see [“Retrieving Input Parameters” on page 35](#). To provide input parameters, add code to the `inputDescriptors()` method. When you define an input parameter using the `ItemDescriptor` object, the method takes four arguments:

- the class of the target activity (Class)
- the name of the input parameter (String)
- the class of the input parameter (Class)
- the object associated with the input parameter (Object)

The following example defines one input parameter of type String.

```
public ItemDescriptor[] inputDescriptors() {
    return (new ItemDescriptor[] {
        ItemDescriptor.createItemDescriptor(
            MyActivityNonGUI.class,
            "serial_number",
            String.class,
            new Object[] {
                ItemDescriptor.SHORTDESCRIPTION,
                "Serial Number"}),
    });
}
```

## Defining Output Parameters

Some activities will provide output data or objects. The activity either creates the output variable or retrieves it from an external source, such as the database or the Plant Operations middle-tier. For example, the activity may create a unit or it may retrieve a unit from the database. For information on setting output parameters, see [“Setting Output Parameters” on page 36](#).

---

**NOTE:** The Application Developer is responsible for retrieving the output parameters at runtime using the activity’s `getOutputItem(...)` method.

---

To define output parameters, add code to the `outputDescriptors()` method. When you define an output parameter using the `ItemDescriptor` object, the method takes four arguments:

- the class of the target activity (Class)
- the name of the output parameter (String)
- the class of the output parameter (Class)
- the object associated with the input parameter (Object)

The following example defines one output parameter of type `Unit`.

```
// The following import statement is included at the top of
// the package.
import com.datasweep.compatibility.client.Unit;

...

public ItemDescriptor[] outputDescriptors() {

    return (new ItemDescriptor[] {
        ItemDescriptor.createItemDescriptor(
            MyActivityNonGUI.class,
            "Unit",
            Unit.class,
            new Object[] {
                ItemDescriptor.SHORTDESCRIPTION,
                "Output unit object."}),
    });
}
```

## Creating Custom Events

You can create custom events for the Application Developer to use. For example, if your activity acts as a JMS queue end-point, you may want to add an event that indicates a message has arrived in the queue.

---

**IMPORTANT:** Custom events are executed synchronously. Activity execution stops until the custom event script written in Process Designer finishes.

---

To use a custom event, you must

- create an array to hold the custom event.
- return the array in the getActivityEvents() method.
- create the arguments you want to pass, if applicable.
- fire the event, as required, using the fireActivityEvaluateScript(...) method.

The fireActivityEvaluateScript(...) method takes two arguments:

- the name of the event (String)
- the arguments to pass to the Application Developer (array)

You create the arguments using a ScriptArgument object. A ScriptArgument object has two parameters:

- the name of the argument (String)
- the value of the argument (Object)

The following example creates one script event that contains two arguments.

```
// The following import statement is included at the top of
// the package.
import com.datasweep.compatibility.ui.ScriptArgument;

...

// The following array is declared at the
// beginning of the class to hold the custom event.
private static final String[] customEvents = {"CustomEvent1"};

...
```



```
// Return the array in the getActivityEvents() method.
public String[] getActivityEvents() {
    return customEvents;
}

...

// The following code is included in the activityExecute()
// method. It creates the arguments and fires the event.
ScriptArgument[] args = new ScriptArgument[2];
args[0] = new ScriptArgument("ArgName1", "ArgValue1");
args[1] = new ScriptArgument("ArgName2", "ArgValue2");

fireActivityEvaluateScript("CustomEvent1", args);
```

When you fire the event, the event name and arguments are passed. Application Developers can access the arguments using the argument names directly as if they were already defined in their script.

---

**NOTE:** Always document your custom events and arguments. Application Developers must know under what circumstances the custom events will fire, and they must know what arguments they can expect to be able to use.

---

The following example is from Process Designer and is written in Pnuts. It displays the argument values from the previous example using the println method.

```
println("ArgName1: " + ArgName1)
println("ArgName2: " + ArgName2)
```

The following displays in the console:

```
ArgName1: ArgValue1
ArgName2: ArgValue2
```

## Adding Visual and Non-Visual Controls

You can add visual and non-visual controls to your GUI activity using Plant Operations classes or Java Swing classes. For a non-GUI activity, you can only use Java Swing classes.

Rockwell Automation strongly recommends that you use Plant Operations UI classes and controls to build your GUI activity. If you require controls not provided by Plant Operations, you can use components from the javax.swing class hierarchy, but be aware of the following limitations:

- Non-Plant Operations controls are not visible in Process Designer. Application Developers will not be able to see them, and they will also not be able to change their look and feel using the Form Designer Interface.
- SWT and AWT (Standard Widget Toolkit and Abstract Windowing Toolkit) are not supported.

## GUI versus Non-GUI Activities

GUI activities derive from the `ActivityControl` class. The `ActivityControl` class extends `Panel`.

```
com.datasweep.compatibility.ui.CComponent
├── com.datasweep.compatibility.ui.CControl
│   ├── com.datasweep.compatibility.ui.CContainer
│   │   ├── com.datasweep.compatibility.ui.Panel
│   │   └── com.datasweep.compatibility.ui.ActivityControl
```

When you add controls to a GUI activity, the GUI activity is a container for those controls. When you add a GUI activity to a form, you can see its Plant Operations controls in the Form Explorer contained in the activity, and you can modify some of their properties.

Non-GUI activities derive from the `ActivityProxy` class, but their superclass is `Activity` (see [“Creating the Package and Class” on page 16](#)).

```
com.datasweep.compatibility.ui.CComponent
└── com.datasweep.compatibility.ui.ActivityProxy
```

A non-GUI activity is not a container. You can only use Java Swing classes for its controls.

## Activities Within Activity Sets

If your activity will be contained in an activity set, keep the following in mind when scripting your activity:

- A GUI activity should not enable or disable itself. The activity set that is controlling the activity should determine when to enable (activate) or disable (deactivate) the GUI activity.
- To execute the activity, call the **complete()** method. This method will determine if the GUI activity is part of a form or an activity set and execute

the activity accordingly. The `activityExecute()` method should not be called directly if the activity is part of an activity set.

## Using Plant Operations UI Controls

You can find Plant Operations UI controls in the following classes:

- **com.datasweep.compatibility.ui:** contains UI controls such as button, edit box, and serial port.
- **com.rockwell.livedata:** contains the LiveData UI control.
- **com.rockwell.scales:** contains the scale UI control.

---

**NOTE:** If you need to have a checkbox in the activity, please use `com.datasweep.compatibility.ui.FlatCheckBox`, not `com.datasweep.compatibility.ui.CheckBox`.

---

See the Process Designer API Method Summary in the Process Designer Online Help for more information on the classes.

In Process Designer, Application Developers **cannot**:

- add controls to or delete controls from an activity.
- write scripts for the events for controls in an activity.
- copy controls that are in an activity.
- rename controls that are in an activity.
- change the visibility of controls that are in an activity.

They can change the look and feel of the visual controls in a GUI activity. For example, they can move a control or assign a message object to it. (If you used a layout style such as flow that limits how much they can move controls around, then they may not be able to move a component.)

## Selecting a Layout Style

Before you create your visual controls for a GUI activity, decide which layout style you want to use. The layout styles to choose from are:

- **Grid:** The controls are laid out in a grid of cells. Each control takes all the available space within its cell, and each cell is exactly the same size.
- **Grid2:** The controls are laid out in a grid of cells. Each control tries to maintain its preferred size, and the height and width of each grid cell can be different.
- **Flow:** The controls are laid out in either horizontal (i.e., controls are laid out left to right, and then top to bottom) or vertical (i.e., controls are laid out top to bottom, and then left to right) flows.

- **Dock and Anchor:** The controls' edges are fixed in relation to the container they are in, and each control maintains its given size regardless of the size of the container. For example, if the container is resized to be larger, the controls will not be resized to become larger.

If you decide to use *Dock and Anchor*, then you should create a mock-up of your UI in Process Designer first because you will need to set X and Y coordinates for each control. If you create a mock-up in the Form Designer Interface first, then you can use the X and Y coordinate location values from the Properties Frame when you create the controls and set their properties in your IDE.

If you use *Grid* or *Grid2*, then remember to set the number of columns and rows that should be in the grid. The default value is 2. If you use *Flow*, then remember to set the flow alignment. The default value is left.

### Naming Controls

When you create a control, you must name it. In addition, you must create a unique *activity name* for it that is used internally by the activity. The naming conventions you choose are up to you. In the example below, the following naming conventions are used:

- **name:** control's base name + constant defined for the control's ID
- **internal activity name:** activity's base name + constant defined for the control's ID

The following example creates a UI for a scan activity that contains a label and an edit control. In this example, the activity's constructor calls the `setUpGUI()` method, which was created to set the layout style, set the activity's size, and then call the methods that were created to add the controls.

```
// The following import statements are included at the top
// of the package.
import com.datasweep.compatibility.ui.LayoutStyle;
import com.datasweep.compatibility.ui.FlatLabel;
import com.datasweep.compatibility.ui.Edit;

...

// The following constants are declared at the
// beginning of the class.

// Activity height, label and edit control width.
private static final int GUI_HEIGHT = 25;
private static final int LABEL_WIDTH = 150;
private static final int EDIT_WIDTH = 100;

// IDs for the edit and label controls.
```

```

private static final String EDIT_ID = "ScannedValue";
private static final String LABEL_ID = "Scanned";

...

// In the constructor, call the method that draws the GUI.
public MyActivityGUI() {
    setUpGUI();
}

...

// The following methods are created within the class.

// Performs the setup that creates the activity's GUI.
public void setUpGUI(){

    // Set the layout style.
    setLayoutStyle(LayoutStyle.GRID2);
    setGridLayoutColumns(2);
    // Zero argument means infinite rows.
    setGridLayoutRows(0);

    // Set the size of the activity.
    setControlSize(this, LABEL_WIDTH + EDIT_WIDTH,
        GUI_HEIGHT);

    // Call the methods to add the label and edit controls.
    add(getScannerLabel());
    add(getScannerEdit());
}

...

// Creates and returns the label control.
private FlatLabel getScannerLabel() {
    String internalName = getBaseName() + LABEL_ID;

    FlatLabel label = (FlatLabel) findActivity
        Control(internalName);

    if (label == null) {
        label = new FlatLabel();
        setControlSize(label, LABEL_WIDTH, GUI_HEIGHT);
        label.setText("Enter tracking ID:");
    }
}

```

```
        label.setName(label.getBaseName() + LABEL_ID);  
        label.setActivityName(internalName);  
    }  
    return label;  
}
```

```

// Creates and returns the edit control.
private Edit getScannerEdit(){
    String internalName = getBaseName() + EDIT_ID;

    Edit edit = (Edit) findActivityControl(internalName);

    if (edit == null) {
        edit = new Edit();
        setControlSize(edit, EDIT_WIDTH, GUI_HEIGHT);
        edit.setName(edit.getBaseName() + EDIT_ID);
        edit.setActivityName(internalName);

        // Activity must implement the
        // CComponentEventListener class to include
        // the following event listener. See
        // "Writing Scripts for Plant Operations Control
        // Events on page 31.
        edit.addCComponentEventListener(this);
    }
    return edit;
}

```

In Process Designer, the activity was added to a form that already contained a label with the title, Tracking ID Form, and looks like the following:



## Writing Scripts for Plant Operations Control Events

You can write code for the events for the Plant Operations controls that you add to your user interface. Application Developers cannot do this. To add code for the events, your activity must implement the CComponentEventListener class, and you must add the listener to the triggering Plant Operations control. The

CComponentEventListener class will listen for events for the activity's controls. Your code needs to find out which control and which event were fired.

When you use the getName() method to find out the control's name, the name that you defined using setName() in [“Using Plant Operations UI Controls” on page 27](#) is returned. The following example shows you how to get the control name and event name.

```
// The following import statements are included at the top
// of the package.
import com.datasweep.compatibility.ui.CComponent;
import com.rockwell.activity.CComponentEvent;
import com.rockwell.activity.CComponentEventListener;

...

```

---

**NOTE:** When you add *implements CComponentEventListener* to the class, you will see an error if the ccomponentEventFired(...) method has not already been added to the activity. Click the error icon to add the unimplemented method.

---

```
// The following code must be included in the method that
// creates the control. In this example, it is an edit control.
// See "Creates and returns the edit control" on page 31 for the
// complete edit control creation example.
edit.addComponentEventListener(this);

// The following method is created within the class.
public Object ccomponentEventFired(CComponentEvent ev) {

    CComponent compFired = null;

    // getSource() returns an Object
    compFired = (CComponent) (ev.getSource());

    // Get the control name and event
    String strCompName = compFired.getName();
    String strEvent = ev.getEvent();

    // Business logic would follow
    return null;
}

```



## Using Non-Plant Operations Controls

If you are creating a non-GUI activity, then you must use non-Plant Operations controls. You can use non-Plant Operations controls in your GUI activity, if required. If you must use components from the `javax.swing` class hierarchy, keep in mind the following limitations:

- Non-Plant Operations controls are not visible in Process Designer. Application Developers will not be able to see them, and they will also not be able to change their look and feel using the Form Designer Interface.
- SWT and AWT are not supported.

If you want Application Developers to be able to change the look and feel of `javax.swing` components, then you can use the activity's configuration or input parameters to provide them with properties they can modify. If you want Application Developers to be able to use `javax.swing` component events, then you can capture them and call an activity's custom event. See the example, [“Using Java Swing Controls” on page 56](#).

## Providing Business Logic

The `activityExecute()` method will contain the core of the activity's business logic. You can use all of the Pnuts functions that you can use in Process Designer. The method `getFunctions()` contains all of the Pnuts functions. It returns the Functions interface documented in the Plant Operations API.

---

**IMPORTANT:** Always use the methods in `getFunctions()` in a try/catch block as an exception could be thrown.

---

Using `this.getFunctions()` could be slow in your IDE as `getFunctions()` is a large file. To see all of the available methods and prevent the system from slowing down, consider opening `com.datasweep.compatibility.pnuts.FunctionsImpl.class` in its own window.

---

**NOTE:** You do not need a separate import statement for the `FunctionsImpl` class. Access to `getFunctions()` is already provided when you import the `Activity` or `ActivityControl` class.

---

## Returning a Response Object

The `activityExecute()` method must return a `Response` object to indicate if the activity was successful. In an error condition, use `createErrorResponse(...)` to return an error response. The following example returns a `Response` object based on the outcome of the activity.

```
// The following is included in the activityExecute()
// method.

try {
    // Logic
    ...
    Response r = new Response();
    return r;

} catch (Exception e) {
    return createErrorResponse(e);
}
```

---

**IMPORTANT:** For a non-GUI activity, you must call `setLastResponse(Response)` in `activityExecute()` before returning the `Response` object. Otherwise the Application Developer will not be able to use the `getLastError()` method.

---

If Application Developers used synchronous execution, then they can access the `Response` object directly in their script. For example:

```
// This is Pnuts script in Process Designer
// activityConsumption is the name of the activity form control
resp = activityConsumption.activityExecute()
if (resp.isError()) {
    arrErrors = resp.getErrors()
    // add logic for error condition
    // ...
}
```

If they used asynchronous execution, then they must access the `Response` object through the activity. The activity returns false in an error condition. For example:

```
// This is Pnuts script in Process Designer
// activityConsumption is the name of the activity form control
bool = activityConsumption.activityStart()
if (bool == false) {
    resp = activityConsumption.getLastError()
    if (resp != null) {
        arrErrors = resp.getErrors()
        // add logic for error condition
        // ...
    }
}
```

## Retrieving Configuration Parameters

To retrieve the configuration parameters, add code to the `activityExecute()` method using the `getConfigurationItem(...)` method. If the Application Developer sets the configuration parameters in runtime rather than through the Customizer dialog in buildtime, then the activity's `configurationItemSet(...)` method will be called after they are set.

The following example retrieves the serial number configuration parameter defined in “[Defining the Configuration Parameters](#)” on page 18.

```
// The following is included in the activityExecute()
// method.

...

// Retrieve the configuration parameter.
String SN = ((String)getConfigurationItem("serial_number"));
```

---

**NOTE:** The Application Developer is responsible for providing the configuration parameters either in buildtime using the Customizer dialog or at runtime using the activity's `setConfigurationItem(...)` method. You do not know if the Application Developer will set the configuration parameters, so always handle null values.

---

## Retrieving Input Parameters

To retrieve the input parameters, add code to the `activityExecute()` method using the `getInputItem(...)` method. After the Application Developer sets the input parameters in runtime, the activity's `inputItemSet(...)` method will be called.

The following example retrieves the serial number input parameter defined in “[Defining Input Parameters](#)” on page 22, and then retrieves a unit.

```
// The following is included in the activityExecute()
// method.

...

// Retrieve the input parameter.
String SN = ((String)getInputItem("serial_number"));

// Get a unit by serial number.
Unit objUnit = null;
try {
    objUnit = getFunctions().getUnitBySerialNumber(SN);
}
catch (Exception e){
    return createErrorResponse(e);
}
```

---

**NOTE:** The Application Developer is responsible for providing the input parameters at runtime using the activity's `setInputItem(...)` method. You do not know if the Application Developer will set the input parameters, so always handle null values.

---

## Setting Output Parameters

To set the output parameters, add code to the `activityExecute()` method using the `setOutputItem(..)` method. The following example sets the Unit output parameter defined in “[Defining Output Parameters](#)” on page 23.

```
// The following is included in the activityExecute()
// method.

...

// Get a unit by serial number.
Unit objUnit = null;
try {
    objUnit = getFunctions().getUnitBySerial
        Number("RA-1234");
}
catch (Exception e){
    return createErrorResponse(e);
}
// Set the output item.
if (objUnit != null){
    setOutputItem("Unit", objUnit);
}

...
```

---

**NOTE:** The Application Developer is responsible for retrieving the output parameters at runtime using the activity's `getOutputItem(...)` method.

---

## Managing Lifecycles

If an Application Developer executes your activity in an asynchronous thread, the activity executes on its own, new thread and current application execution continues. When an activity is executed this way, it has a lifecycle. Based on its lifecycle, an activity has a state. The state can be used to determine if the activity is busy, holding, ready to perform a task, and so on.

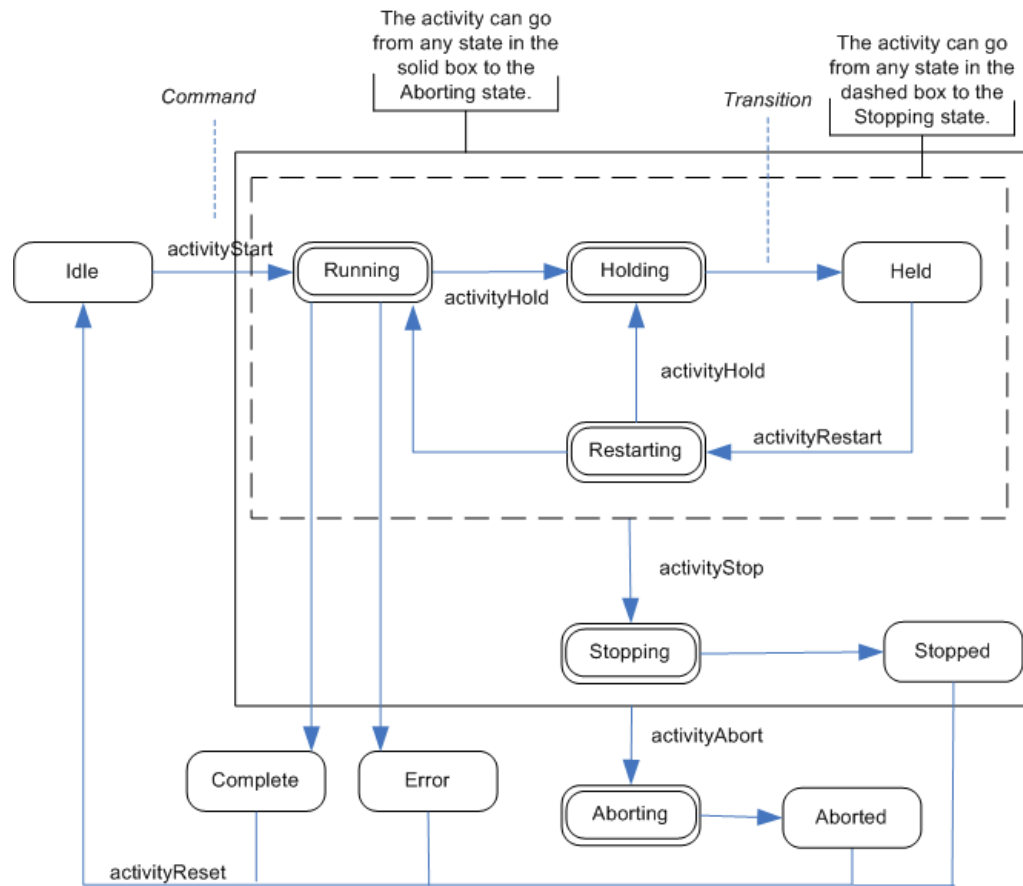
An activity has two types of states:

- **Acting states:** represent what the activity is actually doing at a given time, until certain conditions are met.
- **Waiting states:** represent the activity when it is between acting states, after certain conditions were met, and it is waiting to go to the next state.

**Table 2-3 Acting and Waiting States**

Acting States	Waiting States
Aborting	Aborted
Holding	Complete
Restarting	Error
Running	Held
Stopping	Idle
	Stopped

The following diagram shows the activity state model. The activity state model dictates to which states an activity can go from its current state. For example, you can only start an activity when its current state is Idle.



An activity automatically goes to the Complete state when it successfully completes the conditions of the Running state. It automatically goes to the Error state when something unexpected occurs or when the Activity Developer returns an error response from `activityExecute()`. With asynchronous execution, an Application Developer can use the following methods to attempt to transition an activity to a different state. The thread controls the behavior of the activity, as described below, in all cases except `stop()`.

**activityAbort:** Aborting, then Aborted. This shuts down the activity's thread.

**activityHold:** Holding, then Held. This pauses the activity's execution thread and may result in a deadlock.

**activityReset:** Idle. This resets the activity to Idle. They can only use this method if the activity's current state is Complete.

**activityRestart:** Restarting, then Running. This resumes the activity's paused thread's execution. They can only use this method if the activity's current state is Held.

**activityStart:** Running. They can only use this method when the activity's current state is Idle.

**activityStop:** Stopping, then Stopped. Unlike the other methods, the stop method's behavior is provided by the Activity Developer. It is up to the Activity Developer to decide whether to provide any business logic for this method. If the activity does not honor the request to stop, then execution may continue and OnComplete and OnError may fire.

The Application Developer can also write a script for the following events:

- OnAborted
- OnAborting
- OnComplete
- OnError
- OnHeld
- OnHolding
- OnIdle
- OnRestarting
- OnRunning
- OnStopping
- OnStopped

It is up to you as the Activity Developer to decide whether the activity will monitor its current state and honor the stop request. If required, you can monitor the state and provide logic for the other state change requests, but stop is the only one where the thread itself does not respond to the request. If you provide logic for any state change requests, then you should document this thoroughly so that Application Developers are aware of it.

---

**IMPORTANT:** Always document whether the activity provides stop request behavior, and if it does, what the behavior is.

---

To get the state of the activity, use the getState() method in activityExecute():

```
int activityState = myActivity.getState();
```

The following table summarizes the integer constants provided by IActivity:

**Table 2-4 Activity States**

State	Constant Name	Integer Assignment
Aborted	STATE_ABORTED	9
Aborting	STATE_ABORTING	8
Complete	STATE_COMPLETE	2
Error	STATE_ERROR	10
Held	STATE_HELD	4
Holding	STATE_HOLDING	3
Idle	STATE_IDLE	0
Restarting	STATE_RESTARTING	5
Running	STATE_RUNNING	1

Table 2-4 Activity States

State	Constant Name	Integer Assignment
Stopped	STATE_STOPPED	7
Stopping	STATE_STOPPING	6

---

**NOTE:** When working with activity state constants in code, constant names should be used instead of actual integers. Integer assignments to the names are subject to change, but constant names are not.

---



## Testing and Distributing Activities

### In this chapter

- ❑ **Configuring Default Tab Behavior (Optional)** 42
- ❑ **Testing Activities in Process Designer** 42
- ❑ **Testing Activities in Shop Operations** 42
- ❑ **Running an Activity from the IDE** 43
- ❑ **Adding an Activity to a Form or Event Sheet** 46
- ❑ **Testing and Debugging an Activity** 47
- ❑ **Distributing Activities** 48
  - Setting Up Member Help for Activities 49
- ❑ **Updating Activities** 50

After you create your activity, you need to test and then distribute it. You can test your activity in Process Designer and Shop Operations directly from your development environment (IDE) without having to create an activity object in Process Designer for the activity.

---

**IMPORTANT:** If your activity depends on third-party jar files, you must import those into Process Designer first, using the library object. See the *Process Designer and Objects Help* for more information on using libraries.

---

## Configuring Default Tab Behavior (Optional)

By default, applications launched in Shop Operations and Process Designer exhibit the Metal Look and Feel (LAF) for the Tab key behavior. This results in some unexpected behavior if you are used to working in an environment with the Windows LAF. To configure the Tab key behavior to reflect a Windows LAF, see the *FactoryTalk ProductionCentre Server Installation Guide* specific to your server for additional instructions.

## Testing Activities in Process Designer

You can test your activity by running Process Designer from within your IDE. If you create an activity object in Process Designer, and then test the activity directly from there, you will not be able to see breakpoints that you set in your activity's code. Testing from your IDE allows you to fully debug and step through your code.

To test your activity in Process Designer, perform the following steps.

1. Compile your activity class, if it is not already compiled.
2. Run the activity in Process Designer from within your IDE. See [“Running an Activity from the IDE” on page 43](#).
3. Add the activity to a form or event sheet. See [“Adding an Activity to a Form or Event Sheet” on page 46](#).
4. Test and debug the activity. See [“Testing and Debugging an Activity” on page 47](#).

## Testing Activities in Shop Operations

You can test your activity by running Shop Operations from within your IDE. To test your activity in Shop Operations, perform the following steps.

1. Compile your activity class, if it is not already compiled.

2. Run the activity in Process Designer from within your IDE so that you can add the activity to a form or event sheet. See [“Running an Activity from the IDE” on page 43](#). If you have already run the activity from your IDE in Process Designer and added it to a form or event sheet, then you can skip to Step 3.
3. Run the activity in Shop Operations from within your IDE. See [“Running an Activity from the IDE” on page 43](#).

## Running an Activity from the IDE

From your IDE, you can run the activity directly in either Process Designer or Shop Operations. The first time you do this, you must configure a Java application. The steps below are the same for creating a Process Designer or Shop Operations configuration, except as indicated.

1. Open a Run Dialog:
  - For Eclipse, perform the following steps:
    - a. Select Run > Open Run Dialog.
    - b. In the Run Dialog, right-click Java Application, and then select New.
  - For RAD, perform the following steps:
    - a. Select Run > Run.
    - b. In the Run dialog, select Java Application from the Configurations list, and then click [New].
2. In the Name field:
  - if you are creating the configuration for Process Designer, then enter Process Designer.
  - if you are creating the configuration for Shop Operations, then enter Shop Operations.
3. Click the Main tab, and then enter the following:
  - a. Project: enter the name of your project
  - b. Main class:
    - If you are creating the configuration for Process Designer, then enter `com.datasweep.compatibility.buildtime.Buildtime`.
    - If you are creating the configuration for Shop Operations, then enter `com.datasweep.compatibility.runtime.Runtime`.
4. Click the Arguments tab.
  - a. In the *Program arguments* field,
    - for WebSphere, enter:

```
iiop://<appserver_hostname>:2809
http://<appserver_hostname>:9080
http://<appserver_hostname>/PlantOperations
```

where *<appserver\_hostname>* is the name of the host running the applications server.

---

**NOTE:** 2809 is the WebSphere server's default bootstrap port. This port may be different for your configuration.

---

- for a JBoss application server, enter:

```
remote://<appserver_hostname>:8080
http://<appserver_hostname>:8080
http://<appserver_hostname>/PlantOperations
```

where *<appserver\_hostname>* is the name of the host running the applications server.

- In the *VM arguments* field,

- for WebSphere, enter:

```
-Xbootclasspath/p:"<IBM_JRE_install_path>
\java\jre\lib\ibmorb.jar;<IBM_JRE_install_path>\properties"
-Xmx512m
-Dcom.ibm.CORBA.ConfigURL=http://<machine_name>:
<port_number>/PlantOperations/sas.client.props
-Dcom.ibm.CORBA.Debug.Output=c:\orbtrc.txt
-Dcom.datasweep.plantops.j2eevendor=WebSphere
-Dserver.root="<IBM_JRE_install_path>"
-Djava.ext.dirs="<IBM_JRE_install_path>\jre\lib\ext";"<IBM_JRE_install_path>\jre\lib";
"<IBM_JRE_install_path>\lib"; "<IBM_JRE_install_path>
\plugins";"<IBM_JRE_install_path>\properties";"<project_worksp
ace_path>";"<download_directory>\ProductionCentre\ProcessDesi
gner"
-DuiDefaultButtonFollowFocus=true
-Djava.library.path=<nativelibs_path>
-Drockwell_client_dir=<download_directory>
```

where:

- *<IBM\_JRE\_install\_path>* is the location where the IBM JRE was installed. For example, C:\<version>\jre\lib\ext.
- *<machine\_name>* is the name of the machine where FTPC and WebSphere Application Server are installed.

- `<port_number>` is the HTTP server port number, such as 80.
- `<project_workspace_path>` is the directory that contains your activity class.
- `<nativelibs_path>` is the directory where the native files were extracted (i.e., `<extraction_location>\pcclient\native`).
- `<download_directory>` is the directory where the activity jar files are downloaded. This is `C:\FTPC\AppServer` by default. This directory can be configured. See the *Plant Operations Server Installation Guide for WebSphere* for details.

---

**TIP:** The `-Dcom.ibm.CORBA.Debug.Output=c:\orbtcr.txt` option is optional and is used to debug the WebSphere environment.

The `-DuiDefaultButtonFollowFocus=true` argument is optional and used to support the Windows look and feel for Tab key behavior.

---

- for a JBoss application server, enter `-Xmx512m`  
`-Dcom.datasweep.plantops.j2eevendor=JBoss`  
`-Djava.ext.dirs="<library_directory>";"<jdk>\jre\lib\ext";`  
`"<project_workspace_path>"`  
`-Djava.library.path="<nativelibs_path>"`  
`-Drockwell_client_dir=<download_directory>`  
`-DuiDefaultButtonFollowFocus=true`  
`-Dorg.apache.activemq.SERIALIZABLE_PACKAGES=*`

where:

- `<library_directory>` is the JBoss directory that contains the `jboss-client.jar` file. For example, `C:\JBoss\jboss-<version>\client`.
- `<project_workspace_path>` is the directory that contains your activity class.
- `<jdk>` is the JDK installation location. For example, `C:\<version>\jre\lib\ext`.
- `<nativelibs_path>` is the directory where the native files were extracted (i.e., `<extraction_location>\pcclient\native`).
- `<download_directory>` is the directory where the activity jar files are downloaded. This will depend on your JBoss environment:

**JBoss StandAlone:** `C:\FTPC\<serverName>`

**JBoss Advanced:** `C:\FTPC\AppServer` by default. This directory can be configured. See the *Plant Operations Server Installation Guide for JBoss Advanced* for details.

---

**IMPORTANT:** Note the following:

-Xmx512m should be on a separate line.

-Djava.ext.dirs="*<library\_directory>*";"-jdk\jre\lib\ext";"*<project\_workspace\_path>*" should all be on one line.  
Otherwise, Process Designer cannot be opened via the development environment.

---



---

**TIP:** The -DuiDefaultButtonFollowFocus=true argument is optional and used to support the Windows look and feel for Tab key behavior.

---

5. If you are using WebSphere, then you must perform the following steps:
  - a. Click the Classpath tab.
  - b. Select *User Entries*.
  - c. Click [Add External JARS...].
  - d. Select all of the JARS in the IBM JRE's \AppClient\lib folder that was created as part of [Step 4](#) in “[The Application Framework SDK ZIP File](#)” on page 10, and then click [Open].
6. Click [Debug].
7. Log on to Process Designer or Shop Operations.

## Adding an Activity to a Form or Event Sheet

After Process Designer opens from your IDE, you must add the activity to a form or event sheet so that you can test it.

To add the activity to a form:

1. Open a form, or create a new form.
2. In the toolbox, click [Activity].
3. Select a GUI (ActivityControl) or non-GUI (ActivityProxy) activity control.
4. Place your mouse pointer on the form in the location for the selected activity control, and then click the mouse.

The Activity Control Selector dialog opens, and all the activity class names are displayed (not the activity object names).

5. Select the class name associated with the activity you want to use, and then click [OK].

The activity is added to the form.

6. Add script to the form to launch the activity. For example, to launch the activity synchronously, use `<activity_name>.activityExecute()`. See *Executing Activities* in the Process Designer and Objects Help for more detailed information on executing activities. See *Using Activities* for more information on setting configuration parameters, retrieving output parameters, and so on.

To add the activity to an event sheet:

1. Open an event sheet, or create a new event sheet.
2. In the Event Explorer, click Event Actions, and then select Add Activity Event.

The Activity Control Selector dialog opens, and all the activity class names are displayed (not the activity object names).

---

**IMPORTANT:** Only activities that are type Non-GUI Activity are displayed.

---

3. Select the class name associated with the non-GUI activity you want to use, and then click [OK].

The activity is added to the event sheet.

4. Add script to the event sheet to launch the activity. For example, to launch the activity synchronously, use `<activity_name>.activityExecute()`. See *Executing Activities* in the Process Designer and Objects Help for more detailed information on executing activities. See *Using Activities* for more information on setting configuration parameters, retrieving output parameters, and so on.

## Testing and Debugging an Activity

After you have added the activity to a form or event sheet, you can test and debug the activity by running the form or event sheet.

To test and debug the activity:

1. Open the form or event sheet that the activity is on.
2. Click Run.

The form or event sheet will run.

All of the Process Designer debugging tools are available for the supporting script you write in Process Designer. In addition, any debugging tools you have in your IDE are available for the actual activity code.

## Distributing Activities

Rockwell Automation recommends that Activity Developers create the activity object in Process Designer, export the activity object in a DSX file, and then distribute the activity to Application Developers via that DSX file. Application Developers should not create the activity object.

Activity Developers should create the activity object to ensure that:

- the class name and activity type are set correctly.
- that they have a version of the activity object to maintain if they need to update the activity.

When you create an activity object and associate it with an activity, a unique Globally Unique Identifier (GUID) is assigned. If you need to update the activity, then you must maintain the same GUID. The easiest way to make sure the GUID is the same is to maintain your original activity object in Process Designer, and then only make changes using and distributing that object. See [“Updating Activities” on page 50](#) for more information on updating activities.

To distribute an activity:

1. Create the activity’s JAR file.
2. Create the activity object in Process Designer. Make sure the class name and type are set correctly. The class name must be the fully-qualified class path, including the package name. For example, `com.mycompany.services.activities.MyGUIActivity`. See *Creating Activities* in the Process Designer and Objects Help for more information on creating activity objects.
3. Browse for the activity’s JAR file.
4. Export the activity object to a DSX file.
5. Distribute the activity to Application Developers using the DSX file, and include any documentation they will require to use the activity.

---

**IMPORTANT:** You must provide documentation for your activity to Application Developers. In addition to documenting the activity’s functionality, make sure to document custom events, state change requests logic, the type of execution recommended, and whether Plant Operations form controls were used.

---


Following this procedure ensures that all of the activity’s properties are specified correctly in the activity object and that you will be able to maintain the activity’s GUID if you need to update it later.



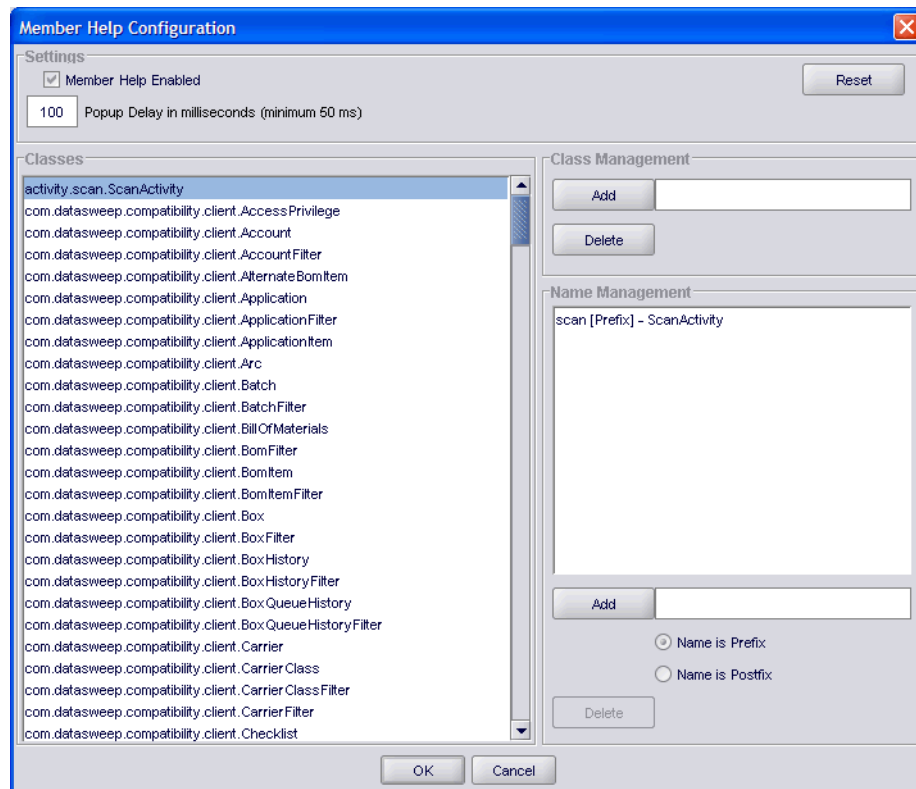
## Setting Up Member Help for Activities

Member Help is a Process Designer feature that provides automatic completion as you write scripts using prefix or postfix keywords. See *Member Help (Intellisense)* in the Process Designer and Objects help for more information on Member Help. To make scripting simpler for Application Developers, you can set up Member Help for your activity. If you set up Member Help, remember to provide Application Developers with the prefix or postfix to use as part of the activity's documentation.

To set up Member Help:

1. In Process Designer, open a form, event sheet, or subroutine. If you open a form, then click the Script tab to change to script mode.
2. Open the Member Help Configuration dialog by clicking the Setup Member Help button in the toolbar. 

**Figure 3-1: Member Help Configuration Dialog**

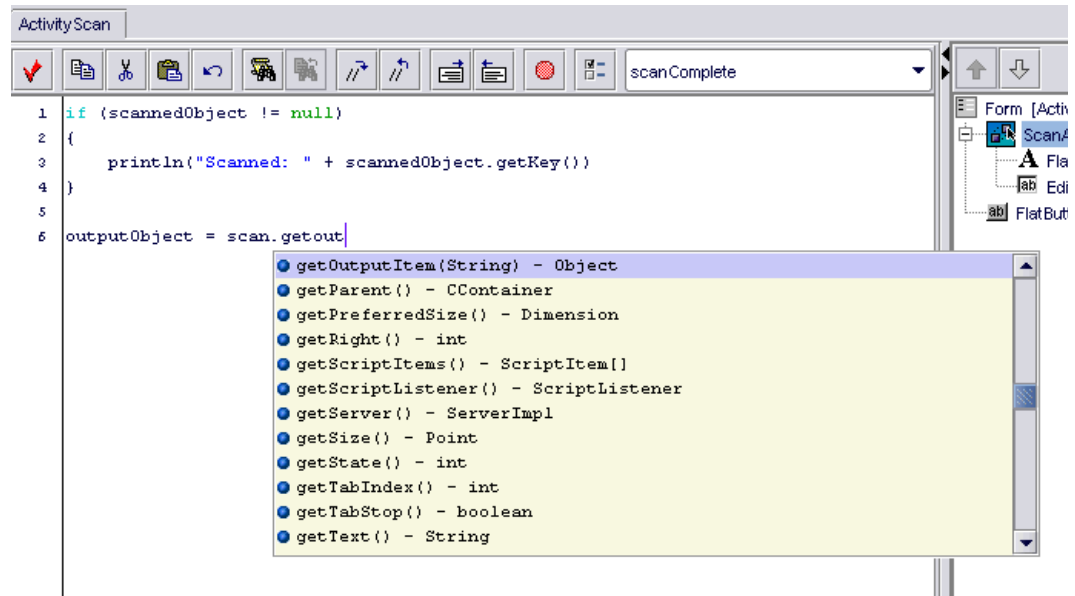


3. To add the activity class, in the Class Management section of the dialog, enter the same fully-qualified class path you entered when you created the activity object, and then click [Add].
4. To add a shortcut keyword, select the class you just created from the Classes list, enter the shortcut keyword in the Name Management section of the dialog, and then click [Add]. Select **Name is Prefix** or **Name is Postfix**

depending on where you want the shortcut keyword to appear in variable names.

5. When Application Developers write script using the prefix or postfix you defined, the Member Help pop-up list will appear with a list of methods they can call for the activity.

**Figure 3-2: Member Help Pop-up List**



## Updating Activities

If you need to update an activity:

- Do not change the class name.
- Maintain compatibility of properties, methods, and events.
- Notify Application Developers of all of your changes so that they can update their applications accordingly.

Remember that Application Developers have already used your activity in their applications. If you change the functionality, then they may need to update their applications. Consider creating a new activity instead if the changes will be significant and the impact on the Application Developers will be high.

In Process Designer, each activity object has a unique GUID associated with it. If you update your activity, then its associated activity object must maintain this GUID. To update an activity and maintain its GUID:

1. Update the JAR file in your Process Designer environment where the original activity object was created. See [“Distributing Activities” on page 48](#).
2. Export the activity object in a DSX file.

3. Distribute the updated activity DSX file to the Application Developers. Be sure to provide details of all of your changes so that they can update their applications.



## Activities Code Examples

### In this chapter

#### ❑ Non-GUI Activities 54

Creating a Basic Hello World Activity 54

Creating Activities to Interface with Other Systems 55

#### ❑ GUI Activities 56

Using Java Swing Controls 56

This chapter provides code examples that you can use to build your application. The code examples are written in Java. The steps and syntax may be different in your environment.

## Non-GUI Activities

This section provides examples of non-GUI activities.

### Creating a Basic *Hello World* Activity

This example will display a *Hello World* dialog. It is a very basic non-GUI activity.

---

**IMPORTANT:** Because this non-GUI activity displays a dialog, you can use it with a form, but you cannot use it with an event sheet.

---

After you create your non-GUI activity class, add the following code.

The `activityExecute()` method should look like:

```
public Response activityExecute() {
    getFunctions().dialogOk("Title", "Hello World!");
    Response r = new Response();
    return r;
}
```

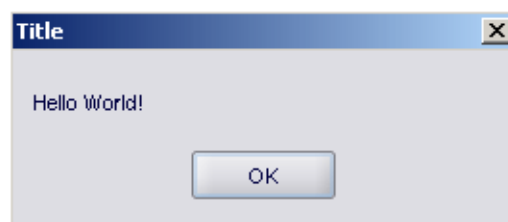
The `getActivityDescription()` method should look like:

```
public String getActivityDescription() {
    return "This activity opens a Hello World dialog. " +
        "It has no configuration, input, or output parameters.";
}
```

In Process Designer, add the activity to a form, and call it *actHelloWorld*. Add the following line of code to the form's activate event:

```
actHelloWorld.activityExecute()
```

When you run the form, the dialog displays:



## Creating Activities to Interface with Other Systems

You can create activities that connect to WebSphere MQ as a mechanism to interface with other systems. For example, if you need to send and receive information to and from xCoupler using WebSphere MQ as a transport mechanism, then you can create two activities to handle this. One activity will send messages to WebSphere MQ, and the other will receive them. The activity that sends messages will require configuration or input parameters, and the activity that receives messages will require custom events. The activities will run on an event sheet on Shop Operations Server.

---

**IMPORTANT:** In addition to creating activity objects for your WebSphere MQ activities, you also must create library objects for the WebSphere IBM Queue JAR files used by your activities.

---

The FTPC online knowledge base contains code examples for WebSphere MQ activities. Contact Rockwell Automation for a user name and password to access the knowledge base.

The basic steps are:

1. Configure the system to communicate with WebSphere MQ. For example, with xCoupler, you must use the xCoupler WorkBench Configuration Utility.
2. In Process Designer, add the WebSphere IBM Queue third-party JAR files using the library object. See *Creating Libraries* in the Process Designer and Objects Help for more information on creating libraries.
3. Using Java, create the WebSphere MQ activities. See the knowledge base for the Java code examples.
4. Configure the activities in Process Designer:
  - a. Add the activities using the activity object. See *Creating Activities* in the Process Designer and Objects Help for more information on creating activities.
  - b. Add the activities to forms or event sheets. See *Form Control - Activities* and *ActivityEvents Container* in the Process Designer and Objects Help for more information on adding activities.
  - c. Configure the activity that sends messages using the configuration or input parameters defined in the activity. See *Setting Configuration Parameters* in the Process Designer and Objects Help for more information on how to set configuration parameters, and see the knowledge base for the Pnuts code example.
  - d. Add Pnuts script to handle the custom events for the activity that receives messages and execute the activities. See *Scripting Events* in the Process Designer and Objects Help for more information on how to set

configuration parameters, and see the knowledge base for the Pnuts code example.

## GUI Activities

This section provides examples of GUI activities.

### Using Java Swing Controls

If you use components from the `javax.swing` class hierarchy, keep in mind the following limitations:

- Non-Plant Operations controls are not visible in Process Designer. Application Developers will not be able to see them, and they will also not be able to change their look and feel using the Form Designer Interface.
- SWT and AWT are not supported.

If you want Application Developers to be able to change the look and feel of `javax.swing` components, then you can use the activity's configuration parameters to provide them with properties they can modify. If you want Application Developers to be able to use `javax.swing` component events, then you can capture them and call an activity's custom event.

This example creates an activity (*RoundButtonActivity*) with two `javax.swing` buttons. It uses the *RoundJButton* class that you must also create. See [“Creating the RoundJButton Class” on page 60](#). Configuration parameters are created that allow the Application Developer to modify the text on the buttons. Custom events are created that are fired based on the button's events. After you create your non-GUI class, add the following code.

1. Add the following import statements to the import statements at the top of the package.

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import com.datasweep.compatibility.ui.LayoutStyle;
import com.rockwell.activity.CComponentEvent;
import com.rockwell.activity.CComponentEventListener;
```



2. To use custom events, modify the class to implement `ComponentEventListener`.

```
public class RoundButtonActivity extends ActivityControl
implements CComponentEventListener {

    // ...

}
```

---

**NOTE:** When you add *implements CComponentEventListener* to the class, you may see an error if the `ccomponentEventFired(...)` method has not already been added to the activity. Click the error icon to add the unimplemented method.

---

3. Declare the following variables at the beginning of the class.

```
private static final String DESCRIPTION = "description";
private static final String OK_BUTTON_TEXT = "OkButtonText";
private static final String
CANCEL_BUTTON_TEXT = "CancelButtonText";
private static final String OK_BUTTON_NAME = "OkButton";
private static final String
CANCEL_BUTTON_NAME = "CancelButton";

private RoundJButton cancelButton;
private RoundJButton okButton;

private static final String
CANCEL_BUTTON_SCRIPT = "cancelButton";
private static final String OK_BUTTON_SCRIPT = "okButton";
```

4. In the constructor, call the method that draws the GUI.

```
public RoundButtonActivity() {
    super();
    setup();
}
```

5. Create the method that draws the GUI.

```
public void setup() {
    setLayoutStyle(LayoutStyle.GRID2);
    setGridLayoutColumns(2);
    setGridLayoutRows(1);
    getJComponent().add(getOKButton());
    getJComponent().add(getCancelButton());
}
```

6. Create the methods that get the buttons. The methods also capture the button click, and then call the applicable custom event. The custom events are created in [step 10 on page 60](#).

```
private RoundJButton getOKButton() {
    if (okButton == null) {
        okButton = new RoundJButton();
        okButton.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                fireActivityEvaluateScript
                    (OK_BUTTON_SCRIPT, null);
            }
        });
        okButton.setName(OK_BUTTON_NAME);
        okButton.setText(OK_BUTTON_TEXT);
    }
    return okButton;
}

private RoundJButton getCancelButton() {
    if (cancelButton == null) {
        cancelButton = new RoundJButton();
        cancelButton.addActionListener(new
            ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    fireActivityEvaluateScript
                        (CANCEL_BUTTON_SCRIPT, null);
                }
            });
    }
}
```

```

        cancelButton.setName(CANCEL_BUTTON_NAME);
        cancelButton.setText(CANCEL_BUTTON_TEXT);
    }
    return cancelButton;
}

```

7. Add functionality to the `activityExecute(...)` method.

```

// Main functionality of the activity.
public Response activityExecute() {
    // Wait to simulate some functionality
    try {
        Thread.sleep(2000);
    }
    catch (InterruptedException e) { /* IGNORE */ }

    return new Response();
}

```

8. Retrieve the configuration parameters and modify the button text. The buttons are *okButton* and *cancelButton*. Remember to change the arguments.

```

protected void configurationItemSet(String key, Object value){
    if (key.equals(OK_BUTTON_TEXT)) {
        okButton.setText((String) value);
    }

    if (key.equals(CANCEL_BUTTON_TEXT)) {
        cancelButton.setText((String) value);
    }
}

```

9. Add the description.

```

public String getActivityDescription() {
    return DESCRIPTION;
}

```

10. Create one custom event for *okButton* and one for *cancelButton*. The events are designed to fire when a user clicks the button.

```
protected String[] getActivityEvents() {
    return new String[] {OK_BUTTON_SCRIPT,
        CANCEL_BUTTON_SCRIPT};
}
```

11. Create the configuration parameters.

```
public ItemDescriptor[] configurationDescriptors() {
    return new ItemDescriptor[] {
        new ItemDescriptor(OK_BUTTON_TEXT,
            RoundButtonActivity.class, String.class),
        new ItemDescriptor(CANCEL_BUTTON_TEXT,
            RoundButtonActivity.class, String.class)
    };
}
```

### Creating the RoundJButton Class

The *RoundButtonActivity* class uses the *RoundJButton* class. Create a *RoundJButton* class that contains the following code:

```
package com.rockwell.activity.display;

import java.awt.AWTEvent;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.event.MouseEvent;
import java.awt.image.ImageObserver;

import javax.swing.JButton;
```

```

public class RoundJButton extends JButton {

    // True if the button is pressed.
    protected boolean pressed = false;

    // Constructs a RoundButton with the specified label.
    // @param label - the label of the button
    public RoundJButton() {
        enableEvents(AWTEvent.MOUSE_EVENT_MASK);
    }

    // Paints the RoundButtons.
    public void paint(Graphics g) {
        int s = Math.min(
            getSize().width - 1,
            getSize().height - 1);
        g.setColor(getBackground());
        g.fillRect(0,0,getSize().width - 1,
            getSize().height - 1);

        // Paint the interior of the button.
        if (pressed) {
            g.setColor(getBackground().darker().
                darker());
        } else {
            g.setColor(getBackground());
        }
        g.fillArc(
            0,
            0,
            s,
            s,
            0,
            360);
    }
}

```

```
// Draw the perimeter of the button.
g.setColor(getBackground().darker().darker().
darker());
g.drawArc(
    0,
    0,
    s,
    s,
    0,
    360);

// Draw the label centered in the button.
Image img = Toolkit.getDefaultToolkit().
getImage(getClass().getResource(
"coloricon.gif"));

int center = s / 2;
int width = s / 2;
int startpos = center - (width/2);
g.drawImage(img, startpos,startpos,width,width,
new ImageObserver() {
    public boolean imageUpdate(Image img, int
infoflags, int x, int y, int width, int
height) {
        System.out.println("test");
        return false;
    }
});

Font f = new Font("Arial", Font.BOLD+Font.
ITALIC, center);
setForeground(Color.red);
```

```
        if (f != null) {  
            FontMetrics fm = getFontMetrics(getFont());  
            g.setColor(getForeground());  
            g.drawString(  
                getText(),  
                s / 2 - fm.stringWidth(getText()) / 2,  
                s / 2 + fm.getMaxDescent());  
        }  
    }  
}
```





# Index

## A

activity  
     asynchronous 8  
         lifecycle 36  
         state model 37  
     creating checklist 14  
     description 18  
     distributing 48  
     examples. See code examples  
     member help 49  
     synchronous 8  
     testing in Process Designer 42  
     testing in Shop Operations 42  
     updating 50  
     WebSphere MQ 55  
     xCoupler 55  
 Activity Developer 8  
 activity sets 8  
 ActivityControl class 26  
 activityExecute() 17  
 ActivityProxy class 26  
 adding controls 25  
 Application Developer 8  
 Application Framework SDK jar file 10  
 asynchronous execution 8

## B

business logic 33

## C

CComponent error 32  
 classes, creating 16  
 code examples  
     configuration parameters

        defining 19  
         defining multiple 20  
         retrieving 35  
         setting default values 21  
         using a drop-down list 20  
 custom events  
     creating 24  
     firing 25  
     retrieving in Pnuts 25  
 description, providing 18  
 GUI, Java Swing controls 56  
 input parameters  
     defining 22  
     retrieving 35  
 non-GUI, basic Hello World 54  
 output parameters  
     defining 23  
     setting 36  
 Plant Operations controls  
     creating 28  
     scripting events 32  
 Response object  
     accessing in Pnuts 34  
     returning 34  
     state, retrieving 39  
 configuration parameters  
     creating drop-down list 19  
     defining 18  
     retrieving 35  
     setting default values 21  
 configurationDescriptors() 17  
 configurationItemSet(...) 17  
 configurationLoaded() 17  
 controls  
     Java Swing 33

- limitations 33
- Plant Operations 27
  - limitations 27
  - writing scripts for events 31
- creating
  - classes 16
  - configuration parameters 18
  - custom events 24
  - input parameters 22
  - output parameters 23
  - packages 16
  - projects 14
- creating activities checklist 14
- custom events
  - creating 24
  - retrieving arguments 25
- D**
  - debugging activity 47
  - default methods 16
  - defining
    - configuration parameters 18
    - input parameters 22
    - output parameters 23
  - development environments, supported 10
  - distributing activities 48
  - downloading Application Framework SDK jar file 10
- E**
  - event sheet, adding activity 46
  - examples. See code examples
  - execution
    - asynchronous 8
    - synchronous 8
- F**
  - form, adding activity 46
- G**
  - getActivityDescription() 17
  - getActivityEvents() 17
  - getBaseName() 17
  - getFunctions() 33
  - GUI activity 14
    - adding controls 25
    - class 26
- I**
  - input parameters
    - defining 22

- retrieving 35
- inputDescriptors() 17
- inputItemSet(...) 17

**J**

- Java Swing controls 33
  - limitations 25
- JRE 14

**L**

- lifecycle 36
  - state model 37
  - states 37
- localization 10
- logic, activity 33

**M**

- managing lifecycles 36
- member help and activities 49

**N**

- non-GUI activity 14
  - class 26

**O**

- output parameters
  - defining 23
  - setting 36
- outputDescriptors() 17

**P**

- packages, creating 16
- Plant Operations controls 27
  - writing scripts for events 31
- projects, creating 14
- providing business logic 33

**R**

- related documents 6
- Response object 33
- retrieving
  - configuration parameters 35
  - input parameters 35
- returning a Response object 33
- running activity from IDE 43

**S**

- scripting Plant Operations control events 31
- setting output parameters 36
- shutdown() 17

- startup() 17
- state model 37
- supported development environments 10
- synchronous execution 8

## T

- Tab Behavior 42
- technical support 6
- testing
  - adding activity to form or event sheet 46
  - debugging activity 47
  - in Process Designer 42
  - in Shop Operations 42
  - running activity from IDE 43
- third-party jar files 42

## U

- updating activities 50
- using Plant Operations UI controls 27

## W

- WebSphere MQ activity 55

## X

- xCoupler activity 55

