



This is Unit #10 of the BPEL Fundamentals course. In past Units we've looked at ActiveVOS Designer, Projects and the Process itself. Next, we looked at Global declarations, Interaction activities and Sequences. Finally, in the last two units we studied Assignments and Correlation. In this Unit we will take a look at a new topic, the Scope activity.

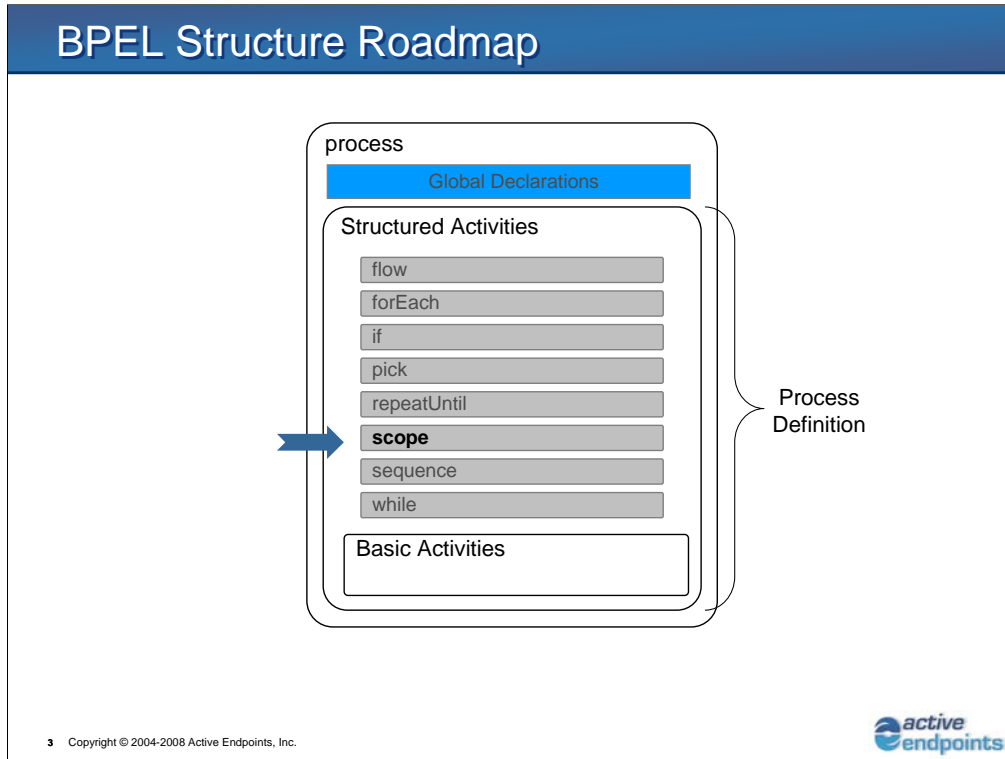
Unit Objectives

- At the conclusion of this unit, you will be familiar with:
 - scope activity
 - Working with scopes

2 Copyright © 2004-2008 Active Endpoints, Inc.



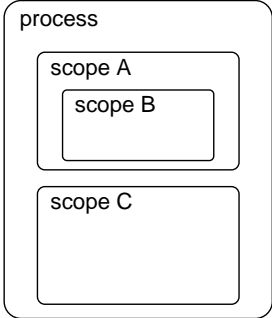
A Scope is a structured activity, and a container activity, in that it can contain other activities.




Scopes allow us to break up our business processes into *logical units of work*. Scopes provide a context for the execution and/or documentation of enclosed activities, and they can have variables that are visible and usable at and within the scope level. Scopes can have both default and defined Fault and Event handling logic, and they can be undone, if necessary. Undoing the work of a Scope involves the concept of *compensation*, which will be covered in a later unit. When designing your BPEL processes they should be organized into logical units of work that can be undone.

scope Activity Overview

- Used to partition a business process into logically organized sections
 - Encapsulates a possibly compensatable, recoverable, unit of work
- Provides processing context for
 - Organization and documentation
 - Variables
 - Fault handlers
 - Compensation handlers
 - Event handlers



The diagram illustrates a BPEL process structure. It consists of a large outer rounded rectangle labeled 'process'. Inside this process, there are three smaller rounded rectangles representing scopes. 'scope A' is positioned at the top left, and 'scope B' is nested inside 'scope A'. 'scope C' is positioned below 'scope A' and 'scope B'.

4 Copyright © 2004-2008 Active Endpoints, Inc. 

Here is an overview of BPEL's Scope activity. As was stated earlier, a Scope is used to partition your process into logical units of work. A Scope is the execution context for a process (and in fact, in the background a process is implemented as a global Scope.) Scopes can be used as a context to create variables, fault/compensation/event handlers and for organizational purposes. They are the basic building blocks used to assemble a BPEL process.

scope Activity Syntax

```

<scope isolated="yes|no"? exitOnStandardFault="yes|no"?
  standard-attributes>
  standard-elements
  <partnerLinks />?
  <messageExchanges />?
  <variables />?
  <correlationSets />?
  <faultHandlers />?
  <compensationHandler />?
  <terminationHandler />?
  <eventHandlers />?
  activity
</scope>

```

5 Copyright © 2004-2008 Active Endpoints, Inc.

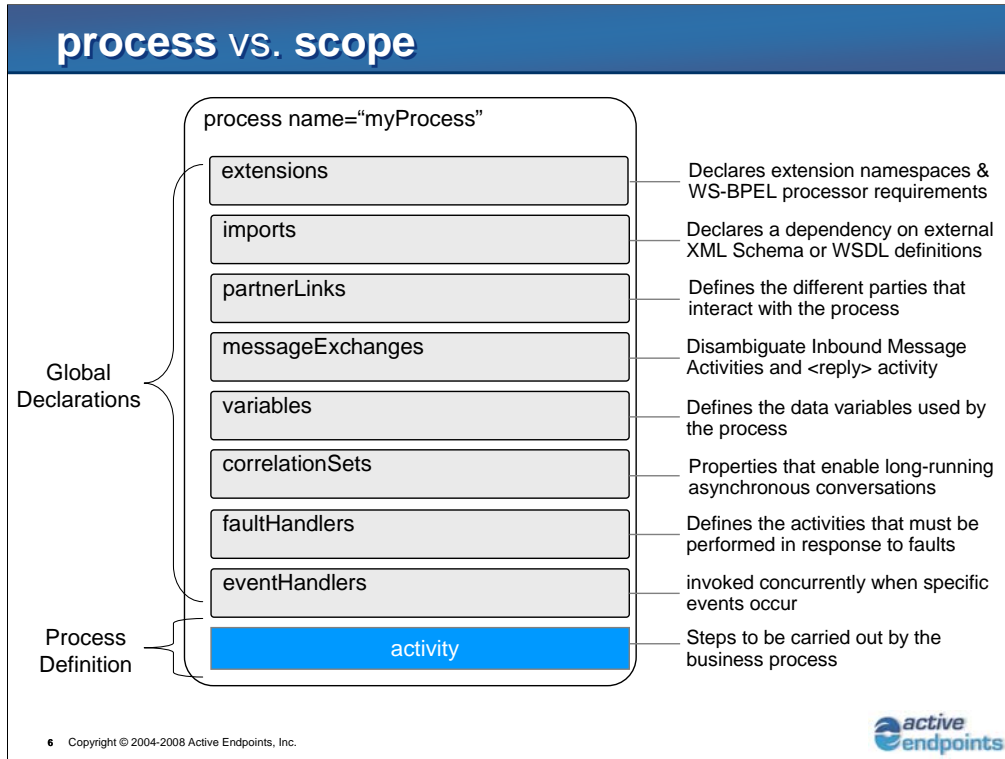


Here a breakdown of the syntax for the Scope activity. First is the *isolated* attribute. We'll cover *Isolated* Scopes in a later unit, when we talk about the Flow activity. Next is the attribute "exitOnStandardFault" which has the value of either yes or no, and the default setting is no. What this means is that if you have it set to "NO" then you will handle standard faults with a user-defined, scope-level fault handler. If any of the standard BPEL faults are thrown when this value is set to "yes", then the process ends, just as if an "exit" activity had been executed. We'll talk more about the Exit activity in a later unit.

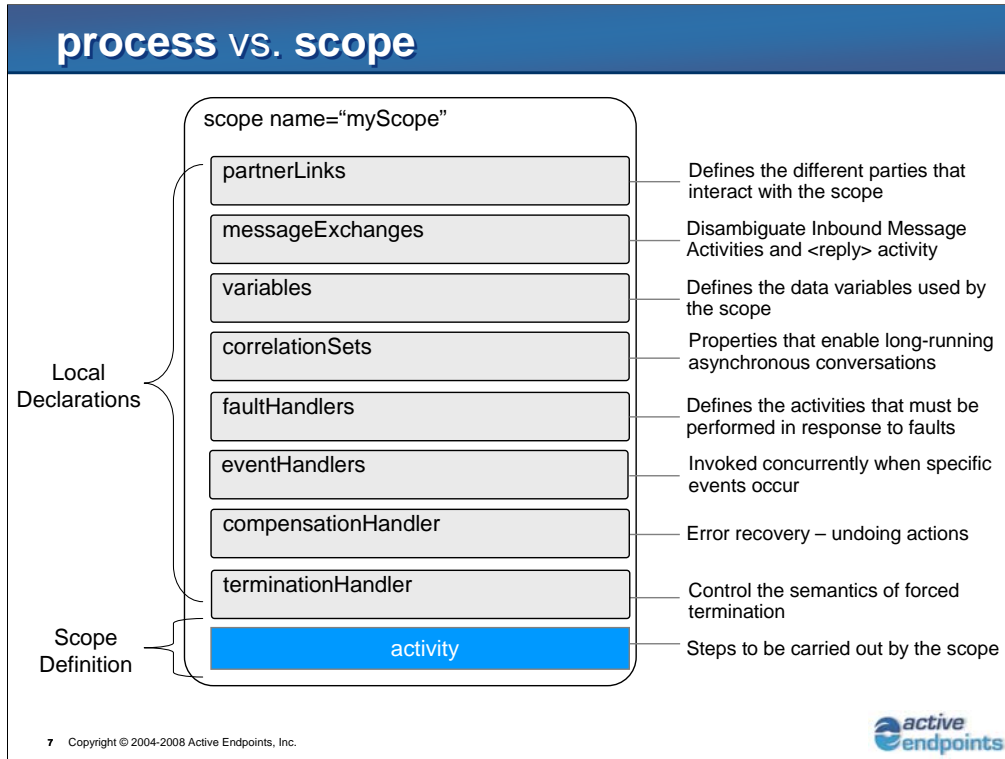
A Scope has all of the usual attributes and elements and can contain these other artifact definitions, as well:

- partnerLinks (which we have already covered)
- messageExchanges (which is an advanced topic and is beyond the scope of this course)
- Variables (which we have covered)
- Correlation Sets (which we went over earlier)

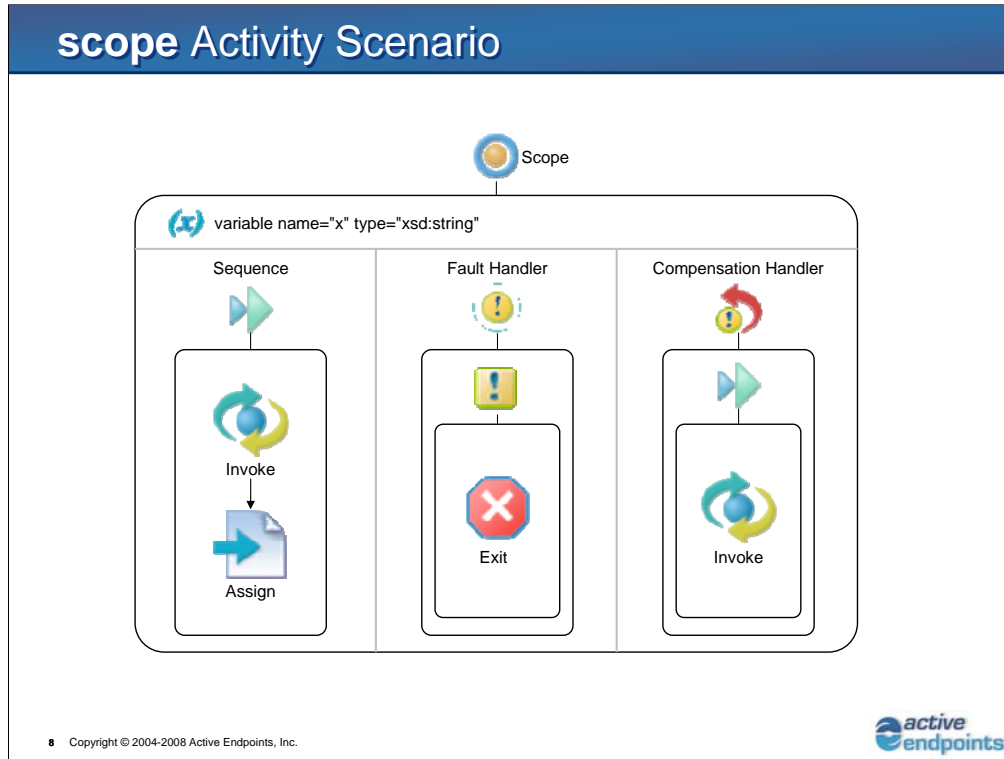
A Scope can also have Fault, Compensation, Termination and Event Handlers, all of which we will cover later in this Unit. The Scope's primary activity is at the bottom, and it can be a container, such as a nested scope. All Scope declarations – obviously - apply at the scope level.



Since many of the declarations at the Scope level can also be made at the process level, the difference between Process Level and Scope Level definitions and declarations must be made clear. The major differences are that a Process can have Import and Extension declarations, but a Scope cannot.



But what Scopes can have – and that a Process *cannot* have - are Compensation and Termination handlers. Note that this is according to the BPEL 2.0 Specification, but ActiveVOS allows Compensation and Termination handlers to be defined at the process level through the use of Extensions. Extensions are an Advanced topic, and is covered in other courses.



Here is a simple scope, as seen in the Process Editor. The outer rounded rectangle with the concentric blue and brown circle icon defines the scope itself. The Scope has a variable named “x”, which is of the type string (as defined in our base schema). (Of course, a process level variable will not actually appear on the process diagram, but is included here for teaching purposes.) This scope has one primary activity, which is a Sequence (shown on the left), as well as a Fault handler (shown in the middle) and a Compensation handler (shown on the right)... The Sequence begins execution and results in one of three possible outcomes:

- 1.) The scope executes its Sequence, which includes the Invoke and the Assign activities, and then terminates normally and the scope is complete.
- 2.) During execution of the Scope’s activities a fault is thrown, the scope terminates *abnormally* and the Scope’s fault handler is invoked.
- 3.) The Scope executes its activities and terminates normally (i.e., successfully) but we need to undo the activities of the scope for some reason, such as a fault being thrown at a higher level of scope. In this case, the compensation handler is invoked and the activities of the Scope are undone.

scope Activity Example

```
<scope>
  <variables>
    <variable name="x" type="xsd:string"/>
  </variables>
  <faultHandlers>
    <catch>
      <exit/>
    </catch>
  </faultHandlers>
  <compensationHandler>
    <sequence>
      <invoke .../>
    </sequence>
  </compensationHandler>
  <!-- Scope's primary activity -->
  <sequence>
    <invoke .../>
    <assign .../>
  </sequence>
</scope>
```

9 Copyright © 2004-2008 Active Endpoints, Inc.



Here is the syntax for the Scope example we saw on the last slide. First we have our scope level variables, here the variable is “x” and is defined as a string (from the schema with the prefix “xsd:”) Next, we define the Fault handlers for the Scope, which in this case uses a Catch that contains an Exit activity. The Exit activity will stop the process immediately. Then we define the Compensation Handlers for the Scope, which in this example starts a Sequence that contains an Invoke. Finally, we have the Scope’s primary activity, which is a Sequence that contains an Invoke followed by an Assign.

scope Activity Semantics

- Follows the same set of semantics as for **process**
 - Process is considered a global `scope`
 - with the addition of `extensions` and `imports`
 - without the compensation and termination handlers
- Can contain other **scopes**
 - Restricted to non-isolated `scopes` only

10 Copyright © 2004-2008 Active Endpoints, Inc.



A Scope is semantically the same as a process, and a process is actually implemented as a global Scope. Let's review the differences between a scope and a process:

- The Process can have Import and/or Extension declarations, but a Scope cannot.
- Scopes can have Compensation and Termination handlers, but a Process cannot. (According to the BPEL 2.0 specification. ActiveVOS allows you to do this.)
- A Process is not an activity, and therefore the standard attributes and elements do not apply.
- Scopes can contain other scopes, if they are not Isolated, but the process is unique.

Variable Semantics in a **scope**

- Variables defined within a **scope** are called local variables
 - Visible only in the `scope` in which it's defined and in all `scopes` nested within this `scope`
- Possible to "hide/mask" a variable declared in an outer **scope** by declaring a variable with the same name and same type in an inner **scope**
 - Variable declared in an inner `scope` with the same name but different type as a variable in an outer `scope` is not permitted

11 Copyright © 2004-2008 Active Endpoints, Inc.



Scopes can have their own variables. Once defined, Scope level variables are seen as local variables to the scope, and can be seen in other scopes that are nested within the defining scope. Therefore, we can say that a variable's visibility goes in or down, not out or up. Building on this idea, we can create a variable in an inner scope with the same type and with the same name as the variable defined in an outer scope. In this case, the one we "see" at that Scope level or below will be the locally defined one, i.e., the "lower level" variable. The variable in the inner Scope must be of same type and have the same name as the one at the higher Scope (or process). We cannot create a scope-level variable that has the same name but a different type as one defined at a higher level. These rules are exactly analogous to those in programming languages with lexical scoping of values.

Unit Objectives

- At the conclusion of this unit, you will be familiar with:
 - ✓ scope activity
 - Working with scopes

12 Copyright © 2004-2008 Active Endpoints, Inc.

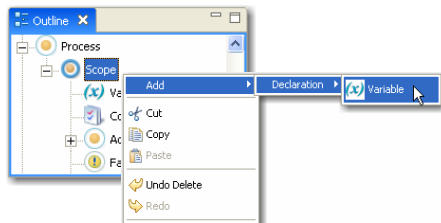


Now that we know something about scopes, lets take a look at how to work with them in the ActiveVOS Designer.

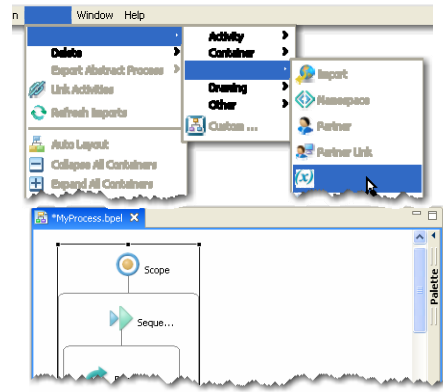
Working with Variables in **scopes** - Adding

- Variables are added to a **scope** manually via the Outline view or Process menu

- Outline view



- Process menu



13 Copyright © 2004-2008 Active Endpoints, Inc.

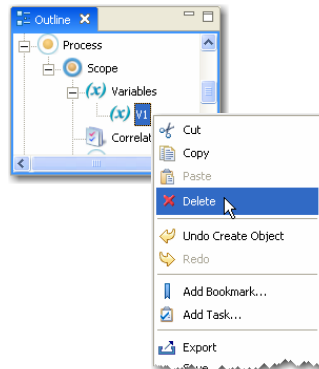


A Scope is a container like any other. Adding a new variable to a Scope is exactly the same as adding a variable to a process. Select the Scope, use the Right Mouse context menu and select Add->Declaration->Variable. This can also be done in the Outline View by selecting the Scope or from the Process menu *if* the target Scope is selected in the Process diagram

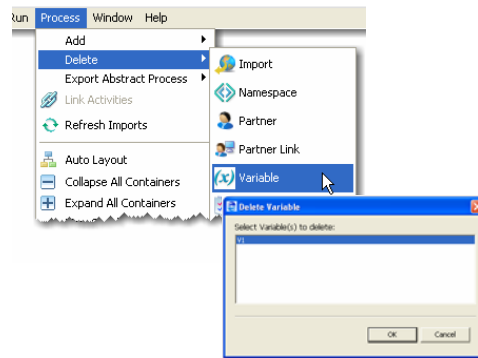
Working with Variables in scopes - Deleting

- Variables are deleted from a **scope** via the Outline view or Process menu

- Outline view



- Process menu
(Scope must first be selected)



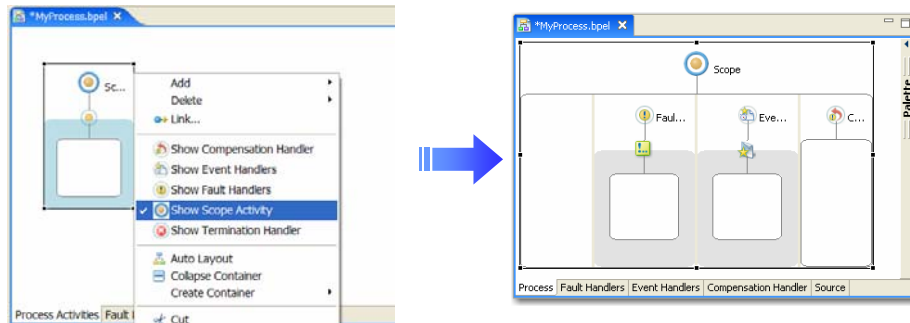
14 Copyright © 2004-2008 Active Endpoints, Inc.



You can delete Scope level variables just as you would any other variable, using the RM from the Outline view or use the Process menu and then Delete->Variable. If you use either the Outline View or the Process menu you can delete multiple variables at once by using the Shift and/or Ctrl keys and selecting the variables to delete.

Show Additional **scope** Views

- By default, **scope** containers only show the process definition
 - Display other sections of the **scope** via the right-click menu



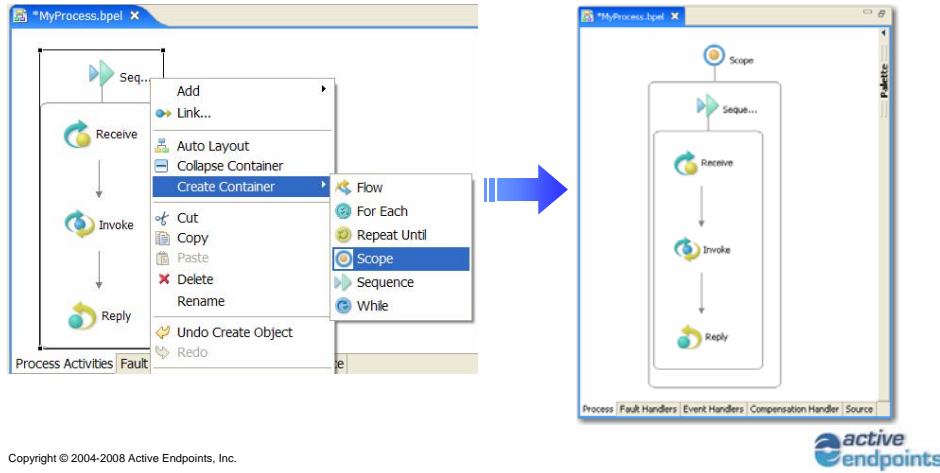
15 Copyright © 2004-2008 Active Endpoints, Inc.



In the process editor, Scopes show as empty containers by default. To see the Scope's internal activities, use the Right Mouse menu and select "Show Scope Activity." (The show/hide values work as toggles.)

Adding Activities for Containment

- Allows for a convenient way to automatically put a selection of activities within certain containers
 - Works with sequence, scope, or while activities

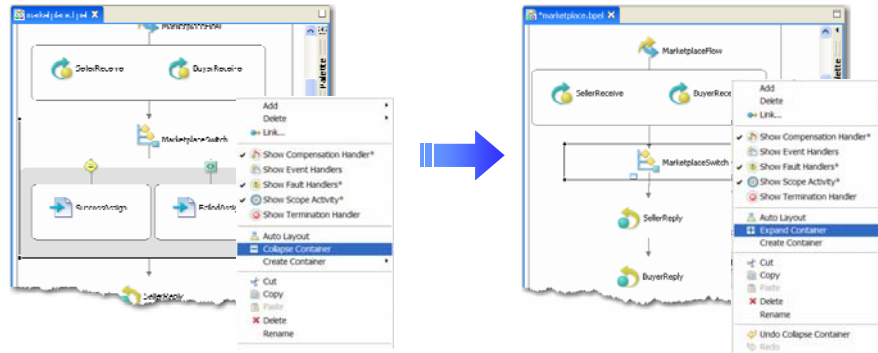


16 Copyright © 2004-2008 Active Endpoints, Inc.

To add activities to a container, select the activities *in the desired order*, then use the Right Mouse menu and select Create Container->Scope. This functionality also works for sequences and while loops.

Expanding and Collapsing Container Activities

- Provides better management of screen real estate by collapsing container activities
 - e.g., sequence, flow, scope, while



17 Copyright © 2004-2008 Active Endpoints, Inc.



You can also expand and collapse containers to change the visibility of their child activities and any nested containers. When dealing with large and complex processes, screen real estate is still in short supply, and must be managed effectively.

Unit Summary

- Now you are familiar with:
 - scope activity
 - Working with scopes