



This is Unit #11 of the BPEL Fundamentals I course. In past Units we've looked at ActiveBPEL Designer, Workspaces and Projects and then we created the Process itself. Next, we looked at declaring Imports, PartnerLinks and Variables and how to create Interaction Activities in various ways. Then, we looked at our first container activity, the Sequence, and at Assignments and Copies and finally, in the last two units we studied Correlation and Scopes. In this Unit we will take a look at a new topic, BPEL Faults.

## Unit Objectives

- At the conclusion of this unit, you will be familiar with:
  - Signaling faults
  - Handling faults

2 Copyright © 2004-2007 Active Endpoints, Inc.



In BPEL, a Fault is a disruption of process execution that results in a signal being sent from the system. Fault Handling is what we do when this happens. Fault Handling can be thought of as a "mode switch" from the normal processing of the scope or process. Its aim is to undo the partial and unsuccessful work of the scope in which the fault occurred.

## Signaling Faults Overview

- During the execution of a BPEL process, an event may occur that disrupts the normal flow of a business process
  - Business-centric faults
    - Invoking Web services may result in a WSDL fault
    - BPEL process may explicitly throw a fault
      - Using the `throw` and `rethrow` activities
  - System-centric faults
    - Faults may be raised automatically via internal or external error conditions
    - Includes the built-in standard BPEL faults
- When a fault occurs, normal processing is halted
  - Control is transferred to the appropriate fault handler

3 Copyright © 2004-2007 Active Endpoints, Inc.



When an event occurs that disrupts the normal flow of our business process, this is called a fault. How do we classify these faults? There are lots of ways that organizations classify faults, here is a simple one. Faults can be divided between those generated by our (or by our partner's) Business Logic and those generated by the system at Runtime, either locally or remotely.

Examples:

Business Faults = *Use the Throw and ReThrow activities*

invalid Customer number (local) throws a fault defined in our WSDL

credit check problems (remote) throws a fault defined in our partner's WSDL

Runtime Faults = *Are generated by the system and includes all standard BPEL faults*

invalid variable type used as a parameter (local)

site of the invoked service is temporarily down (remote)

When a fault is thrown, normal processing halts and control is sent to Fault Handler, if one is defined.

Note: Invoked faults are those returned when we attempt to Invoke a Web Service. The BPEL 2.0 Spec differentiates faults as either internal faults or invoked faults. The ActiveBPEL Engine has a "Retry" setting that tells it whether to try again if an invoked site throws a runtime fault. This has settings for whether we try again and, if we do, how long to wait. (These settings can be linked to any SLAs that apply to the invoked service.)

## Business Faults in WSDL

- Invoking synchronous operations on partner-provided Web services may return a WSDL fault message
  - WSDL faults are identified in BPEL by their qualified name
    - Comprised of the WSDL's target namespace and the ncname of the fault

```
<portType name="CalculatorPortType">  
  <operation name="CalculatorOperation">  
    <input message="tns:CalculatorInput" />  
    <output message="tns:CalculatorOutput" />  
    <fault name="calcFault" message="tns:BadArgumentFault" />  
  </operation>  
</portType>
```

4 Copyright © 2004-2007 Active Endpoints, Inc.



Here we see an example of a fault that is thrown when a bad argument is used in a process. Starting at the top in the WSDL file: The portType is “CalculatorPortType”, the Operation is “CalculatorOperation”, and there are both input and output messages, “CalculatorInput” and “CalculatorOutput.” In this example, the fault’s name is “calcFault” and the Fault’s QName is “tns:BadArgumentFault”, which is defined somewhere in one of our other WSDL files.

### BPEL Standard Faults

- ambiguousReceive
- completionConditionFailure
- conflictingReceive
- conflictingRequest
- correlationViolation
- invalidBranchCondition
- invalidExpressionValue
- invalidVariables
- joinFailure
- mismatchedAssignmentFailure
- missingReply
- missingRequest
- scopeInitializationFailure
- selectionFailure
- subLanguageExecutionFault
- uninitializedPartnerRole
- uninitializedVariable
- unsupportedReference
- xsltInvalidSource
- xsltStylesheetNotFound

5 Copyright © 2004-2007 Active Endpoints, Inc.



Here is a list of the standard BPEL faults. We have mentioned these before and will be referring to them continuously through this Unit.

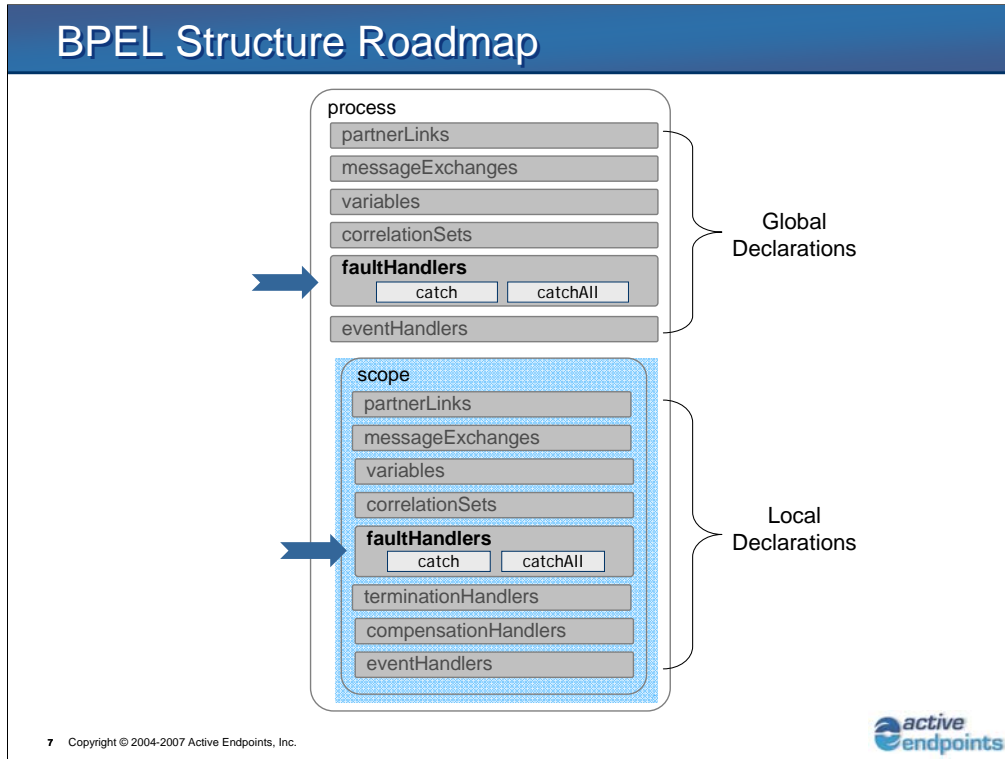
## Unit Objectives

- At the conclusion of this unit, you will be familiar with:
  - ✓ Signaling faults
  - Handling faults

6 Copyright © 2004-2007 Active Endpoints, Inc.



So, now that a fault has been signalled, what do we do?



We can *handle* them. Here we're showing the roadmap to include both scope and process level fault handlers. We can define them globally, i.e., at the process level, or we can define them locally - at a particular Scope's level - and we can define one or more Catches and a catchAll within the same handler.

## Fault Handler Overview

- Used to explicitly handle faults that might occur in a business process
  - Fault handlers can be defined
    - For the `process`
    - For a `scope`
    - Inline for the `invoke` activity
  - Bound to a particular kind of fault using a `catch`
    - Any fault not caught by a more specific fault handler
      - using a `catchAll`
- Once a **scope** receives a fault, normal processing of all nested activities are halted
  - `scope` is considered to have ended abnormally

8 Copyright © 2004-2007 Active Endpoints, Inc.



The purpose of the handler is to explicitly handle faults that occur in a process. Fault handlers can be defined for the process, for the scope level or inline for a specific Invoke activity. We'll cover inline fault handlers later on. Once thrown, a fault goes to a Catch (for a specific fault) or to a catchAll (for anything not handled otherwise.) Once the scope receives a fault, all processing is halted, including nested activities, and the scope is considered to have ended abnormally. At this point the Fault Handler takes over.

## Fault Handler Overview (cont.)

- Fault handlers provide a way to separate out normal process logic from error handling logic
  - Makes normal processing logic less cluttered and therefore easier to read
- Position fault handlers strategically
  - Catch and handle all exceptions from where you want your process to recover

© Copyright © 2004-2007 Active Endpoints, Inc.



Now let's step back and look at fault handlers from a higher level. We don't *need* Fault Handling, but we should have it. Like airbags and smoke alarms, you don't need them, but they are highly recommended. Fault Handlers separate normal process logic from fault logic, i.e., the execution of normal activities vs. error handling logic that is in the Fault Handler. You should position your Fault Handlers strategically. By doing this, you don't have to add fault logic for each individual *activity*, but rather only for *logical units of work*. This reduces the overall volume of fault handling code and makes the process less cluttered and much easier to read. Position Fault Handlers at those points where you would want the process to recover from, or to, depending on how you look at it.

## Fault Handler Syntax

```

<faultHandlers>?
<!-- Note: There must be at least one faultHandler -->
  <catch faultName="QName"?
    faultVariable="BPELVariableName"?
    ( faultMessageType="QName" |
      faultElement="QName" )? >*
    activity
  </catch>
  <catchAll>?
    activity
  </catchAll>
</faultHandlers>

```

10 Copyright © 2004-2007 Active Endpoints, Inc.



Here is the BPEL Fault Handler syntax. If we declare a “faultHandlers” section we must have something inside that definition. Therefore, we need to define one or more Catches and/or a catchAll. If we define a Catch, it will optionally have a single associated faultName and optionally a single associated faultVariable. The faultVariable can also optionally have data that is of the faultMessageType or faultElement type. If you notice the “\*” character after the closing angle bracket of the Catch, you can see that we can declare as many Catches as are needed. The catchAll is optional, and if we declare a catchAll, we can have only one. Both Catches and catchAlls have a primary activity to handle the actual fault recovery's actions. Definitions to consider when evaluating the fault handler selection rules that follow:

- The *Fault* is the QName (i.e., the namespace prefix + ":" + the fault name) of the Fault that is actually thrown.

- The *Fault Variable* is an optional variable associated with the fault. It holds the fault's data. It is implicitly declared by virtue of being used as the fault handler's "fault variable" attribute and is local to the fault handler. It is not visible or usable outside of the fault handler in which it is declared.

- The *Fault Variable Element QName* is the data that is inside the variable, i.e., its contents. It can be either a MessageType (and the attribute will be of the faultMessageType) or an Element type (and the attribute will be of the elementType), and there can only be one. Note that the variable data is only considered by the handler's selection process if the data is a "user-defined" i.e., non-standard - Schema Element Type. What this means is that if the variable data is a Standard Schema Type (such as a boolean or a string) then that data is not

Copyright © 2004-2007 Active Endpoints, Inc. defined one of your own Schemas (or imported by one of your WSDLs) then the handler assumes that the type match is significant and therefore usable in the

## catch Overview

- Used to handle a specific fault, based on matching:
  - Both the Name of the fault and the type of variable associated with the fault
  - Name of the fault
    - Globally unique fault QName
  - Type of variable associated with the fault
    - Two types:
      - WSDL Message Type (`faultMessageType`) or
      - XML Schema Element (`faultElement`)
- Fault variables must be explicitly declared in the associated **catch**

11 Copyright © 2004-2007 Active Endpoints, Inc.



We will look at the "Catch" first. Each Catch is created to handle a specific fault. There are different data points used to match a Catch to particular Fault:

1.) by FaultName and faultVariable type

2.) By Fault Name only (which must be a Globally Unique QName) // otherwise it is ambiguous...

3.) By faultVariable Type

- WSDL Message Type (the `faultMessageType`) or

- XML Schema Element (the `faultElement` type)

As was discussed earlier, Fault Variables must be explicitly defined inside the Catch that will handle that particular fault.

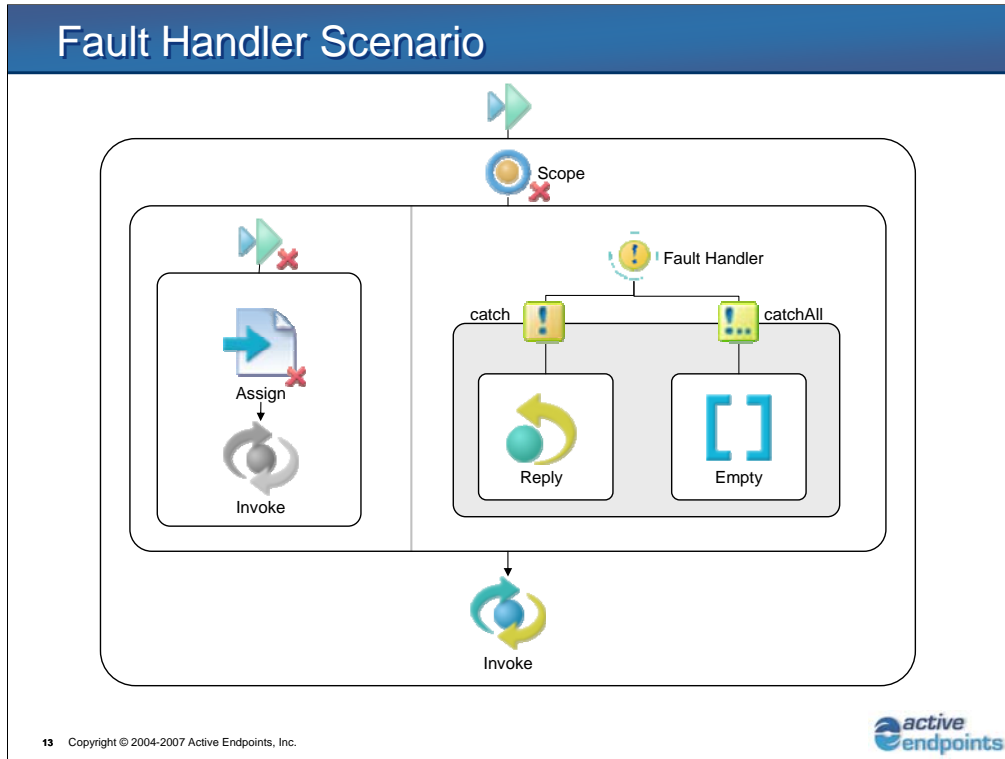
## catchAll Overview

- Catches any fault not caught by a more specific **catch** handler

12 Copyright © 2004-2007 Active Endpoints, Inc.



The "catchAll" activity is created to catch anything not handled by a catch. This activity functions like the Default Case of a Case/Switch/Select statement in other languages. You can also think of it as being like a coin sorter... and the Catch is the bucket that gets the coins that don't fit into any other slot.



Here we have a typical Fault Handling scenario. We have an outer Sequence activity which contains a Scope, followed by an Invoke activity. The Scope's primary activity is also a Sequence, which is on the left, and has an Assign followed by an Invoke. If we a fault is thrown during the execution of the Assign, we end the activity immediately, and the Invoke that comes after it is never called. Since the Scope doesn't complete normally, control passes to the defined Fault Handler for the scope. The Fault Handler is on the right. Inside the fault handler we have one Catch for a specific fault, that calls a Reply activity. Any fault other than the one specifically associated with the Catch will not be routed to the Catch, so we have a catchAll, which contains an Empty activity. Once the Fault Handling for the Scope is completed the Scope itself is completed, and then the Sequence's final activity, the Invoke, is fired.

## Fault Handler Example

```
<sequence>
  <scope>
    <faultHandlers>
      <catch faultName="ns1:somethingWentWrong">
        <reply ... />
      </catch>
      <catchAll>
        <empty />
      </catchAll>
    </faultHandlers>
    <sequence>
      <assign ... />
      <invoke ... />
    </sequence>
  </scope>
  <invoke ... />
</sequence>
```

14 Copyright © 2004-2007 Active Endpoints, Inc.



Here is the Fault Handling syntax for the previous example. We have the enclosing Sequence, and then the child Scope, and inside the scope is Fault Handler, which has one Catch and a catchAll, then we see the Sequence which has an Assign and an Invoke. Below the Scope is the rest of the original Sequence, which is simply the Invoke activity at the bottom. The Catch is there to handle a specific fault called "ns1:somethingwentwrong", which it will handle by executing a Reply activity. If any other Faults are encountered, they are handled by the catchAll, which executes the Empty activity. Once the Fault Handling is complete, the Scope is complete, and the Invoke activity will then execute and complete the Sequence.

## Fault Handler Selection Rules Review

- Fault with no associated fault data
  1. `catch` handler that specifies only a matching `faultName` value will be selected
  2. `catchAll` handler will be selected
  3. `fault` will be handled by the default fault handler

15 Copyright © 2004-2007 Active Endpoints, Inc.



Here is a brief review of the Fault Handling rules in BPEL where there is a fault thrown, but there is *no* associated fault data.

- 1.) If the Fault Name matches the Fault Name in a Catch then it is handled by the Catch.
- 2.) If the Fault Name does not match then it is handled by the catchAll.
- 3.) If no catchAll is defined, it is handled by the Default Fault Handler.

## Fault Handler Selection Rules Review (cont.)

- Fault has associated fault data
  1. catch handler that has a matching `faultName` value and a `faultVariable` whose type matches the fault's data will be selected
  2. catch handler that has a matching `faultName` value and a `faultVariable` whose element QName matches the fault's data element QName will be selected
  3. catch handler that has a matching `faultName` and no `faultVariable` defined will be selected
  4. catch handler with no specified `faultName` and with a `faultVariable` whose type matches the fault's data will be selected
  5. catch handler with no specified `faultName` and a `faultVariable` whose element QName matches the fault's data element QName will be selected
  6. `catchAll` handler will be selected
  7. `fault` will be handled by the default fault handler


16 Copyright © 2004-2007 Active Endpoints, Inc.



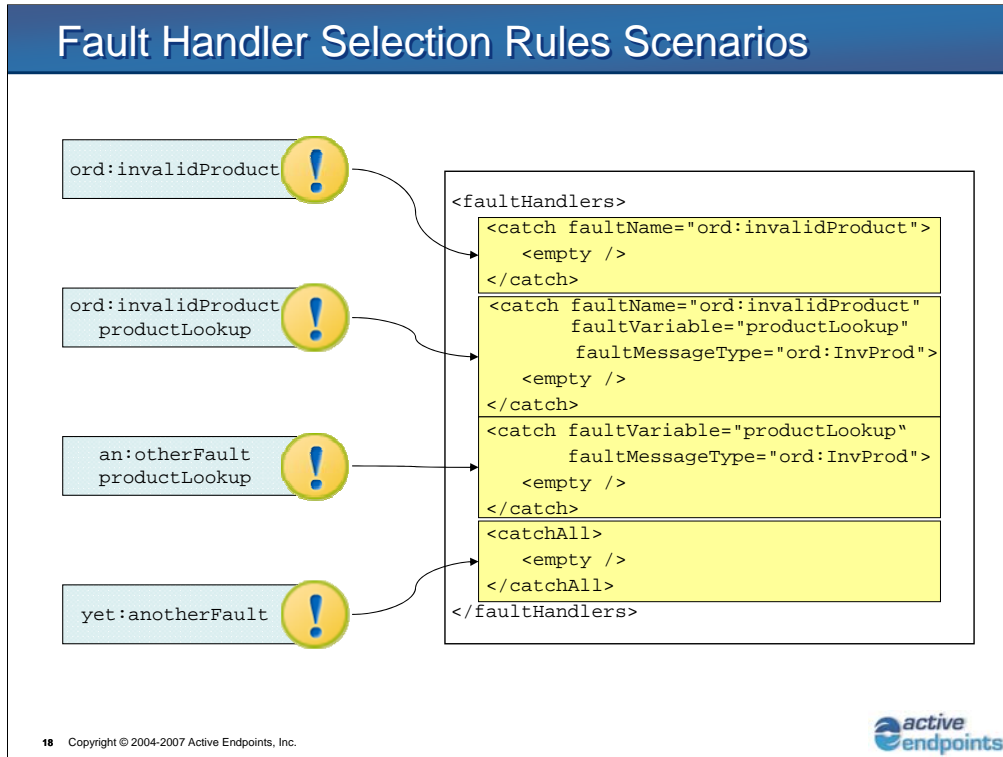
Here are seven situations for handling a fault that is thrown when that fault *has* associated data. Data thrown with a fault can be either a WSDL `MessageType` or an XML Schema Element. As we look at these, remember that there are three things the system is evaluating at execution time: The Fault Name, the Fault Variable and the Fault Variable's Element QName. The Fault examines each case, looking for the “best match”, *not* the first match.

1. We match both the Fault name and the Fault variable's type - which will always be the best match - because it doesn't get any better.
2. The Fault Name matches, and the Fault Variable's element QName matches the Fault Data Element QName.
3. The Fault Name matches but there is no Fault Variable defined.
4. There is no Fault Name to match, but the Fault Variable's type matches the Fault's data type.
5. There is no Fault Name to match, the Fault Variable's type does not match, but the Fault Variable has an element whose QName matches the Fault's data Element QName.
6. The `catchAll` is used if none of the above apply.
7. If no `catchAll` is defined, we use the default Fault Handler.

Fault Handler Selection Table (with fault data)			
	Fault Name	Fault Var. Type	Fault Var. Elem. QName
Best	Yes	Yes	No
2 <sup>nd</sup>	Yes	No	Yes
3 <sup>rd</sup>	Yes	No	No
4 <sup>th</sup>	No	Yes	No
5 <sup>th</sup>	No	No	Yes
6 <sup>th</sup>	catchAll		
Worst	Default Fault Handler		

17 Copyright © 2004-2007 Active Endpoints, Inc. 

The logic in this table comes from the BPEL 2.0 Specification. You can go to the specification's Section 12.5 to see the information on "Fault Handlers." Note that there are important caveats and special cases included in the BPEL Specification's explanations.



### Animated slide!

The Fault Handler selection rules are shown in this example.

Note that Fault handling in BPEL is done using a “*best match*” algorithm, not a “first match” algorithm.

1 = Fault Name “ord:invalidProduct” matches the Fault Name in the first catch, so it goes to the first Catch.

Note: the Empty activity is an activity that does nothing. It works as a placeholder for future work, or it can be used as a way to synchronize the execution of activities.

2 = Fault Name “ord:invalidProduct” and the Fault Variable “productLookup” match both the Fault Name and Fault Variable defined in the second catch, so it goes to the second Catch.

3 = Fault Variable Name does not match (because there isn’t one), but the Fault Variable does match the third catch so it goes to the third Catch.

4 = Fault Name does not match and there is no fault variable, so it goes to the catchAll.

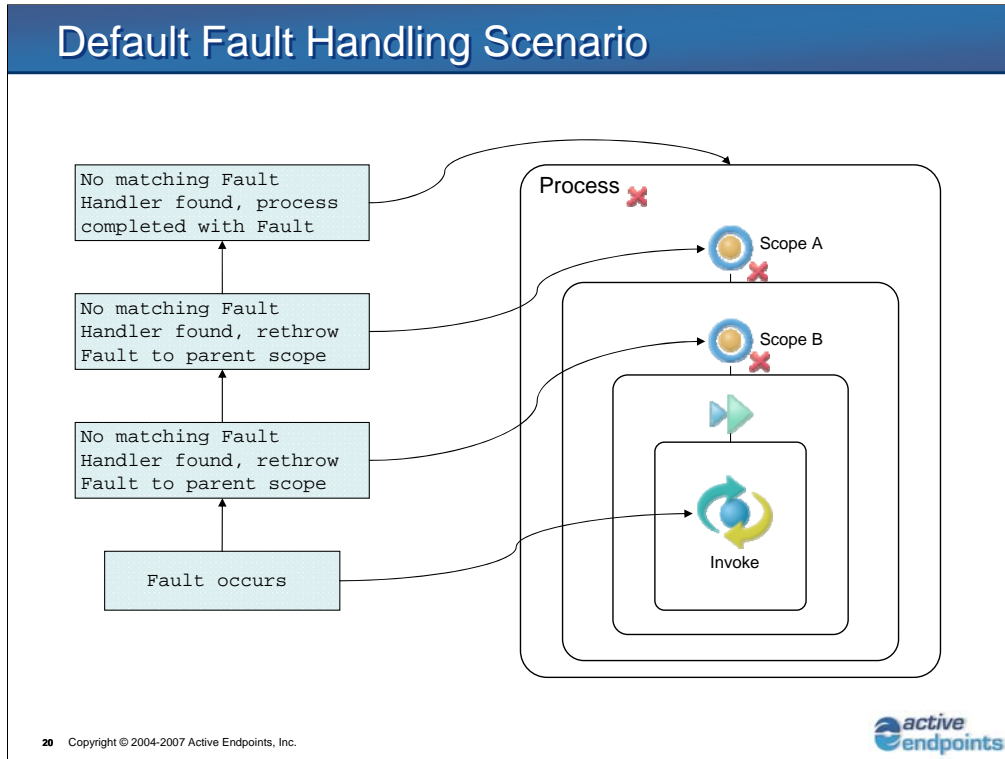
## Default Fault Handling

- When there is no fault handler present or no specific **catch** is selected for a given scope
  - Default `faultHandler` is implicitly created
    - Behavior is to re-throw the fault to the parent `scope`
    - This process continues until the global `process` scope is reached
- If the fault occurs in (or is re-thrown to) the global `process` scope, and there is no matching fault handler the `process` terminates abnormally

19 Copyright © 2004-2007 Active Endpoints, Inc.



Default Fault Handling is used on a fault if no existing `Catch` matches and no `catchAll` is defined. In BPEL, the Default Fault Handler is implicit and does not have to be defined. It automatically re-throws the fault to the next level of scope (i.e., the parent of the current scope.) If the fault reaches the process level without being handled by a fault handler, then the process terminates abnormally. (Note that this behavior can be overridden.)



We have a Process with a Scope A, and a Scope B that is nested inside Scope A. Inside Scope B is a Sequence, and in that Sequence is an Invoke, which throws a fault.

- 1.) Scope B's Invoke activity throws a fault, but there is no fault handler defined for Scope B, so the fault is thrown to the parent scope, which is Scope A.
- 2.) If there is no Fault Handler defined for Scope A, it throws the fault to its parent Scope, which is the process itself.
- 3.) If the process has no fault handler defined, then the Process is terminated abnormally.

## Inline Fault Handling

- The **invoke** activity provides a special shortcut to directly handle faults related to the invoke
  - Can handle WSDL faults for synchronous operations and other run-time related faults

```

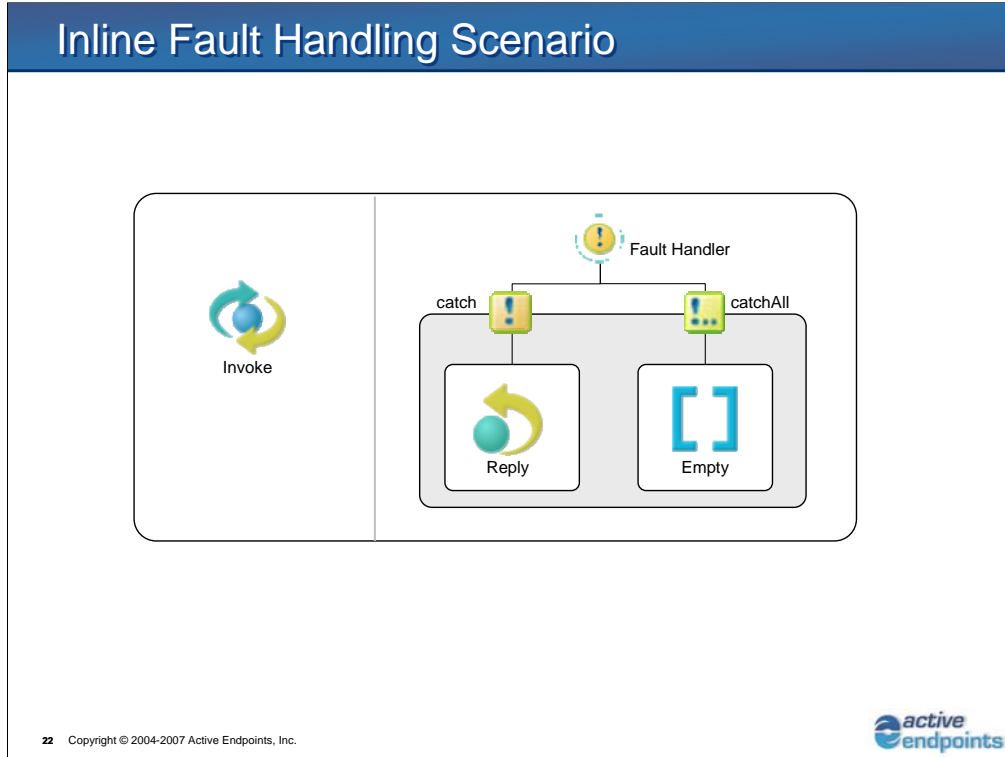
<invoke partnerLink="ncname" portType="qname"? operation="ncname"
  inputVariable="ncname"? outputVariable="ncname"?
  standard-attributes>
  standard-elements
  ...
  <catch faultName="QName"? faultVariable="BPELVariableName"?
    faultMessageType="QName"? faultElement="QName"?>*
    activity
  </catch>
  <catchAll?
    activity
  </catchAll>
  ...
</invoke>

```

21 Copyright © 2004-2007 Active Endpoints, Inc.



We have seen that we can define fault handling at the scope or process levels. We can also define **Inline** fault handlers, which are provided specifically for the Invoke activity. This is a special shortcut to handle faults generated when Invoking a web service. If we invoke a service and the invocation returns a fault, we will use this to handle it immediately. In the syntax, we see the Invoke defined, with its partnerLink, portType, Operation and Input and Output Variables, as well as all of the standard attributes and elements. Following the basic definitions - but still inside the Invoke - is a Catch, followed by a catchAll. Note that the Catch is followed by the "\*", which allows multiple Catches, just as in a "normal" fault handler.



Now let's look at an Inline Fault Handler scenario. If an Invoke has an inline fault handler, this is how it looks in ABD. A single Invoke activity is on the left, with a fault handler defined on the right, here containing a single Catch and one catchAll. If a fault is returned on the Invoke, control passes to the fault handler, evaluating the Catch first. If it matches to it, then it fires the Reply. If it does not match the Catch, then it goes to the catchAll, which will fire the Empty activity.

## Inline Fault Handling Example

```
<invoke partnerLink="calcServicePL"
  portType="calc:CalculatorPortType"
  operation="CalculatorOperation"
  inputVariable="CalculatorInput"
  outputVariable="CalculatorOutput" >
  <catch faultName="calc:InvalidArgument" >
    <reply ... />
  </catch>
  <catchAll>
    <empty/>
  </catchAll>
</invoke>
```

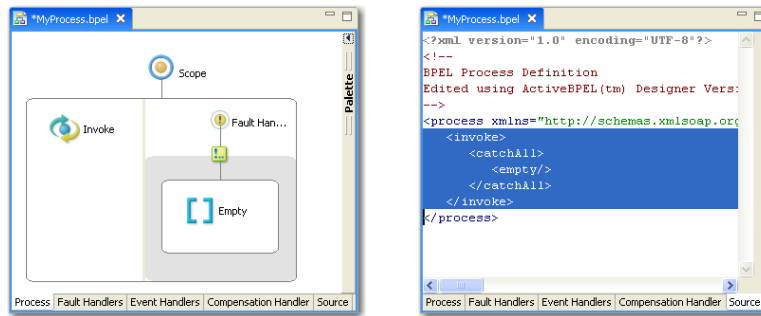
23 Copyright © 2004-2007 Active Endpoints, Inc.



Here is the syntax for the previous example. We have a partnerLinkType “calcServicePL”, a portType “CalculatorPortType”, and an operation called “CalculatorOperation.” We have both input and output variables defined and we have a single Catch defined, which will handle a fault with the Fault Name “calc:InvalidArgument.” The Catch will fire a Reply activity when this specific fault is generated. If any other type of fault is thrown, the catchAll will handle it.

## Working with Inline Fault Handling

- Modeling inline Fault Handlers using ActiveBPEL Designer
  - Requires a `scope` activity which contains a single `invoke` activity
  - Enable Show Fault Handlers view on `scope`



24 Copyright © 2004-2007 Active Endpoints, Inc.



To add a fault handler to an Invoke activity, wrap the Invoke inside a scope, then enable fault handling for the scope. This is an otherwise normal scope with a single invoke activity. If we check the source code (on the right) you'll see that it defines a single catchAll with an Empty activity.

## Unit Summary

- Now you are familiar with:
  - Signaling faults
  - Handling faults