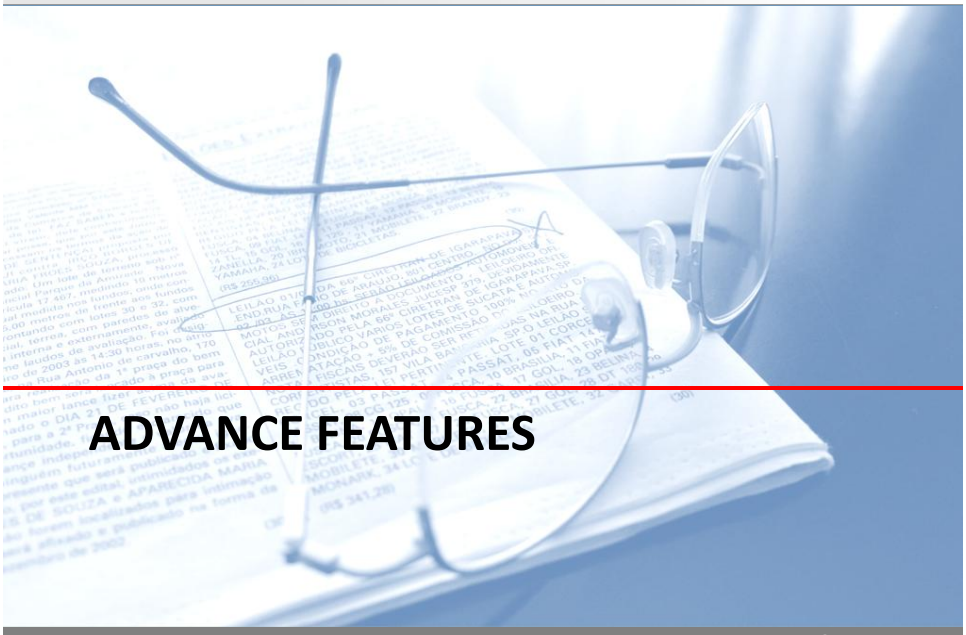


ADVANCE FEATURES

Chained Subprocess

- Chained Subprocess provides same functionality as Subprocess node with few exceptions

Subprocess	Chained Subprocess
Synchronous	Asynchronous
Parent process waits for Subprocess to complete	Parent process continues execution after invoking Subprocess
Data can be exchanged from parent to sub (input) as well as from sub to parent (output on completion)	Data can be exchanged only from parent to Subprocess as input.

Remote Subprocess

FUJITSU

- Remote Subprocess node allows to execute/invoke a process outside the BPM system boundary.
- BPM Process can be invoked on another BPM server
- Interstage BPM supports two open protocols for communication between workflow servers, these protocols pass XML messages over HTTP between workflow servers
  - ASAP: Asynchronous Service Access Protocol (***recommended***)
  - SWAP: Simple Workflow Access Protocol
- To invoke Remote process, provide the ASAP access URL for the remote process in the “Data Mapping” settings.

XML UDA

FUJITSU

- XML UDA allow to store xml data in process as process attribute.
- XML data/UDA is useful when working with Web Services
- Provides a way to add UDAs dynamically (XML attributes) in process without modifying/editing process.
- XML UDAs can be displayed on the form using Table widgets, or XPath can also be used to display attributes separately.
- XML UDA can be
  - Well formed XML: create an XML UDA and ensure to store valid XML structure
  - Valid XML: specify schema to validate XML structure against
- Studio generates XPath wherever applicable for easy mapping of XML (UDA) attributes with string (UDA) value and vice versa.

XML Actions

FUJITSU

- There are out-of-the-box XML actions to help working with XML UDAs
- Available Actions:
  - Add Substructure in XML
  - Assign UDA from XPath
  - Assign XML to UDA
  - Delete from XML
  - Set Substructure in XML
  - Set Text or Attribute Value in XML

Add substructure in XML

FUJITSU

- This action allows to add/insert an XML part in an XML UDA at a specified location.

XML UDA

```
<?xml version="1.0" encoding="UTF-8"?>
<Body>
 <Customers>
  <Customer>
   <id>1234</id>
   <firstName>John</firstName>
   <lastName>Doe</lastName>
  </Customer>
 </Customers>
<Body/>
```

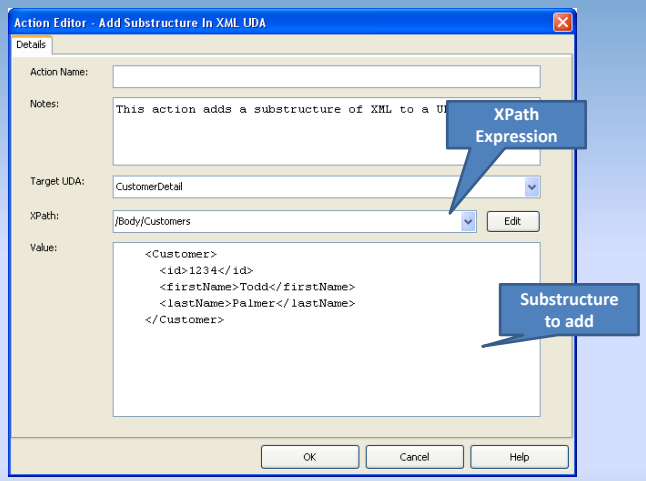
Substructure

```
<Customer>
   <id>1235</id>
   <firstName>Todd</firstName>
   <lastName>Palmer</lastName>
</Customer>
```

Updated XML UDA

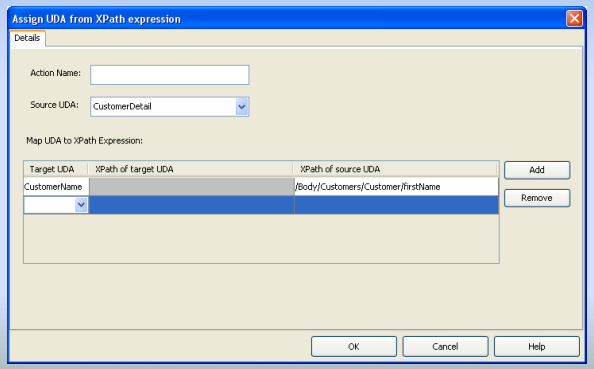
```
<?xml version="1.0" encoding="UTF-8"?>
<Body>
 <Customers>
  <Customer>
   <id>1234</id>
   <firstName>John</firstName>
   <lastName>Doe</lastName>
  </Customer>
  <Customer>
   <id>1234</id>
   <firstName>Todd</firstName>
   <lastName>Palmer</lastName>
  </Customer>
 </Customers>
<Body/>
```


Add substructure in XML



Assign UDA from XPath

- Assign UDA value from an XML UDA using XPath
- Source UDA is XML
- Target UDA
  - XML – use XPath for source and target
  - String – use XPath to get value from Source UDA



XML Actions

FUJITSU

- Assign XML to UDA
  - Assign value to an XML UDA or update value
- Delete from XML
  - Delete value of an element, specified by XPath Expression
- Set Substructure in XML
  - Set a substructure in an XML UDA
  - As opposed to “Add Substructure..” action, this action replaces existing substructure with new one.
- Set Text or Attribute Value in XML
  - Set value of an element of attribute using XPath expression.

Exception Handling Actions

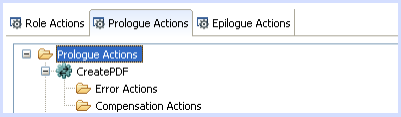
FUJITSU

- OnSuspend Action
  - When a process or task is suspended, these actions will execute before suspending
  - Useful to perform task before suspension, e.g. send email, update database etc.
- OnResume Action
  - When a suspended process or task is resumed, these actions will execute on resuming
  - Useful to perform task before resuming, e.g. send email, load data etc.
- OnAbort Action
  - When a process or task is aborted, these actions will execute immediately before abort.
  - Useful to perform task before aborting, e.g. send email, delete data etc.
- OnError Action (only for Process and Remote Subprocess Node)
  - Execute actions to perform task when remote Subprocess fails to start

Error Handling

FUJITSU

- Actions in an action set execute in a single transaction
- If any action fails, all updates are rolled back and process goes to error state
- Error handling allows catching of exceptions in actions and preventing the process from going to error state
- There are two sets of actions available for error handling
  - Error Action
  - Compensation Action



- Each action may have zero or more error and compensation actions defined.

Error Handling

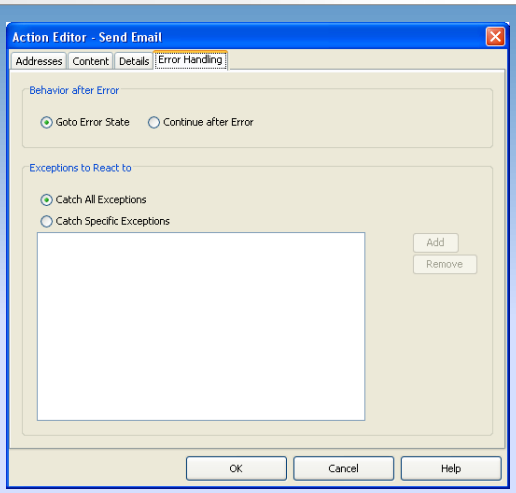
FUJITSU

- Error Action
  - Executes when an action fails and throws exception
  - Catch exception
    - Perform pre-defined corrective action
    - Continue process w/o error
- Compensation Action
  - Executes when action fails and throws exception
  - *Does not execute if error action handles exception and process continues without error*
  - Useful to rollback updates outside the BPM transaction context.
  - Executes in reverse order for all actions in the Action Set
- On Task Recall
  - Compensation actions for target node's "Prologue Action Set" and for source node's "Epilogue Action Set" are executed.

Error Action

FUJITSU

- Actions added as Error Action have additional "Error Handling" tab
- Catch "All" or specific exceptions
- Continue to error state or continue without error.



Application Variable

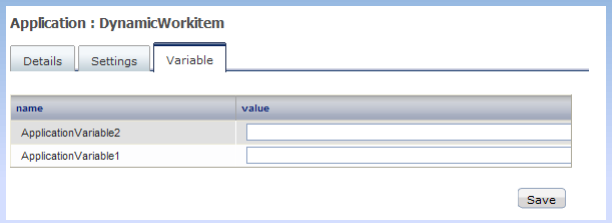
FUJITSU

- Allows to share data across processes in an application
- Great for defining common configuration information.
- usage example
  - WSDL URL can be defined as an application variable and shared across processes
  - If URL changes, change impact is less and change is easy.
- To define application variables:
  - Right click application → select *Properties* → select "*Application Variable*"
  - All app variables are stored in XML format in Appvariable.xml

Application Variable

FUJITSU

- Can be created only in Studio
- Can be updated in BPM Console



Process Scheduler

FUJITSU

- Process Scheduler is a periodic workflow application-level timer that *starts process instances, of specified process definitions* contained in a workflow application, based on the schedule set in the timer
  - Can be used with business calendar to control scheduling
  - Can be set with an expiration time, after which scheduling stops
  - Process Definition must be in **Published** state
- To setup scheduling
  - Right click on “resources” folder and select “New→Process Scheduler”
  - Scheduler configuration is defined in *ProcessScheduler.xml* file

Process Scheduler

FUJITSU

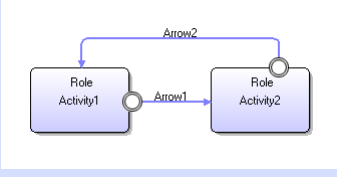
```
<ProcessScheduler>
  <Timers>
    <Timer>
      <Name>Timer Name</Name>
      <ProcessDefinitions>
        <ProcessDefinition>Process Definition Name1</ProcessDefinition>
        <ProcessDefinition>Process Definition Name2</ProcessDefinition>
      </ProcessDefinitions>
      <Calendar>Business Calendar File Name</Calendar>
      <Schedule>Business Calendar Value</Schedule>
      <ExpirationDate>Expiration Date of Timer</ExpirationDate>
    </Timer>
  </Timers>
</ProcessScheduler>
```

Tag	Sample Value
Calendar	System uses default if not defined.
Schedule	e.g. *WN(1);BT(01:00:00)* Next week day (after today) and 1 hour after current time
ExpirationDate	e.g. 2012/12/31

Looping Setting

FUJITSU

- Activity and Subprocess node can be configured to behave as multi instance nodes that simulates the while loop behavior.
  - Processing individual line items in an order
  - sending a response to a list of targets.
- Alternate (old) way to provide looping was to create a looping arrow.

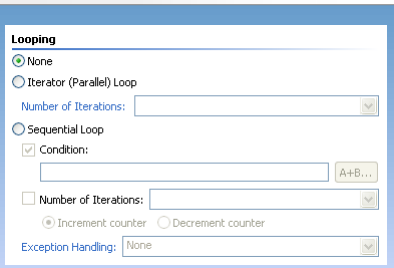


- Looping setting provides easier way to manage while loops, with much more control over execution settings

Looping Settings

FUJITSU

- By Default nodes do not have any looping setting (*None*)
- Looping can be Sequential (looping) or Parallel (iterator)
- Number of loops can be defined as “Number of Iterations” using a numerical counter
- Increment or Decrement counter setting
  - Supported only for Sequential looping
- Exception Handling
  - Stop execution in case of error or ignore and continue.



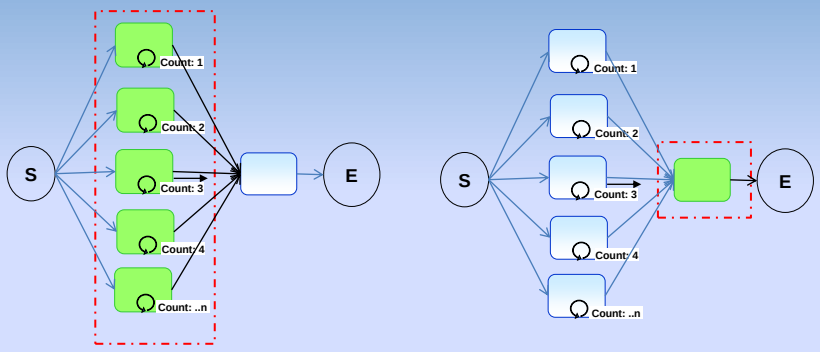
Parallel Looping Behavior

FUJITSU

- Supported for Activity, Subprocess and Chained Subprocess nodes.
- Iterator count is defined by a UDA

- Activity Node
  - “n” number of activity node instances are created depending on UDA value
  - Process waits for all node instances to complete.
- Subprocess Node
  - “n” number of Subprocess node instances and “n” number of child sub processes are instantiated
  - Process waits for all Subprocess instances to ***complete***.
- Chained Subprocess Node
  - 1 instance of chained Subprocess node instance and “n” number of child subprocesses are instantiated
  - Process waits for all Subprocess instances to ***start***.

Parallel Looping Behavior



Parallel Looping Behavior

- In parallel looping, all instances are created and execute together.
- Action Execution
  - Prologue and Epilogue Actions are executed only once irrespective of the number of iterations
  - Role Actions are executed for each iterated node instance
- Due Date applies to the entire loop not individual nodes in the loop.
- Any update to a UDA is instantly available to all instances.
- Iterator instances can be accessed using index variable *$index*

Parallel Looping Behavior

FUJITSU

- Task Recall: When Iterator node is source for Task Recall, recall is possible if at least one of the instances is completed.
  - If all iterator instances are completed
    - The next activated activity (Target) is de-activated
    - Compensation actions for prologue of target and epilogue of source (iterator) are executed
    - Recalled iterator activity instance (source) is re-activated and a task is created and assigned to (recalling) user.
  - If all instances are not completed
    - One of the closed iterator instance is re-activated and assigned to current user.

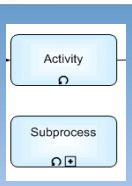
Parallel Looping Restriction

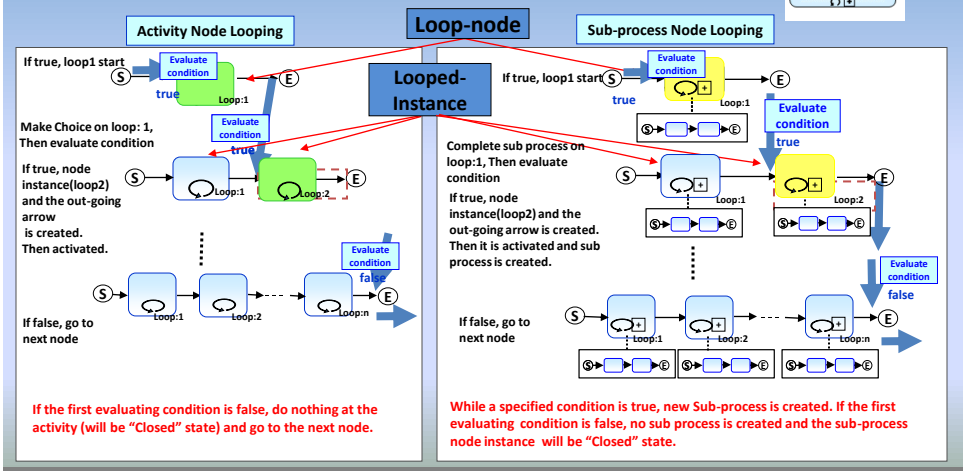
FUJITSU

- The number of outgoing arrows from an activity iterator node is restricted to one
- You cannot use triggers on an iteration-enabled Activity node
- Each iterated instance has the same properties (name, description, and so on).
- Future work items are not supported for iterated nodes.

Sequential Looping Behavior

- Supported for Activity and Subprocess Nodes





Sequential Looping Behavior

- Number of iterations or "max loop count" can be specified using a UDA or a constant value.
- Increment or decrement counter specified is evaluated after every loop instance completes to check if more needs to be created.
- Conditions can be used to control the instance creation
  - Complex condition can be created using UDAs and constants
  - If condition evaluates to false, instance is not created and next loop count is evaluated.
  - Example: *if uda.EmpType == "temp"*

Sequential Looping Behavior

FUJITSU

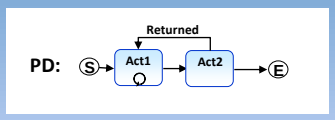
- Exception handling allows to control the behavior if one of the loop instance fails.
  - None
    - no exception handling will be done and process will go to error state
  - Ignore and continue the loop
    - error instance will be ignored and next instance in the loop will be activated
  - Break the loop
    - loop is broken and process moves to next step.

These options will work only for loop-node if an error occurs in prologue, Role, Epilogue Actions or Agent execution

Sequential Looping Behavior

FUJITSU

- If sequential loop node has more than one incoming arrows, closed instances are reused.

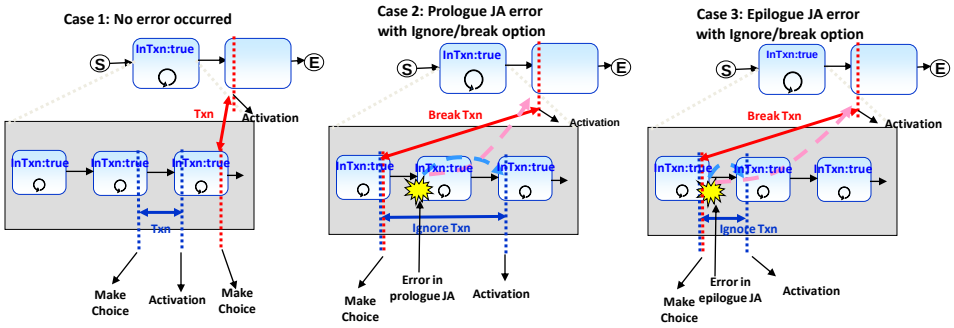


- Example:
  - Act1 activity has sequential node setting
  - Max count = 10
  - 3 instances are created and loop was broken, count UDA is updated to 7
  - On “Returned” Act1 gets activated again and this time it tries to create 7 instances
  - 3 completed instances are reactivated and 4 new are created (unless loop is broken before that)

Sequential Looping Behavior

FUJITSU

- Transaction Management

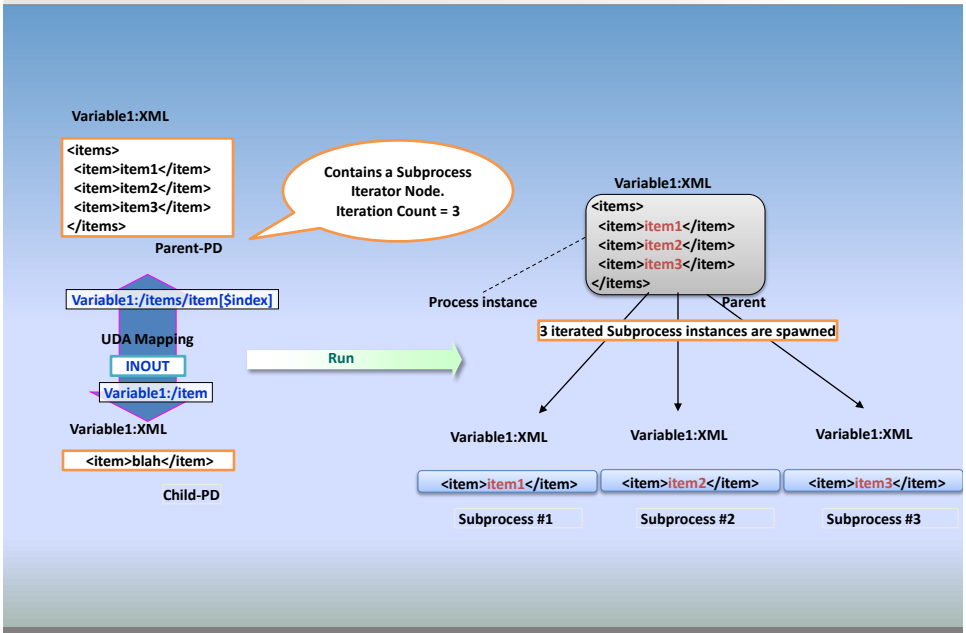


Sequential Looping Restrictions

FUJITSU

- Future work items not supported
- Recalling work items work as normal
- Timer or Due Date cannot be used on Sequential Loop Node
- Process definitions and instances which have Sequential Loop Nodes, cannot be archived or migrated.

Iterator Node – Example



FUJITSU

shaping tomorrow with you