IBM

WebSphere Application Server

Unit 3

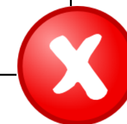
JSR-352 Concepts

© 2017 IBM Corporation

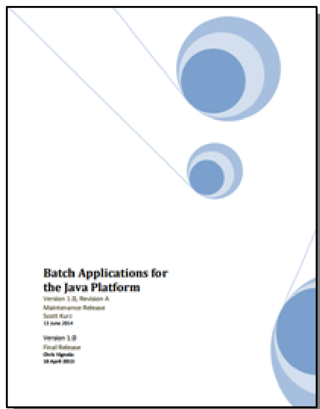
Objective of This Unit

Okay, I understand the key concepts and can carry on a conversation with developers

I'm going deep into Java coding ...



```
public class SleepyBatchlet extends AbstractBatchlet {

    private final static Logger logger = Logger.getLogger(SleepyBatchlet.class.getName());

    /**
     * Logging helper.
     */
    protected static void log(String method, Object msg) {
        System.out.println("SleepyBatchlet: " + method + ": " + String.valueOf(msg));
        // logger.info("SleepyBatchlet: " + method + ": " + String.valueOf(msg));
    }
```



This is based on the JSR-352 Specification, which can be found here:

https://www.jcp.org/en/jsr/detail?id=352

JSRs: Java Specification Requests
JSR 352: Batch Applications for the Java Platform

Stage	Access
Maintenance Release	Download page
Maintenance Review Ballot	View results
Maintenance Draft Review	Download page

Useful Techdoc ...

© 2017 IBM Corporation

A Very Useful Document

http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102706

WebSphere Application Server

Understanding Java Batch (JSR-352)

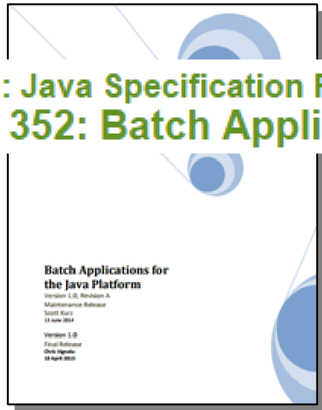
This document can be found on the web at:
www.ibm.com/support/techdocs
Search for document number WP102706 under the category of "White Papers"

This document explains the concepts and some of the details of the JSR-352 specification.

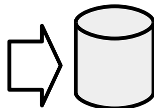
The standard specification ...

© 2017 IBM Corporation

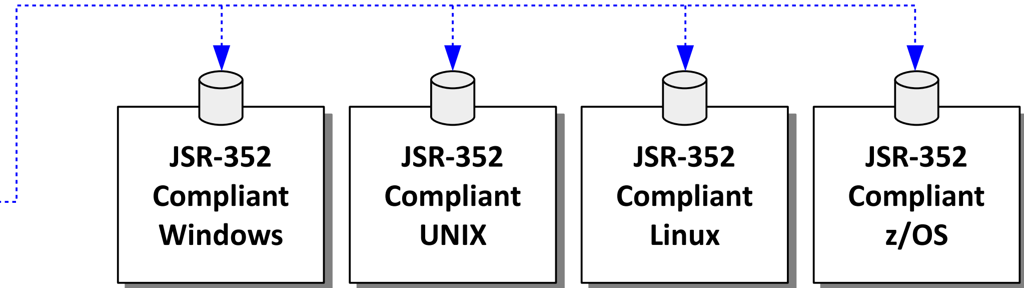
JSR-352 is a *Standard*, Which Means Programs Written to the Standard are Portable



JSRs: Java Specification Requests
JSR 352: Batch Applications for the Java Platform



Java Batch application written to the JSR-352 interface specification



IBM *Operational* Enhancements

JSR-352 Standard

The things you'll learn about in this workshop are nearly all IBM operational enhancements built around the core JSR-352 standard

- The jobOperator implementation: REST interface, batchManager, batchManagerZos
- Job logging, batch events, z/OS SMF records, multi-JVM design, etc.

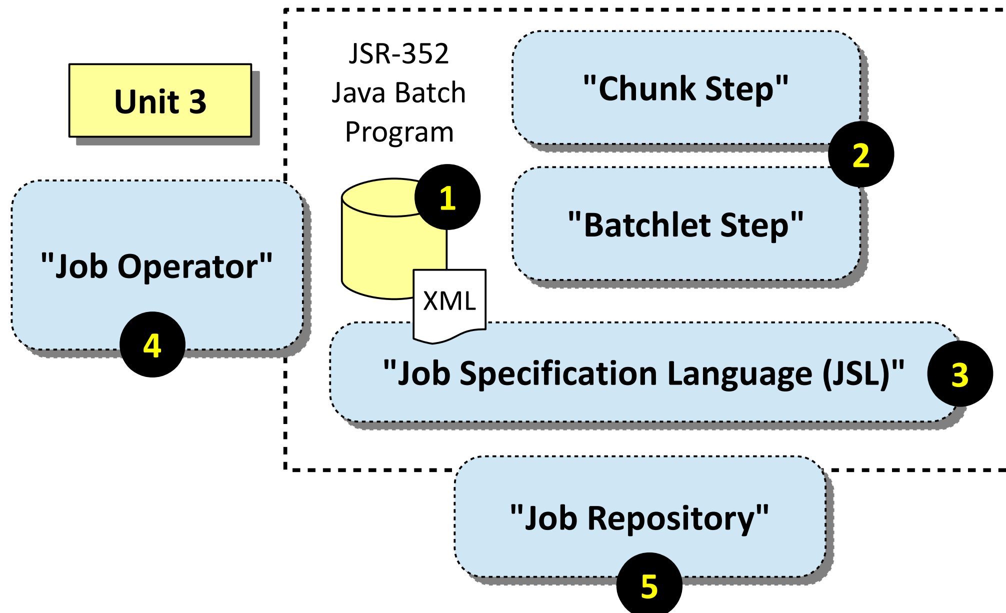
The application has no direct awareness of any of that. The code has no specific requirement needed to use any of that.

© 2017 IBM Corporation

JSR-352 Overview

© 2017 IBM Corporation

Our Picture from Unit 1 - Overview



1. Java Batch Program

You write this based on the defined JSR-352 requirements. This is packaged as a servlet (WAR) and deployed into Liberty just like any other application would be deployed.

2. Job Step Programming Types

Two job step types defined:

Chunk: the looping model we most often associate with batch processing. This includes functions such as checkpointing, commits and rollbacks, and job restarts.

Batchlet: a simple "invoke and it runs" model. This is useful for non-looping functions such as file FTP steps.

3. Job Specification Language (JSL)

An XML specification to describe the batch job: the steps, the Java programs that implement the steps, and the flow of steps within the job.

4. Job Operator

This interface defines how to submit and control jobs. This workshop focuses on the IBM enhancements around this job operator definition.

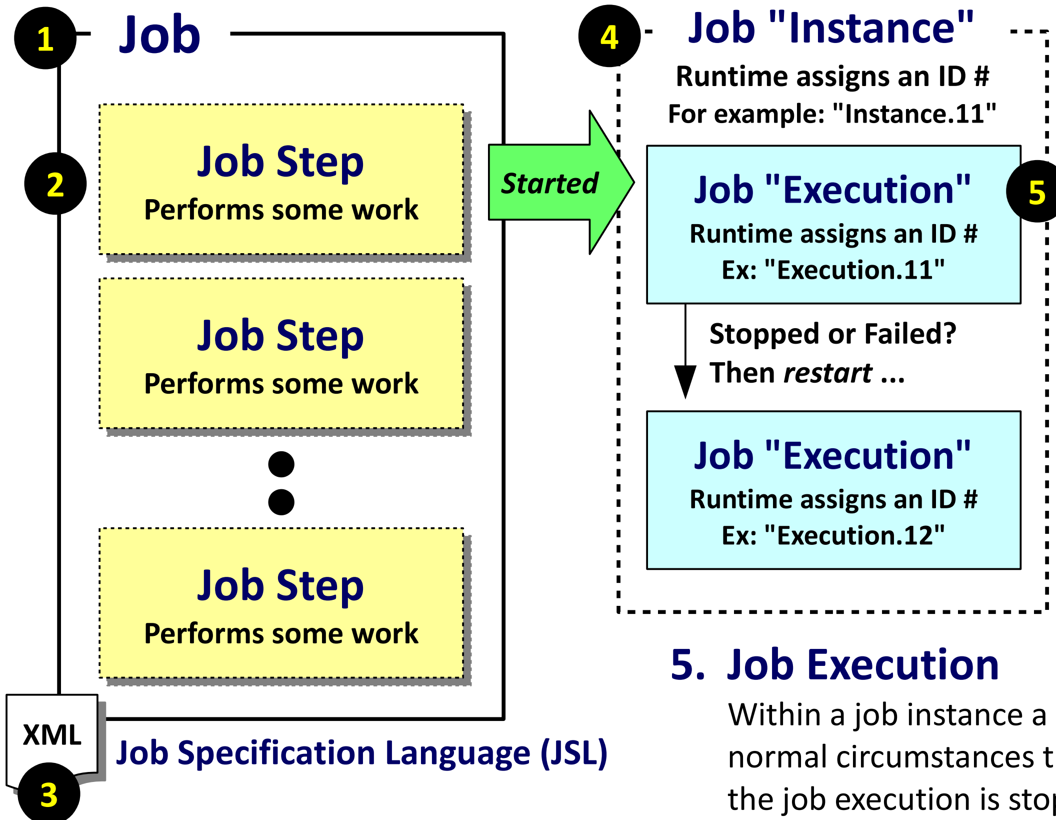
5. Job Repository

The specification calls for a repository to track job submissions and results, but leaves it to the vendor to implement. We'll use IBM DB2 z/OS in workshop.

Key terminology ...

© 2017 IBM Corporation

Some Key Terminology from the JSR-352 Specification



1. Job

A "job" encapsulates all the artifacts of a given batch process.

2. Job Step

A "step" implements a particular portion of your batch job. Your job may have 1 to many steps.

3. JSL

The Job Specification Language describes the components of the job, and defines the flow of execution (called "orchestration").

4. Job Instance

When a job is started, a "job instance" is created and assigned an instance ID.

5. Job Execution

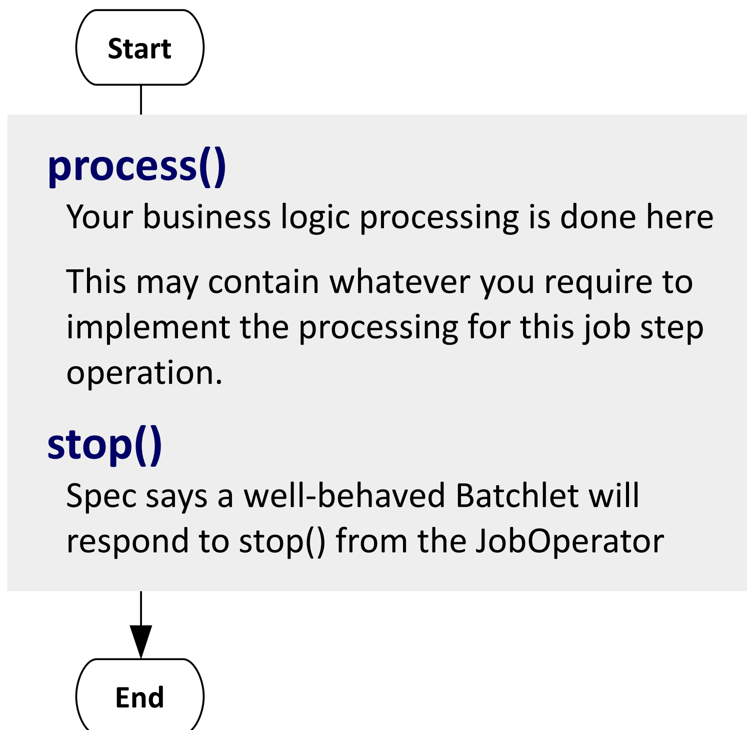
Within a job instance a "job execution" is created and is assigned an execution ID. Under normal circumstances the execution and instance complete and the job completes. But if the job execution is stopped or failed, then you can *restart* within the same job instance. A new "job execution" is created.

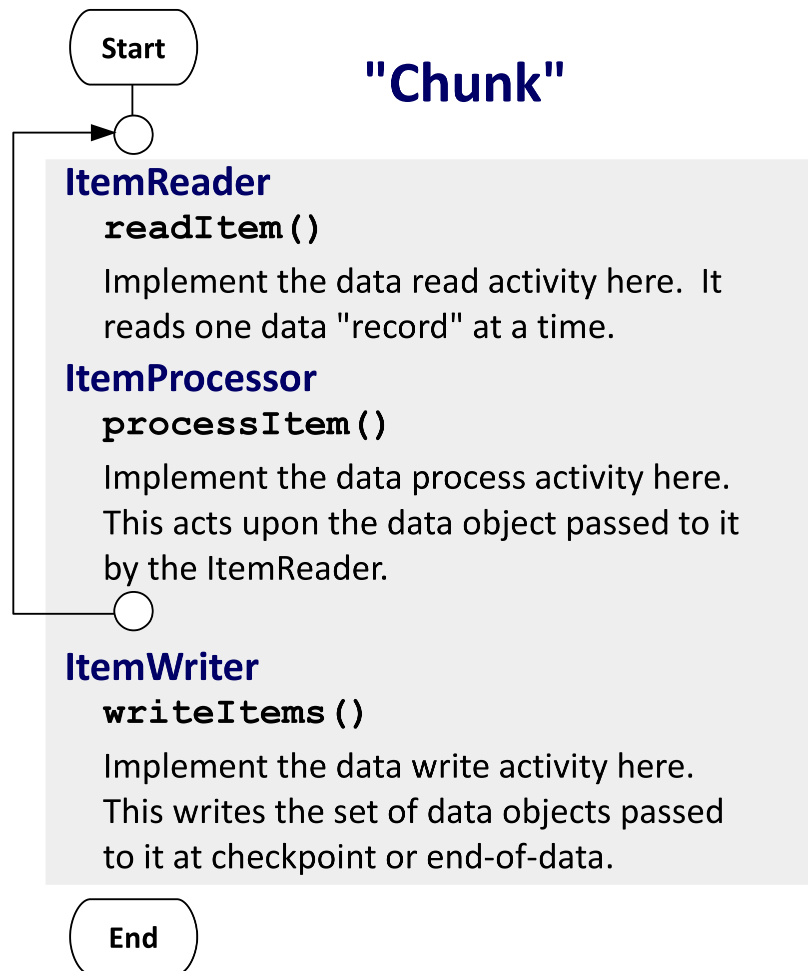
If your job completes successfully and you start it again some time later, it gets a new Job Instance ID and a new Job Execution ID.

Job Step Types ...

© 2017 IBM Corporation

Two Step Types – Batchlet and Chunk





Batchlet code stub example ...

© 2017 IBM Corporation

IBM

Example: Batchlet Step Outline Generated by WDT Tooling

```
package com.ibm.test;
import javax.batch.api.Batchlet;
public class MyBatchlet implements Batchlet {

    /**
     * Default constructor.
     */
    public MyBatchlet() {
        // TODO Auto-generated constructor stub
    }

/**
     * @see Batchlet#stop()
     */
    public void stop() {
        // TODO Auto-generated method stub
    }

/**
     * @see Batchlet#process()
     */
    public String process() {
        // TODO Auto-generated method stub
return null;
    }
}
```

The IBM WebSphere Development Tool (WDT) plugin to Eclipse has code to support JSR-352 programming.

This is what the a batchlet step looks like when it's first generated.

Your batchlet step code goes here

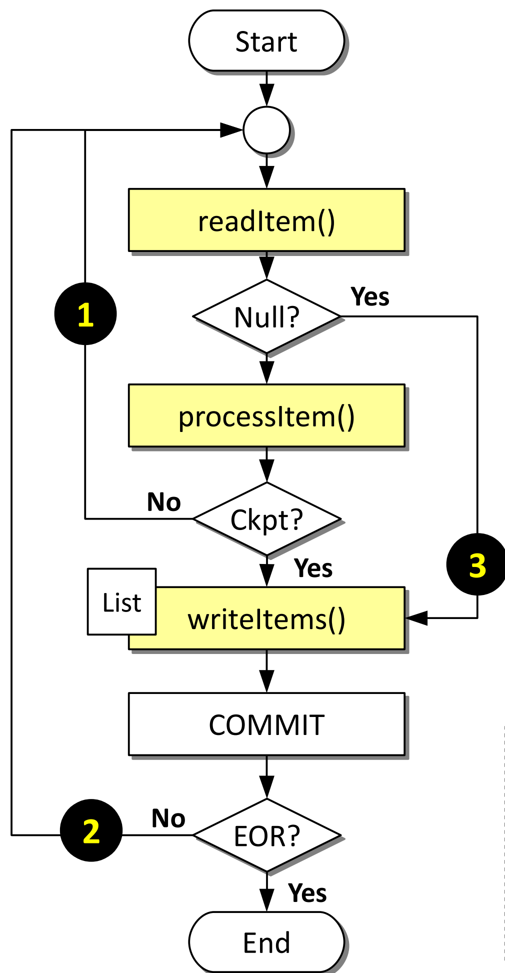
Your batchlet stop processing code goes here

The code for ItemReader, ItemProcessor, and ItemWriter is similar ... but a bit longer and with more methods as per the spec requirements

© 2017 IBM Corporation

Overview of chunk processing ...

Overview of Chunk Processing



1. The Read / Process loop

This loop processes until either a checkpoint interval is met (more on this coming up), or the reader returns a null, which means end-of-records.

The item returned from processItem() is added to a list of items which is eventually passed to the ItemWriter and written.

2. The Checkpoint / Write loop

If a checkpoint interval is reached (more on this coming up), control goes to the ItemWriter. The container passes the ItemWriter a list of items to write. Here your code may either iterate through the list, or perform a bulk insert if the output data resource permits that. At this point a transactional commit is processed if data resource supports transactional context.

3. The End of Records final write and exit

At some point you'll run out of records your ItemReader reads from, and that will trigger a final call to the ItemWriter and a final commit.

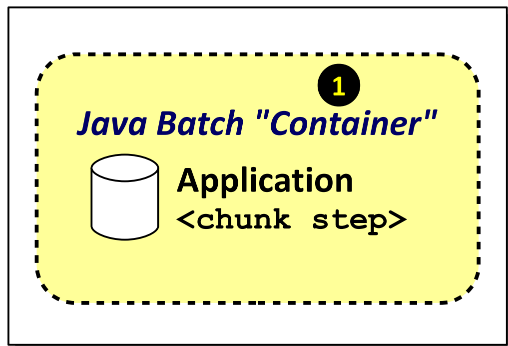
Some detail not shown:

- The open() method in both the ItemReader and the ItemWriter, which the container calls at the start
- The close() method in both the ItemReader and the ItemWriter, which the container calls at the end
- The checkpointInfo() method of both ItemReader and ItemWriter, which is used to maintain information about where within the records the last read and write was accomplished
- The transaction wrapper maintained by the container for transactional resources

Chunk checkpoint control ...

© 2017 IBM Corporation

Chunk Step Checkpoint Control



JSL Job Specification File

```
<chunk
   checkpoint-policy="{item|custom}"
   item-count="{value}"
   time-limit="{value}"
   skip-limit="{value}"
   retry-limit="{value}" />
```

Your chunk code does not handle iterative looping, and does not do checkpoint processing. That is the role of the Batch Container.

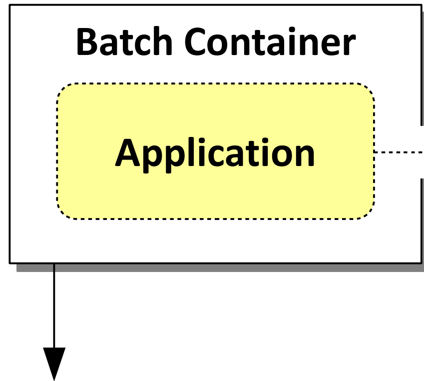
Batchlet steps are another matter … if you're looping in there, that's up to you.

1. **The "batch container"**
   This is the JSR-352 implementation inside the Java EE 7 runtime.
2. **checkpoint-policy**
   This is defined on the <chunk> element of the JSL, and you may either specify "item" (container handles) or "custom" (your own code handles)
3. **item-count and time-limit**
   "item-count" specifies the number of iterations of ItemReader / ItemProcessor for a checkpoint interval (or end-of-data reached). "time-limit" specifies a time interval. This can be specified with item-count. If time-limit reached before item-count, then checkpoint taken.
4. **skip-limit and retry-limit**
   These control the behavior when configured skippable or retryable exceptions are encountered. This allows a job step to survive occasional or intermittent errors you've configured to skip and retry.

Status Codes ...

© 2017 IBM Corporation

Status Codes: "Batch Status" and "Exit Status"



"Exit Status"

- **Under the control of the application**

  If the application does not explicitly set, then the "exit status" ends up being what the container sets for the "batch status"

  Otherwise, the application can override and set its own exit status

- **Container classes: JobContext and StepContext**

  Two container-supplied classes that provide methods accessible to applications to get various context information, and to set context values. For example:

  Job: getJobName(), getProperties(), getBatchStatus(), getExitStatus(), **setExitStatus()**
  Step: getStepName(), getProperties(), getBatchStatus(), getExitStatus(), **setExitStatus()**

- **This is useful in controlling flow of "orchestration"**

  We're going to look at "orchestration" – the flow of steps within a job – in detail.

  Key point here: the JSR-352 specification provides a way to control the flow based on the exit status of jobs. This is very similar to z/OS JCL condition codes. The Job Specification Language (JSL) provides an "On (exit) To (next)" mechanism.

"Batch Status"

- STARTING
- STARTED
- STOPPING
- STOPPED
- FAILED
- COMPLETED
- ABANDONED

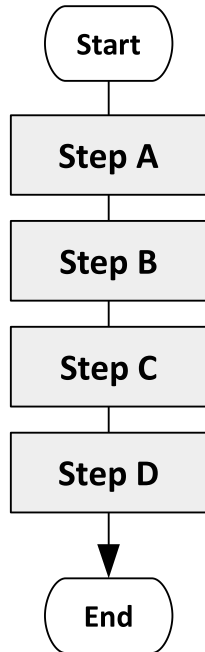
Container sets this for each step and the job overall

Flows and splits ...

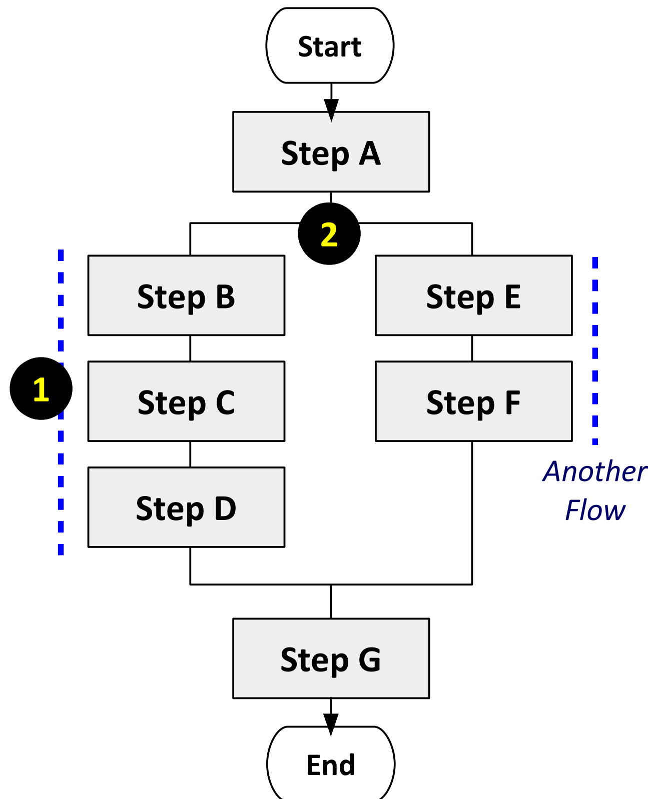
© 2017 IBM Corporation

Job Step "Flows" and "Splits"

A simple sequential processing of steps:



This is what we commonly think of as "batch," and this may be what you do



1. Flow

From the spec: "A flow defines a sequence of execution elements that execute together as a unit."

2. Split

From the spec: "A split defines a set of flows that execute concurrently. A split may include only flow elements as children."

This picture shows two splits, but you may have more if you wish

Key Points:

- This is optional; you don't have to utilize this
- These flows and splits are defined in the Job Specificaton Language (JSL) file; they are not part of the Java code itself.
- The flows and splits for a given job operate within the same JVM; it will utilize separate threads. This is not "step partitioning," which can parallelize across JVMs.

Step partitioning ...

© 2017 IBM Corporation

Job Step "Partitioning"

From the spec: "A partitioned step runs as multiple instances of the same step definition across multiple threads, one partition per thread."

Key Points:

- This is optional; you don't have to utilize this
- Not all processing lends itself to effective partitioning. The data may be arranged in such a way that serial processing is better.

  However, if applicable, this can be an effective way to shorten processing time by parallelizing the processing.
- With IBM Liberty Java Batch, the partitions may run on multiple threads in the same server, or on threads across JVMs.
- This does not happen by magic. The spec: *"Each partition needs the ability to receive unique parameters to instruct it which data on which to operate."*

  You must have knowledge of your data layout, and you must code your partition step to accept and operate on the data-range parameters.

The details of this is beyond the scope of this workshop

It is great technology, but at some point we have to draw a line to keep this workshop 2 days.

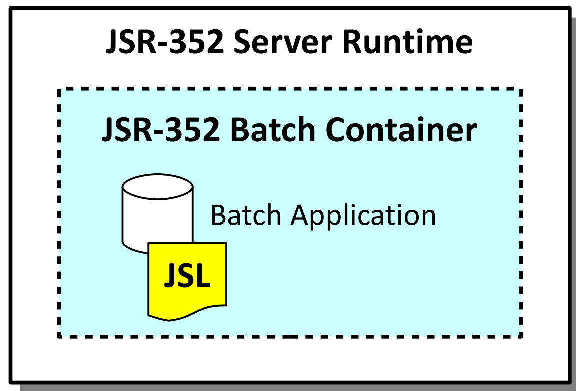
© 2017 IBM Corporation

Job Specification Language (JSL)

This defines and controls the execution of the job

© 2017 IBM Corporation

The Role of the Job Specification Language (JSL) File



The JSL file tells the story about what the batch job is comprised of, and how it is to run:

- Number of steps
- Step type (batchlet or chunk)
- Java class files that implement the steps
- Chunk type checkpoint policies
- Splits and flow processing
- Other control information externalized from the application

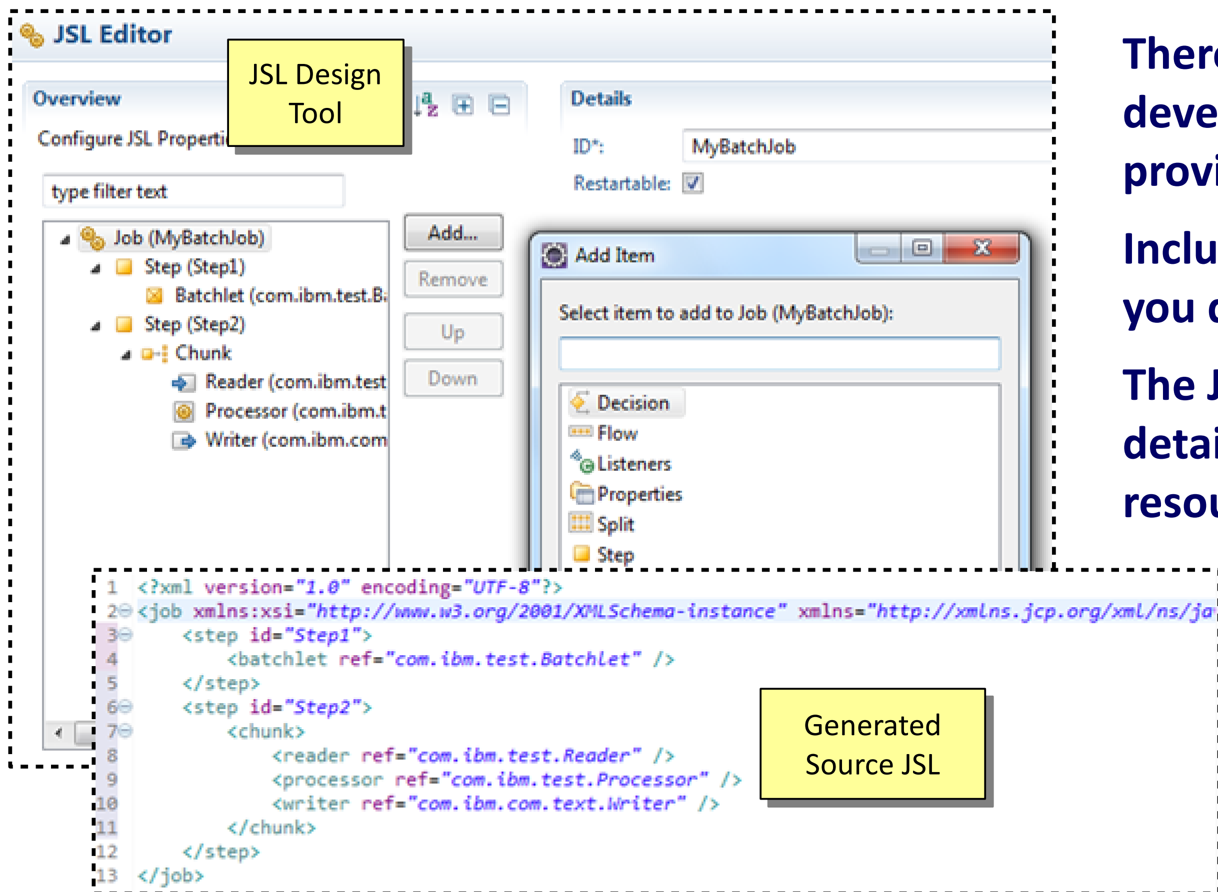
Syntax is XML; the elements are defined in the JSR-352 standard specification document

JSL is typically packaged with the batch application, but IBM Liberty Java Batch supports separate file pointed to at time of submission (called "inline JSL")

WDT Tool to compose JSL ...

© 2017 IBM Corporation

JSL Editor in the WebSphere Developer Tool (WDT) for Eclipse



There is IBM tooling support for Java Batch development. The Techdoc shown below provides details on installing into Eclipse.

Included is a "JSL Editor," which can help you design the job and the associated JSL.

The JSR specification document has further details on the JSL elements. Many other resources exist for details on the XML.

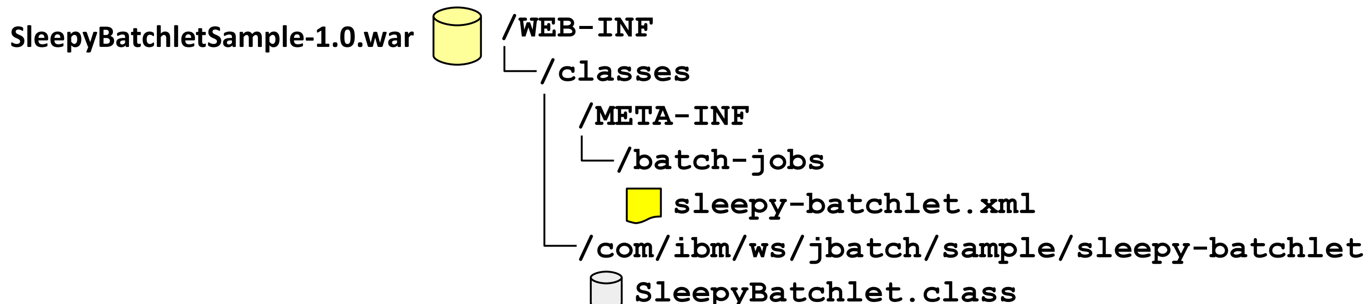
Use this to create the JSL initially. If you wish to modify, then do so manually, or use the JSL Editor.

Techdoc http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102639

SleepyBatchlet JSL ...

© 2017 IBM Corporation

Real JSL – the JSL Packaged with the SleepyBatchlet Sample Program



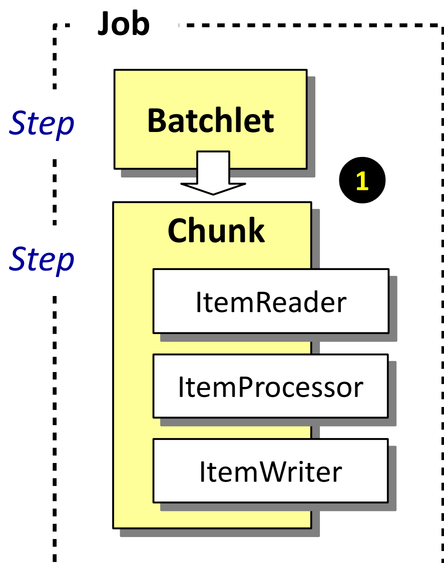

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<job id="sleepy-batchlet" xmlns="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
  1 <step id="step1">                                                       3
     2 <batchlet ref="com.ibm.ws.jbatch.sample.sleepybatchlet.SleepyBatchlet" >
          <properties>
              <property name="sleep.time.seconds" value="#{jobParameters['sleep.time.seconds']}" />
          </properties>                 4
      </batchlet>
   </step>
</job>
```

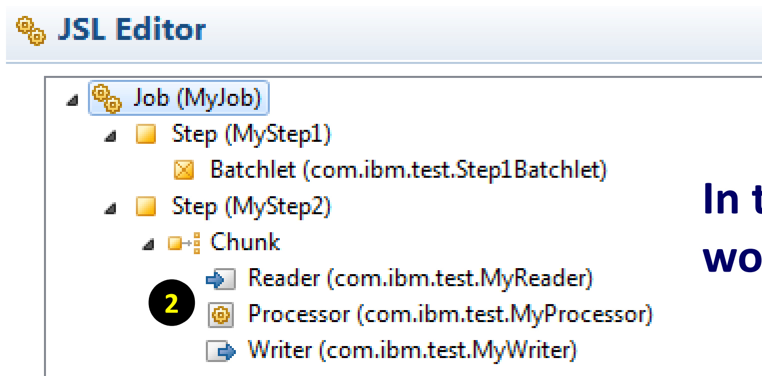
1. **One Step**
   This sample job has one step, which is the minimum.
2. **Step is a batchlet**
   That one step is defined as a "batchlet"
3. **The Java class that implements the job step**
   The ref= points to the Java class for the batchlet
4. **One property is defined**
   You can pass in the number of seconds to "sleep"

Hypothetical two-step job ...

© 2017 IBM Corporation

A Hypothetical Two-Step Job





In the JSL Editor the job and steps would be composed like this

The JSL that gets created looks like this:

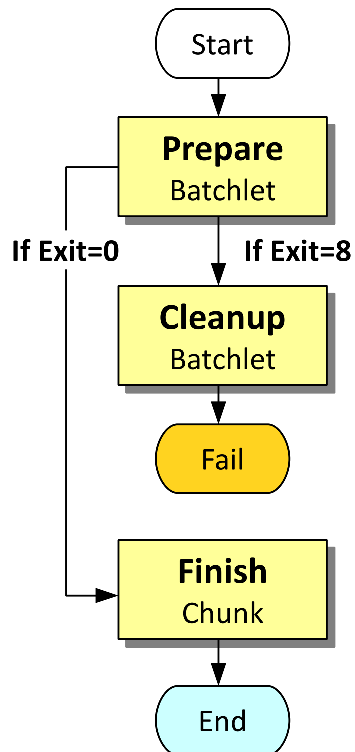
```
3 <?xml version="1.0" encoding="UTF-8"?>
<job ... id="MyJob" restartable="true" version="1.0">
     <step id="MyStep1" next="MyStep2"> 4
          <batchlet ref="com.ibm.test.Step1Batchlet" />
     </step>
     <step id="MyStep2">
          <chunk checkpoint-policy="item" item-count="1000"> 5
               <reader ref="com.ibm.test.MyReader" />
            6  <processor ref="com.ibm.test.MyProcessor" />
               <writer ref="com.ibm.test.MyWriter" />
          </chunk>
     </step>
</job>
```

See speaker notes for notes that correspond to the numbered circles on this chart

Hypothetical with conditional next ...

© 2017 IBM Corporation

Another Hypothetical ... Showing Conditional "Next" Processing



JSL Editor

- Job (MyJob)
  - Step (Prepare)
    - Batchlet (com.ibm.test.Prepare)
    - Next (0)
    - Next (8)
  - Step (Bad-Cleanup)
    - Batchlet (com.ibm.test.Cleanup)
    - Fail (12)
  - Step (Good-Finish)
    - Chunk
      - Reader (com.ibm.test.FinalStep
      - Processor (com.ibm.test.FinalS
      - Writer (com.ibm.test.FinalStep\

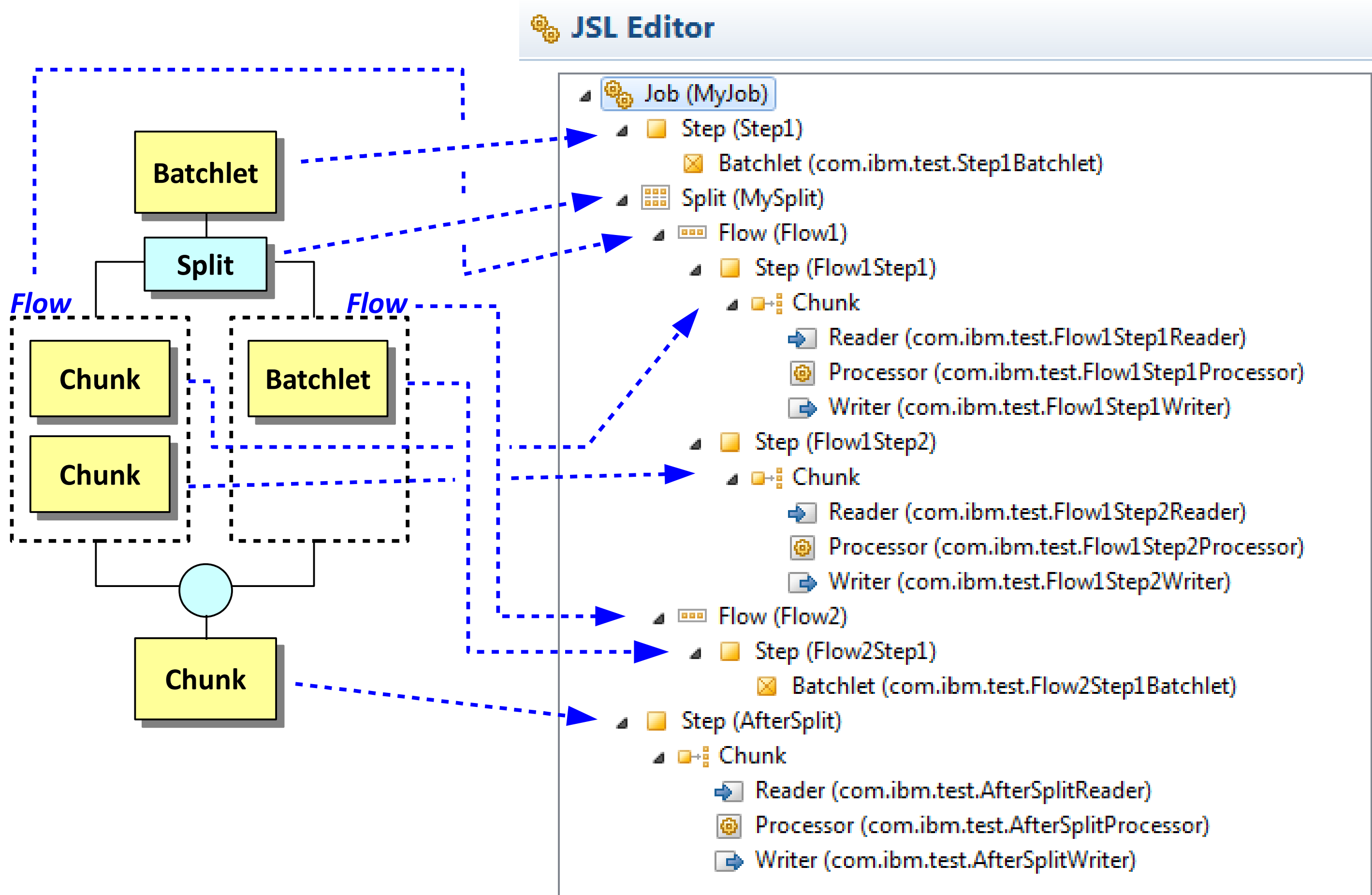
Assumption that Prepare sets the exit Status to 0 or 8 depending on how finishes, and that the Cleanup batchlet always sets an exitStatus of 12.

```
<?xml version="1.0" encoding="UTF-8"?>
<job  ... id="MyJob" restartable="true" version="1.0">
      <step id="Prepare">
            <batchlet ref="com.ibm.test.Prepare" />
            <next on="0" to="Good-Finish" />
            <next on="8" to="Bad-Cleanup" />
      </step>
      <step id="Bad-Cleanup">
            <batchlet ref="com.ibm.test.Cleanup" />
            <fail on="12" exit-status="Post-Cleanup"/>
      </step>
      <step id="Good-Finish">
            <chunk>
                  <reader ref="com.ibm.test.FinalStepReader" />
                  <processor ref="com.ibm.test.FinalStepProcessor" />
                  <writer ref="com.ibm.test.FinalStepWriter" />
            </chunk>
      </step>
</job>
```

Split and flows ...

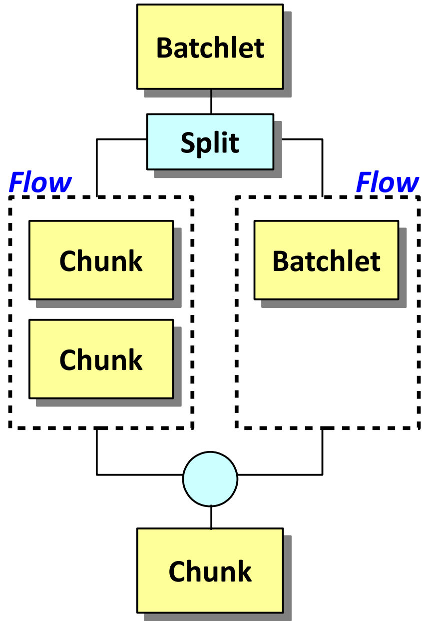
© 2017 IBM Corporation

Yet Another Hypothetical … Split and Two Flows



JSL for this hypothetical …

Same Hypothetical with the Generated JSL




```
<?xml version="1.0" encoding="UTF-8"?>
<job ... id="MyJob" restartable="true" version="1.0">          (1)
  <step id="Step1" next="MySplit">
      <batchlet ref="com.ibm.test.Step1Batchlet" />
  </step>
(2) <split id="MySplit" next="AfterSplit">
      <flow id="Flow1">
          <step id="Flow1Step1" next="Flow1Step2">            (3)
              <chunk>
                  <reader ref="com.ibm.test.Flow1Step1Reader" />
                  <processor ref="com.ibm.test.Flow1Step1Processor" />
                  <writer ref="com.ibm.test.Flow1Step1Writer" />
              </chunk>
Flow 1    </step>
          <step id="Flow1Step2">
              <chunk>
                  <reader ref="com.ibm.test.Flow1Step2Reader" />
                  <processor ref="com.ibm.test.Flow1Step2Processor" />
                  <writer ref="com.ibm.test.Flow1Step2Writer" />
              </chunk>
          </step>
      </flow>
      <flow id="Flow2">
          <step id="Flow2Step1">
Flow 2 (4)    <batchlet ref="com.ibm.test.Flow2Step1Batchlet" />
          </step>
      </flow>
  </split>
  <step id="AfterSplit">              After both flows complete (5)
      <chunk>
          <reader ref="com.ibm.test.AfterSplitReader" />
          <processor ref="com.ibm.test.AfterSplitProcessor" />
          <writer ref="com.ibm.test.AfterSplitWriter" />
      </chunk>
  </step>
</job>
```

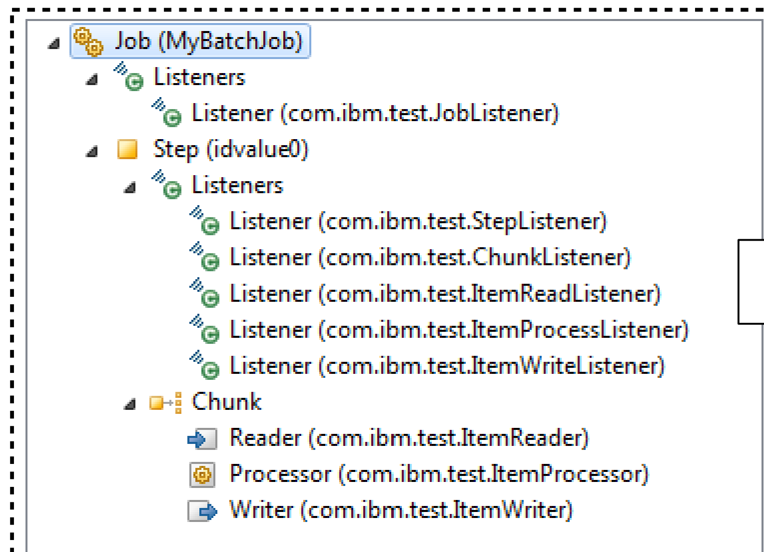
You're not limited to just two flows in a split. And it's possible to have splits inside of splits.

Listeners ...

© 2017 IBM Corporation

"Listeners" – Job, Step, Chunk, ItemRead, ItemProcess, ItemWrite, Skip, and Retry

"Listeners" are callable interfaces behind which you may implement your own code to get control at various points in a job process: start of job, end of job, start of step, end of step, beginning of chunk, end of chunk, etc.



This is what job, step, chunk, read, process and write listeners look like in the JSL Editor

```
<?xml version="1.0" encoding="UTF-8"?>
<job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns=
    <listeners>
        <listener ref="com.ibm.test.JobListener" />
    </listeners>
    <step id="idvalue0">
        <listeners>
            <listener ref="com.ibm.test.StepListener" />
            <listener ref="com.ibm.test.ChunkListener" />
            <listener ref="com.ibm.test.ItemReadListener" />
            <listener ref="com.ibm.test.ItemProcessListener" />
            <listener ref="com.ibm.test.ItemWriteListener" />
        </listeners>
        <chunk>
            <reader ref="com.ibm.test.ItemReader" />
            <processor ref="com.ibm.test.ItemProcessor" />
            <writer ref="com.ibm.test.ItemWriter">
            </writer>
        </chunk>
    </step>
</job>
```

Generated JobListener stub ...

© 2017 IBM Corporation

Example: The Job Listener Generate Stub Code

JobListener.java

```
 1  package com.ibm.test;
 2
 3
 4  public class JobListener implements javax.batch.api.listener.JobListener {
 5
 6      /**
 7       * Default constructor.
 8       */
 9      public JobListener() {
10          // TODO Auto-generated constructor stub
11      }
12
13      /**
14       * @see JobListener#afterJob()
15       */
16      public void afterJob() {
17          // TODO Auto-generated method stub
18      }
19
20      /**
21       * @see JobListener#beforeJob()
22       */
23      public void beforeJob() {
24          // TODO Auto-generated method stub
25      }
26
27  }
28
```

The other listeners are similar in design

The batch container will call these interfaces at the beginning and the end of the job

You may implement any processing here that you wish.

Your code gets control, does what it does, and returns

© 2017 IBM Corporation

Passing Parameters to the Batch Program

Job submission command

```
./batchManager submit ... --applicationName=SleepyBatchletSample-1.0
                    --jobXMLName=sleepy-batchlet.xml --jobParameter=sleep.time.seconds=99 --wait
```

Step Property in JSL

```
<property name="sleep.time.seconds" value="#{jobParameters['sleep.time.seconds']}" />
```

```
import javax.batch.api.BatchProperty;
import javax.inject.Inject;

@Inject
@BatchProperty(name = "sleep.time.seconds")
String sleepTimeSecondsProperty;
private int sleepTime_s = 15;
            Default if nothing passed in
@Override
public String process() throws Exception {

    if (sleepTimeSecondsProperty != null) {
        sleepTime_s = Integer.parseInt(sleepTimeSecondsProperty);
    }

    int i;
    for (i = 0; i < sleepTime_s && !stopRequested; ++i) {
        log("process", "[" + i + "] sleeping for a second...");
        Thread.sleep(1 * 1000);
    }
```

The SleepyBatchlet sample provides the ability to pass in a job parameter and have it injected into the batch program.

The property is the time to "sleep" ... which controls how long the batchlet runs.

A <property> on the batchlet step in the JSL defines the property and opens it to being passed in as a job parameter.

The property is then available to the batch program.

If a non-null value is passed in, then the value passed in overrides the default, and the batchlet "sleeps" for the time specified.

Summary ...

© 2017 IBM Corporation

Summary

This is a big topic, and our objective here was to provide an understanding of the essential framework

We did not get into actual Java coding, but we did highlight how the WDT tool generates the stub classes for you to complete.

The key things to take-away:

- **Two programming models: batchlet and chunk**
- **Job orchestration is accomplished with Job Specification Language (JSL)**
- **Your job can be simple (one step) or sophisticated (splits and flows)**
- **You can pass parameters into jobs from job submission**

A Techdoc to be aware of: "Understanding Java Batch (JSR-352)"

http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102706

© 2017 IBM Corporation

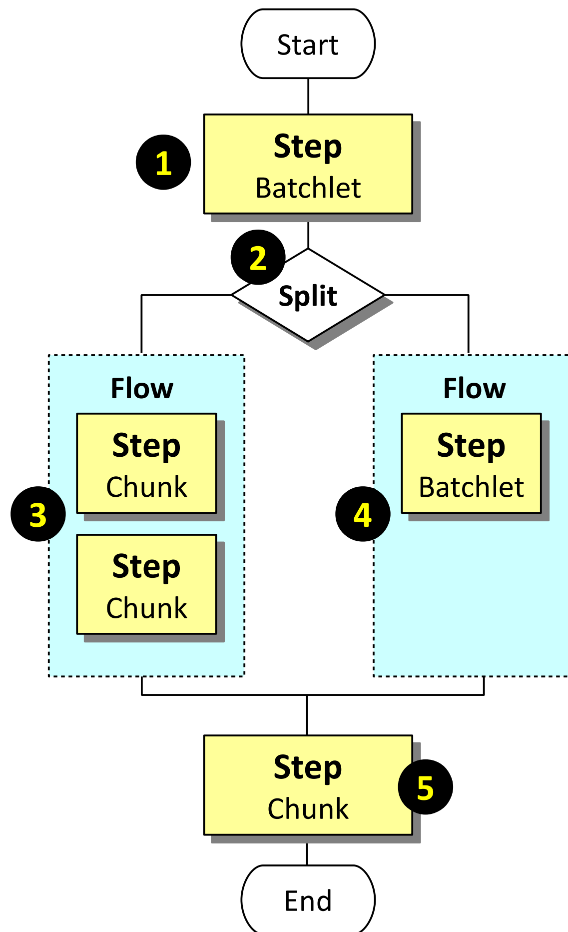
IBM

Screen Shots of JSL Editor

This illustrates how the "split and two flow" example is built

© 2017 IBM Corporation

A Hypothetical Job We'll Use to Illustrate WDT and JSL Editor



1. **Start with a batchlet step**

   We start with a batchlet step for no reason other than it's relatively easy to illustrate.

2. **Start a split with two flows**

   A split may contain only flows, not steps outside of a flow. So we'll create two flows under the split and populate those flows with steps.

3. **Flow with two steps, both chunk**

   For this flow we'll show two steps in the flow. We'll make them both chunk steps for illustration.

4. **Flow with one step, which is a batchlet**

   For the other flow we'll have only one step to show that scenario. We'll make it a batchlet just to show a flow can contain either chunk or batchlets.

5. **Finish with a final chunk step**

   We'll bring the split together with a final chunk step. After that the job ends.

We're not going to show Java coding. We're just going to show what it looks like to compose a job in the JSL editor, and use the generated JSL to explain the elements.

Initial step ...

© 2017 IBM Corporation

Starting Out … Creating the Initial Step

Overview

Configure JSL Properties in this section.

type filter text

Job (MyBatchJob)

Add...

Add → Decision, Flow, Listeners, Properties, Split, Step

Remove

Move Up

Move Down

Details

ID*: MyBatchJob

Restartable:

The Job was created with File → New → Other → Batch Job

Then right-mouse click, Add → Step

type filter text

Job (MyBatchJob)

Step (idvalue0)

Add...

Start limit:

Next:

Add → Batchlet, Chunk, End, Fail, Listeners, Next, Partition, Properties, Stop

Remove

Move Up

Move Down

Then right-mouse click on the new step and Add → Batchlet

The red X's are because the batchlet reference is not yet provided

Job (MyBatchJob)

Step (idvalue0)

Batchlet ()

Add...

Remove

Up

Down

Create batchlet …

Creating the Batchlet

Overview
Configure JSL Properties in this section.
type filter text
Job (MyBatchJob)
Step (idvalue0)
Batchlet ()
Add...
Remove

Details
Reference*:

Click on "Reference"

Create Batchlet Class
Specify class file destination

Project: MyBatch
Source folder: \MyBatch\src
Java package: com.ibm.test
Class name: Batchlet1
Superclass:
Use abstract Java Batch class as superclass
Interfaces: javax.batch.api.Batchlet
Browse...
Add...

Provide package and class name

Click "Finish"

Finish
Cancel

*MyBatchJob.xml
Batchlet1.java

```java
package com.ibm.test;

import javax.batch.api.Batchlet;

public class Batchlet1 implements Batchlet {

    /**
     * Default constructor.
     */
    public Batchlet1() {
        // TODO Auto-generated constructor stub
    }

    /**
     * @see Batchlet#stop()
     */
    public void stop() {
        // TODO Auto-generated method stub
    }

    /**
     * @see Batchlet#process()
     */
    public String process() {
        // TODO Auto-generated method stub
            return null;
    }

}
```

Result is an editor with Java framework for a batchlet step. The required interfaces are pre-populated; you code from there.

Create the split ...

© 2017 IBM Corporation

Create the Split

type filter text

Restartable:

Job (MyBatchJob)
Step (Step1)
Batchlet (co
Add...
Add
Remove
Decision
Flow
Listeners
Properties
Split
Step

Highlight the Job, then click on Add → Split

You can rename created elements here. The split is renamed to "Split1" and the initial step was renamed to "Step1"

JSL Editor

Overview

Configure JSL Properties in this section.

type filter text

Job (MyBatchJob)
Step (Step1)
Batchlet (com.ibm.test.B
Split (Split1)

Add...
Remove
Up
Down

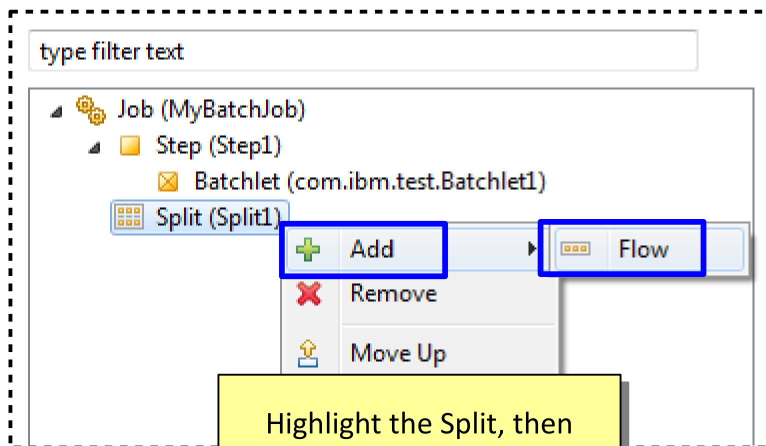
Details
ID*: Split1
Next:

The split appears in sequence after the first step

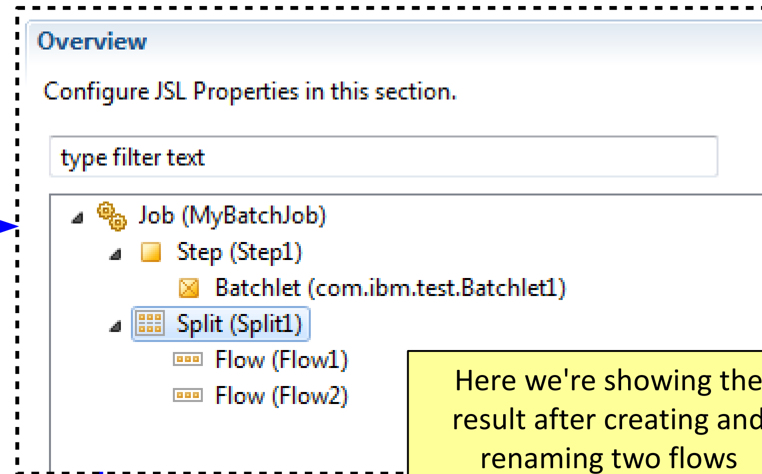
Add flows under split ...

© 2017 IBM Corporation

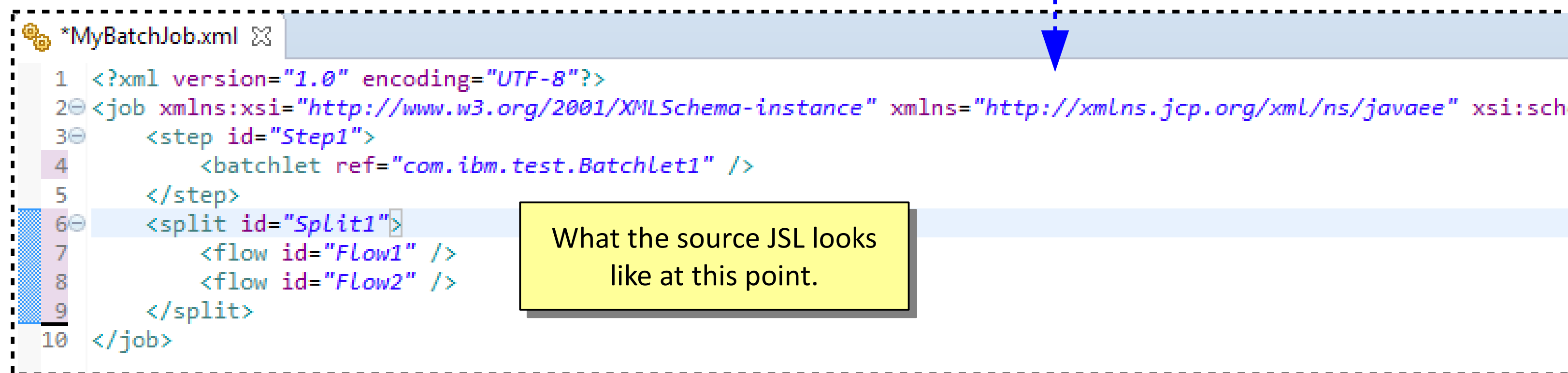
Add Flows Under the Defined Split



Highlight the Split, then click on Add → Flow



Here we're showing the result after creating and renaming two flows



What the source JSL looks like at this point.

Add steps to the flows ...

Add Steps to the Flows

Job (MyBatchJob)
Step (Step1)
Batchlet (com.ibm.test.Batchlet1)
Split (Split1)
Flow (Flow1)
Flow (Flow2)

Add
Remove
Move Up
Move Down

Decision
End
Fail
Flow
Next
Split
Step
Stop

Add...
Remo
Up

Highlight the Flow, then click on Add → Step

Job (MyBatchJob)
Step (Step1)
Batchlet (com.ibm.test.Batchlet1)
Split (Split1)
Flow (Flow1)
Step (Flow1Step1)
Step (Flow1Step2)
Flow (Flow2)

Here we're showing the result after creating and renaming two steps

Next ... add a "chunk" under each of those steps

© 2017 IBM Corporation

Add chunks to the flow steps ...

Add Chunk to the First Step in the Flow

Job (MyBatchJob)
Step (Step1)
Batchlet (com.ibm.test.Batchlet1)
Split (Split1)
Flow (Flow1)
Step (Flow1Step1)
Step (Flow1Step2)
Flow (Flow2)

Add...
Remove
Up
Down

Add
Remove
Move Up
Move Down

Batchlet
Chunk
End
Fail
Listeners
Next
Partition

Highlight the Step, then click on Add → Chunk

Split (Split1)
Flow (Flow1)
Step (Flow1Step1)
Chunk
Rea
Wri
Step (Flow

Up
Down
Checkpoint

Add
Remove
Move Up
Move Down

Checkpoint algorithm
No rollback exception classes
Processor
Retryable exception classes
Skippable exception classes

Highlight the Chunk, then click on Add → Processor

Job (MyBatchJob)
Step (Step1)
Batchlet (com.ibm.test.Batchlet1)
Split (Split1)
Flow (Flow1)
Step (Flow1Step1)
Chunk
Reader ()
Processor ()
Writer ()
Step (Flow1Step2)
Flow (Flow2)

The red X's are because the chunk references are not yet provided

Resolve ItemReader reference ...

IBM

Resolve the References for the ItemReader of the Chunk Step

JSL Editor 3 errors detected

Overview

Configure JSL Properties in this section.

type filter text

- Job (MyBatchJob)
  - Step (Step1)
    - Batchlet (com.ibm.test.Batchlet1)
  - Split (Split1)
    - Flow (Flow1)
      - Step (Flow1Step1)
        - Chunk
          - Reader ()
          - Processor ()
          - Writer ()
      - Step (Flow1Step2)
    - Flow (Flow2)

Add... | Remove | Up | Down

Details

Reference*:

Click on "Reference"

Highlight the Reader

New Item Reader Class

Create Item Reader Class

Specify class file destination

Project: MyBatch

Source folder: \MyBatch\src — Browse...

Java package: com.ibm.test — Browse...

Class name: Flow1Step1Reader

Superclass: — Browse...

Use abstract Java Batch class as superclass

Interfaces: javax.batch.api.chunk.ItemReader — Add... Remove

Provide package and class name

Click "Finish"

Finish | Cancel

Generated ItemReader code ...

© 2017 IBM Corporation

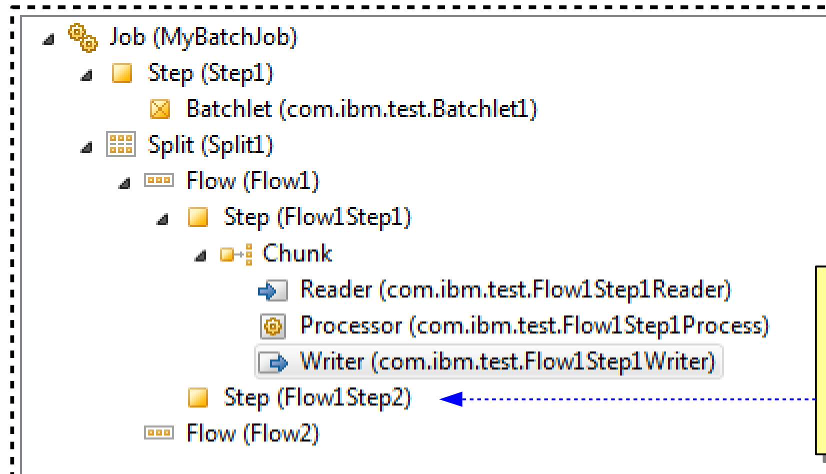
Result: Generated Java for the ItemReader Class that Implements that Reader

*MyBatchJob.xml | Flow1Step1.java

```java
package com.ibm.test;

import java.io.Serializable;

public class Flow1Step1 implements ItemReader {

    /**
     * Default constructor.
     */
    public Flow1Step1() {
        // TODO Auto-generated constructor stub
    }

    /**
     * @see ItemReader#readItem()
     */
    public Object readItem() {
        // TODO Auto-generated method stub
        return null;
    }

    /**
     * @see ItemReader#open(Serializable)
     */
    public void open(Serializable arg0) {
        // TODO Auto-generated method stub
    }

    /**
     * @see ItemReader#close()
     */
    public void close() {
        // TODO Auto-generated method stub
    }

    /**
     * @see ItemReader#checkpointInfo()
     */
    public Serializable checkpointInfo() {
        // TODO Auto-generated method stub
        return null;
    }
```

Result is an editor with Java framework for a ItemReader class. The required interfaces are pre-populated; you code from there.

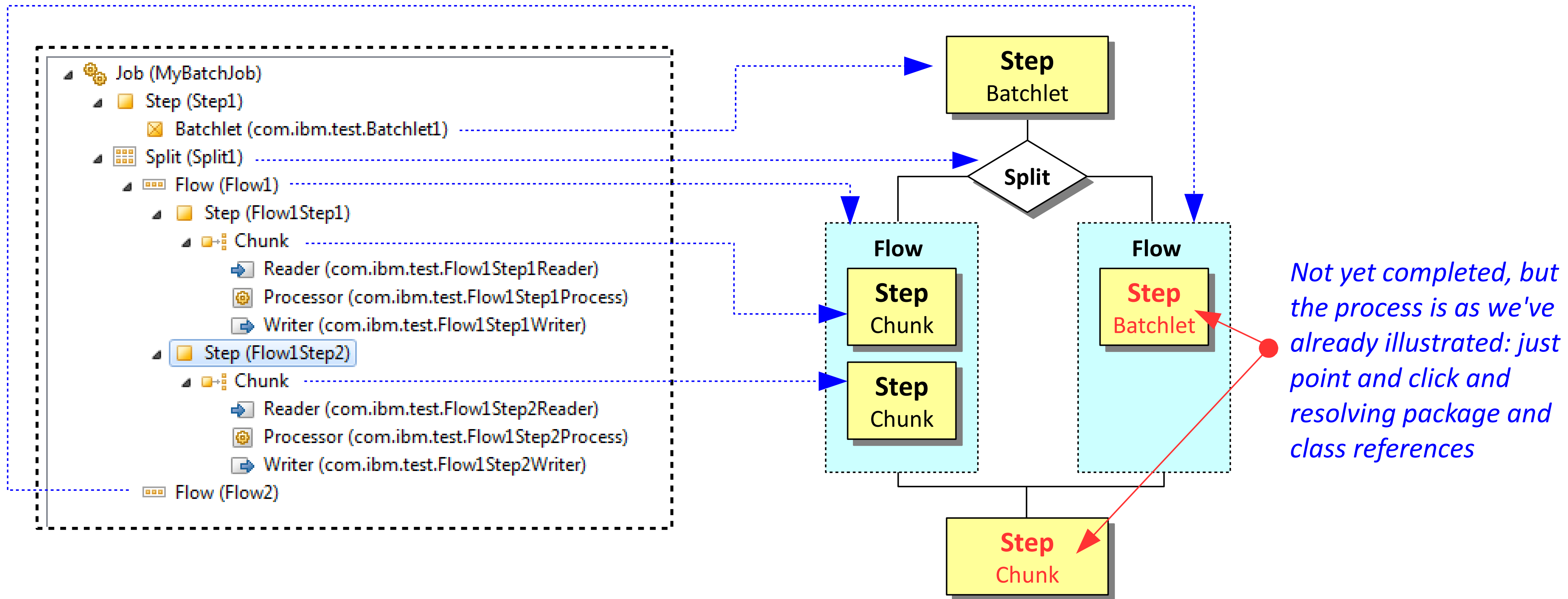
Do the same for ItemProcessor and ItemWriter. Result:



Result after creating second chunk step ...

Add Chunk to Second Step in the Flow and Resolve the References

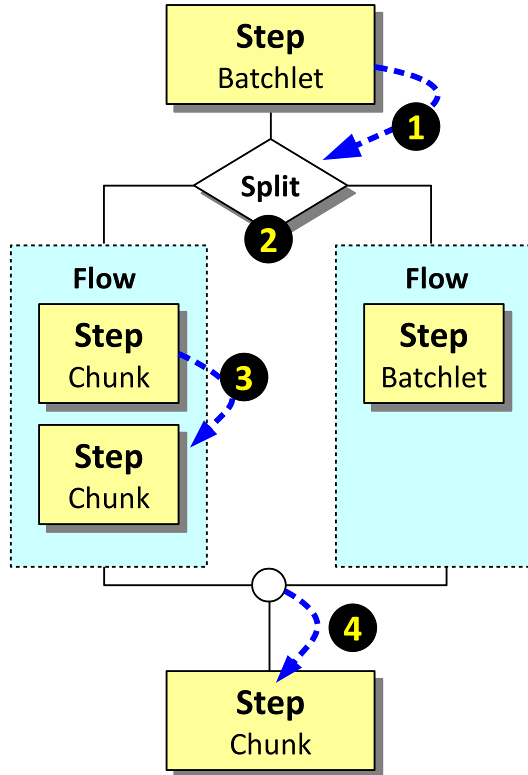
We're now starting to repeat the same steps over and over, so we won't show detail



© 2017 IBM Corporation

IBM

Add "Next" References so Flow Through Job Specified in the JSL



1. Set "Next" on Step1 to Indicate "Split1"

Configure JSL Properties in this section.

type filter text

- Job (MyBatchJob)
  - Step (Step1)
    - Batchlet (com.ibm.test.Batchlet)

ID*: Step1

Start limit:

Next: Split1

Allow start if complete:

2. Split will go down each defined flow in the split

You *do* define a "Next" on the Split, but not to the flows. You define "Next" to indicate where to go *after the split comes back together*. See #4 below.

3. Set "Next" on Flow1Step1 to Indicate "Flow1Step2"

Details

ID*: Flow1Step1

Start limit:

Next: Flow1Step2

4. Set "Next" on split to go to "FinalStep"

Details

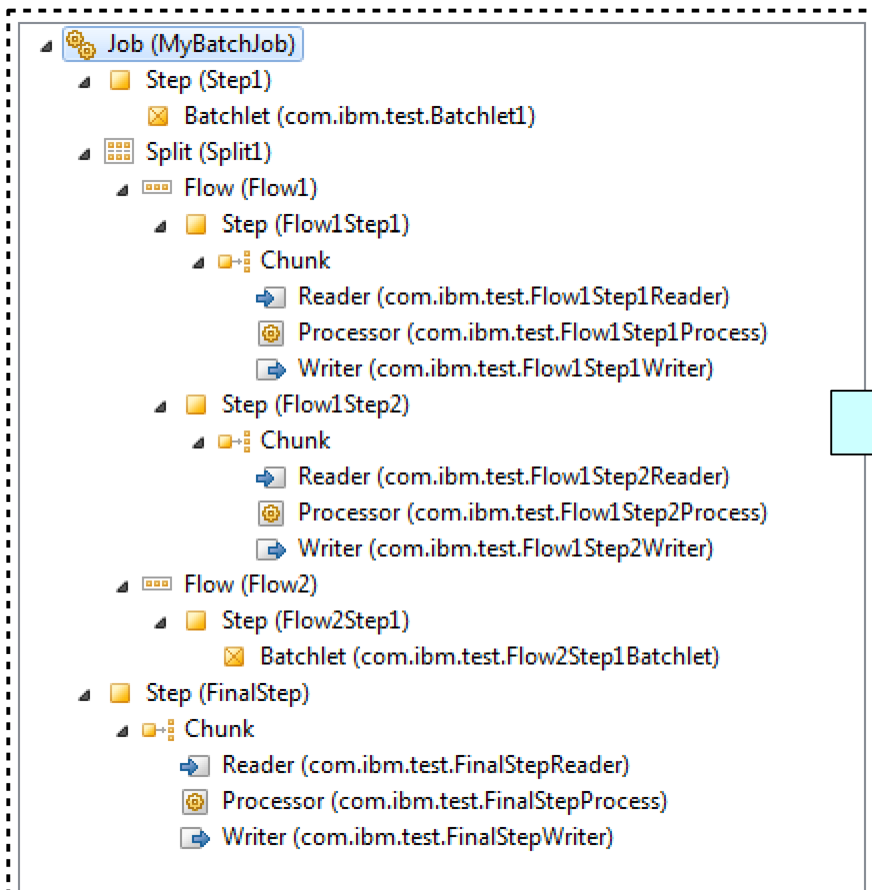
ID*: Split1

Next: FinalStep

The final result ...

© 2017 IBM Corporation

All the Steps, Splits and Flows … and the Generated JSL




```xml
1  <?xml version="1.0" encoding="UTF-8"?>
2  <job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http:/
3      <step id="Step1" next="Split1">
4          <batchlet ref="com.ibm.test.Batchlet" />
5      </step>
6      <split id="Split1" next="FinalStep">
7          <flow id="Flow1">
8              <step id="Flow1Step1" next="Flow1Step2">
9                  <chunk>
10                     <reader ref="com.ibm.test.Flow1Step1Reader" />
11                     <processor ref="com.ibm.test.FlowStep1Process" />
12                     <writer ref="com.ibm.test.Flow1Step1Writer" />
13                 </chunk>
14             </step>
15             <step id="Flow1Step2">
16                 <chunk>
17                     <reader ref="com.ibm.test.Flow1Step2Reader" />
18                     <processor ref="com.ibm.test.Flow1Step2Process" />
19                     <writer ref="com.ibm.test.Flow1Step2Writer" />
20                 </chunk>
21             </step>
22         </flow>
23         <flow id="Flow2">
24             <step id="Flow2Step1">
25                 <batchlet ref="com.ibm.test.Flow2Step1Batchlet" />
26             </step>
27         </flow>
28     </split>
29     <step id="FinalStep">
30         <chunk>
31             <reader ref="com.ibm.test.FinalStepReader" />
32             <processor ref="com.ibm.test.FinalStepProcess" />
33             <writer ref="com.ibm.test.FinalStepWriter" />
34         </chunk>
35     </step>
36 </job>
```

© 2017 IBM Corporation