

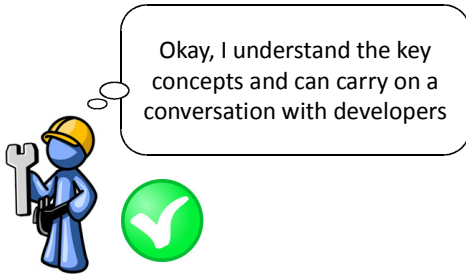


WebSphere Application Server

Unit 3

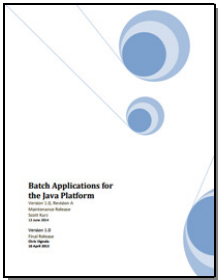
JSR-352 Concepts

Objective of This Unit



```
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

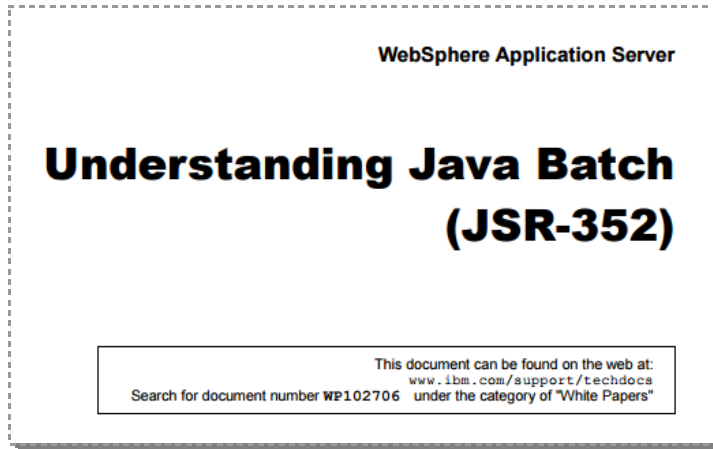
The subject of this unit is one of those where it's important to define the objective so the proper expectations are set. We'll start by saying what this unit is **not** about: it is *not* a deep review of Java coding practices and the JSR-352 specification. We will cover *some* of that, but the anticipated audience for this workshop is not really focused Java programmers; it's more aimed at those who will be responsible for overseeing the proper operations of the IBM WebSphere Liberty Java Batch environment on z/OS.

The objective of this unit is to give the non-programmer – or the Java programmer who is not familiar with batch programming – enough to be conversant in the key terminology of the standard and how it is structured.

For those interested in the programming specifics of the JSR-352 specification, we encourage you to go to the source: the JSR-352 specification itself. That can be found at the URL shown on the chart, and there you can download the PDF that includes the specifics of every element of the defined specification.

A Very Useful Document

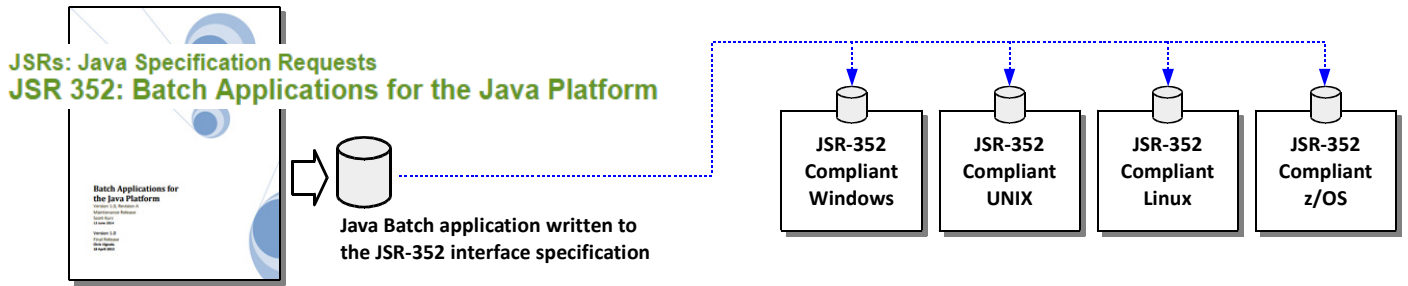
<http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102706>



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

The Techdoc referenced on this chart is useful to help you understand the key concepts of the JSR-352 specification. The Techdoc is written in an approachable style. It is a very helpful document.

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



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.

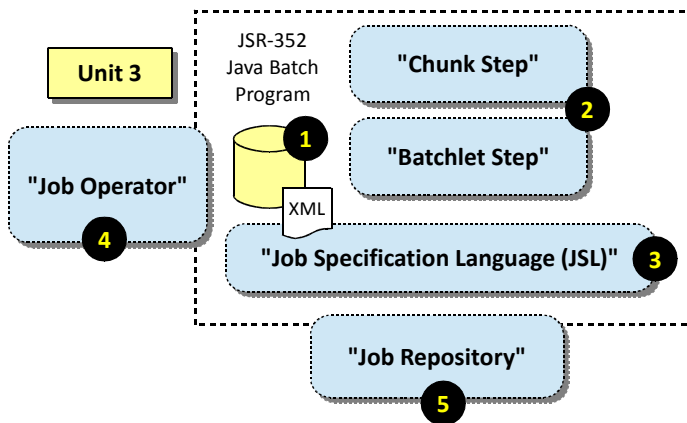
Before we get going, it's important to remember that the JSR-352 open standard is an *open standard*. That means programs written to the open standard can be run on *any* platform that implements a standard-compliant version of the standard.

In this workshop we are covering quite a few IBM operational enhancements, but none of those change the open standard programming specification. Any JSR-352 compliant application can run in the IBM WebSphere Liberty Java Batch runtime, even with those operational enhancements in play. The application has no direct awareness of that. The code does not need to be modified in any way to run in IBM's runtime, or to benefit from those operational enhancements.



JSR-352 Overview

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

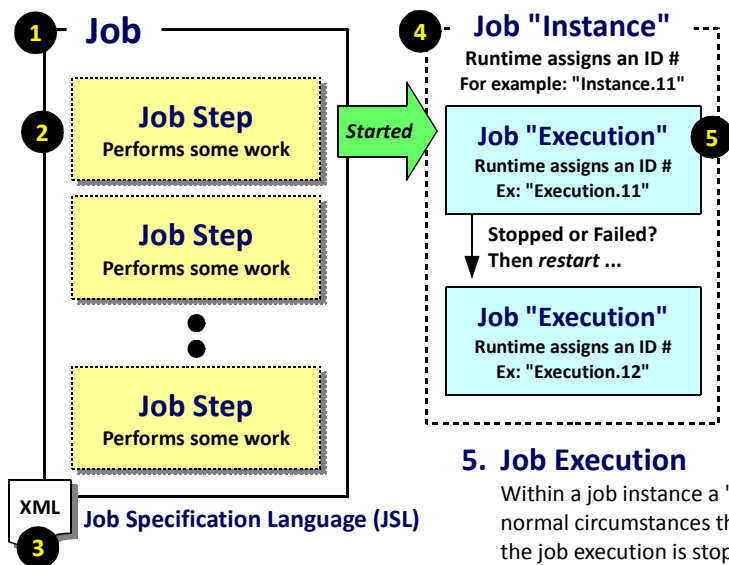
This interface defines how to submit and control jobs. This workshop focuses on the IBM enhancements around this job operator definition.

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.

This is the picture we showed back in Unit 1 of this workshop. It shows five high-level concepts of a the JSR-352 specification. In this unit we're going to focus mostly on items #2 and #3, and we'll add a few other JSR-352 concepts as well.

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.

Let's start with some terminology associated with the JSR-352 specification:

- 1. Job** – a "job" is a logical representation of the activities that make up a given batch process. Jobs are made up of one or more "steps" that do some specific batch task. We're careful to differentiate "job" from a job that has run or is running because those cases are covered under "job instance" and "job execution." Think of a "job" as a set of batch artifacts that together will perform a defined batch process.
- 2. Job Step** – a "job step" is what performs the batch processing. A batch job may have between 1 and many job steps. The JSR-352 specification defines two *types* of steps: *batchlet* and *chunk*. We'll cover those in more detail in an upcoming chart.
- 3. Job Specification Language** – the artifacts that comprise a job require some mechanism to describe the artifacts so the runtime can order the execution in the intended way. This is done with "job specification language," which is an XML file that defines the job, the job properties, the steps, the step properties, the order in which the steps are executed, as well as other properties related to the intended execution of the job.

Note: for those familiar with z/OS JCL, the concept here is identical to the concept of JCL. The difference is the syntax. JCL uses // and other syntax expressions; JSL uses XML.

- 4. Job Instance** – a "job instance" is a representation of a job that has been started within a certain context; for example, the "every day at 10:00pm this job is run" context. When the job is started at 10:00pm the batch container will assign a "job instance" number to the started job. If the job runs to successful completion, that job context is completed. But suppose that job fails for some reason and you have to correct something and restart it. You restart at 10:42pm. The restarted job is still within the "every day at 10:00pm" context, so we don't create a new job instance. We do, however, create a new "job execution," which we describe next.

This is best described in conjunction with "job execution," so let's get that discussed, then we'll revisit "instance" and "execution" together.

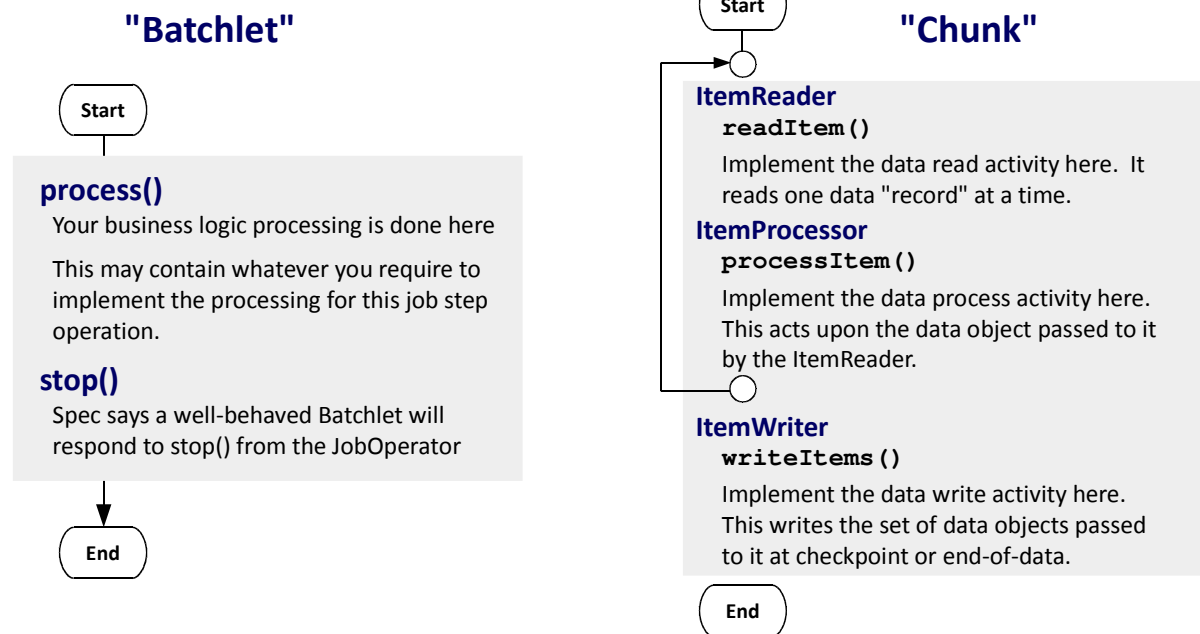
- 5. Job Execution** – a "job execution" is a discrete running of the job. When a job is started, the runtime assigns a "job execution" number. If the job fails and is restarted, a new "job execution" number is assigned.

Scenario 1 -- Imagine a job that you start and everything works perfectly. When you start the job, a "job instance number" is created, and a "job execution" number is created as well. The job completes successfully. An hour later you run the job *again* and it runs successfully. You have a new instance number a new execution number.

Scenario 2 – You start the job and a new "job instance number is created," and a "job execution" number is created as well. The job fails. You correct the problem and restart. The same instance number is used, but a new execution number is created. The job fails again. You correct the problem and restart. The same instance number is used, but another execution number is created. This time the job runs to completion. In this scenario you have one "instance" and three "executions."

Understanding this is important because it goes directly to job logging and retrieving job logs. The way you uniquely identify a job log is by citing the job-instance / job-execution numbers.

Two Step Types – Batchlet and Chunk



© 2017 IBM Corporation

8

Batchlet code stub example ...

Okay, let's dig into the difference between a "batchlet" step and a "chunk" step.

A "batchlet" step is one that is called by the batch container – it calls process() -- the batchlet does whatever it's coded to do, and it stops. There is no container-managed looping or checkpointing going on. If there's any looping or committing of data, it's your custom code doing it. The batchlet is a very simple "invoke, it runs, it completes" model. The JSR-352 specification says a properly written batchlet should implement the stop() method, and your code should provide a way for the calling of stop() to cease execution of the batchlet step and return to the container.

Batchlets are handy for batch steps that don't necessarily iterate over data, but do something like FTP a file, or sort a large set of data. It's also a handy means of taking existing Java main() programs you've written and bringing them into the JSR-352 framework.

A "chunk" step is called that because it operates on data in "chunks" – or a collection of data gathered up during a looping process and written out at a "chunk" (or checkpoint) interval. The specification says a chunk step must implement an ItemReader, an ItemProcessor, and an ItemWriter. This is what we provided in the notes of the previous chart:

- **ItemReader** – this interface is called by the batch container to read a "record" of data. What this does is invoke *your* code that you've implemented behind the interface. A "record" is whatever you define that term to mean. The container calls ItemReader, and your code gets a "record."
- **ItemProcessor** – this interface is called by the batch container to process the record retrieved by ItemReader. What this does is invoke your code you've implemented behind the interface. It "processes" the data from ItemReader in whatever fashion you've provided in the ItemProcessor code.
- **ItemWriter** – this interface is only called when a checkpoint interval is reached, or ItemReader indicates no more records are available. Calling ItemWriter causes your code you've implemented behind the interface to write the data collected by the iterative calling of ItemReader and ItemProcessor since the last checkpoint.

Chunk steps are what we commonly think of as "batch" ... that is, a looping process that works through a large set of data doing some processing on each set of data.

It's tempting to think that *you* control the looping, checkpointing, and commit processing in this model, but that's *not* the case. Your code, which sits behind the ItemReader, ItemProcessor, and ItemWriter interfaces, is limited to just those three functions – reading, processing, and writing. The looping and checkpointing is handled by the *container*, and the parameters that control this behavior are externalized to the Job Specification Language (JSL) file for the job. Let's look at that next.

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

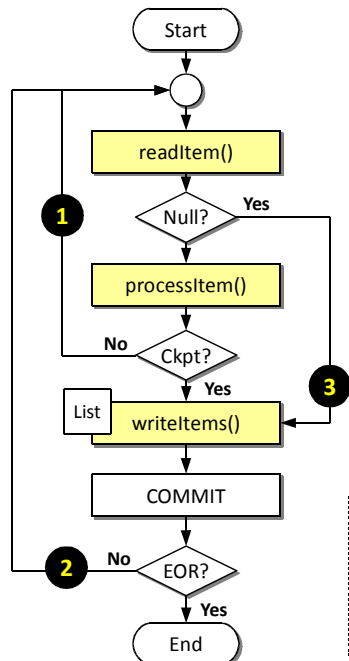
To help make this a bit more "real," let's see an example of the batchlet code stub generated by the IBM WebSphere Development Toolkit plugin to Eclipse for JSR-352 coding. This shows what is generated for a batchlet step.

Note: showing ItemReader, or ItemProcessor, or ItemWriter is more difficult on a chart because they're much longer due to the specification calling for more methods in those classes. Our objective here is *not* to go into programming specifics, but rather to show you that tooling generates the framework and you fill in the blanks.

We have already mentioned that a batchlet is basically a "invoke it and it runs until it's done" model. The process() step is what the container calls when a batchlet step is defined. The process() step is highlighted. Your code is placed there.

The specification says a properly-behaved batchlet step implements a mechanism for the container to call a stop() method to stop the batchlet processing. You can see the stop() method highlighted on the chart. Your stop processing would go there.

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

© 2017 IBM Corporation

10

Chunk checkpoint control ...

The idea of a "chunk" step is that the program will iterate over a set of data until there's no more data to be read. The important thing to keep in mind is your code does not do the iteration; that's the role of the batch container. The "chunk" step type defines three code implementations you provide -- `ItemReader`, `ItemProcessor`, and `ItemWriter` -- and the container does the looping.

There are two "loops" that take place -- (1) the read/process loop, and the (2) checkpoint/write loop:

- 1. Read/Process** -- In this loop the container calls your `ItemReader` to get a data record (from wherever your `ItemReader` is coded to get that data, and however many items of data your `ItemReader` is coded to retrieve). Your code is executed, and it returns an "item" object (again, however your code has chosen to construct this object). The container then calls your `ItemProcessor` and passes it the item returned from `ItemReader`. Your `ItemProcessor` code then does the processing against the data object defined by your business requirements. The `ItemProcessor` returns an object. This loop continues until either (a) a checkpoint interval is reached, or (b) a null is returned from your `ItemReader`, indicating that there's no more data to be read.

Note: as it's doing this loop it's building an "object list" of processed data objects from the `ItemProcessor`. That "object list" is what's passed to your `ItemWriter` where the code implemented for the `writelItems()` method works through the list and writes the data, either one record at a time, or as a multi-record write operation. That part is up to you, based on your knowledge of what the output data resource is capable of doing.

- 2. Checkpoint/Write** -- at some point you will either encounter a checkpoint interval, or you'll run out of input records. Let's assume for now you reach a checkpoint interval. (We'll show how that's defined next.) The container breaks out of the "read/process" loop and calls the `ItemWriter`, passing it the list of data objects created in the read/process loop. The code that implements the `writelItems()` method works through the list of objects and does the writing, either record by record or in bulk, depending on the capabilities of the data resource being written to.

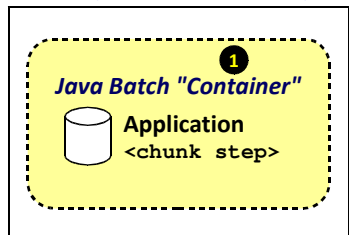
Note: here's where we get into some detail that's not illustrated on the chart. In reality the container is maintaining a transaction, and a commit processing will take place at this point. If the data resource does not honor transactions, the commit will be ignored and the data is written. But if the data resource is something like DB2, then the commit tells DB2 to write the records. Another bit of detail: at this point your code should use `checkpointInfo()` to maintain information about the last record processed so if the job needs to be restarted, the container can know where it left off.

- 3. End of Records** -- at some point your `ItemReader` will return a null, which will tell the container there's nothing more to read. This will trigger a call to `ItemWriter` (regardless of where you are in the checkpoint interval) where a final write/commit process will take place. From there the chunk step completes, and you move on to the next step in your job, or the job completes.

We have hinted at some details but not spelled them out -- specifically, how your code would keep track of records read and written, and the transactional context maintained by the container. For this unit we'll skim over that and move on to other concepts.

Chunk Step Checkpoint Control

Java EE Runtime Server
Liberty z/OS for this Workshop



JSL Job Specification File

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

Numbered circles in the original image: 2 is above 'checkpoint-policy', 3 is above 'time-limit', and 4 is above 'retry-limit'.

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.

As we discussed in the notes of the previous chart, the iterative looping and checkpointing of a chunk step is under the control of the batch container, not your code. Your code reads a record (ItemReader), processes a record (ItemProcessor), and when a checkpoint interval is reached, it writes a record (ItemWriter).

Note: your code is not totally unaware of this. One of the things your ItemReader and ItemWriter must do is keep track of *where* within the data the records have been read. This is so a restart can be done from the last checkpoint can be known and the iteration can pick up from there. For example, if you're reading from DB2 you'd maintain your cursor position in the ItemReader code. The checkpointInfo() method of ItemReader returns the current checkpoint data for this reader.

Let's follow the numbered circles in the picture:

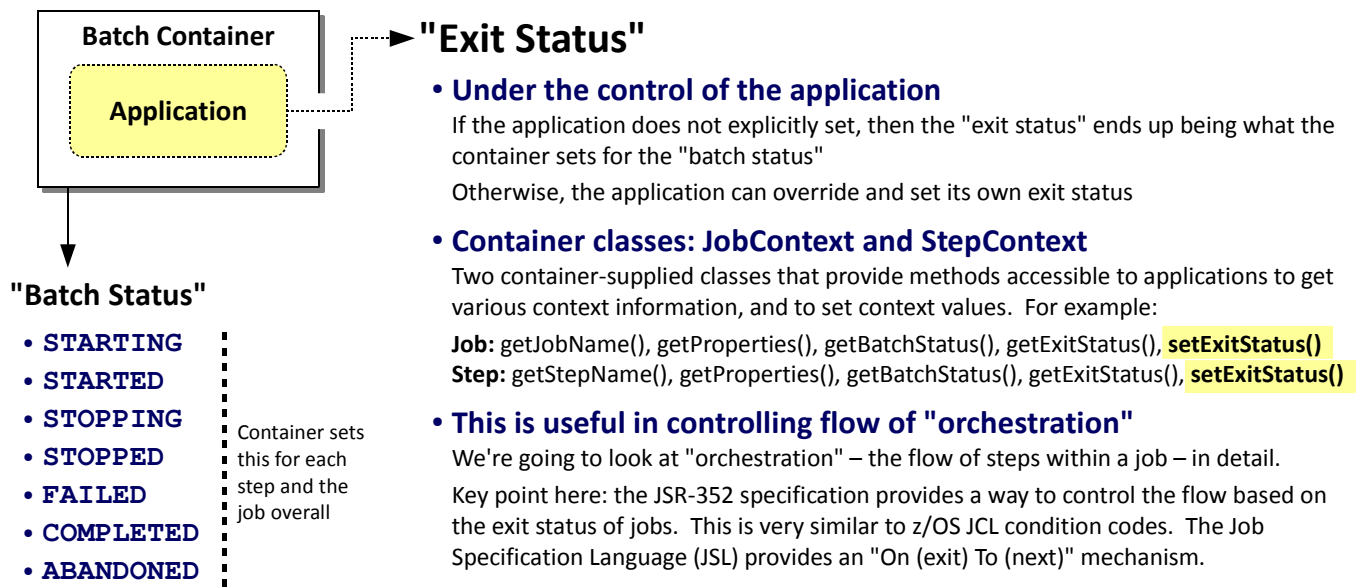
1. The "batch container" is the runtime code that implements the JSR-352 standard specification within the Java EE 7 standard implementation. For IBM, that's any runtime that supports Java EE 7. That includes Liberty, which is the focus of this workshop.
2. In the Job Specification Language (JSL) file for the job, the chunk step has properties that define the checkpoint processing. There are two approaches: "item," which means the container will do the checkpoint processing based on item-count and/or time-limit; or "custom," which means your own custom code will handle the checkpoint processing based on the item-count interval.

Note: if you specify "custom," you also specify a "checkpoint-algorithm ref=" that points to the class that implements your checkpoint processing. The class has standard-defined methods that must be implemented. The code behind those defined methods is your code doing your custom checkpoint processing.

3. The container is keeping track of *when* to call for a checkpoint, and this can be done by with both item-count and time-limit. The item-count option is just that: the number of times ItemReader/ItemProcessor is invoked, for example: "every 1000 records." The time-limit option specifies the amount of time that has passed since the last checkpoint. When the checkpoint policy is "item," you can specify both options. If, for example, you specify item-count="1000" and time-limit="300" (that's expressed in seconds), then a checkpoint will be taken when either 1000 records is reached or when 300 seconds has elapsed, whichever comes first.
4. The skip-limit and retry-limit values control how the container will behave when exceptions are seen in ItemReader processing. The objective of this function is to avoid stopping the whole job just because of some occasional read or write problem that you know comes up every once in a while. You can define the exception and tell the container it may skip X number of times before failing the step. The retry-limit indicates how many times it should attempt a retry before failing the step. We're not going to dig into this level of detail in the workshop; the important thing is to know the specification provides for a way to define values to help "survive" little glitches that may come up.

To summarize: the checkpoint processing control values are externalized from the batch job. They are contained in the JSL file, and the container is what handles this.

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



Job, and the steps within a job, end ... either with success, failure, or something in-between. To indicate the status of steps and jobs, the JSR-352 specification defines two categories of status: "Batch Status" and "Exit Status":

Batch Status

This is controlled by the batch container. The values set are defined and deterministic: the status values are the seven shown on the chart. These batch status values are set for each step and the job overall. If everything ran well, the value COMPLETED will be seen. If something went wrong, we'd see FAILED. If the job was stopped with a command, then STOPPED would be seen.

Exit Status

This is controlled by the application, and if set by the application this can be any value you want. The container provides two classes to set job and step exit status values. Those classes are JobContext and StepContext respectively. The methods to set the exit status are:

```
JobContext.setExitStatus() -- set the job exit status
StepContext.setExitStatus() -- set the step exit status
```

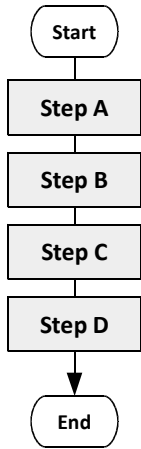
These are string values, though you can set a number value in the string so it appears to be a number for the purposes of flow control.

As the chart indicates, if your application doesn't do anything then the batch status values become the exit status values. That can be used for flow control, but the granularity is much less than is possible when you set your own.

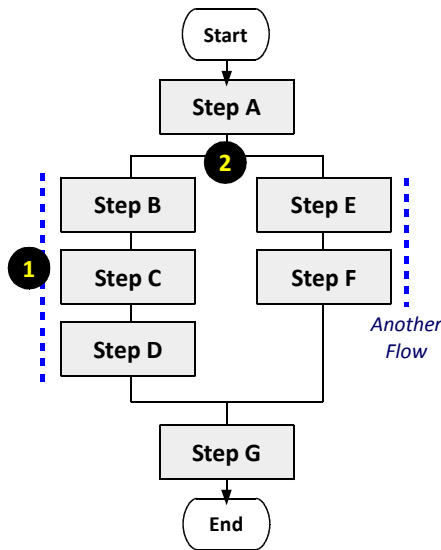
We're going to look at the ability to control the flow of processing within a job in more detail. For now, the key point is that the exit status can be used to determine where to go next in the step processing. The JSL provides a mechanism to conditionally "go to" based on defined exit values.

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 Specification 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.

When we think of "batch jobs," we often think of steps executed in linear sequence ... such as what's illustrated on the left side of this chart. The JSR-352 specification allows this, and you may in fact do this. But you should also understand the JSR-352 specification defines a way to compose more sophisticated step flows within a job. This is where the discussion of "splits" and "flows" comes into the picture.

The picture on the right side of the chart illustrates a job with seven steps arranged into a split and two flows. Before we get into any details of this, let's explore *why* you may do this. The value of this is it allows the container to process steps concurrently – on separate JVM threads – if you have defined such concurrent processing. You may have a batch job that has steps that can process concurrently because they're not directly related to one another. However, some later step requires the results of the earlier concurrent processing, so you have to bring things back together ... which is what the picture illustrates with the lines coming back before proceeding to Step G.

A "flow" is a series of steps that execute in sequence to form a logical unit of processing. A flow can have one to many steps included in it. The concept of a flow is important because the JSR-352 specification says that a "split" (which we define next) can process only flows. A "flow" is really just a logical structure consisting of a step or steps. It is defined in the JSL file. XML there says: "Here's a flow, and it contains the following steps."

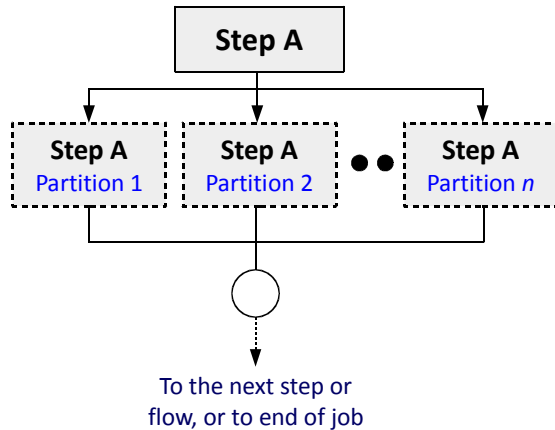
A "split" is how concurrent processing occurs. A split is defined in the JSL, and under the split definition there is a number of flows specified. If you have two flows defined under the split, both flows will execute concurrently. If you have three flows under the split, all three will execute concurrently.

The picture above is a relatively simple split-and-flow illustration. You can make this as sophisticated as you need to make it, including more than two flows in a split, and splits within splits.

Key Point: the splits execute *within the same JVM* as the job itself. The concurrent flows execute on separate JVM threads, but do *not* run in other JVMs. The ability to "parallelize" batch processing is called "step partitioning," and is something different from splits and flows. We'll cover step partitioning on the next chart.

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.

Suppose you have a job step that you feel lends itself to splitting into "sub-steps" and running in parallel. The reason you'd want to do this is to save time – if the job step by itself takes 10 hours to run, and you can split it into 10 sub-steps and run at the same time, then theoretically it would take 1 hour to complete the step when run in parallel.

Doing that within the JSR-352 programming model is called "job step partitioning."

Note: doing this is predicated on a data layout that lends itself to parallel processing. If there's data contention and locking between the parallel partitions, this isn't going to work very well. But if the data is laid out such that one partition can process on a set of data separately from another partition, then this becomes a viable solution.

As the chart says, this does not happen by magic; you have to structure your code in such a way that it can be passed parameters to indicate which range of data to work on. This is accounted for in the JSR-352 specification, and the IBM implementation of JSR-352 in the WebSphere Liberty Java Batch product is capable of doing this.

This, however, is beyond the scope of this workshop. Interesting as it may be, to go down the path of getting into the details of this would take more time than we have. So we'll draw a line and fence this topic away from this workshop. Understand that this capability is present. And understand this is different job splits and flows, which your step code does not have to be aware of.

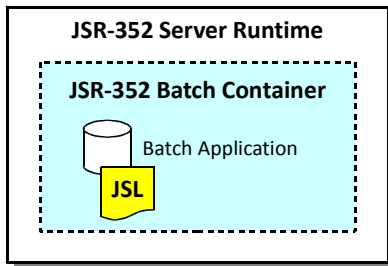
Next we're going to talk about "orchestration," which is a fancy word for "controlling how a job processes." This is really an exploration of the Job Specification Language (JSL) file, and as part of that discussion we'll show how the WDT tool helps you compose the JSL.



Job Specification Language (JSL)

This defines and controls the execution of the job

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")

Batch applications are packaged and deployed into the Liberty server just like any other application. So the Liberty runtime is aware there's an application present, but it's not a mind-reader and it can't know the specifics of the batch processing flow unless you tell it. That mechanism is the Job Specification Language (JSL) file, which is an XML file you specify at time of job submission. The batch container reads the JSL file, and from that it understands how you want the batch job processed.

Note: in concept this is just like mainframe JCL. The syntax is different, but the concepts are almost identical.

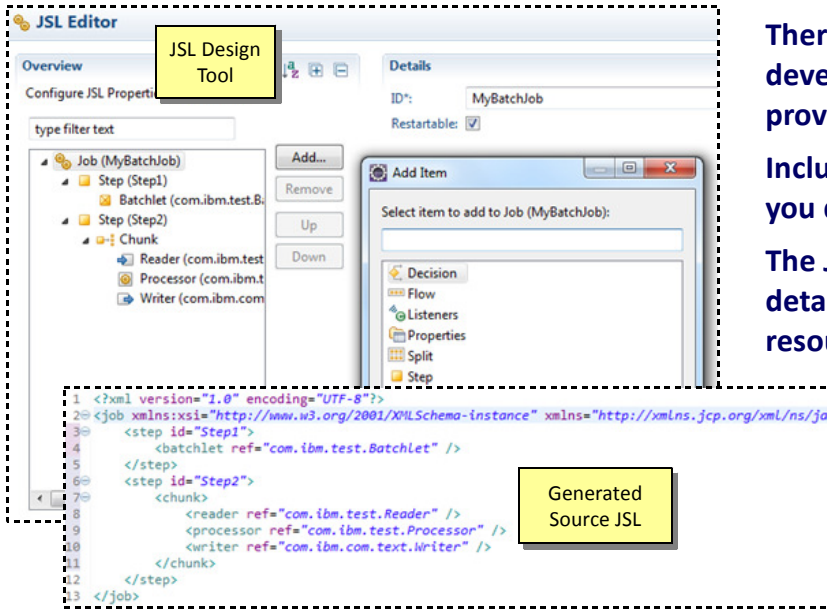
For example, the JSL will define how many steps are in the job, and what kind of steps they are (batchlet, or chunk). For a given step, it will specify what Java classes implement the step. It will tell the container about checkpoint intervals. It will tell the container how to process from step to step, and whether there are any splits and flows in the batch processing.

In short, the JSL file "tells the story" of the batch job to be executed.

Typically this XML is packaged with the batch application, and at time of job submission you simply name the file. That's how the SleepyBatchlet job was run in the earlier lab. But with the IBM Liberty Java Batch support, you may also store the XML file at a separate file system location, and at time of job submission you can point to the path and file of the JSL file you wish to use.

The XML elements for the JSL file is a documented standard, and is spelled out in the JSR-352 specification document. In addition, there is an Eclipse plugin that can be used to develop the batch code, and with it comes a JSL editor to compose the XML file. We'll touch on that next.

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



The screenshot displays the JSL Editor interface. On the left, a tree view shows a job named 'MyBatchJob' containing two steps: 'Step1' and 'Step2'. 'Step1' contains a 'Batchlet' and 'Step2' contains a 'Chunk' with 'Reader', 'Processor', and 'Writer' components. A 'JSL Design Tool' label is placed over the tree view. In the center, an 'Add Item' dialog is open, showing a list of items to add to the job, including 'Decision', 'Flow', 'Listeners', 'Properties', 'Split', and 'Step'. On the right, a 'Details' pane shows the ID 'MyBatchJob' and the 'Restartable' checkbox checked. At the bottom, a code editor displays the 'Generated Source JSL' XML code:

```

1 <?xml version="1.0" encoding="UTF-8"?>
2 <job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/xml/ns/job">
3   <step id="Step1">
4     <batchlet ref="com.ibm.test.Batchlet" />
5   </step>
6   <step id="Step2">
7     <chunk>
8       <reader ref="com.ibm.test.Reader" />
9       <processor ref="com.ibm.test.Processor" />
10      <writer ref="com.ibm.com.text.Writer" />
11    </chunk>
12  </step>
13 </job>

```

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>

© 2017 IBM Corporation

17

SleepyBatchlet JSL ...

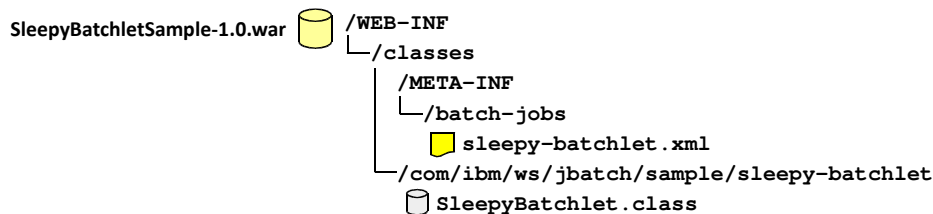
If we told you coding a JSR-352 application and coding the associated JSL was done by hand in a simple text editor, you'd likely not be very happy. You *can* do it that way, but thankfully you do not *have* to do it that way.

IBM has created a plugin to Eclipse to provide assistance in developing JSR-352 batch code and the JSL that defines the job execution behavior. At the bottom of the chart you can see a URL to a Techdoc where the details of getting Eclipse and downloading the plugin tool is spelled out in detail.

The tool is like any other Eclipse-based tool, with menus and wizards that generate code and XML files. The JSL editor, in particular, is useful in composing the overall orchestration of the job flow, and it produces the JSL file according to your composition. That's what we're going to illustrate over the next several charts.

Note: the official JSR-352 specification document is very handy for understanding the specifics of the XML elements of the JSL. The Eclipse tool is handy for creating the file and getting the structure right; the JSR-352 document is handy as a reference for the specifics of a given XML element within the JSL file.

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">
    2 <batchlet ref="com.ibm.ws.jbatch.sample.sleepybatchlet.SleepyBatchlet" 3 >
      <properties>
        <property name="sleep.time.seconds" value="#{jobParameters['sleep.time.seconds']}" /> 4
      </properties>
    </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"

Here's a very simple example of a Job Specification Language (JSL) file. This comes from the SleepyBatchlet sample you ran in the previous lab. That sample was a single-step job where the step was a batchlet. The batchlet simply looped for a number of seconds and then ended. It's about as simple a batch job as is possible.

The JSL file for the SleepyBatchlet sample was located in the WAR file, at the location indicated on the chart.

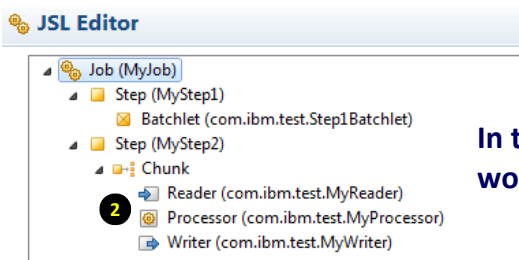
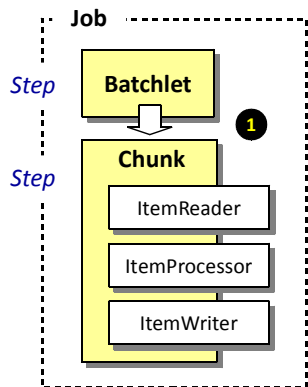
Note: with IBM WebSphere Liberty Java Batch you may also have the JSL for a job *outside* the application package file, and refer to it on the job submission command using either `batchManager` or `batchManagerZos`. We'll see that in the next unit.

Let's take a look at what the JSL for SleepyBatchlet is telling us by following the numbered circles:

1. The `<step>` element defines a job step. There's only one `<step>` element in this JSL because SleepyBatchlet has only one step.
2. The `<batchlet>` element defines the job as a batchlet type step.
3. The `ref=` tag points to the package and class file that implements the batchlet. In the picture at the top of the page you can see where that class file is located within the WAR file.
4. The SleepyBatchlet batchlet step has one job property defined. It determines the number of seconds the job will loop-and-sleep. The default is 15 seconds, but you can override this default. One way you could override it is to code the `value=""` in the JSL to specify a different value. Another way you can override it with IBM WebSphere Liberty Java Batch is to pass the value into the job with a jobParameter name/value pair on the job submission command. We'll see the details of this a bit later, but for now understand that it is possible to pass in parameters at the time of job submission. In fact, you can do this by pointing to a file that includes the job parameters. That's handy if you have a lot of job parameters for a job.

That's the basics of a JSL file. Let's now look at the JSL editor that's part of the WebSphere Development Tool (WDT) plugin to Eclipse.

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:

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

```

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" />
        <processor ref="com.ibm.test.MyProcessor" /> 6
        <writer ref="com.ibm.test.MyWriter" />
      </chunk>
    </step>
  </job>
    
```

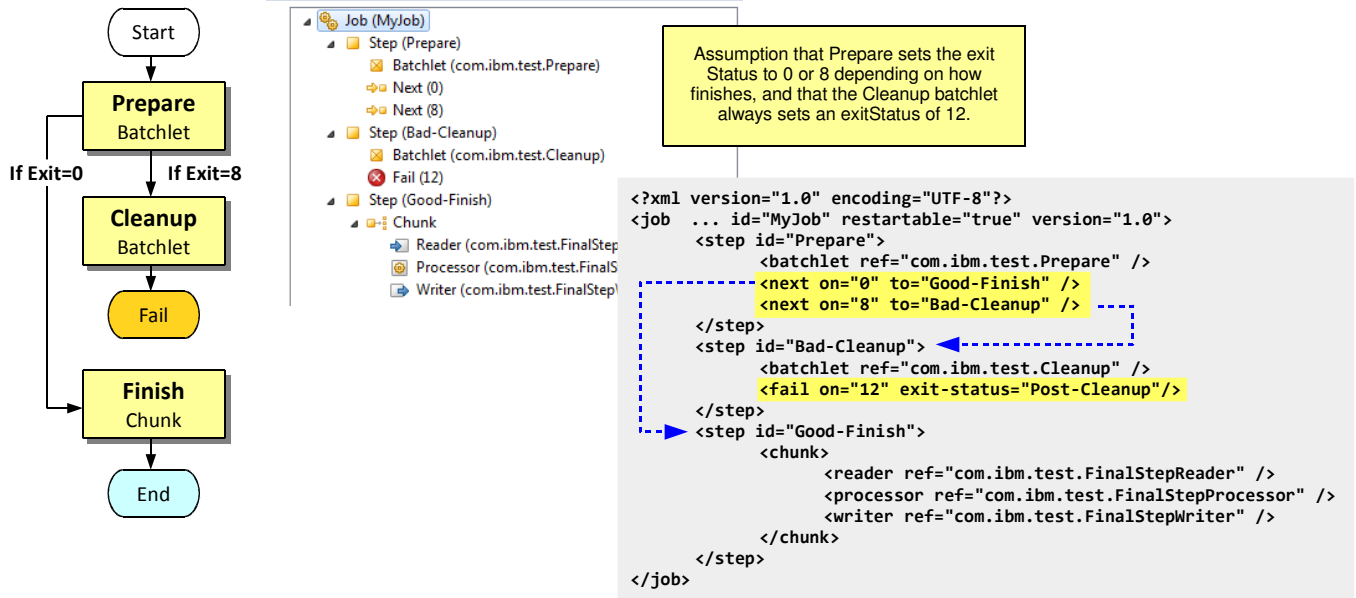
Hypothetical with conditional next ...

Here we're showing a hypothetical two-step job, with the first step a batchlet, and the second step a chunk step. The graphic on the chart shows the high-level of the job, along with what the WDT JSL editor would look like for that. In the lower right we have the JSL that's created for this. The circled numbered on the chart correspond to the notes below.

1. This picture illustrates the hypothetical job. It has two steps -- a batchlet and a chunk -- and the chunk step has an ItemReader, and ItemProcessor, and an ItemWriter.
2. This is a bitmap clip of what this hypothetical job would look like in the JSL editor that comes with the WDT plugin to eclipse. Here we're highlighting the Reader, Processor, and Writer. Notice how package that implements your code is specified for each -- com.ibm.test.MyReader, etc. The WDT plugin will generate code stubs that have all the JSR-352 methods included. You may then write your implementation code to each method as needed by your business logic.
3. The tool will generate the JSL XML file. We're showing you the XML that was associated with the JSL Editor composition in the graphic above this XML.
4. One of the things we want to point out here is that you must tell the order of execution for steps in a mult-step job. It is not implied by the order seen in the XML file. So for the first step we have a next= attribute, and this point to the ID of the second step in the job. (The last step in the job does not need a next= attribute.)
5. The second step in this hypothetical job is a chunk step, so here we show the checkpoint policy definition ("item") then item-count="1000". The "item-count" attribute tells the container the checkpoint policy is based on the number of records read and processed. In this case it's defined as 1000 records. If "time-limit" was specified, then it would be 1000 seconds.
6. Finally, the references to the packages that implement the reader, processor, and writer are specified.

This shows how the JSL Editor can make composition of the JSL much easier.

Another Hypothetical ... Showing Conditional "Next" Processing



© 2017 IBM Corporation

20

Split and flows ...

The next topic we'll discuss is "next" processing, which is a JSL value that tells the container where to go upon certain conditions. We're going to simplify the job design a bit to more clearly show this next processing. The more simplified hypothetical has three steps with no splits or flows (although "next" processing works with those as well).

The first step is a batchlet. If that ends with a return code of 8 we want to go to another batchlet that does some cleanup work, then fail the job. But if the first batchlet ends with RC=0, we want to go to a chunk step to finish up the job.

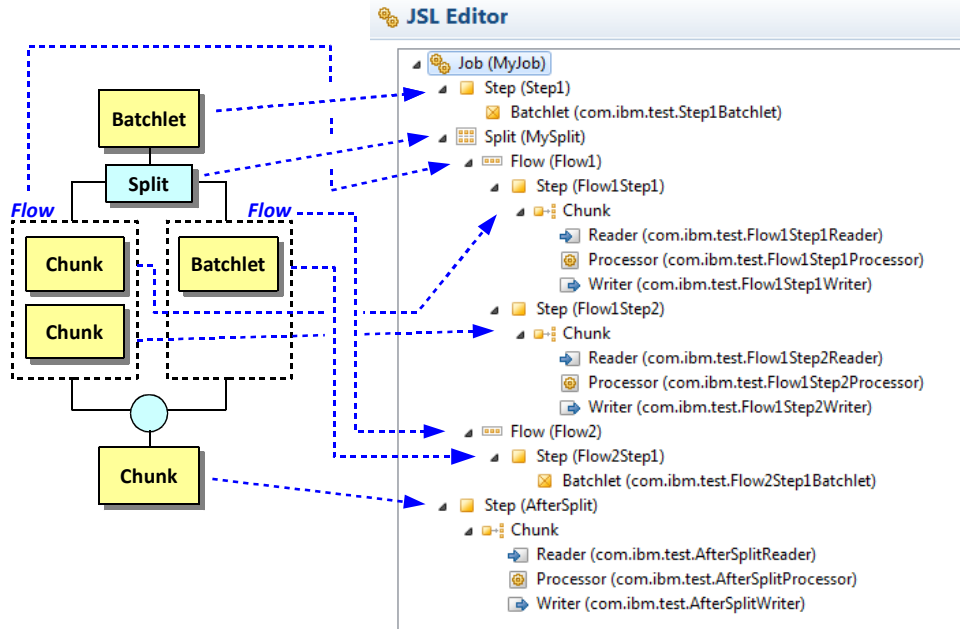
In the editor we add a step and a batchlet just like we saw earlier. Then we add two "Next" objects after that. We define the first next with "on RC=0 go to step Good-Finish". The other next has "on RC=8 go to step Bad-Cleanup."

For the "cleanup" batchlet we want to fail the job after that's done because if we're in the "cleanup" step it's because something went wrong in the "prepare" step. We don't want the overall job to end as if things went well because they didn't.

Note: the assumption here is that the "Prepare" step sets the exit status to 0 if things go well, and 8 if things do not go well. Also that the "cleanup" step always ends with a RC=12. You may have different exist status codes; there's nothing that says values of 0, 8, and 12 are what you must use.

The generated JSL is shown on the chart. Notice how under the "Prepare" step there are two <next> elements – one defines where to go when "0" is seen; the other where to go when "8" is seen. Under the "Cleanup" step there's a <fail> that says when the batchlet finishes with an exit code of "12," the job is to fail with an exit status string of "Post-Cleanup." Again, these are just examples; you can have your own exit codes as meet your local standards.

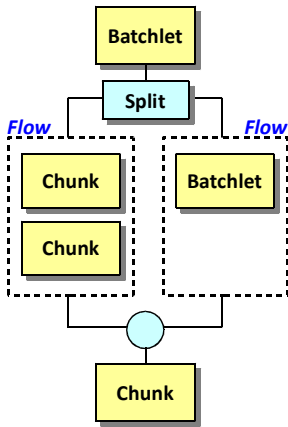
Yet Another Hypothetical ... Split and Two Flows



Here's another hypothetical job. This starts with a batchlet step, then flows into a split which has two flows. Flow 1 has two chunk steps, and flow 2 has a batchlet step. When the two flows complete, the split comes back together and it processes a final chunk step.

The JSL Editor would look like what's show on the chart, with the blue dotted lines showing the association of the elements for the picture on the left to the editor elements on the right.

Same Hypothetical with the Generated JSL



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

```

<?xml version="1.0" encoding="UTF-8"?>
<job ... id="MyJob" restartable="true" version="1.0">
  <step id="Step1" next="MySplit">
    <batchlet ref="com.ibm.test.Step1Batchlet" />
  </step>
  <split id="MySplit" next="AfterSplit">
    <flow id="Flow1">
      <step id="Flow1Step1" next="Flow1Step2">
        <chunk>
          <reader ref="com.ibm.test.Flow1Step1Reader" />
          <processor ref="com.ibm.test.Flow1Step1Processor" />
          <writer ref="com.ibm.test.Flow1Step1Writer" />
        </chunk>
      </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">
        <batchlet ref="com.ibm.test.Flow2Step1Batchlet" />
      </step>
    </flow>
  </split>
  <step id="AfterSplit">
    <chunk>
      <reader ref="com.ibm.test.AfterSplitReader" />
      <processor ref="com.ibm.test.AfterSplitProcessor" />
      <writer ref="com.ibm.test.AfterSplitWriter" />
    </chunk>
  </step>
</job>
    
```

Here's the same hypothetical job, but this time with the generated JSL shown.

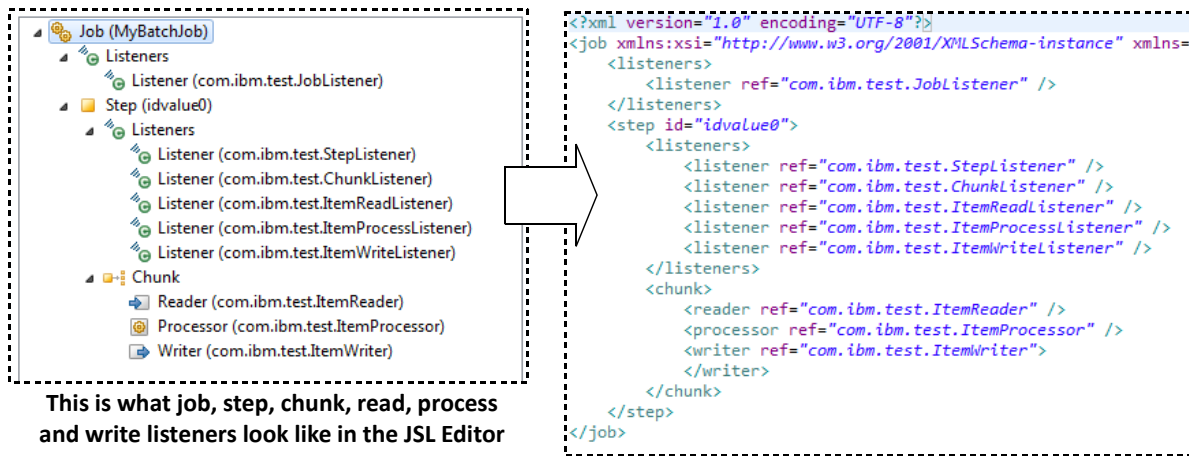
1. The first step is the batchlet step. The package that implements the batchlet is specified, and the "next" tag tells the container where to go after this batchlet completes. The "next" tag names the ID of the split. So the container will look for the XML identified by this "next" and the process what it sees there.
2. This is the split identified by "next" on the first step. This split has two flows inside of it. The container will process those two flows "in parallel" on two separate threads in the JVM.
3. Inside of the first flow of the split we see two steps, both of which are chunk steps. Within a flow it's necessary to point to the next processing element after a step is complete, and we show that here. After the first chunk step completes, the "next" tag says to go to the next step in the flow.

Why? Because you don't necessarily have to process through a flow in sequential order. You could have conditional flow processing based on the exit status of the step. Rather than try to resolve potentially ambiguous processing caused by unspecified "next" processing, the specification calls for "next" to be named for steps in a flow.

4. This shows the second flow with its single batchlet.
5. The <split> tag has a "next" attribute, which tells the container where to go after the split "comes back together." The container keeps track of processing which the flows defined to a split, and when each flow's final step completes, the split itself is marked complete. At that point the container goes to the defined "next" for the split. In this case it's the final chunk step.

"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.



"Listeners" are points in job processing where your code can get control to do whatever you might wish to do, such as log some information. There are quite a few defined listeners in the JSR_352 specification:

- JobListener Interface
- StepListener Interface
- ChunkListener Interface
- ItemReadListener Interface
- ItemProcessListener Interface
- ItemWriteListener Interface
- Skip Listener Interfaces
- RetryListener Interface

Take for example the "JobListener Interface" – that gets called at the start of a job, and again at the end of a job. The same thing happens for a "StepListener" – at the start of the step, and again at the end of the step.

In the JSL editor we showed you earlier the listeners become objects you add to the job or the step, as we're showing in the left side of the chart illustration. In turn, it generates the JSL with the <listener> elements, which have a ref= that points to your Java class that implements the listener. The tool will also generate a Java source stub with the methods for the listener as defined by the JSR-352 specification.

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

Here is an example of what the tool generates as the code stub for a Job Listener. It populates it with a `beforeJob()` method and an `afterJob()` method. You would add your code to those methods to do whatever you would want to do at the start of a job and at the end of the job.

That's all we're going to say about "listeners" ... we wanted to introduce you to them so you would be aware of them. But with that we'll set the topic aside. If you wish to learn more about listeners, see the JSR-352 specification.

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.

Earlier we showed you the JSL for the SleepyBatchlet sample and mentioned that it was possible to pass in parameters at time of job submission and have those values flow into the batch program. Here's how that works with SleepyBatchlet:

- At the top of the chart we show you a batchManager command to submit the SleepyBatchlet job. Notice how we have a jobParameter= option with a name/value pair of "sleep.time.seconds=99". That means we are going to pass into the job a value of 99 which will get substituted in for the job property sleep.time.seconds.
- In the JSL file we saw a <property> element which defined the sleep.time.seconds property. Notice how the "value" defined in the JSL specifies "#{jobParameters['sleep.time.seconds']}" – this means it's exposed as a job parameter that can be passed in at time of job submission. So the value of "99" ends up being assigned to the job property of sleep-time.seconds.
- Down in the Java code for SleepyBatchlet we see where the property gets injected, then assigned to the string "sleepTimeSecondsProperty." So in this example at this point the value of sleepTimeSecondsProperty is 99. Notice the integer variable sleepTime_s set to "15" ... that's the default value if nothing is passed in.
- We move down to the "if" statement ... it's saying, "If the sleepTimeSecondsProperty is *not* null, then set the integer sleepTime_s to the value of the sleepTimeSeconds string." In other words, override the default 15 seconds when a sleep time value has been passed in as a job parameter. If nothing is passed in, then the default value for sleepTime_s remains 15 seconds.
- Finally we get into the sleep loop where it iterates from 0 to the value of sleepTime_s. On each iteration it sleeps for one second, then loops again. When the "for" loop condition is met it ends.

The ability to pass parameters in at time of job submission provides you the ability to tailor job submission at time of execution. And though we've not yet come to the unit where we spell out the details of batchManager or batchManagerZos, understand there is an ability to specify a job parameters *file* for cases where the number of job parameter name/value pairs would make encapsulating it all on a command line somewhat cumbersome.

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>

We're going to close this unit and put you into another hands-on lab.

The coding of Java batch jobs can turn into a lengthy discussion about many things, but our objective here was to provide the essential framework of the JSR-352 specification and show you how the tooling works to aid in the construction of a batch job. We intentionally avoided getting into too many specific coding discussions, though we did illustrate a few things.

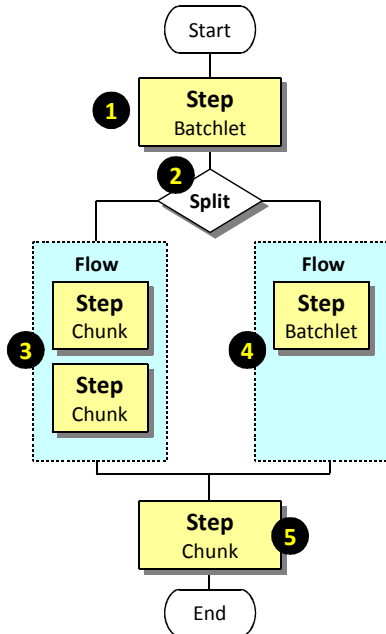
The key things to take away from this unit are what's expressed in the bullet on the chart. With those key things in mind, and knowledge of from where the JSR-352 specification document can be downloaded, you should have a good start on coding up your first JSR-352 batch program.



Screen Shots of JSL Editor

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

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.

Before we launch into bitmap captures of the Eclipse panels to compose a job flow, let's draw a picture of what we're going to create.

Note: this is a *hypothetical* illustration. We're going to define the job in the tool and see how it generates the JSL file, but we're *not* going to actually code the Java for the steps. So use your imagination as to what these steps actually do. With respect to the key points we're making in this section it doesn't really matter what those steps do.

We're going to illustrate a job with five steps – two batchlet and three chunk steps – with a split and two flows:

1. The first step is a batchlet step. Let's assume that does some setup work prior to running the chunk steps.
2. Then we move into a split. The split is going to go down two paths, each with a flow.
3. The flow on the left is going to have two chunk steps that process in sequence.
4. The flow on the right is going to have a single batchlet steps. The JSR-352 specification says that a split can only branch to flows, so this helps illustrate that even if you have a single step in the branch of a split, you still need a flow.
5. The flows of the split come back together and we run a final chunk step. After that, the job ends.

Let's go look at how this would be created in the tool ...

Starting Out ... Creating the Initial Step

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

Then right-mouse click, Add → Step

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

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

© 2017 IBM Corporation

29

Create batchlet ...

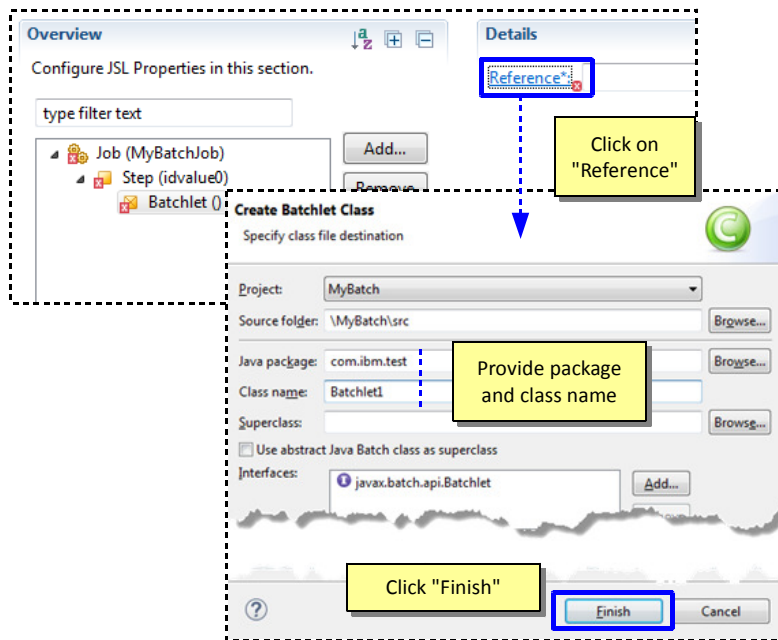
We start out in the tool and we create a batch job using *File* → *New* → *Other* → *Batch Job* (we're not showing you that; it's just a following of the menu). We name the batch job "MyBatchJob."

Recall our picture ... the first step in the job is a batchlet step. So we create that with a right-click on the new batch job and click on *Add* → *Step*. When the step is created we right click on the newly created step and click on *Add* → *Batchlet*. What results is what you see in the upper-right of the chart: a hierarchy showing a Job/Step/Batchlet.

That's just the start; we're going to show adding all the other things as well – splits, flows, etc.

But this is the general process – the tool provides you with menus you can use to define things.

Creating the Batchlet



```

1 package com.ibm.test;
2
3 import javax.batch.api.Batchlet;
4
5 public class Batchlet1 implements Batchlet {
6
7     /**
8      * Default constructor.
9      */
10    public Batchlet1() {
11        // TODO Auto-generated constructor stub
12    }
13
14    /**
15     * @see Batchlet#stop()
16     */
17    public void stop() {
18        // TODO Auto-generated method stub
19    }
20
21    /**
22     * @see Batchlet#process()
23     */
24    public String process() {
25        // TODO Auto-generated method stub
26        return null;
27    }
28
29 }
30

```

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

The new batchlet has "red X" symbols next to it, which means something is not yet complete about the definition. The problem is that the batchlet is created, but we've not yet specified the Java package name or the Java class name for the batchlet. So we highlight the new batchlet and then click on the "Reference" link (which also has a red-X), which opens up a panel for us to resolve the package name and class. We specify `com.ibm.test` for the package (your actual package names will be different) and the class name as `Batchlet1`. When we click finish the red-X's go away and a Java editor pops open with the framework for a batchlet. The methods are there, but they're just stubs without any actual code. This is where you would code whatever Java you need to perform the tasks for that opening batchlet. We're not going to illustrate that here.

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

To create the split we highlight the job, right-click and click on *Add* → *Split*. This adds the job.

Note: when the tool creates steps and splits and other artifacts, it creates a default name. You can rename by highlighting the artifact and then changing the "ID" value over to the right. Notice the Step name in the upper left of the chart ... initially that was created with a value of "idvalue0". We renamed that to "Step1" to reflect the picture we drew at the beginning.

Note also the "Up" and "Down" buttons in the tool. If you want to reposition an artifact within the flow, you can use those buttons to do it. The tool by default positioned the split after the first step because by default the tool adds new artifacts to the bottom, and that was where we wanted it. But be aware you do have the ability to move things around.

Add Flows Under the Defined Split

type filter text

Job (MyBatchJob)

- Step (Step1)
 - Batchlet (com.ibm.test.Batchlet1)
 - Split (Split1)
 - Flow

Highlight the Split, then click on Add → Flow

Overview

Configure JSL Properties in this section.

type filter text

Job (MyBatchJob)

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

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

*MyBatchJob.xml

```

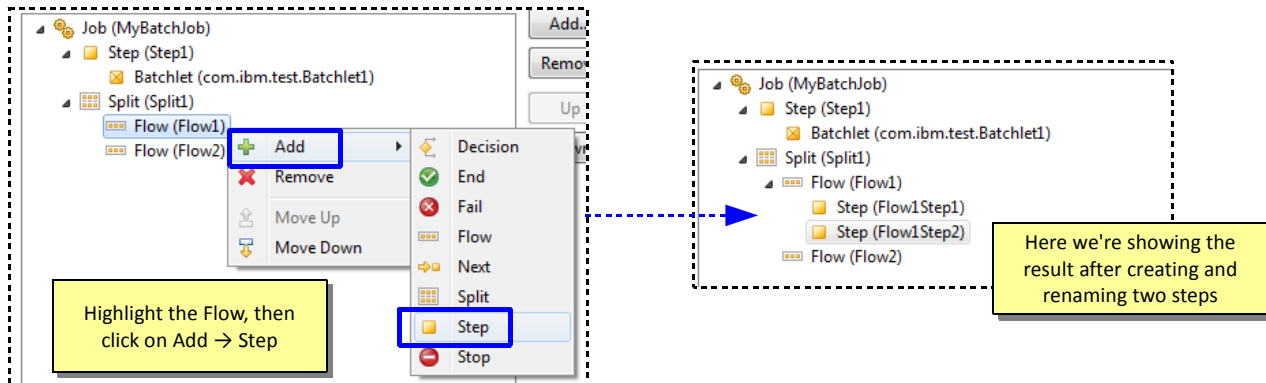
1 <?xml version="1.0" encoding="UTF-8"?>
2 <job xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://xmlns.jcp.org/xml/ns/javaee" xsi:sch
3 <step id="Step1">
4   <batchlet ref="com.ibm.test.Batchlet1" />
5 </step>
6 <split id="Split1">
7   <flow id="Flow1" />
8   <flow id="Flow2" />
9 </split>
10 </job>
  
```

What the source JSL looks like at this point.

We have the Job, the first step, and the split defined. Now we add the flows under the split. Recall from our picture our split will include two flows, so we create both here by highlighting the split and clicking on *Add → Flow*. Do that twice and you what's shown in the upper right of the chart. Notice there's no steps yet defined to the flows. That comes next.

In the bottom of the chart we're showing you the Job Specification Language (JSL) file that's been generated by the tool so far. This is simply the XML representation of what the JSL Editor is showing you in graphics. Notice how there's a `<job>` element, then a `<step>` element, and then `<split>` that includes two `<flow>` elements.

Add Steps to the Flows

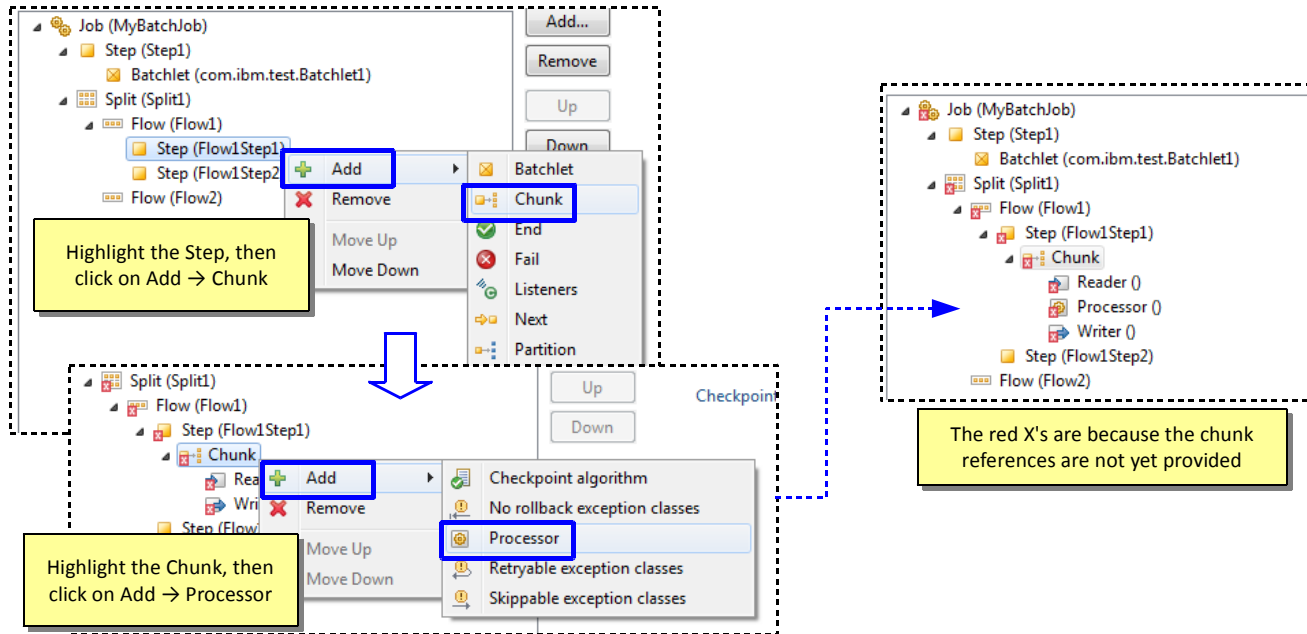


Next ... add a "chunk" under each of those steps

Next we add the two steps that we are defining under the first flow. This is done with *Add* → *Step*. Do that twice, then rename each step to something that's meaningful to you.

Note that at this point the steps aren't defined as chunk or batchlet. That comes next.

Add Chunk to the First Step in the Flow



© 2017 IBM Corporation

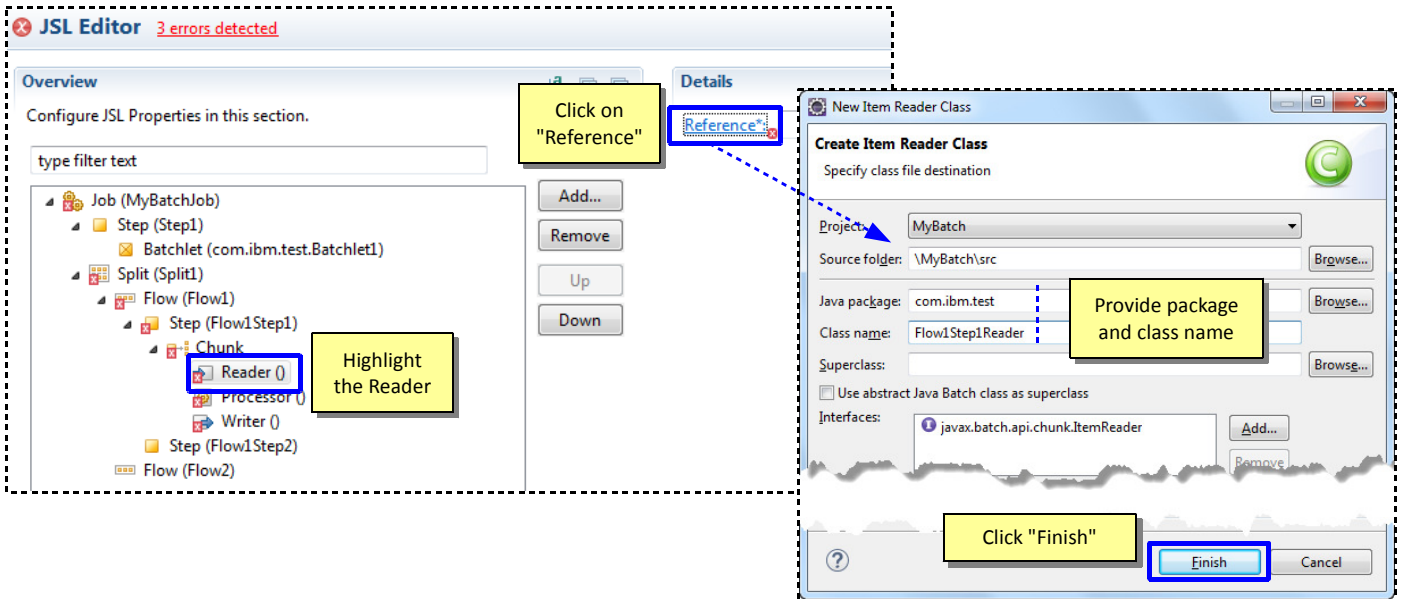
34

Resolve ItemReader reference ...

Add a chunk to the first step in the flow by highlighting the step and clicking *Add* → *Chunk*. This will add the chunk and add a *Reader()* and a *Writer()*, but not a *Processor()*. So we simply click on the newly-created chunk, click *Add* → *Processor* and we have what's shown on the right side of the chart, which is a chunk step with an *ItemReader*, *ItemProcessor* and an *ItemWriter*.

Remember, those are just placeholders at this time. And you'll notice there are red-X's all over the place. That's because the package name and class name references for the *ItemReader*, *ItemProcessor*, and *ItemWriter* are not yet created. We do that next.

Resolve the References for the ItemReader of the Chunk Step



The screenshot illustrates the process of resolving references for an ItemReader class in the JSL Editor. The Overview tab shows a tree view of the job structure, with the 'Reader ()' element highlighted. The Details tab shows the 'Reference' link. A dialog box titled 'New Item Reader Class' is open, showing the 'Create Item Reader Class' form. The form includes fields for Project (MyBatch), Source folder (\MyBatch\src), Java package (com.ibm.test), and Class name (Flow1Step1Reader). The 'Finish' button is highlighted.

© 2017 IBM Corporation

35

Generated ItemReader code ...

To resolve the references we click on the Reader() and then on the "Reference" link for the reader. We provide a package name and a class name and click "Finish." We do that for the Reader, the Processor, and the Writer. When all three are done, then the red X's go away.

Again ... all that does is create code stubs. You have to go into the created stubs and build out the reader, processor and writer according to your specific needs.

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

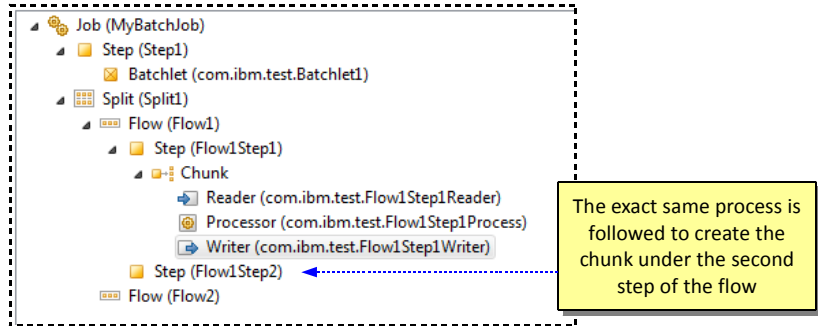
```

1 package com.ibm.test;
2
3 import java.io.Serializable;
4
5 public class Flow1Step1 implements ItemReader {
6
7     /**
8      * Default constructor.
9      */
10    public Flow1Step1() {
11        // T000 Auto-generated constructor stub
12    }
13
14    /**
15     * @see ItemReader#readItem()
16     */
17    public Object readItem() {
18        // T000 Auto-generated method stub
19        return null;
20    }
21
22    /**
23     * @see ItemReader#open(Serializable)
24     */
25    public void open(Serializable arg0) {
26        // T000 Auto-generated method stub
27    }
28
29    /**
30     * @see ItemReader#close()
31     */
32    public void close() {
33        // T000 Auto-generated method stub
34    }
35
36    /**
37     * @see ItemReader#checkpointInfo()
38     */
39    public Serializable checkpointInfo() {
40        // T000 Auto-generated method stub
41        return null;
42    }
43
44    ..

```

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:



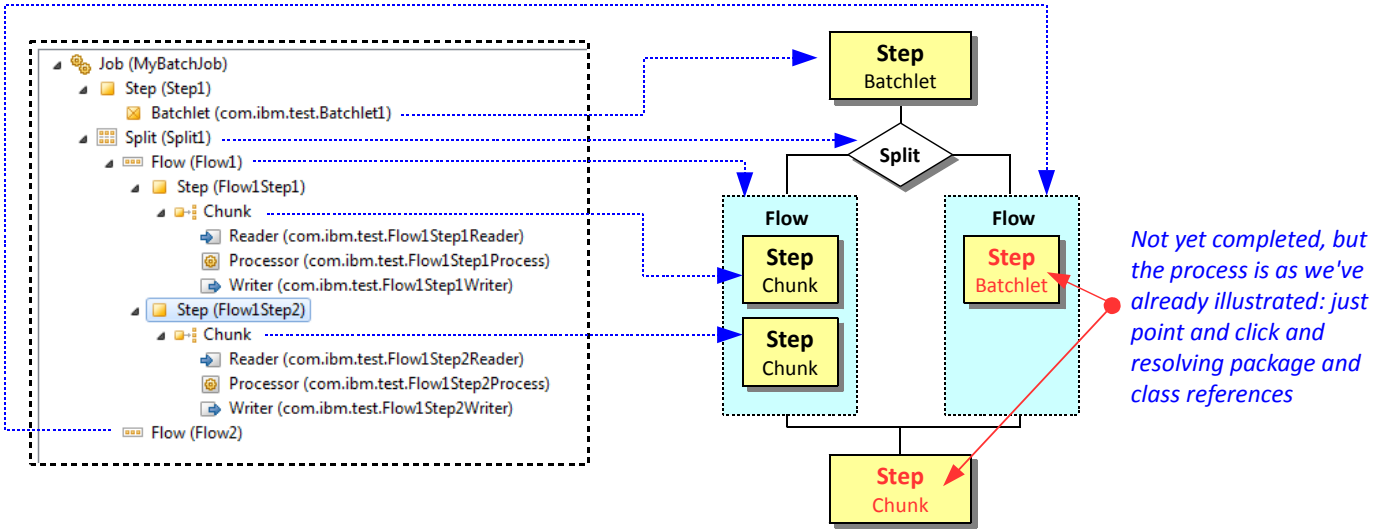
Here's an example of what the tool created for the ItemReader. All the JSR-352 specification requirements for methods are included. But there's no code inside those methods; they're just stubs waiting for you to supply the Java code you require for your business.

What you end up with is what we are showing on the chart – a chunk step inside the first flow, with a Reader, Processor, and Writer.

The exact same process is done for the second step in the flow, which we planned to be another chunk step. We're not going to show you the flows for that because, as we mentioned, it's the exact same process.

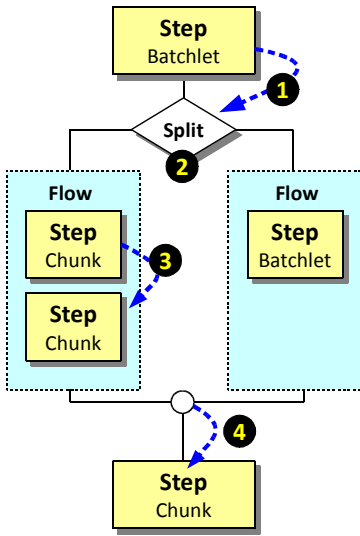
Add Chunk to Second Step in the Flow and Resolve the References

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



Here's where you would be after completing the two chunk steps in the first flow. The design tree would be as we're showing here. What's not yet done is the definition of the step under the second flow, and the final step to complete the job. We're not going to show you the step-by-step for that because it's just the same things over and over again ... highlight, add, resolve, etc. And that's the key point here – the tool makes this relatively easy.

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



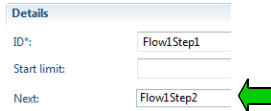
1. Set "Next" on Step1 to Indicate "Split1"



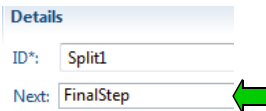
2. Split will go down each defined flow in the split

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"



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



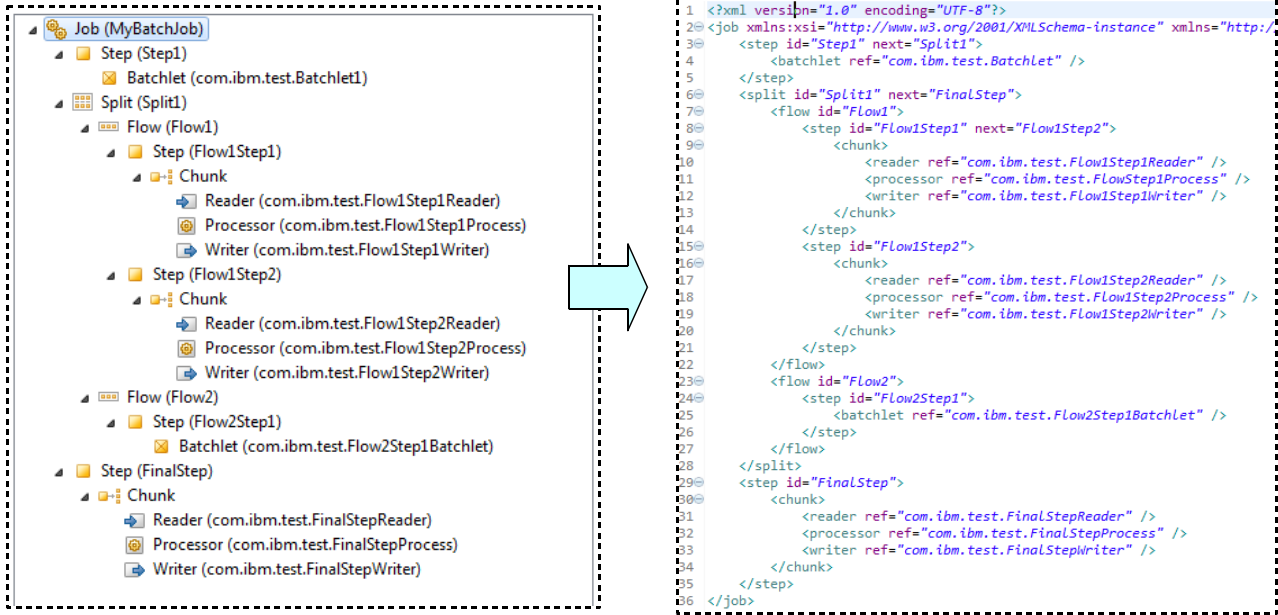
With all the pieces of the job constructed, you then define the "Next" for each of the component that requires a definition of where to go once that component is finished. This "Next" is specified by highlighting the component, then specifying the "next" component in the "Next" field. (You can also use a pulldown to select the "next" component, or you can type it yourself.)

For instance, the first batchlet step goes to the split after it completes. So you highlight the first step, then specify "Split1" in the "Next" field. ("Split1" is the ID we gave the split in this example.)

Splits are interesting. You do specify a next on the split, but you do not specify the flows in the split, you specify where to go once the split flows come back together. So on the "split" object you'd specify "FinalStep" for the next field. The JSL defines two flows under the split, and it will automatically begin processing the defined flows in the split. But when the two flows come back together, then it goes to "FinalStep."

Interestingly, inside a multi-step flow -- which is what the left-hand flow in the picture is -- you have to specify the "Next" on steps in the flow that have steps that follow. So in this example, "Flow1Step1" has a "Next" that indicates "Flow1Step2" as the next step to go to after the first step completes. The last steps in the flows in the split will proceed to the point where the flows come back together, then it proceeds to the "Next" defined on the Split.

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



Here we're showing the final result with all the steps defined. The JSL that's defined is shown as well, though it's probably too small to read, but the point of this chart is not for you to look at the details of the generated JSL, but to understand that the tool generates the JSL based on the composition of the job you do in the editor.

End of Unit