Batch Applications for the Java Platform

Table of Contents

License 2
Acknowledgements 3
Forward 4
Table of Contents 5
Introduction to JSR 352 6
Applicability of Specification 7
Domain Language of Batch 8
Job 8
JobInstance 9
JobParameters 9
JobExecution 9
Step 10
StepExecution 11
JobOperator 11
Job Repository 11
ItemReader 11
ItemWriter 11
ItemProcessor 11
Chunk-oriented Processing 12
Batch Checkpoints 13
Job Specification Language 14
Job 14
Job Level Listeners 14
Job Level Exception Handling 15
Job Level Properties 15
Step 15
Chunk 16
Reader 17
Reader Properties 18
Processor 18
Processor Properties 18
Writer 19
Writer Properties 19
Chunk Exception Handling 20
Skipping Exceptions 20
Retrying Exceptions 21

Retry and Skip the Same Exception . . . 23
Default Retry Behavior - Rollback . . . 23
Preventing Rollback During Retry . . . 23
Checkpoint Algorithm . . . 24
Checkpoint Algorithm Properties . . . 24
Batchlet . . . 25
Batchlet Exception Handling . . . 25
Batchlet Properties . . . 25
Step Level Properties . . . 26
Step Level Listeners . . . 26
Step Level Listener Properties . . . 27
Step Sequence . . . 28
Step Partitioning . . . 28
Partition Plan . . . 29
Partition Properties . . . 30
Partition Mapper . . . 31
Mapper Properties . . . 32
Partition Reducer . . . 32
Partition Reducer Properties . . . 33
Partition Collector . . . 33
Partition Collector Properties . . . 34
Partition Analyzer . . . 34
Partition Analyzer Properties . . . 35
Step Exception Handling . . . 35
Flow . . . 36
Split . . . 36
Split Termination Processing . . . 37
Decision . . . 37
Decision Properties . . . 38
Decision Exception Handling . . . 39
Transition Elements . . . 39
Next Element . . . 39
Fail Element . . . 40
End Element . . . 40
Stop Element . . . 41
Batch and Exit Status . . . 42
Batch and Exit Status for Steps . . . 44
Exit Status for Partitioned Steps . . . 45

Job XML Substitution . . . 45
Substitution Processing Rules . . . 46
jobParameters Substitution Operator . . . 46
jobProperties Substitution Operator . . . 47
systemProperties Substitution Operator . . . 48
partitionPlan Substitution Operator . . . 48
Substitution Expression Default . . . 49
Property Resolution Rule . . . 49
Undefined Target Name Rule . . . 50
Job Restart Rule . . . 50
Examples . . . 50
Transitioning Rules . . . 51
Combining Transition Elements . . . 51
Transitioning Precedence Rules . . . 52
Loop definition . . . 53
Transitioning From Within Flows . . . 53
Flow-level Transitions . . . 54
Batch Programming Model . . . 56
Steps . . . 56
Chunk . . . 56
ItemReader Interface . . . 56
ItemProcessor Interface . . . 58
ItemWriter Interface . . . 59
CheckpointAlgorithm Interface . . . 62
Batchlet Interface . . . 64
Listeners . . . 66
JobListener Interface . . . 66
StepListener Interface . . . 68
ChunkListener Interface . . . 70
ItemReadListener Interface . . . 72
ItemProcessListener Interface . . . 74
ItemWriteListener Interface . . . 76
Skip Listener Interfaces . . . 77
RetryListener Interface . . . 79
Batch Properties . . . 81
@BatchProperty . . . 81
Scope of property definitions for @BatchProperty . . . 84
Batch Contexts . . . 85

Batch Contexts . . . 85
Batch Context Lifecycle and Scope . . . 85
Parallelization . . . 86
PartitionMapper Interface . . . 86
PartitionReducer Interface . . . 87
PartitionCollector Interface . . . 90
PartitionAnalyzer Interface . . . 91
Decider Interface . . . 93
Transactionality . . . 95
Batch Runtime Specification . . . 96
Batch Properties Reserved Namespace . . . 96
Job Metrics . . . 96
Job Runtime Identifiers . . . 96
JobOperator . . . 97
Job XML Loading . . . 98
Application Packaging Model . . . 98
META-INF/batch.xml . . . 98
META-INF/batch-jobs . . . 99
Restart Processing . . . 99
Job Parameters on Restart . . . 99
Job XML Substitution during Restart . . . 100
Execution Sequence on Restart – Overview . . . 100
Execution Sequence on Restart – Detailed Rules . . . 100
PartitionMapper on Restart . . . 102
partitionsOverride = False . . . 102
Number of Partitions Must Be Same . . . 102
Partition Properties Populated From Current Plan . . . 102
"Numbering" of Partitions via Partition Properties . . . 102
partitionsOverride = True . . . 103
Supporting Classes . . . 103
JobContext . . . 103
StepContext . . . 105
Metric . . . 108
PartitionPlan . . . 108
BatchRuntime . . . 112
BatchStatus . . . 113
JobOperator . . . 114
JobInstance . . . 118

JobExecution . . . 119
StepExecution . . . 120
Batch Exception Classes . . . 121
Job Runtime Lifecycle . . . 123
Batch Artifact Lifecycle . . . 123
Job Repository Artifact Lifecycle . . . 123
Job Processsing . . . 123
Regular Batchlet Processsing . . . 124
Partitioned Batchlet Processsing . . . 124
Regular Chunk Processing . . . 125
Partitioned Chunk Processing . . . 126
Chunk with Listeners (except RetryListener) . . . 127
Chunk with RetryListener . . . 129
Chunk with Custom Checkpoint Processing . . . 132
Split Processing . . . 133
Flow Processing . . . 133
Stop Processing . . . 133
Batch XML XSD . . . 135
Job Specification Language . . . 136
Validation Rules . . . 136
JSL XSD . . . 136
Credits . . . 146
Change Log . . . 147
Version 1.0 Revision A - Maintenance Release . . . 147
Issues List . . . 147

Version 1.0, Revision A

Maintenance Release

Scott Kurz

Final Release: **Chris Vignola**, **18 April 2013**

License

IBM Corporation (the "Spec Lead") for the Java Batch specification (the "Specification") hereby grants permission to copy and display the Specification, in any medium without fee or royalty, provided that you include the following on ALL copies, or portions thereof, that you make:

1. A link or URL to the Specification at this location: http://jcp.org/aboutJava/communityprocess/final/jsr352/index.html
2. The copyright notice as shown herein.

```
The Spec Lead offers to grant a perpetual, non-exclusive, worldwide, fully paid-up,
royalty free, irrevocable license under its licensable copyrights and patent claims for
which there is no technically feasible way of avoiding infringement in the course of
implementing the Specification, provided that you:

a. fully implement the Specification including all its required interfaces and
functionality.
a. do not modify, subset, superset or otherwise extend the Specification's name space;
a. pass the TCK for this Specification; and
a. grant a reciprocal license to any of your patents necessary to implement required
portions of the Specification on terms consistent with the provisions of Section 6.A of
the Java Specification Participation Agreement.

THE SPECIFICATION IS PROVIDED "AS IS," AND THE SPEC LEAD AND ANY OTHER AUTHORS MAKE NO
REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO,
WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR
TITLE; THAT THE CONTENTS OF THE SPECIFICATION ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE
IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE
ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS. THE SPEC LEAD AND ANY
OTHER AUTHORS WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL OR
CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE SPECIFICATION OR THE PERFORMANCE OR
IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the Spec Lead or any other Authors may NOT be used in any
manner, including advertising or publicity pertaining to the Specification or its
contents without specific, written prior permission. Title to copyright in the
Specification will at all times remain with the Authors. No other rights are granted by
implication, estoppel or otherwise.
```

Acknowledgements

A number of individuals deserve special recognition for their contributions to forming this specification:

- Kevin Conner
- Tim Fanelli
- Cheng Fang
- Mahesh Kannan
- Scott Kurz
- Wayne Lund
- Simon Martinelli
- Michael Minella
- Kaushik Mukherjee
- Joe Pullen

Forward

This specification describes the job specification language, Java programming model, and runtime environment for batch applications for the Java platform. It is designed for use on both the Java SE and Java EE platforms. Additionally, it is designed to work with dependency injection (DI) containers without prescribing a particular DI implementation.

Table of Contents

Introduction to JSR 352

Batch processing is a pervasive workload pattern, expressed by a distinct application organization and execution model. It is found across virtually every industry, applied to such tasks as statement generation, bank postings, risk evaluation, credit score calculation, inventory management, portfolio optimization, and on and on. Nearly any bulk processing task from any business sector is a candidate for batch processing.

Batch processing is typified by bulk-oriented, non-interactive, background execution. Frequently long-running, it may be data or computationally intensive, execute sequentially or in parallel, and may be initiated through various invocation models, including ad hoc, scheduled, and on-demand.

Batch applications have common requirements, including logging, checkpointing, and parallelization. Batch workloads have common requirements, especially operational control, which allow for initiation of, and interaction with, batch instances; such interactions include stop and restart.

Applicability of Specification

This specification applies to Java SE and Java EE environments. It requires Java 6 or higher.

Domain Language of Batch

To any experienced batch architect, the overall concepts of batch processing used by JSR 352 should be familiar and comfortable. There are "Jobs" and "Steps" and developer supplied processing units called ItemReaders and ItemWriters. However, because of the JSR 352 operations, callbacks, and idioms, there are opportunities for the following:

a. significant improvement in adherence to a clear separation of concerns
b. clearly delineated architectural layers and services provided as interfaces
c. significantly enhanced extensibility

The diagram below is a simplified version of the batch reference architecture that has been used for decades. It provides an overview of the components that make up the domain language of batch processing. This architecture framework is a blueprint that has been proven through decades of implementations on the last several generations of platforms (COBOL/Mainframe, C/Unix, and now Java/anywhere). JCL and COBOL developers are likely to be as comfortable with the concepts as C, C# and Java developers. JSR 352 specifies the layers, components and technical services commonly found in robust, maintainable systems used to address the creation of simple to complex batch applications.

[image] | *images/image003.png*

The diagram above highlights the key concepts that make up the domain language of batch. A Job has one to many steps, which has no more than one ItemReader, ItemProcessor, and ItemWriter. A job needs to be launched (JobOperator), and meta data about the currently running process needs to be stored (JobRepository).

Job

A Job is an entity that encapsulates an entire batch process. A Job will be wired together via a Job Specification Language. However, Job is just the top of an overall hierarchy:

[http://static.springsource.org/spring-batch/trunk/reference/html-single/images/job-heirarchy.png] |

images/image005.png

With JSR 352, a Job is simply a container for Steps. It combines multiple steps that belong logically together in a flow and allows for configuration of properties global to all steps, such as restartability. The job configuration contains:

1. The simple name of the job
2. Definition and ordering of Steps
3. Whether or not the job is restartable

JobInstance

A JobInstance refers to the concept of a logical job run. Let's consider a batch job that should be run once at the end of the day, such as the 'EndOfDay' job from the diagram above. There is one 'EndOfDay' Job, but each individual run of the Job must be tracked separately. In the case of this job, there will be one logical JobInstance per day. For example, there will be a January 1st run, and a January 2nd run. If the January 1st run fails the first time and is run again the next day, it is still the January 1st run. Usually this corresponds with the data it is processing as well, meaning the January 1st run processes data for January 1st, etc. Therefore, each JobInstance can have multiple executions (JobExecution is discussed in more detail below); one or many JobInstances corresponding to a particular Job can be running at a given time.

The definition of a JobInstance has absolutely no bearing on the data that will be loaded. It is entirely up to the ItemReader implementation used to determine how data will be loaded. For example, in the EndOfDay scenario, there may be a column on the data that indicates the 'effective date' or 'schedule date' to which the data belongs. So, the January 1st run would only load data from the 1st, and the January 2nd run would only use data from the 2nd. Because this determination will likely be a business decision, it is left up to the ItemReader to decide. What using the same JobInstance will determine, however, is whether or not the 'state' from previous executions will be available to the new run. Using a new JobInstance will mean 'start from the beginning' and using an existing instance will generally mean 'start from where you left off'.

JobParameters

Job parameters can be specified each time a job is started or restarted. Job parameters are keyword/value string pairs. The JobOperator start and restart operations support the specification of job parameters. See section 10.4 for further details on JobOperator.

JobExecution

A JobExecution refers to the technical concept of a single attempt to run a Job. Each time a job is started or restarted, a new JobExecution is created, belonging to the same JobInstance.

Step

A Step is a domain object that encapsulates an independent, sequential phase of a batch job. Therefore, every Job is composed entirely of one or more steps. A Step contains all of the information necessary to define and control the actual batch processing. This is a necessarily vague description because the contents of any given Step are at the discretion of the developer writing it. A Step can be as simple or complex as the developer desires. A simple Step might load data from a file into the database, requiring little or no code, depending upon the implementations used. A more complex Step may have complicated business rules that are applied as part of the processing. As with Job, a Step has an individual StepExecution that corresponds with a unique JobExecution:

[http://static.springsource.org/spring-batch/trunk/reference/html-

single/images/jobHeirarchyWithSteps.png] | *images/image007.png*

StepExecution

A StepExecution represents a single attempt to execute a Step. A new StepExecution will be created each time a Step is run, similar to JobExecution. However, if a step fails to execute because the step before it fails, there will be no execution persisted for it. A StepExecution will only be created when its Step is actually started.

JobOperator

JobOperator provides an interface to manage all aspects of job processing, including operational commands, such as start, restart, and stop, as well as job repository related commands, such as retrieval of job and step executions. See section 10.4 for more details about JobOperator.

Job Repository

A job repository holds information about jobs currently running and jobs that have run in the past. The JobOperator interface provides access to this repository. The repository contains job instances, job executions, and step executions. For further information on this content, see sections 10.9.8, 10.9.9, 10.9.10, respectively.

Note the implementation of the job repository is outside the scope of this specification.

ItemReader

ItemReader is an abstraction that represents the retrieval of input for a Step, one item at a time. An ItemReader provides an indicator when it has exhausted the items it can supply. See section 9.1.1.1 for more details about ItemReaders.

ItemWriter

ItemWriter is an abstraction that represents the output of a Step, one batch or chunk of items at a time. Generally, an item writer has no knowledge of the input it will receive next, only the item that was passed in its current invocation. See section 9.1.1.3 for more details about ItemWriters.

ItemProcessor

ItemProcessor is an abstraction that represents the business processing of an item. While the ItemReader reads one item, and the ItemWriter writes them, the ItemProcessor provides access to transform or apply other business processing. See section 9.1.1.2 for more details about ItemProcessors.

Chunk-oriented Processing

JSR 352 specifies a 'Chunk Oriented' processing style as its primary pattern. Chunk oriented processing refers to reading the data one item at a time, and creating 'chunks' that will be written out, within a transaction boundary. One item is read in from an ItemReader, handed to an ItemProcessor, and aggregated. Once the number of items read equals the commit interval, the entire chunk is written out via the ItemWriter, and then the transaction is committed.

[http://static.springsource.org/spring-batch/trunk/reference/html-single/images/chunk-oriented-

processing.png] | *images/image009.png*

Batch Checkpoints

For data intensive batch applications - particularly those that may run for long periods of time - checkpoint/restart is a common design requirement. Checkpoints allow a step execution to periodically bookmark its current progress to enable restart from the last point of consistency, following a planned or unplanned interruption.

Checkpoints work naturally with chunk-oriented processing. The end of processing for each chunk is a natural point for taking a checkpoint.

JSR 352 specifies runtime support for checkpoint/restart in a generic way that can be exploited by any chunk-oriented batch step that has this requirement.

Since progress during a step execution is really a function of the current position of the input/output data, natural placement of function suggests the knowledge for saving/restoring current position is a reader/writer responsibility.

Since managing step execution is a runtime responsibility, the batch runtime must necessarily understand step execution lifecycle, including initial start, execution end states, and restart.

Since checkpoint frequency has a direct effect on lock hold times, for lockable resources, tuning checkpoint interval size can have a direct bearing on overall system throughput.

Job Specification Language

Job Specification Language (JSL) specifies a job, its steps, and directs their execution. The JSL for JSR 352 is implemented with XML and will be henceforth referred to as "Job XML".

Job

The 'job' element identifies a job.

Syntax:

```
<job id="{name}" restartable="{true|false}">
```

Where:

id	Specifies the logical *name* of the job and is used for identification purposes. It must be a valid XML string value. This is a required attribute.
restartable	Specifies whether or not this job is restartable . It must specify *true* or *false*. This is an optional attribute. The default is *true*.

Job Level Listeners

Job level listeners may be configured to a job in order to intercept job execution. The listener element may be specified as child element of the job element for this purpose. Job listener is the only listener type that may be specified as a job level listener.

Multiple listeners may be configured on a job. However, there is no guarantee of the order in which they are invoked.

Syntax:

```
<listeners>
 <listener ref="{name}">
 ...
</listeners>
```

Where:

ref	Specifies the name of a batch artifact.

Job Level Exception Handling

Any unhandled exception thrown by a job-level listener causes the job to terminate with a batch status of FAILED. In this context, "unhandled" simply means an exception thrown by the listener back to the runtime implementation.

Job Level Properties

The 'properties' element may be specified as a child element of the job element. It is used to expose properties to any batch artifact belonging to the job and also to the batch runtime. Any number of properties may be specified. Job level properties are available through the JobContext runtime object. See section 9.4 for further information about Job Context.

Syntax:

```
<properties>
  <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

name	Specifies a unique property name within the current scope. It must be a valid XML string value. This is a required attribute.
value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Step

The 'step' element identifies a job step and its characteristics. Step is a child element of job. A job may contain any number of steps. Each step may be either a chunk type step or batchlet type step. See section 8.2.1 for information on chunk type steps and section 8.2.2 for information on batchlet type steps.

Syntax:

```
<step id="{name}"
      start-limit="{integer}"
      allow-start-if-complete ="{true|false}"
      next="{flow-id|step-id|split-id|decision-id}">
```

Where:

id	Specifies the logical *name* of the step and is used for identification purposes. It must be a valid XML string value. This is a required attribute.
start-limit	Specifies the number of times this step may be started or restarted. It must be a valid XML integer value. This is an optional attribute. The default is 0, which means no limit. If the limit is exceeded, the job is placed in the FAILED state.
allow-start-if-complete	Specifies whether this step is allowed to start during job restart, even if the step completed in a previous execution. It must be *true* or *false*. A value of *true* means the step is allowed to restart. This is an optional attribute. The default is *false*.
next	Specifies the next step, flow, split, or decision to run after this step is complete. It must be a valid XML string value. This is an optional attribute. The default is this step is the last step in the job or flow. Note: next attributes cannot be specified such that a loop occurs among steps.

Chunk

The 'chunk' element identifies a chunk type step. It is a child element of the step element. A chunk type step is periodically checkpointed by the batch runtime according to a configured checkpoint policy. Items processed between checkpoints are referred to as a "chunk". A single call is made to the ItemWriter per chunk. Each chunk is processed in a separate transaction. See section 9.7 for more details on transactionality. A chunk that is not complete is restartable from its last checkpoint. A chunk that is complete and belongs to a step configured with allow-start-if-complete=true runs from the beginning when restarted.

Syntax:

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

Where:

checkpoint-policy	Specifies the checkpoint policy that governs commit behavior for this chunk. Valid values are: "item" or "custom". The "item" policy means the chunk is checkpointed after a specified number of items are processed. The "custom" policy means the chunk is checkpointed according to a checkpoint algorithm implementation. Specifying "custom" requires that the checkpoint-algorithm element is also specified. See section 8.2.1.5 for checkpoint-algorithm. It is an optional attribute. The default policy is "item".
item-count	Specifies the number of items to process per chunk when using the item checkpoint policy. It must be valid XML integer. It is an optional attribute. The default is 10. The item-count attribute is ignored for "custom" checkpoint policy.
time-limit	Specifies the amount of time in seconds before taking a checkpoint for the item checkpoint policy. It must be valid XML integer. It is an optional attribute. The default is 0, which means no limit. When a value greater than zero is specified, a checkpoint is taken when time-limit is reached or item-count items have been processed, whichever comes first. The time-limit attribute is ignored for "custom" checkpoint policy.
skip-limit	Specifies the number of exceptions a step will skip if any configured skippable exceptions are thrown by chunk processing. It must be a valid XML integer value. It is an optional attribute. The default is no limit.
retry-limit	Specifies the number of times a step will retry if any configured retryable exceptions are thrown by chunk processing. It must be a valid XML integer value. It is an optional attribute. The default is no limit.

Reader

The 'reader' element specifies the item reader for a chunk step. It is a child element of the 'chunk' element. A chunk step must have one and only one item reader.

Syntax:

```
<reader ref="{name}"/>
```

Where:

ref	Specifies the name of a batch artifact.

Reader Properties

The 'properties' element may be specified as a child element of the reader element. It is used to pass property values to a item reader. Any number of properties may be specified.

Syntax:

```
<properties>
  <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

name	Specifies a unique property name within the current scope. It must be a valid XML string value. If it matches a named property in the associated batch artifact, its value is assigned to that property. If not, it is ignored. This is a required attribute.
value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Processor

The 'processor' element specifies the item processor for a chunk step. It is a child element of the 'chunk' element. The processor element is optional on a chunk step. Only a single processor element may be specified.

Syntax:

```
<processor ref="{name}"/>
```

Where:

ref	Specifies the name of a batch artifact.

Processor Properties

The 'properties' element may be specified as a child element of the processor element. It is used to pass

property values to a item processor. Any number of properties may be specified.

Syntax:

```
<properties>
 <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

name	Specifies a unique property name within the current scope. It must be a valid XML string value. If it matches a named property in the associated batch artifact, its value is assigned to that property. If not, it is ignored. This is a required attribute.
value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Writer

The 'writer' element specifies the item writer for a chunk step. It is a child element of the 'chunk' element. A chunk type step must have one and only one item writer.

Syntax:

```
<writer ref="{name}"/>
```

Where:

ref	Specifies the name of a batch artifact.

Writer Properties

The 'properties' element may be specified as a child element of the writer element. It is used to pass property values to a item writer. Any number of properties may be specified.

Syntax:

```
<properties>
 <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

name	Specifies a unique property name within the current scope. It must be a valid XML string value. If it matches a named property in the associated batch artifact, its value is assigned to that property. If not, it is ignored. This is a required attribute.
value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Chunk Exception Handling

By default, when any batch artifact that is part of a chunk type step throws an exception to the Batch Runtime, the job execution ends with a batch status of FAILED. The default behavior can be overridden for a reader, processor, or writer artifact by configuring exceptions to skip or to retry. The default behavior can be overridden for the entire step by configuring a transition element that matches the step's exit status.

Skipping Exceptions

The skippable-exception-classes element specifies a set of exceptions that chunk processing will skip. This element is a child element of the chunk element. It applies to exceptions thrown from the reader, processor, writer batch artifacts of a chunk type step. It also applies to exceptions thrown during checkpoint commit processing. A failed commit will be treated the same as a failed write. The total number of skips is set by the skip-limit attribute on the chunk element. See section 8.2.1 for details on the chunk element.

A given exception will be skipped if it "matches" an include child element of the skippable-exception-classes element, though this might be negated (and the exception not skipped) if it also "matches" an exclude child element of skippable-exception-classes.

The behavior is determined by the "nearest superclass" in the class hierarchy.

To elaborate, in this context, "matches" means the following: For an include (or exclude) element C with @class attribute value T, an exception E "matches" C when either E is of type T or E's type is a subclass of T.

When an exception E "matches" both one or more include and one or more exclude elements, then there will be one type T1 among all the matching include/exclude elements such that all other distinct matching element types are superclasses of T1 (because of Java's single inheritance). If T1 only occurs in a matching include element then include (skip) this exception. If T1 appears in a matching exclude element (even if it also appears in a matching include element), then exclude (don't skip) this exception.

Optional Skip Listener batch artifacts can be configured to the step. A Skip Listener receives control

after a skippable exception is thrown by the reader, processor, or writer. See section 9.2.7 for details on the Skip Listener batch interfaces.

Syntax:

```
<skippable-exception-classes>
 <include class="{class name}"/>
 <exclude class="{class name}"/>
</skippable-exception-classes>
```

Where:

include class	Specifies the class name of an exception or exception superclass to skip. It must be a fully qualified class name. Multiple instances of the include element may be specified. The include child element is optional. However, when specified, the class attribute is required.
exclude class	Specifies a class name of an exception or exception superclass to not skip. 'Exclude class' reduces the number of exceptions eligible to skip as specified by 'include class'. It must be a fully qualified class name. Multiple instances of the exclude element may be specified. The exclude child element is optional. However, when specified, the class attribute is required.

Example:

```
<skippable-exception-classes>
 <include class="java.lang.Exception"/>
 <exclude class="java.io.FileNotFoundException"/>
</skippable-exception-classes>
```

The preceding example would skip all exceptions except `java.io.FileNotFoundException`, (along with any subclasses of `java.io.FileNotFoundException`).

Retrying Exceptions

The retryable-exception-classes element specifies a set of exceptions that chunk processing will retry. This element is a child element of the chunk element. It applies to exceptions thrown from the reader, processor, or writer batch artifacts of a chunk type step. It also applies to exceptions thrown by checkpoint commit processing. The total number of retry attempts is set by the retry-limit attribute on the chunk element. See section 8.2.1 for details on the chunk element.

The list of exceptions that will be retried (or not retried) is specified in the retryable-exception-classes element on the child include element. This list, however, may be modified using one or more child exclude elements. The rules for deciding whether to retry or not retry a given exception when a combination of include and exclude elements are used are analogous to the rules described in the discussion in section 8.2.1.4.1 for skipping exceptions.

Optional Retry Listener batch artifacts can be configured on the step. A Retry Listener receives control after a retryable exception is thrown by the reader, processor, or writer. See section 9.2.8 for details on the Retry Listener batch artifact.

Syntax:

```
<retryable-exception-classes>
 <include class="{class name}"/>
 <exclude class="{class name}"/>
</retryable-exception-classes>
```

Where:

include class	Specifies a class name of an exception or exception superclass to retry. It must be a fully qualified class name. Multiple instances of the include element may be specified. The include child element is optional. However, when specified, the class attribute is required.
exclude class	Specifies a class name of an exception or exception superclass to not retry. 'Exclude class' reduces the number of exceptions eligible for retry as specified by 'include class'. It must be a fully qualified class name. Multiple instances of the include element may be specified. The exclude child element is optional. However, when specified, the class attribute is required.

Example:

```
<retryable-exception-classes>
 <include class="java.io.IOException"/>
 <exclude class="java.io.FileNotFoundException"/>
</retryable-exception-classes>
```

The result is that all IOExceptions except `FileNotFoundException` (and its subclasses) would be retried.

Retry and Skip the Same Exception

When the same exception is specified as both retryable and skippable, retryable takes precedence over skippable during regular processing of the chunk. While the chunk is retrying, skippable takes precedence over retryable since the exception is already being retried.

This allows an exception to initially be retried for the entire chunk and then skipped if it recurs. When retrying with default retry behavior (see section 8.2.1.4.4) the skips can occur for individual items, since the retry is done with an item-count of 1.

Default Retry Behavior - Rollback

When a retryable exception occurs, the default behavior is for the batch runtime to rollback the current chunk and re-process it with an item-count of 1 and a checkpoint policy of item. If the optional ChunkListener is configured on the step, the onError method is called before rollback. The default retry behavior can be overridden by configuring the no-rollback-exception-classes element. See section 8.2.1.4.5 for more information on specifying no-rollback exceptions.

Preventing Rollback During Retry

The no-rollback-exception-classes element specifies a list of exceptions that override the default behavior of rollback for retryable exceptions. This element is a child element of the chunk element. If a retryable exception is thrown the default behavior is to rollback before retry. If an exception is specified as both a retryable and a no-rollback exception, then no rollback occurs and the current operation is retried. Retry Listeners, if configured, are invoked. See section 9.2.8 for details on the Retry Listener batch artifact.

The rules for determining whether a combination of include and exclude child elements of no-rollback-exception-classes results in the "no rollback" behavior or not are analogous to the rules described in the discussion in section 8.2.1.4.1 for skipping exceptions.

Syntax:

```
<no-rollback-exception-classes>
 <include class="{class name}"/>
 <exclude class="{class name}"/>
</no-rollback-exception-classes>
```

Where:

include class	Specifies a class name of an exception or exception superclass for which rollback will not occur during retry processing. It must be a fully qualified class name. Multiple instances of the include element may be specified. The include child element is optional. However, when specified, the class attribute is required.
exclude class	Specifies a class name of an exception or exception superclass for which rollback will occur during retry processing. It must be a fully qualified class name. Multiple instances of the include element may be specified. The exclude child element is optional. However, when specified, the class attribute is required.

Checkpoint Algorithm

The checkpoint-algorithm element specifies an optional custom checkpoint algorithm. It is a child element of the chunk element. It is valid when the chunk element checkpoint-policy attribute specifies the value 'custom'. A custom checkpoint algorithm may be used to provide a checkpoint decision based on factors other than only number of items, or amount of time. See section 9.1.1.4 for further information about custom checkpoint algorithms.

Syntax:

```
<checkpoint-algorithm ref="{name}"/>
```

Where:

ref	Specifies the name of a batch artifact.

Checkpoint Algorithm Properties

The 'properties' element may be specified as a child element of the checkpoint algorithm element. It is used to pass property values to a checkpoint algorithm. Any number of properties may be specified.

Syntax:

```
<properties>
 <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

Name	Specifies a unique property name within the current scope. It must be a valid XML string value. If it matches a named property in the associated batch artifact, its value is assigned to that property. If not, it is ignored. This is a required attribute.
Value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Batchlet

The batchlet element specifies a task-oriented batch step. It is specified as a child element of the step element. It is mutually exclusive with the chunk element. See 9.1.2 for further details about batchlets. Steps of this type are useful for performing a variety of tasks that are not item-oriented, such as executing a command or doing file transfer.

Syntax:

```
<batchlet ref="{name}"/>
```

Where:

Ref	Specifies the name of a batch artifact.

Batchlet Exception Handling

This section is superseded by section 8.2.7.

Batchlet Properties

The 'properties' element may be specified as a child element of the batchlet element. It is used to pass property values to a batchlet. Any number of properties may be specified.

Syntax:

```
<properties>
 <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

Name	Specifies a unique property name within the current scope. It must be a valid XML string value. If it matches a named property in the associated batch artifact, its value is assigned to that property. If not, it is ignored. This is a required attribute.
value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Step Level Properties

The 'properties' element may be specified as a child element of the step element. It is used to expose properties to any step level batch artifact and also to the batch runtime. Any number of properties may be specified. Step level properties are available through the StepContext runtime object. See section 9.4 for further information about StepContext.

Syntax:

```
<properties>
 <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

name	Specifies a unique property name within the current scope. It must be a valid XML string value. This is a required attribute.
value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Step Level Listeners

Step level listeners may be configured to a job step in order to intercept step execution. The listener element may be specified as child element of the step element for this purpose. The following listener types may be specified according to step type:

- chunk step - step listener, item read listener, item process listener, item write listener, chunk listener, skip listener, and retry listener
- batchlet step - step listener

Multiple listeners may be configured on a step. However, there is no guarantee of the order in which they are invoked.

Syntax:

```
<listeners>
 <listener ref="{name}">
 ...
</listeners>
```

Where:

ref	Specifies the name of a batch artifact.

Step Level Listener Properties

The 'properties' element may be specified as a child element of the step-level listeners element. It is used to pass property values to a step listener. Any number of properties may be specified.

Syntax:

```
<properties>
 <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

name	Specifies a unique property name within the current scope. It must be a valid XML string value. If it matches a named property in the associated batch artifact, its value is assigned to that property. If not, it is ignored. This is a required attribute.
value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Example:

```
<listener ref="{name}">
 <properties>
  <property name="Property1" value="Property1-Value"/>
 </properties>
</listener>
```

Step Sequence

The first step, flow, or split defines the first step (flow or split) to execute for a given Job XML. "First" means first according to order of occurrence as the Job XML document is parsed from beginning to end. The 'next' attribute on the step, flow, or split defines what executes next. The next attribute may specify a step, flow, split, or decision. For the purpose of discussing transitioning it is convenient to group these four with the term "execution elements". The next attribute is supported on step, flow, and split elements. Steps, flows, and decisions may also use the "next" *element* to specify what executes next. The next attribute and next element may not be used in a way that allows for looping among job execution elements.

Syntax:

```
<next on="{exit status}" to="{id}" />
```

Where:

on	Specifies an exit status to match to the current next element. It must be a valid XML string value. Wildcards of "**" and "" may be used. **"" matches zero or more characters. "" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.
to	Specifies the id of another step, split, flow, or decision, which will execute next. It must be a valid XML string value. It must match an id of another step, split, flow, or decision in the same job. For a step inside a flow, the id must match another step in the same flow. This is a required attribute.

See section 8.6 for more details about transition elements and section 8.9 for details on transitioning rules.

Step Partitioning

A batch step may run as a partitioned step. A partitioned step runs as multiple instances of the same step definition across multiple threads, one partition per thread. The number of partitions and the number of threads is controlled through either a static specification in the Job XML or through a batch artifact called a partition mapper. Each partition needs the ability to receive unique parameters to instruct it which data on which to operate. Properties for each partition may be specified statically in the Job XML or through the optional partition mapper. Since each thread runs a separate copy of the step, chunking and checkpointing occur independently on each thread for chunk type steps.

There is an optional way to coordinate these separate units of work in a partition reducer so that

backout is possible if one or more partitions experience failure. The PartitionReducer batch artifact provides a way to do that. A PartitionReducer provides programmatic control over logical unit of work demarcation that scopes all partitions of a partitioned step.

The partitions of a partitioned step may need to share results with a control point to decide the overall outcome of the step. The PartitionCollector and PartitionAnalyzer batch artifact pair provide for this need.

The 'partition' element specifies that a step is a partitioned step. The partition element is a child element of the 'step' element. It is an optional element. Syntax:

```
<partition>
```

Example:

The following Job XML snippet shows how to specify a partitioned step: <step id="Step1"> <chunk .../> or <batchlet ... /> <partition .../> </step>

Partition Plan

A partition plan defines several configuration attributes that affect partitioned step execution. A partition plan specifies the number of partitions, the number of partitions to execute concurrently, and the properties for each partition. A partition plan may be defined in a Job XML declaratively or dynamically at runtime with a partition mapper.

The 'plan' element is a child element of the 'partition' element. The 'plan' element is mutually exclusive with partition mapper element. See section 9.5.1 for further details on partition mapper.

Note the specification does not attempt to guarantee order of partition execution with respect to the order within a statically or dynamically-defined plan.

Syntax:

```
<plan partitions="{number}" threads="{number}"/>
```

Where:

Partitions	Specifies the number of partitions for this partitioned step. This is a an optional attribute. The default is 1.

threads	Specifies the maximum number of threads on which to execute the partitions of this step. Note the batch runtime cannot guarantee the requested number of threads are available; it will use as many as it can up to the requested maximum. This is an optional attribute. The default is the number of partitions.

Example:

The following Job XML snippet shows how to specify a step partitioned into 3 partitions on 2 threads:

```
<step id="Step1">
  <chunk .../>
  <partition>
    <plan partitions="3" threads="2"/>
  </partition>
</step>
```

Partition Properties

When defining a statically partitioned step, it is possible to specify unique property values to pass to each partition directly in the Job XML using the property element. See section 9.5.1 for further information on partition mapper.

Syntax:

```
<properties partition="_partition-number_">
 <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

partition	Specifies the logical partition number to which the specified properties apply. This must be a non-negative integer value, starting at 0.
name	Specifies a unique property name within the current scope . It must be a valid XML string value. If it matches a named property in the associated batch artifact, its value is assigned to that property. If not, it is ignored. This is a required attribute.

value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Example:

The following Job XML snippet shows a step of 2 partitions with a unique value for the property named "filename" for each partition:

```
<partition>
 <plan partitions="2">
  <properties partition="0">
   <property name="filename" value="/tmp/file1.txt"/>
  </properties>
  <properties partition="1">
   <property name="filename" value="/tmp/file2.txt"/>
  </properties>
 </plan>
</partition>
```

Partition Mapper

The partition mapper provides a programmatic means for calculating the number of partitions and threads for a partitioned step. The partition mapper also specifies the properties for each partition. The mapper element specifies a reference to a PartitionMapper batch artifact; see section 9.5.1 for further information. Note the mapper element is mutually exclusive with the plan element.

Syntax:

```
<mapper ref="{name}">
```

Where:

ref	Specifies the name of a batch artifact.

Example:

```
<partition>
 <mapper ref="MyStepPartitioner"/>
</partition>
```

Mapper Properties

The 'properties' element may be specified as a child element of the mapper element. It is used to pass property values to a PartitionMapper batch artifact. Any number of properties may be specified.

Syntax:

```
<properties>
 <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

name	Specifies a unique property name within the current scope. It must be a valid XML string value. If it matches a named property in the associated batch artifact, its value is assigned to that property. If not, it is ignored. This is a required attribute.
value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Partition Reducer

A partitioned step may execute with an optional partition reducer. A partition reducer provides a kind of unit of work demarcation around the processing of the partitions. Programmatic interception of the partitioned step's lifecycle is possible through the partition reducer. The reducer element specifies a reference to a PartitionReducer batch artifact; see section 9.5.2 for further information.

The 'reducer' element is a child element of the 'partition' element.

Syntax:

```
<reducer ref="{name}">
```

Where:

ref	Specifies the name of a batch artifact.

Example:

```
<partition>
<reducer ref="MyStepPartitionReducer"/>
</partition>
```

Partition Reducer Properties

The 'properties' element may be specified as a child element of the PartitionReducer element. It is used to pass property values to a PartitionReducer batch artifact. Any number of properties may be specified.

Syntax:

```
<properties>
 <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

name	Specifies a unique property name within the current scope. It must be a valid XML string value. If it matches a named property in the associated batch artifact, its value is assigned to that property. If not, it is ignored. This is a required attribute.
value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Partition Collector

A Partition Collector is useful for sending intermediary results for analysis from each partition to the step's Partition Analyzer. A separate Partition Collector instance runs on each thread executing a partition of the step. The collector is invoked at the conclusion of each checkpoint for chunking type steps and again at the end of partition; it is invoked once at the end of partition for batchlet type steps. A collector returns a Java Serializable object, which is delivered to the step's Partition Analyzer. See section 9.5.4 for further information about the Partition Analyzer. The collector element specifies a reference to a PartitionCollector batch artifact; see section 9.5.3 for further information.

The 'collector' element is a child element of the 'partition' element.

Syntax:

```
<collector ref="{name}">
```

Where:

ref	Specifies the name of a batch artifact.

Example:

```
<partition>
<collector ref="MyStepCollector"/>
</partition>
```

Partition Collector Properties

The 'properties' element may be specified as a child element of the collector element. It is used to pass property values to a PartitionCollector batch artifact. Any number of properties may be specified.

Syntax:

```
<properties>
 <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

name	Specifies a unique property name within the current scope. It must be a valid XML string value. If it matches a named property in the associated batch artifact, its value is assigned to that property. If not, it is ignored. This is a required attribute.
value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Partition Analyzer

A Partition Analyzer receives intermediary results from each partition sent via the step's Partition Collector. A Partition analyzer runs on the step main thread and serves as a collection point for this data. The PartitionAnalyzer also receives control with the partition exit status for each partition, after that partition ends. An analyzer can be used to implement custom exit status handling for the step, based on the results of the individual partitions. The analyzer element specifies a reference to a PartitionAnalyzer batch artifact; see section 9.5.4 for further information.

Syntax:

```
<analyzer ref="{name}">
```

Where:

ref	Specifies the name of a batch artifact.

Example:

```
<partition>
<analyzer ref="MyStepAnalyzer"/>
</partition>
```

Partition Analyzer Properties

The 'properties' element may be specified as a child element of the analyzer element. It is used to pass property values to a PartitionAnalyzer batch artifact. Any number of properties may be specified.

Syntax:

```
<properties>
 <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

name	Specifies a unique property name within the current scope. It must be a valid XML string value. If it matches a named property in the associated batch artifact, its value is assigned to that property. If not, it is ignored. This is a required attribute.
value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Step Exception Handling

Any unhandled exception thrown by any step-level artifact during step processing causes the step to terminate with a batch status of FAILED. In this context, "unhandled" means an exception thrown by the execution of the artifact back to the runtime implementation which does not result in a skip or a retry as described in section 8.2.1.4. See section 8.9.2 for complete details on transitioning after an unhandled exception.

Flow

A flow defines a sequence of execution elements that execute together as a unit. When the flow is finished, it is the entire flow that transitions to the next execution element. A flow may transition to a step, split, decision, or another flow. A flow may contain step, flow, decision, and split execution elements. See section 8.5 for more on decisions. See section 8.4 for more on splits. The execution elements within a flow may only transition among themselves; they may not transition to elements outside of the flow. A flow may also contain the transition elements next, stop, fail, and end. See section 8.6 for more on transition elements.

Syntax:

```
<flow id="{name}"next="{flow-id|step-id|split-id|decision-id}">
 <step> ... </step> ...
</flow>
```

Where:

id	Specifies the logical *name* of the flow and is used for identification purposes. It must be a valid XML string value. This is a required attribute.
next	Specifies the next step, flow, split, or decision to run after this step is complete. It must be a valid XML string value. This is an optional attribute. The default is this flow is the last execution element in the job. Note: next attributes cannot be specified such that a loop occurs among steps.

Split

A split defines a set of flows that execute concurrently. A split may include only flow elements as children. See section 8.3 for more on flows. Each flow runs on a separate thread. The split is finished after all flows complete. When the split is finished, it is the entire split that transitions to the next execution element. A split may transition to a step, flow, decision, or another split.

Syntax:

```
<split id="{name}"next="{flow-id|step-id|split-id|decision-id}">
 <flow> ... </flow>
 ...
</split>
```

Where:

id	Specifies the logical *name* of the split and is used for identification purposes. It must be a valid XML string value. This is a required attribute.
next	Specifies the next step, flow, split, or decision to run after this step is complete. It must be a valid XML string value. This is an optional attribute. The default is this split is the last execution element in the job. Note: next attributes cannot be specified such that a loop occurs among steps.

Split Termination Processing

Incomplete

The effort of the initial 1.0 final release specification to define split termination processing is recognized as incomplete. This is related to the recognition that flow transitioning is incomplete (section 8.9.5).

As such, there is no well-defined mechanism for "passing back" status from the individual child flows of a split and aggregating them into a status at the split level. There is, accordingly, no termination based on the status of the constituent flows performed after a split execution.

However, the implementor must be aware that a split may have a child flow where the flow itself or a flows child (step, decision, etc.) causes the job execution to terminate. This could be via an end, stop, or fail transition element, or via an unhandled exception.

In such a case the job should then cease execution before transitioning past the current, containing split, on to the next execution element.

Typically only one such element (in one single flow) would terminate job execution, with a corresponding batch and exit status that would then be set by the implementation as the job-level batch status and exit status, since typically the whole split would be intended to complete.

The spec does not make an effort, then, to define the outcome if more than one flow within a split produced a terminating status. A suggestion, though, is that a FAILED batch status should be given preference to STOPPED, which should be given preference to COMPLETED status, and a natural corollary might be to bubble up the associate exit status as the job-level exit status as well.

Decision

A decision provides a customized way of determining sequencing among steps, flows, and splits. The decision element may follow a step, flow, or split. A job may contain any number of decision elements. A decision element is the target of the "next" attribute from a job-level step, flow, split, or another decision. A decision must supply a decider batch artifact (see section 9.6). The decider’s purpose is to decide the next transition. The decision uses any of the transition elements, stop, fail, end, and next

elements to select the next transition. See section 8.6 for further information on transition elements. The decider return value will also be set as the current value of the job exit status, in addition to being matched against the decisions own child transition elements to decide the next transition.

Syntax:

```
<decision id="{name}" ref="{ref_-_name}">
```

Where:

id	Specifies the logical *name* of the decision and is used for identification purposes. It must be a valid XML string value. This is a required attribute.
ref	Specifies the name of a batch artifact.

Example:

```
<decision id="AfterFlow1" ref="MyDecider">
...
</decision>
```

Decision Properties

The 'properties' element may be specified as a child element of the decision element. It is used to pass property values to a decider. Any number of properties may be specified.

Syntax:

```
<properties>
 <property name="{property-name}" value="{name-value}"/>
</properties>
```

Where:

name	Specifies a unique property name within the current scope. It must be a valid XML string value. If it matches a named property in the associated batch artifact, its value is assigned to that property. If not, it is ignored. This is a required attribute.

value	Specifies the value corresponding to the named property. It must be a valid XML string value. This is a required attribute.

Decision Exception Handling

Any exception thrown by a batch artifact invoked during decision handling will end the job with a batch status of FAILED. This exception is visible to job-level listeners.

Transition Elements

Transition elements may be specified in the containment scope of a step, flow, or decision (but not a split) to direct job execution sequence or to terminate job execution. There are fo ur transition elements:

1. next - directs execution flow to the next execution element.
2. fail - causes a job to end with FAILED batch status.
3. end - causes a job to end with COMPLETED batch status.
4. stop - causes a job to end with STOPPED batch status.

Fail, end, and stop are considered "terminating elements" because they cause a job execution to terminate.

Next Element

The next element is used to transition execution to the next execution element. Multiple next elements may be specified in the current containment scope. Syntax:

```
<next on="{exit status}" to="{step id_|_flow id_|_split id}"/>
```

Where:

on	Specifies the exit status value that activates this end element. It must be a valid XML string value. Wildcards of "**" and "" may be used. **"" matches zero or more characters. "" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.

to	Specifies the execution element to which to transition after this decision. It must be a valid XML string value. This is a required attribute. Note: the to value cannot specify the next execution element such that a loop occurs in the batch job.

Example:

<step id="Step1"> <next on="*" to="Step2"/> </step>

Fail Element

The fail element is used to terminate the job at the conclusion of the current step or flow. The job batch status is set to FAILED. This does not, however, directly affect the batch status of the step containing the fail element. Multiple fail elements may be specified in the current containment scope. The fail element is supported as a child of the step, flow, and decision elements.

Syntax:

```
<fail on="{exit status}" exit-status="{exit status}"/>
```

Where:

on	Specifies the exit status value that activates this fail element. It must be a valid XML string value. Wildcards of "**" and "" may be used. "" matches zero or more characters. "" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.
exit-status	Specifies the new exit status for the job. It must be a valid XML string value. This is an optional attribute. If not specified, the job-level exit status is unchanged. This attribute does not directly change any step exit status (particularly the step which contains this fail element).

Example:

<step id="Step1"> <fail on="FAILED" exit-status="EARLY COMPLETION"/> </step>

End Element

The end element is used to terminate the job at the current step. The job batch status is set to COMPLETED. This does not, however, directly affect the batch status of the step containing the end

element. Multiple end elements may be specified in the current containment scope. The end element is supported as a child of the step, flow, and decision elements.

Syntax:

```
<end on="{exit status}" exit-status="{exit status}"/>
```

Where:

on	Specifies the exit status value that activates this end element. It must be a valid XML string value. Wildcards of "**" and "" may be used. ""** matches zero or more characters. "" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.
exit-status	Specifies the new exit status for the job. It must be a valid XML string value. This is an optional attribute. If not specified, the job-level exit status is unchanged. This attribute does not directly change any step exit status (particularly the step which contains this end element).

Example:

```
<step id="Step1">
 <end on="COMPLETED" exit-status="EARLY COMPLETION">
</step>
```

Stop Element

The stop element is used to terminate the job after the current step or flow. If the stop element matches the exit status, the job-level batch status is then set to STOPPED. This does not, however, directly affect the batch status of the step containing the . Multiple stop elements may be specified in the current containment scope. The stop element is supported as a child of step, flow, and decision elements.

<stop on="{exit status}" exit-status="{exit status}" restart="{step id_|*flow id*|_split id}"/>

Where:

on	Specifies the exit status value that activates this end element. It must be a valid XML string value. Wildcards of "**" and "" may be used. ""** matches zero or more characters. "" matches exactly one character. It must match an exit status value in order to have effect. This is a required attribute.
exit-status	Specifies the exit status for the job. It must be a valid XML string value. This is an optional attribute. If not specified, the job-level exit status is unchanged. This attribute does not directly change any step exit status (particularly the step which contains this stop element).
restart	Specifies the job-level step, flow, or split at which to restart when the job is restarted. It must be a valid XML string value. This is an optional attribute.

Example:

<step id="Step1"> <stop on="COMPLETED" restart="step2"/> </step>

Batch and Exit Status

Batch execution reflects a sequence of state changes, culminating in an end state after a job has terminated. These state changes apply to the entire job as a whole, as well as to each step within the job. These state changes are exposed through the programming model as status values. There is both a runtime status value, called "batch status", as well as a user-defined value, called "exit status".

A job and each step in a job end with a batch status and exit status value. Batch status is set by the batch runtime; exit status may be set through the Job XML or by the batch application. The exit status for a job and a step will be initially set to null. At the time that the job or step completes execution, if the exit status is equal to null, it will then be set by the runtime implementation to the string value of the batch status, which will be its final value. The batch and exit status values are available in the JobContext and StepContext runtime objects, and the exit status can be set explicitly via any batch artifact. The overall batch and exit status for the job are available through the JobOperator interface. Batch and exit status values are strings. The following batch status values are defined:

Value	Meaning
STARTING	Batch job has been passed to the batch runtime for execution through the JobOperator interface start or restart operation. A step has a status of STARTING before it actually begins execution.

STARTED	Batch job has begun execution by the batch runtime. A step has a status of STARTED once it has begun execution.
STOPPING	Batch job has been requested to stop through the JobOperator interface stop operation or by a <stop> element in the Job XML. A step has a status of STOPPING as soon as JobOperator.stop receives control.
STOPPED	Batch job has been stopped through the JobOperator interface stop operation or by a <stop> element in the Job XML. A step has a status of STOPPED once it has actually been stopped by the batch runtime.
FAILED	Batch job has ended due to an unresolved exception or by a <fail> element in the Job XML. A step has a status of FAILED under the same conditions.
COMPLETED	Batch job has ended normally or by an <end> element in the Job XML. A step has a status of COMPLETED under the same conditions.
ABANDONED	Batch job has been marked abandoned through the JobOperator interface abandon operation. An abandoned job is still visible through the JobOperator interface, but is not running, nor can it be restarted. It exists only as a matter of history.

A job execution will end under the following conditions:

1. A job-level execution element (step, flow, or split) finishes execution, without specifying a "next" attribute and without the exit status matching any transition elements. (See section 8.9.2 for details). In this case, the batch status is set to COMPLETED.
2. A step throws an exception to the batch runtime that does not match skip or retry criteria, with the exit status not matching any transition elements. In this case, the batch status is set to FAILED. (See section 8.9.2 for details). In the case of partitioned or concurrent (split) step execution, all other still-running parallel instances are allowed to complete before the job ends with FAILED batch status.
3. A step, flow, or decision terminates execution with a stop, end, or fail element. In this case, the batch status is STOPPED, COMPLETED, or FAILED, respectively .

The batch and exit status of the job is set as follows:

1. Batch status is initially set to STARTING by the batch runtime. Immediately before starting the first step, the batch runtime sets the batch status to STARTED .

2. Exit status can be overridden by any artifact by invoking the exit status setter method on the JobContext object.
3. Exit status can be overridden by a decision element.
4. Exit status can be overridden by a terminating transition element on a step, flow, or split. See section 8.6.
5. Final batch status is set by the batch runtime depending on the outcome of the job. See table above. Exit status is set to the final batch status if it was not overridden by any of the override means described earlier in this list. Note the last override to set exit status during the course of job execution takes precedence over all others.

In addition to these conditions and events which are well-defined by this specification, it is also recognized that the runtime may be forced to make another transition of job and step batch status.

For example, a JVM hang may cause a job to appear in STARTED state even though it is no longer running. The specification forbids running multiple executions of a given job instance at the same time. In order to recover and allow restart it is expect that a batch runtime implementation might provide a mechanism to automatically or through user intervention mark the appropriate job and step execution(s) as FAILED (i.e. set the batch status as FAILED).

The details are left entirely to the implementation, we are just recognizing here that this is a valid state transition.

Batch and Exit Status for Steps

Step batch status is set initially, and then again at the conclusion of the step, by the batch runtime. **Step exit status is initially set to the same value as batch status.** Step exit status may be set by any batch artifact configured to the step by invoking the exit status setter method in the StepContext object. See section 9.4 for further information about the StepContext object. Setting the step exit status does not alter the execution of the current step, but rather, is available to influence the execution of subsequent steps via transition elements (see 8.6) and deciders (see9.6). If no batch artifact sets the exit status, the batch runtime will default the value to the string form of the batch status value of the step when step execution completes. An important point to note is that transition elements do not affect the batch and exit status of their containing step (for a step with one or more child transition elements), but only potentially affect the batch and exit status of the job.

Example:

```
<step id="FS1">
 <batchlet >
  <next on="RC0" />
  <fail on="RC4" exit-status="BAD"/>
  <fail on="RC8" />
</step>
```

Suppose for the above example JSL snippet, FS1s batchlet executes normally with an exit status of "RC4". Then step FS1s batch status will end up as COMPLETED, and FS1s exit status will end up as "RC4". The jobs batch status will end up as FAILED and the jobs exit status will end up as "BAD". Likewise, if the batchlet completes with an exit status of "RC8" the steps batch and exit status will be COMPLETED and "RC8", respectively, while the jobs batch and exit status will be FAILED and "FAILED" (assuming the job exit status hasnt been set and defaults in this case).

Note the implications for restart processing. For example, a completed step wont re-run just because the step includes a transition element failing the job on the original step executions exit status. See section 10.8 for more on restart processing.

Exit Status for Partitioned Steps

The exit status for a partitioned step follows the same rules as for a regular step except for an exit status set by batch artifacts processing individual partitions . This means any batch artifact running on the main thread of the partitioned step can set the steps exit status via the exit status setter method on the StepContext object , the same as for a non-partitioned step.E.g. a steps partition analyzer, partition reducer, or step listener could each potentially set the steps exit status in this simple manner (since each of these artifacts run on the initial thread, not the threads processing an individual partition). If the exit status is not set it defaults to batch status at the end of step execution , the same as for a non-partitioned step.

For a partitioned batchlet, each thread processing a partition may return a separate exit status. However, these exit status values are ignored unless a partition analyzer is used to coalesce these separate exit status values into a final exit status value for the step.

The batch runtime maintains a StepContext clone per partition. For a partitioned batchlet or chunk, any batch artifact running on any of the threads processing a partition would merely set a separate exit status through the StepContext clone. These exit status values are ignored unless a partition analyzer is used to coalesce these separate exit status values into a final exit status value for the step.

Job XML Substitution

Job XML supports substitution as part of any attribute value. The following expression language is supported on all attributes:

```
<attribute-value> ::= ' " ' <principle-value-expression>
[<default-expression>] ' " '

<principle-value-expression> ::= <value-expression>

<value-expression> ::= "#\{"<operator-expression>"}" | <string-literal>
[ <value-expression> ]

<default-expression> ::= ":" <value-expression> ";"

<operator-expression> ::= <operator1> | <operator2> | <operator3> |
<operator4> | <operator5>

<operator1> ::= "jobParameters" "[" <target-name> "]"

<operator2> ::= "jobProperties" "[" <target-name> "]"

<operator3> ::= "systemProperties" "[" <target-name> "]"

<operator4> ::= "partitionPlan" "[" <target-name> "]"

<target-name> ::= " ' " <string-literal> " ' "

<string-literal> is a valid XML string value.
```

Substitution Processing Rules

Substitution expressions are processed for both initial job start and on job restart. All substitution expressions must be resolved before the job can be started or restarted, except for the partitionPlan operator, which has deferred resolution - see section 8.8.1.4 for more on that. After substitution expression resolution, the resultant XML document must be checked for validity, according to the guidelines outlined in section 13, Job Specification Language XSD.

A substitution expression may reference a job parameter or a job property by specifying the name of the parameter or property through a substitution expression operator. This name is referred to generally in substitution expression syntax as a "target name". There are four substitution operators:

1. jobParameters - specifies to use a named parameter from the job parameters.
2. jobProperties - specifies to use a named property from among the job's properties.
3. systemProperties - specifies to use a named property from the system properties.
4. partitionPlan - specifies to use a named property from the partition plan of a partitioned step.

jobParameters Substitution Operator

The jobParameters substitution operator resolves to the value of the job parameter with the specified

target name.

jobProperties Substitution Operator

The jobProperties substitution operator resolves to the value of the job property with the specified target name. This property is found by recursively searching from the innermost containment scope (this includes earlier properties within the current scope) to the outermost scope until a property with the specified target name is found.

E.g. The batch runtime would attempt resolution of the jobProperties operator specification in each of the two following reader property definitions by first searching for earlier property definitions within the reader properties collection, then the step properties collection (there are none in this example), then the job properties collection (if any). The search stops at the first occurrence of the specified target name.

```
 <job id="job1">
 <properties>
 <property name="filestem" value="postings"/>

<property name="outputlog" value="jobmessages"/>

</properties>
 <step id="step1">

<properties/>
 <chunk>

<reader ref="MyReader">

<properties>
 <property name="infile.name"
value="#\{jobProperties['filestem']}.txt"/>

<property name="outputlog" value="readermessages"/>

<property name="outfile.name"
value="#\{jobProperties['outputlog']}.txt"/>
 </properties>

</reader>
 </chunk>

</step>

</job>
```

The resolved value for reader property "infile.name" would be "postings.txt".

The resolved value for reader property "outfile.name" would be "readermessages.txt".

systemProperties Substitution Operator

The systemProperties substitution operator resolves to the value of the system property with the specified target name.

partitionPlan Substitution Operator

The partitionPlan substitution operator resolves to the value of the partition plan property with the specified target name from the PartitionPlan returned by the PartitionMapper. Partition plan properties are in scope only for the step to which the partition plan is defined. The partitionPlan operator is resolved separately for each partition before the partition execution begins.

E.g. Given job, job1:

```
<job id="job1">
 <step id="step1">
 <chunk>
 <reader  ref="MyReader>
 <properties>
 <property name="infile.name"
 value="file#\{partitionPlan['myPartitionNumber']}.txt"/>

<property name="outfile.name"
 value="#\{partitionPlan['outFile']}"/>
 </properties>
 </reader>
 <writer ref="MyWriter"/>
 </chunk>

<partition>
 <mapper ref="MyMapper "/>
 </partition>
 </step>
 </job>
```

And MyMapper implementation:

```
public class MyMapper implements PartitionMapper \{
  public PartitionPlan mapPartitions() \{
    PartitionPlanImpl pp= new PartitionPlanImpl();
    pp.setPartitions(2);

    Properties p0= new Properties();
    p0.setProperty("myPartitionNumber", "0");
    p0.setProperty("outFile", "outFileA.txt");

    Properties p1= new Properties();
    p1.setProperty("myPartitionNumber", "1");
    p1.setProperty("outFile", "outFileB.txt");

    Properties[] partitionProperties= new Properties[2];
    partitionProperties[0]= p0;
    partitionProperties[1]= p1;
    pp.setPartitionProperties(partitionProperties);

    return pp;
  }
}
```

The step1 chunk would run as two partitions, with the itemReader property "infile.name" resolved to "file0.txt" and "file1.txt" for partitions 0 and 1, respectively. Also, itemReader property "outfile.name" would resolve to "outFileA.txt", and "outFileB.txt" for partitions 0 and 1, respectively.

Substitution Expression Default

Substitutions expressions may include a default value using the ":" operator. The default is applied if the substitution's principle value expression resolves to the empty string "".

Property Resolution Rule

Properties specified by a substitution operator must be defined before they can be used in a substitution expression.

Examples:

Resolvable Property Reference

The batch runtime will resolve a substitution reference to a property that occurs before it is referenced. In the following example, property "infile.name" is defined before it is used to form the value of property "tmpfile.name". This is a resolvable reference.E.g.

```
 <property name="infile.name" value="in.txt" />
 <property name="tmpfile.name"
value="#\{jobProperties['infile.name']}.tmp" />
```

The batch runtime resolves a resolvable reference with the resolved value of the specified property reference.

Unresolvable Property Reference

The batch runtime will not resolve a substitution reference to a property whose first occurrence is after it is referenced. In the following example, property "infile.name" is defined after it is used to form the value of property "tmpfile.name". This is a unresolvable reference.E.g.

```
 <property name="tmpfile.name"
value="in.txt#\{jobProperties[infile.name]}" />
 <property name="infile.name" value="in.txt" />
```

The batch runtime resolves an unresolvable reference in XML to the empty string "".

Undefined Target Name Rule

A substitution expression operator that specifies an undefined target name is assigned the empty string in XML.

Job Restart Rule

Job Parameters may be specified on job restart. Substitution expression resolution occurs on each restart. This makes it possible for new values to be used in Job XML attributes during job restart. While all substitution expressions resolve the same way on restart as on initial start, there is a special rule for the number of partitions in a partitioned step:

The number of partitions in a partition plan

The batch runtime determines the number of partitions in a partitioned step the first time the step is attempted. The batch runtime remembers that decision and applies it to that step on the next job execution, once the previous job execution is restarted. The decision cannot be altered by a substitution expression. The decision can be altered, however, through a PartitionMapper artifact by specifying the "override" option in the PartitionPlan object. See section 10.9.4 for details on the PartitionPlan class.

Examples

```
       <property name="infile.name" value="in.txt" />
```

Resolved property: infile.name="in.txt"

```
       <property name="infile.name"
value="#\{jobParameters['infile.name']}" />
```

Resolved property: infile.name= value of infile.name job parameter

```
       <property name="infile.name"
value="#\{systemProperties['infile.name']}" />
```

Resolved property: infile.name= value of infile.name system property

```
       <property name="infile.name"
value="#\{jobProperties['infile.name']}" />
```

Resolved property: infile.name= value of infile.name job property

```
       <property name="infile.name"
value="#\{partitionPlan['infile.name']}" />
```

Resolved property: infile.name= value of infile.name from partition plan for the current partition

```
       <property name="infile.name"
value="#\{jobParameters['infile.name']}:in.txt;" />
```

Resolved property: infile.name = value of infile.name job parameter or "in.txt" if infile.name job parameter is unspecified.

Transitioning Rules

Combining Transition Elements

Any combination of transition elements can be included at the end of a step, flow, or decision definition. Combinations can include zero, one, or more than one instance of a single type of execution element,E.g. next.

Transition elements are evaluated in sequential order as they occur within the JSL document. I.e. the appropriate exit status is compared with the on attribute value of the first transition element in the sequence and, if it matches, then the corresponding transition is perfomed, and the rest of the transition elements are ignored. If not, the second transition element is evaluated, etc.

Example:

```
<step id="Step1">
 <next on="RC0" to="Step2"/>

<next on="RC4" to="Step3"/>

<end on="RC4" exit-status="DONE"/>

<fail on="*"/> <!-- Matches anything, so only makes sense as last
transition element-->

</step>
```

Transitioning Precedence Rules

The transition elements are always "evaluated" first, and if a match is found, execution transitions accordingly (either to another execution element or the job is stopped or failed).

If a match is not found among the transition elements (which would always be the case if there are no transition elements), then transition proceeds as follows:

1. If execution resulted in an unhandled exception, then the job ends with batch status of FAILED.
2. If execution ended normally, and the execution element whose execution is completing contains a next attribute, then execution transitions to the element named by this next attribute value.
3. If execution ended normally, and the execution element whose execution is completing does not contain a next attribute, then the job ends normally (with COMPLETED batch status). For transitioning from a step within a flow, this statement doesnt apply. See section 8.9.4 for details.

The following examples illustrate how the above rules might be employed:

Example 1: Transition to Step2, unless exit status of RC_ABORT seen, in which case fail the job

```
<step id="Step1" next="Step2">
 <fail on="RC_ABORT" exit-status="ABORTED"/>

</step>
```

Example 2: Transition to Step2, but if exception thrown, transition to RecoveryStep.

```
<step id="Step1" next="Step2">

<!-- Assumes step exit status defaults to step batch status (FAILED)
-->
 <next on="FAILED" to="RecoveryStep"/>

<fail on="*"/>

</step>
```

Note that the second example shows it is possible for a job to executed to COMPLETED status, even though a constituent step ends with FAILED batch status (See section 8.2.7).

Loop definition

The specification prohibits next and to attribute values that result in a "loop". More precisely, this means that no execution element can be transitioned to twice within a single job execution.

This wording is purposely written this way rather than merely saying no execution element can be executed twice within a single job execution. Say "step1" executed to completion during an initial execution which ultimately failed, and upon restart we transitioned past "step1" without executing it since it had already completed, but we subsequently transitioned (back) to "step1". This may only be a single execution of "step1" during a single job execution, but it still violates the looping prohibition.

The runtime may detect potential loops in an initial validation phase, as described in section 13.1, or may only detect loops once they occur.

Transitioning From Within Flows

As mentioned in section 8.3, an execution element which is a child of a flow may only transition to another execution element within the same flow. The flows transition elements, however, would transition execution to the next execution element at the level of the execution scope containing the flow ,E.g. the job.

For terminating transitions (stop, end, fail) as well as failures caused by unhandled exceptions, it is the entire job execution which is terminated. It is not just the case that the flow alone is somehow failed or ended yet with another level of transitioning occurring at the containing (e.g. job) level.

1. Note: transition via next outside of the flow is not permitted. If this is not detected during job validation (see section 13.1), then at runtime the job execution will end at this point with batch status of FAILED.

When a child of a flow completes normally, and when there are no matching transition elements as well as no next attribute at the level of this child of a flow, then the flow ends.

Another way of stating rules #2 and #3 in this section would be to say that all the rules in section 8.9.2

apply to transitions within flows (i.e. among children of flows) and are effective at the job level, except for rule #3 in section 8.9.2 (this case does not necessarily end the job).

See the example at the end of section 8.9.5 for further clarification.

Flow-level Transitions

Undefined

It is recognized that the specification is incomplete with respect to how exactly flow transition elements are evaluated. Though the list in section 10.8.4 has an assertion in rule 3.e. that suggests using the exit status of the last contained execution element as a flow-level exit status, this does not seem to be a complete definition. For example, what if the last execution element within the flow is a split

This might be rectified in a later revision of this specification. In the meantime it is suggested to avoid using flow-level transition elements in light of this ambiguity.

On the other hand, a transition from a flow via the next attribute of the flow element is well-defined at the current spec level, and is suggested.

Example:

```
<flow id="Flow1" next="StepX">

<step id="FS1">

<next on="RC1" to="FS2A"/>

<next on="RC2" to="FS2B"/>

<!-- ILLEGAL - would be illegal, since one can only transition within
the flow

<next on="RC3" to="StepX"/>

-->

</step>

<step id="FS2A" >

<fail on="FAILED"/> <!-- FAILS job, doesn't "fail flow"-->

</step>

<step id="FS2B" >

<fail on="FAILED"/> <!-- FAILS job, doesn't "fail flow"-->

</step>

<next on="F*" to="StepY"/> <!-- UNDEFINED -->

</flow>
 <step id="StepX">
```

As noted in the comments inline, this example makes the following points:

- that a child of a flow can only transition to another child of the same flow (Item 1. in section 8.9.4)
- that a terminating transition terminates the job, not just the flow somehow (Item 2. in section 8.9.4)
- that a transition element which is a direct child of the flow itself is currently UNDEFINED (section 8.9.5)

Batch Programming Model

The batch programming model is described by interfaces, abstract classes, and field annotations. Interfaces define the essential contract between batch applications and the batch runtime. Most interfaces have a corresponding abstract class that provides default implementations of certain methods for developer convenience.

Steps

A batch step is either chunk or batchlet.

Chunk

A chunk type step performs item-oriented processing using a reader-processor-writer batch pattern and does checkpointing.

ItemReader Interface

An ItemReader is used to read items for a chunk step. ItemReader is one of the three batch artifact types that comprise a chunk type step. The other two are ItemProcessor and ItemWriter.

The ItemReader interface may be used to implement an ItemReader batch artifact:

ItemReader.java

```
package javax.batch.api.chunk;

import java.io.Serializable;

/*
 * ItemReader defines the batch artifact that reads
 * items for chunk processing.
 */
public interface ItemReader {
  /**
   * The open method prepares the reader to read items
   * The input parameter represents the last checkpoint
   * for this reader in a given job instance. The
   * checkpoint data is defined by this reader and is
   * provided by the checkpointInfo method. The checkpoint
   * data provides the reader whatever information it needs
   * to resume reading items upon restart. A checkpoint value
   * of null is passed upon initial start.
   *
   * @param checkpoint specifies the last checkpoint
   * @throws Exception is thrown for any errors.
```

```
    */
    public void open(Serializable checkpoint) throws Exception;

    /**
     * The close method marks the end of use of the
     * ItemReader. The reader is free to do any cleanup
     * necessary.
     * @throws Exception is thrown for any errors.
     */
    public void close() throws Exception;

    /**
     * The readItem method returns the next item
     * for chunk processing.
     * It returns null to in dicate no more items, which
     * also means the current chunk will be committed and
     * the step will end.
     * *@return* next item or null
     * @throws Exception is thrown for any errors.
     */
    public Object readItem() throws Exception;

    /**
     * The checkpointInfo method returns the current
     * checkpoint data for this reader. It is
     * called before a chunk checkpoint is committed.
     * *@return* checkpoint data
     * @throws Exception is thrown for any errors.
     */
    public Serializable checkpointInfo() throws Exception;
}
```

AbstractItemReader.java

```
package javax.batch.api.chunk;
import java.io.Serializable;
public abstract class AbstractItemReader implements ItemReader
{
    @Override
    public void open(Serializable checkpoint)throws Exception {
    }

    @Override
    public void close()throws Exception {
    }

    @Override
    public abstract Object readItem() throws Exception;

    @Override
    public Serializable checkpointInfo() throws Exception {
        return null;
    }
}
```

ItemProcessor Interface

An ItemProcessor is used to process items for a chunk step. ItemProcessor is one of the three batch artifact types that comprise a chunk type step. An ItemProcessor is an optional artifact on a chunk type step. The other two are ItemProcessor and ItemWriter.

The ItemProcessor interface may be used to implement an ItemProcessor batch artifact:

ItemProcessor.java

```
package javax.batch.api.chunk;
/**
* ItemProcessor is used in chunk processing
* to operate on an input item and produce
* an output item.
*
*/
public interface ItemProcessor {
    /**
    * The processItem method is part of a chunk
    * step. It accepts an input item from an
    * item reader and returns an item that gets
    * passed onto the item writer. Returning null
    * indicates that the item should not be continued
    * to be processed. This effectively enables processItem
    * to filter out unwanted input items.
    * @param item specifies the input item to process.
    * *@return* output item to write.
    * @throws Exception thrown for any errors.
    */
    public Object processItem(Object item) throws Exception;
}
```

ItemWriter Interface

An ItemWriter is used to write a list of output items for a chunk step. ItemWriter is one of the three batch artifact types that comprise a chunk type step. The other two are ItemProcessor and ItemReader.

The ItemWriter interface may be used to implement an ItemWriter batch artifact:

ItemWriter.java

```
package javax.batch.api.chunk;
import java.io.Serializable;
import java.util.List;
/**
*
* ItemWriter defines the batch artifact that writes to a
* list of items for chunk processing.
*
*/
public interface ItemWriter {
    /**
    * The open method prepares the writer to write items.
    *
```

```
    * The input parameter represents the last checkpoint
    * for this writer in a given job instance. The
    * checkpoint data is defined by this writer and is
    * provided by the checkpointInfo method. The checkpoint
    * data provides the writer whatever information it needs
    * to resume writing items upon restart. A checkpoint value
    * of null is passed upon initial start.
    *
    * @param checkpoint specifies the last checkpoint
    * @throws Exception is thrown for any errors.
    */
    public void open(Serializable checkpoint) throws Exception;
    /**
    * The close method marks the end of use of the
    * ItemWriter. The writer is free to do any cleanup
    * necessary.
    * @throws Exception is thrown for any errors.
    */
    public void close() throws Exception;
    /**
    * The writeItems method writes a list of item
    * for the current chunk.
    * @param items specifies the list of items to write.
    * This may be an empty list (e.g. if all the
    * items have been filtered out by the
    * ItemProcessor).
    * @throws Exception is thrown for any
    errors.
    */
    public void writeItems(List<Object> items) throws Exception;
    /**
    * The checkpointInfo method returns the current
    * checkpoint data for this writer. It is
    * called before a chunk checkpoint is committed.
    * *@return* checkpoint data
    * @throws Exception is thrown for any errors.
    */
    public Serializable checkpointInfo() throws Exception;
}
```

AbstractItemWriter.java

```
package javax.batch.api.chunk;
import java.io.Serializable;
import java.util.List;
/**
* The AbstractItemWriter provides default implementations
```

```
 * of not commonly implemented methods.
 */
public abstract class AbstractItemWriter implements ItemWriter
{
    /**
    * Override this method if the ItemWriter requires
    * any open time processing.
    * The default implementation does nothing.
    *
    * @param last checkpoint for this ItemReader
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void open(Serializable checkpoint) throws Exception {
    }
    /**
    * Override this method if the ItemWriter requires
    * any close time processing.
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void close() throws Exception {
    }
    /**
    * Implement write logic for the ItemWriter in this
    * method.
    *
    * @param items specifies the list of items to write.
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public abstract void writeItems(List<Object> items) throws
    Exception;
    /**
    * Override this method if the ItemWriter supports
    * checkpoints.
    * The default implementation returns null.
    *
    * *@return* checkpoint data
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public Serializable checkpointInfo() throws Exception {
        return null;
    }
}
```

CheckpointAlgorithm Interface

A CheckpointAlgorithm implements a custom checkpoint policy for a chunk step. The CheckpointAlgorithm interface may be used to implement an CheckpointAlgorithm batch artifact:

CheckpointAlgorithm.java

```
package javax.batch.api.chunk;
/**
* CheckpointAlgorithm provides a custom checkpoint
* policy for chunk steps.
*
*/
public interface CheckpointAlgorithm {
    /**
    * The checkpointTimeout is invoked at the beginning of a new
    * checkpoint interval for the purpose of establishing the checkpoint
    * timeout.
    * It is invoked before the next chunk transaction begins. This
    * method returns an integer value, which is the timeout value
    * (expressed in seconds) which will be used for the next chunk
    * transaction.
    * This method is useful to automate the setting of the
    * checkpoint timeout based on factors known outside the job
    * definition.
    * A value of '0' signifies no maximum established by this
    * CheckpointAlgorithm, i.e. the maximum permissible timeout allowed by
    * the runtime environment.
    * @return the timeout interval (expressed in seconds)
    * to use for the next checkpoint interval
    * @throws Exception thrown for any errors.
    */
    public int checkpointTimeout() throws Exception;
    /**
    * The beginCheckpoint method is invoked before the
    * next checkpoint interval begins (before the next
    * chunk transaction begins).
    * @throws Exception thrown for any errors.
    */
    public void beginCheckpoint() throws Exception;
    /**
    * The isReadyToCheckpoint method is invoked by
    * the batch runtime after each item is processed
    * to determine if now is the time to checkpoint
    * the current chunk.
    * *@return* boolean indicating whether or not
    * to checkpoint now.
    * @throws Exception thrown for any errors.
```

```
    */
    public boolean isReadyToCheckpoint() throws Exception;
    /**
    * The endCheckpoint method is invoked after the
    * last checkpoint is taken (after the chunk
    * transaction is committed).
    * @throws Exception thrown for any errors.
    */
    public void endCheckpoint() throws Exception;
}
```

AbstractCheckpointAlgorithm.java

```
package javax.batch.api.chunk;
/**
* The AbstractCheckpointAlgorithm provides default
* implementations of less commonly implemented
* methods.
*/
public abstract class AbstractCheckpointAlgorithm implements
CheckpointAlgorithm {
    /**
    * Override this method if the CheckpointAlgorithm
    * establishes a checkpoint timeout.
    * The default implementation returns 0, which means
    * the maximum permissible timeout allowed by the
    * runtime environment.
    *
    * @return the timeout interval (expressed in seconds)
    * to use for the next checkpoint interval
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public int checkpointTimeout() throws Exception {
        return 0;
    }
    /**
    * Override this method for the CheckpointAlgorithm
    * to do something before a checkpoint interval
    * begins (before the next chunk transaction begins).
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void beginCheckpoint() throws Exception {
    }
```

```
    /**
    * Implement logic in this method
    * to decide if a checkpoint should be taken now.
    *
    * *@return* boolean indicating whether or not
    * to checkpoint now.
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public abstract boolean isReadyToCheckpoint() throws Exception;
    /**
    * Override this method for the CheckpointAlgorithm
    * to do something after a checkpoint is taken (after
    * the chunk transaction is committed).
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void endCheckpoint() throws Exception {
    }
}
```

Batchlet Interface

A Batchlet-type step implements a roll your own batch pattern. This batch pattern is invoked once, runs to completion, and returns an exit status.

The Batchlet interface may be used to implement a Batchlet batch artifact:

Batchlet.java

```
package javax.batch.api;
/**
*
* A batchlet is type of batch step
* that can be used for any type of
* background processing that does not
* explicitly call for a chunk oriented
* approach.
* <p>
* A well-behaved batchlet responds
* to stop requests by implementing
* the stop method.
*
*/
public interface Batchlet {
    /**
    * The process method does the work
    * of the batchlet. If this method
    * throws an exception, the batchlet
    * step ends with a batch status of
    * FAILED.
    * *@return* exit status string
    * @throws Exception if an error occurs.
    */
    public String process() throws Exception;
    /**
    * The stop method is invoked by the batch
    * runtime as part of JobOperator.stop()
    * method processing. This method is invoked
    * on a thread other than the thread on which
    * the batchlet process method is running.
    *
    * @throws Exception if an error occurs.
    */
    public void stop() throws Exception;
}
```

AbstractBatchlet.java

```
package javax.batch.api;
/**
* The AbstractBatchlet provides default
* implementations of less commonly implemented methods.
*/
public abstract class AbstractBatchlet implements Batchlet {
    /**
    * Implement process logic for the Batchlet in this
    * method.
    *
    * *@return* exit status string
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public abstract String process() throws Exception;
    /**
    * Override this method if the Batchlet will
    * end in response to the JobOperator.stop()
    * operation.
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void stop() throws Exception {
    }
}
```

[image] | *images/image010.png*

Listeners

Use Listeners to interpose on batch execution.

JobListener Interface

A job listener receives control before and after a job execution runs, and also if an exception is thrown during job processing. The JobListener interface may be used to implement an JobListener batch artifact:

JobListener.java

```
package javax.batch.api.listener;
/**
* JobListener intercepts job execution.
*
*/
public interface JobListener {
    /**
    * The beforeJob method receives control
    * before the job execution begins.
    * @throws Exception throw if an error occurs.
    */
    public void beforeJob() throws Exception;
    /**
    * The afterJob method receives control
    * after the job execution ends.
    * @throws Exception throw if an error occurs.
    */
    public void afterJob() throws Exception;
}
```

AbstractJobListener.java

```
package javax.batch.api.listener;
/**
* The AbstractJobListener provides default
* implementations of less commonly implemented methods.
*/
public abstract class AbstractJobListener implements JobListener
{
    /**
    * Override this method if the JobListener
    * will do something before the job begins.
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void beforeJob() throws Exception {
    }
    /**
    * Override this method if the JobListener
    * will do something after the job ends.
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void afterJob() throws Exception {
    }
}
```

StepListener Interface

A step listener can receive control before and after a step runs, and also if an exception is thrown during step processing. The StepListener interface may be used to implement an StepListener batch artifact:

StepListener.java

```
package javax.batch.api.listener;
/**
* StepListener intercepts step execution.
*
*/
public interface StepListener {
    /**
    * The beforeStep method receives control
    * before a step execution begins.
    * @throws Exception throw if an error occurs.
    */
    public void beforeStep() throws Exception;
    /**
    * The afterStep method receives control
    * after a step execution ends.
    * @throws Exception throw if an error occurs.
    */
    public void afterStep() throws Exception;
}
```

AbstractStepListener.java

```
package javax.batch.api.listener;
/**
* The AbstractStepListener provides default
* implementations of less commonly implemented methods.
*/
public abstract class AbstractStepListener implements
StepListener {
    /**
    * Override this method if the StepListener
    * will do something before the step begins.
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void beforeStep() throws Exception {
    }
    /**
    * Override this method if the StepListener
    * will do something after the step ends.
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void afterStep() throws Exception {
    }
}
```

ChunkListener Interface

A chunk listener can receive control at the beginning and the end of chunk, and upon an exception thrown back to the runtime implementation. The ChunkListener interface may be used to implement a ChunkListener batch artifact:

ChunkListener.java

```
package javax.batch.api.chunk.listener;
/**
* ChunkListener intercepts chunk processing.
*
*/
public interface ChunkListener {
    /**
    * The beforeChunk method receives control
    * before processing of the next
    * chunk begins. This method is invoked
    * in the same transaction as the chunk
    * processing.
    * @throws Exception throw if an error occurs.
    */
    public void beforeChunk() throws Exception;
    /**
    * The onError method receives control
    * before the chunk transaction is rolled back.
    * Note afterChunk is not invoked in this case.
    * @param ex specifies the exception that
    * caused the roll back.
    * @throws Exception throw if an error occurs.
    */
    public void onError(Exception ex) throws Exception;
    /**
    * The afterChunk method receives control
    * after processing of the current
    * chunk ends. This method is invoked
    * in the same transaction as the chunk
    * processing.
    * @throws Exception throw if an error occurs.
    */
    public void afterChunk() throws Exception;
}
```

AbstractChunkListener.java

```
package javax.batch.api.chunk.listener;
/**
* The AbstractChunkListener provides default
* implementations of less commonly implemented methods.
*/
public abstract class AbstractChunkListener implements
ChunkListener {
    /**
    * Override this method if the ChunkListener
    * will do something before the chunk begins.
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void beforeChunk() throws Exception {
    }
    /**
    * Override this method if the ChunkListener will do
    * something before the chunk transaction is rolled back.
    * Note afterChunk is not invoked in this case.
    * @param ex specifies the exception that
    * caused the roll back.
    * @throws Exception (or subclass) throw if an error occurs.
    */
    @Override
    public void onError(Exception ex) throws Exception {
    }
    /**
    * Override this method if the ChunkListener
    * will do something after the chunk ends.
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void afterChunk() throws Exception {
    }
}
```

ItemReadListener Interface

An item read listener can receive control before and after an item is read by an item reader, and also if the reader throws an exception. The ItemReadListener interface may be used to implement an ItemReadListener batch artifact:

ItemReadListener.java

```
package javax.batch.api.chunk.listener;
/**
* ItemReadListener intercepts item reader
* processing.
*
*/
public interface ItemReadListener {
    /**
    * The beforeRead method receives control
    * before an item reader is called to read the next item.
    * @throws Exception is thrown if an error occurs.
    */
    public void beforeRead() throws Exception;
    /**
    * The afterRead method receives control after an item
    * reader reads an item. The method receives the item read as
    * an input.
    * @param item specifies the item read by the item reader.
    * @throws Exception is thrown if an error occurs.
    */
    public void afterRead(Object item) throws Exception;
    /**
    * The onReadError method receives control after an item reader
    * throws an exception in the readItem method.
    * This method receives the exception as an input.
    * @param ex specifies the exception that occurred in the item reader.
    * @throws Exception is thrown if an error occurs.
    */
    public void onReadError(Exception ex) throws Exception;
}
```

AbstractItemReadListener.java

```
package javax.batch.api.chunk.listener;
/**
* The AbstractItemReadListener provides default
* implementations of less commonly implemented methods.
*/
public abstract class AbstractItemReadListener implements
ItemReadListener {
    /**
    * Override this method if the ItemReadListener
    * will do something before the item is read.
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void beforeRead() throws Exception {
    }
    /**
    * Override this method if the ItemReadListener
    * will do something after the item is read.
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void afterRead(Object item) throws Exception {
    }
    /**
    * Override this method if the ItemReadListener
    * will do something when the ItemReader readItem
    * method throws an exception.
    * The default implementation does nothing.
    *
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void onReadError(Exception ex) throws Exception {
    }
}
```

ItemProcessListener Interface

An item processor listener can receive control before and after an item is processed by an item processor, and also if the processor throws an exception. The ItemProcessListener interface may be used to implement an ItemProcessListener batch artifact:

ItemProcessListener.java

```
package javax.batch.api.chunk.listener;
/**
* ItemProcessListener intercepts item processing.
*
*/
public interface ItemProcessListener {
    /**
    * The beforeProcess method receives control before
    * an item processor is called to process the next item.
    * The method receives the item to be processed as an input.
    * @param item specifies the item about to be processed.
    * @throws Exception if an error occurs.
    */
    public void beforeProcess(Object item) throws Exception;
    /**
    * The afterProcess method receives control after an item
    * processor processes an item. The method receives the item processed
    * and the result item as an input.
    * @param item specifies the item processed by the item processor.
    * @param result specifies the item to pass to the item writer.
    * @throws Exception if an error occurs.
    */
    public void afterProcess(Object item, Object result) throws
    Exception;
    /**
    * The onProcessError method receives control after an
    * item processor processItem throws an exception. The method
    * receives the item sent to the item processor as input.
    * @param item specifies the item the processor attempted to process.
    * @param ex specifies the exception thrown by the item processor.
    * @throws Exception if an error occurs
    */
    public void onProcessError(Object item, Exception ex) throws
    Exception;
}
```

AbstractItemProcessListener.java

```
package javax.batch.api.chunk.listener;
/**
* The AbstractItemProcessListener provides default
* implementations of less commonly implemented methods.
*
*/
public abstract class AbstractItemProcessListener implements
```

```
ItemProcessListener {
    /**
    * Override this method if the ItemProcessListener
    * will do something before the item is processed.
    * The default implementation does nothing.
    *
    * @param item specifies the item about to be processed.
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void beforeProcess(Object item) throws Exception {
    }
    /**
    * Override this method if the ItemProcessListener
    * will do something after the item is processed.
    * The default implementation does nothing.
    *
    * @param item specifies the item about to be processed.
    * @param result specifies the item to pass to the item writer.
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void afterProcess(Object item, Object result) throws
    Exception {
    }
    /**
    * Override this method if the ItemProcessListener
    * will do something when the ItemProcessor processItem
    * method throws an exception.
    * The default implementation does nothing.
    *
    * @param item specifies the item about to be processed.
    * @param ex specifies the exception thrown by the item processor.
    * @throws Exception (or subclass) if an error occurs.
    */
    @Override
    public void onProcessError(Object item, Exception ex) throws
    Exception {
    }
}
```

ItemWriteListener Interface

A item write listener can receive control before and after an item is written by an item writer, and also if the writer throws an exception. The ItemWriteListener interface may be used to implement an ItemWriteListener batch artifact:

ItemWriteListener.java

```
package javax.batch.api.chunk.listener;
import java.util.List;
/**
* ItemWriteListener intercepts item writer
* processing.
*
*/
public interface ItemWriteListener {
    /**
    * The beforeWrite method receives control before
    * an item writer is called to write its items. The
    * method receives the list of items sent to the item
    * writer as an input.
    * @param items specifies the items about to be
    * written.
    * @throws Exception is thrown if an error occurs.
    */
    public void beforeWrite(List<Object> items) throws Exception;
    /**
    * The afterWrite method receives control after an
    * item writer writes its items. The method receives the
    * list of items sent to the item writer as an input.
    * @param items specifies the items written by the item writer.
    * @throws Exception is thrown if an error occurs.
    */
    public void afterWrite(List<Object> items) throws Exception;
    /**
    * The onWriteError method receives control after an
    * item writer writeItems throws an exception. The method
    * receives the list of items sent to the item writer as input.
    * @param items specifies the items which the item writer
    * attempted to write.
    * @param ex specifies the exception thrown by the item
    * writer.
    * @throws Exception is thrown if an error occurs.
    */
    public void onWriteError(List<Object> items, Exception ex) throws
    Exception;
}
```

Skip Listener Interfaces

A skip listener can receive control when a skippable exception is thrown from an item reader, processor, or writer. Three interfaces are provided to implement these listeners:

SkipReadListener.java

```
package javax.batch.api.chunk.listener;
/**
* SkipReadListener intercepts skippable
* itemReader exception handling.
*/
public interface SkipReadListener {
    /**
    * The onSkipReadItem method receives control
    * when a skippable exception is thrown from an
    * ItemReader readItem method. This method receives the
    * exception as an input.
    * @param ex specifies the exception thrown by the ItemReader.
    * @throws Exception is thrown if an error occurs.
    */
    public void onSkipReadItem(Exception ex) throws Exception;
}
```

SkipProcessListener.java

```
package javax.batch.api.chunk.listener;
/**
* SkipProcessListener intercepts skippable
* itemProcess exception handling.
*/
public interface SkipProcessListener {
    /**
    * The onSkipProcessItem method receives control when
    * a skippable exception is thrown from an ItemProcess
    * processItem method.
    * This method receives the exception and the item to process
    * as an input.
    * @param item specifies the item passed to the ItemProcessor.
    * @param ex specifies the exception thrown by the
    * ItemProcessor.
    * @throws Exception is thrown if an error occurs.
    */
    public void onSkipProcessItem(Object item, Exception ex) throws
    Exception;
}
```

SkipWriteListener.java

```
package javax.batch.api.chunk.listener;
import java.util.List;
/**
* SkipWriteListener intercepts skippable
* itemWriter exception handling.
*/
public interface SkipWriteListener {
    /**
    * The onSkipWriteItems method receives control when a
    * skippable exception is thrown from an ItemWriter
    * writeItems method. This
    * method receives the exception and the items that were
    * skipped as an input.
    * @param items specifies the list of item passed to the
    * item writer.
    * @param ex specifies the exception thrown by the
    * ItemWriter.
    * @throws Exception is thrown if an error occurs.
    */
    public void onSkipWriteItem(List<Object> items, Exception ex)
    throws Exception;
}
```

RetryListener Interface

A retry listener can receive control when a retryable exception is thrown from an item reader, processor, or writer. Three interfaces are provided to implement these listeners:

RetryReadListener.java

```
package javax.batch.api.chunk.listener;
/**
* RetryReadListener intercepts retry processing for
* an ItemReader.
*/
public interface RetryReadListener {
    /**
    * The onRetryReadException method receives control
    * when a retryable exception is thrown from an ItemReader
    * readItem method.
    * This method receives the exception as input. This method
    * receives control in the same checkpoint scope as the
    * ItemReader. If this method throws a an exception, the job
    * ends in the FAILED state.
    * @param ex specifies the exception thrown by the item
    * reader.
    * @throws Exception is thrown if an error occurs.
    */
    public void onRetryReadException(Exception ex) throws Exception;
}
```

RetryProcessListener.java

```
package javax.batch.api.chunk.listener;
/**
* RetryProcessListener intercepts retry processing for
* an ItemProcessor.
*
*/
public interface RetryProcessListener {
    /**
    * The onRetryProcessException method receives control
    * when a retryable exception is thrown from an ItemProcessor
    * processItem method. This method receives the exception and the item
    * being processed as inputs. This method receives control in same
    * checkpoint scope as the ItemProcessor. If this method
    * throws a an exception, the job ends in the FAILED state.
    * @param item specifies the item passed to the ItemProcessor.
    * @param ex specifies the exception thrown by the ItemProcessor.
    * @throws Exception is thrown if an error occurs.
    */
    public void onRetryProcessException(Object item, Exception ex)
    throws Exception;
}
```

RetryWriteListener.java

```
package javax.batch.api.chunk.listener;
import java.util.List;
/**
* RetryWriteListener intercepts retry processing for
* an ItemWriter.
*
*/
public interface RetryWriteListener {
    /**
    * The onRetryWriteException method receives control when a
    * retryable exception is thrown from an ItemWriter writeItems
    * method. This method receives the exception and the list of items
    * being written as inputs.
    * This method receives control in same checkpoint scope as the
    * ItemWriter. If this method throws a an exception, the job ends
    * in the FAILED state.
    * @param items specify the items passed to an item writer.
    * @param ex specifies the exception thrown by an item
    * writer.
    * @throws Exception is thrown if an error occurs.
    */
    public void onRetryWriteException(List<Object> items, Exception ex)
    throws Exception;
}
```

Batch Properties

Batch applications need a way to receive parameters when a job is initiated for execution. Properties can be defined by batch programming model artifacts, then have values passed to them when a job is initiated. Batch properties are string values.

Note batch properties are visible only in the scope in which they are defined (see Section 9.3.2). However batch properties values can be formed from other properties according to Job XML Substitution Rules. See section 8.8 for further information on substitution.

@BatchProperty

The @BatchProperty annotation identifies a class field injection as a batch property. A batch property has a name (name) and default value. The @BatchProperty may be used on a class field for any class identified as a batch programming model artifact -E.g. ItemReader, ItemProcessor, JobListener, etc..

@BatchProperty must be used with the standard @Inject annotation (javax.inject.Inject). @BatchProperty is used to assign batch artifact property values from Job XML to the batch artifact itself.

A field annotated with the @BatchProperty annotation must not be static and must not be final.

```
 Note: the batch runtime must ensure @Inject works with @BatchProperty,
whether or not the execution environment includes an implementation of
JSR 299 or 330. This means the batch properties may always be injected.
Whether or not other injections are supported is dependent upon the
batch runtime implementation.
```

Syntax:

```
package: javax.batch.api

@Inject @BatchProperty(name="<property-name>") String <field-name>;
```

Where:

<property-name>	is the optional name of this batch property. The default is the Java field name.
<field-name>	is the field name of the batch property.

BatchProperty.java

```
package javax.batch.api;
import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;
import javax.enterprise.util.Nonbinding;
import javax.inject.Qualifier;
/**
* Annotation used by batch artifacts to declare a
* field which is injectable via a JSL-defined value
* (possibly leveraging Job XML substitutions).
*
*/
@Qualifier
@Target({
    ElementType._FIELD_, ElementType._METHOD_,
    ElementType._PARAMETER_
}
)
@Retention(RetentionPolicy._RUNTIME_)
public @interface BatchProperty {
    @Nonbinding
    public String name() default "";
}
```

The value of the annotated field is assigned by the batch runtime if a corresponding property element with a matching name is specified in the JSL in the scope that applies to the batch artifact in question. If the JSL property value resolves to the empty string (either explicitly set to the empty string literal or resolving to an empty string via property substitution – see section 8.8), no assignment is made and the resulting value is undefined by the batch specification. The resulting value might simply be the Java default value, however using various dependency injection technologies may produce different results. The resultant behavior may be defined by the particular dependency injection technology used in the runtime environment and so is outside the scope of this specification. Example:

```
 import javax.inject.Inject;
 import javax.batch.api.BatchProperty;
 public class MyItemReaderImpl \{

@Inject @BatchProperty String fname;

}
```

Behavior:

When the batch runtime instantiates the batch artifact (item reader in this example), it assigns the

value of the property with name equal to fname provided in the job XML to the corresponding @BatchProperty field named fname. If no value is defined in JSL, the Java default (null) is assigned or some other default is provided by a particular dependency injection technology.

Scope of property definitions for @BatchProperty

injection

The rules governing the definition of properties for injection via @BatchProperty deserve some extra explanation and an example.

For a given artifact, the only properties that are injectable via @BatchProperty are those which are defined at the level of the artifact itself (i.e. as children of the "properties" element which is in turn a child of the very element defining the artifact: batchlet, reader, listener, etc.).

In particular, just because an artifact definition is contained (at some level of nesting) within a job element and (for most artifacts) within a step element as well, it is NOT the case that the job properties and step properties are themselves injectable into that artifact via @BatchProperty. This is the case even though these job and step properties are available for resolving the artifact-level property definitions via the jobProperties substitution mechanism (see section 8.8.1.2) .

The following example should make this more clear:

Example JSL

```
<job>

<properties>

<property name="x" value="xVal"/>

...

<step id="step1">

<batchlet ref="MyBatchlet">

<properties>

<property name="y" value="#\{jobProperties['x']}"/>

</properties>
</job>
```

Example Java (MyBatchlet from JSL above):

```
// WONT WORK! - There is no property 'x' in scope for this injection
@Inject @BatchProperty(name="x");

// WILL WORK – Gets value 'xVal'
@Inject @BatchProperty(name="y");
```

Batch Contexts

Context objects are supplied by the batch runtime and provide important functions to a batch application. Contexts provide information about the running batch job, provide a place for a batch job to store interim values, and provide a way for the batch application to communicate important information back to the batch runtime. Contexts can be injected into an application as member variables. There is a context for both job and step. The job context represents the entire job. The step context represents the current step executing within the job.

Batch Contexts

Batch artifact access to batch contexts is by injection using the standard @Inject annotation (javax.inject.Inject). A field into which a batch context is injected must not be static and must not be final. E.g.:

```
@Inject JobContext _jctxt;
```

```
@Inject StepContext _sctxt;
```

The batch runtime is responsible to ensure the correct context object is injected according to the job or step currently executing.

Note: the batch runtime must ensure @Inject works with JobContext and StepContext, whether or not the execution environment includes an implementation of JSR 299 or 330. This means the batch contexts may always be injected. Whether or not other injections are supported is dependent upon the batch runtime implementation. See section 10.9.1 for definition of JobContext class. See section 10.9.2 for definition of StepContext class.

Batch Context Lifecycle and Scope

A batch context has thread affinity and is visible only to the batch artifacts executing on that particular thread. A batch context injected field may be null when out of scope. Each context type has a distinct scope and lifecycle as follows:

1. JobContext

There is one JobContext per job execution. It exists for the life of a job. There is a distinct JobContext for each sub-thread of a parallel execution (e.g. partitioned step).

2. StepContext+

   There is one StepContext per step execution. It exists for the life of the step. For a partitioned step, there is one StepContext for the parent step/thread; there is a distinct StepContext for each sub-thread and each StepContext has its own distinct persistent user data for each sub-thread.

Parallelization

Batch jobs may be configured to run some of their steps in parallel. There are two supported parallelization models:

Partitioned:

In the partitioned model, a step is configured to run as multiple instances across multiple threads. Each thread runs the same step or flow. This model is logically equivalent to launching multiple instances of the same step. It is intended that each partition processes a different range of the input items.

The partitioned model includes several optional batch artifacts to enable finer control over parallel processing:

1. PartitionMapper provides a programmatic means for calculating the number of partitions and unique properties for each.
2. PartitionReducer provides a unit of work demarcation around partition processing.
3. PartitionCollector provides a means for merging interrim results from individual partitions.
4. PartitionAnalyzer provides a means to gather interrim and final results from individual partitions for single point of control processing and decision making.

Concurrent:

In the concurrent model, the flows defined by a split are configured to run concurrently on multiple threads, one flow per thread.

PartitionMapper Interface

A partition mapper receives control at the start of a partitioned execution. The partition mapper is responsible to provide unique batch properties for each partition. The PartitionMapper interface may be used to implement a PartitionMapper batch artifact:

PartitionMapper.java

```
package javax.batch.api.partition;
import javax.batch.api.partition.PartitionPlan;
/**
* PartitionMapper receives control at the start of a partitioned
* execution. A PartitionMapper is responsible to provide unique
* batch properties for each partition.
*
*/
public interface PartitionMapper {
    /**
    * The mapPartitions method that receives control at the
    * start of partitioned step processing. The method
    * returns a PartitionPlan, which specifies the batch properties
    * for each partition.
    * *@return* partition plan for a partitioned step.
    * @throws Exception is thrown if an error occurs.
    */
    public PartitionPlan mapPartitions( ) throws Exception;
}
```

See section 10.9.4 for details on the PartitionPlan result value type.

The PartitionMapper, when defined, is invoked upon every execution, including restarted executions. For a full discussion of the behavior on restart, including how to override particular details of the PartitionPlan built by the previous execution, see section 10.8.5.

PartitionReducer Interface

A partition reducer provides a unit of work demarcation across partitions. It is not a JTA transaction; no resources are enlisted. Rather, it provides transactional flow semantics to facilitate finalizing merge or compensation logic. The PartitionReducer interface may be used to implement an PartitionReducer batch artifact:

PartitionReducer.java

```
package javax.batch.api.partition;
/**
* PartitionReducer provides unit of work demarcation across
* partitions. It is not a JTA transaction;
no resources are
* enlisted. Rather, it provides transactional flow semantics
* to facilitate finalizing merge or compensation logic.
*
*/
public interface PartitionReducer {
```

```
    public *enum* PartitionStatus {
        COMMIT_, _ROLLBACK
    }
    /**
    * The beginPartitionedStep method receives
    * control at the start of partition processing.
    * It receives control before the PartitionMapper
    * is invoked and before any partitions are started.
    * @throws Exception is thrown if an error occurs.
    */
    public void beginPartitionedStep() throws Exception;
    /**
    * The beforePartitionedStepCompletion method
    * receives control at the end of partitioned
    * step processing. It receives control after all
    * partitions have completed. It does not receive
    * control if the PartitionReducer is rolling back.
    * @throws Exception is thrown if an error occurs.
    */
    public void beforePartitionedStepCompletion() throws Exception;
    /**
    * The rollbackPartitionedStep method receives
    * control if the runtime is rolling back a partitioned
    * step. Any partition threads still running are
    * allowed to complete before this method is invoked. This method
    * receives control if any of the following conditions
    * are true:
    * <p>
    * <ol>
    * <li>One or more partitions end with a Batch Status of
    * STOPPED or FAILED.</li>
    * <li>Any of the following partitioned step callbacks
    * throw an exception:</li>
    * <ol>
    * <li>PartitionMapper</li>
    * <li>PartitionReducer</li>
    * <li>PartitionCollector</li>
    * <li>PartitionAnalyzer</li>
    * </ol>
    * <li>A job with partitioned steps is restarted.</li>
    * </ol>
    * @throws Exception is thrown if an error occurs.
    */
    public void rollbackPartitionedStep() throws Exception;
    /**
    * The afterPartitionedStepCompletion method receives control
    * at the end of a partition processing. It receives a status
    * value that identifies the outcome of the partition processing.
```

```
    * The status string value is either "COMMIT" or "ROLLBACK".
    * @param status specifies the outcome of the partitioned step. Values
    * are "COMMIT" or "ROLLBACK".
    * @throws Exception is thrown if an error occurs.
    */
    public void afterPartitionedStepCompletion(PartitionStatus status)
    throws Exception;
}
```

AbstractPartitionReducer.java

```
package javax.batch.api.partition;
/**
* The AbstractPartitionReducer provides default
* implementations of less commonly implemented methods.
*/
public abstract class AbstractPartitionReducer implements
PartitionReducer {
    /**
    * Override this method to take action before
    * partitioned step processing begins.
    *
    * @throws Exception is thrown if an error occurs.
    */
    @Override
    public void beginPartitionedStep() throws Exception {
    }
    /**
    * Override this method to take action before
    * normal partitioned step processing ends.
    *
    * @throws Exception is thrown if an error occurs.
    */
    @Override
    public void beforePartitionedStepCompletion() throws Exception {
    }
    /**
    * Override this method to take action when a
    * partitioned step is rolling back.
    *
    * @throws Exception is thrown if an error occurs.
    */
    @Override
    public void rollbackPartitionedStep() throws Exception {
    }
    /**
    * Override this method to take action after
```

```
     * partitioned step processing ends.
     *
     * @param status specifies the outcome of the partitioned step.
     * Values are "COMMIT" or "ROLLBACK".
     * @throws Exception is thrown if an error occurs.
     */
     @Override
     public void afterPartitionedStepCompletion(PartitionStatus status)
     throws Exception {
     }
 }
```

PartitionCollector Interface

A partition collector provides a way to send data from individual partitions to a single point of control running on the parent thread. The PartitionAnalyzer is used to receive and process this data. See section 9.5.4 for further information about the PartitionAnalyzer. The PartitionCollector interface may be used to implement an PartitionCollector batch artifact:

PartitionCollector.java

```
package javax.batch.api.partition;
import java.io.Serializable;
/**
* PartitionCollector provides a way to pass data from
* individual partitions to a single point of control running on
* the step's parent thread. The PartitionAnalyzer is used to
* receive and process this data.
*
*/
public interface PartitionCollector {
    /**
    * The collectPartitionData method receives control
    * periodically during partition processing.
    * This method receives control on each thread processing
    * a partition as follows:
    * <p>
    * <ol>
    * <li>for a chunk type step, it receives control after
    * every chunk checkpoint and then one last time at the
    * end of the partition;
    </li>
    * <li>for a batchlet type step, it receives control once
    * at the end of the batchlet.</li>
    * </ol>
    * <p>
    * Note the collector is not called if the partition
    * terminates due to an unhandled exception.
    * <p>
    * *@return* an Serializable object to pass to the
    * PartitionAnalyzer.
    * @throws Exception is thrown if an error occurs.
    */
    public Serializable collectPartitionData() throws Exception;
}
```

PartitionAnalyzer Interface

A partition analyzer receives control to process data and final results from partitions. If a partition collector is configured on the step, the partition analyzer receives control to process the data and results from the partition collector. While a separate partition collector instance is invoked on each thread processing a partition, the partition analyzer runs on a single, consistent thread each time it is invoked. The PartitionAnalyzer interface may be used to implement an PartitionAnalyzer batch artifact:

PartitionCollector.java

```
package javax.batch.api.partition;
import java.io.Serializable;
import javax.batch.runtime.BatchStatus;
/**
* PartitionAnalyzer receives control to process
* data and final results from each partition. If
* a PartitionCollector is configured on the step,
* the PartitionAnalyzer receives control to process
* the data and results from the partition collector.
* While a separate PartitionCollector instance is
* invoked on each thread processing a step partition,
* a single PartitionAnalyzer instance runs on a single,
* consistent thread each time it is invoked.
*
*/
public interface PartitionAnalyzer {
    /**
    * The analyzeCollectorData method receives
    * control each time a Partition collector sends
    * its payload. It receives the
    * Serializable object from the collector as an
    * input.
    * @param data specifies the payload sent by a
    * PartitionCollector.
    * @throws Exception is thrown if an error occurs.
    */
    public void analyzeCollectorData(Serializable data) throws
    Exception;
    /**
    * The analyzeStatus method receives control each time a
    * partition ends. It receives the batch and exit
    * status strings of the partition as inputs.
    * @param batchStatus specifies the batch status of a partition.
    * @param exitStatus specifies the exit status of a partition.
    * @throws Exception is thrown if an error occurs.
    */
    public void analyzeStatus(BatchStatus batchStatus, String
    exitStatus) throws Exception;
}
```

AbstractPartitionAnalyzer.java

```
package javax.batch.api.partition;
import java.io.Serializable;
import javax.batch.runtime.BatchStatus;
/**
* The AbstractPartitionAnalyzer provides default
* implementations of less commonly implemented methods.
*/
public abstract class AbstractPartitionAnalyzer implements
PartitionAnalyzer {
    /**
    * Override this method to analyze PartitionCollector payloads.
    *
    * @param data specifies the payload sent by the
    * PartitionCollector.
    * @throws Exception is thrown if an error occurs.
    */
    @Override
    public void analyzeCollectorData(Serializable data) throws
    Exception {
    }
    /**
    * Override this method to analyze partition end status.
    * @param batchStatus specifies the batch status of a partition.
    * @param exitStatus specifies the exit status of a partition.
    * @throws Exception is thrown if an error occurs.
    */
    @Override
    public void analyzeStatus(BatchStatus batchStatus, String
    exitStatus)
    throws Exception {
    }
}
```

Decider Interface

A decider may be used to determine batch exit status and sequencing between steps, splits, and flows in a Job XML. The decider returns a String value which becomes the exit status value on which the decision chooses the next transition. The Decider interface may be used to implement an Decider batch artifact:

Decider.java

```
package javax.batch.api;
import javax.batch.runtime.StepExecution;
/**
* A Decider receives control as part of a decision element
* in a job. It is used to direct execution flow during job
* processing. It returns an exit status that updates the
* current job execution's exit status. This exit status
* value also directs the execution transition based on
* next, end, stop, fail child elements configured on the
* same decision element as the decider.
*/
public interface Decider {
    /**
    * The decide method sets a new exit status for a job.
    * It receives an array of StepExecution objects as input.
    * These StepExecution objects represent the execution
    * element that transitions to this decider as follows:
    * <p>
    * <ul>
    * <li>Step</li>
    * <p>
    * When the transition is from a step, the decide method
    * receives the StepExecution corresponding
    * to the step as input.
    * <li>Split</li>
    * <p>
    * When the transition is from a split, the decide method
    * receives a StepExecution from each flow defined to the split
    * as input.
    * <li>Flow</li>
    * <p>
    * When the transition is from a flow, the decide method
    * receives a StepExecution corresponding
    * to the last execution element that completed in the flow.
    * This will be a single StepExecution if the last element
    * was a step and multiple StepExecutions if the last element
    * was a split.
    * </ul>
    * @param executions specifies the StepExecution(s) of the preceding
    * element.
    * *@return* updated job exit status
    * @throws Exception is thrown if an error occurs.
    */
    public String decide(StepExecution[] executions) throws Exception;
}
```

Transactionality

Chunk type check points are transactional. The batch runtime uses global transaction mode on the Java EE platform and local transaction mode on the Java SE platform. Global transaction timeout is configurable at step-level with a step-level property:

'javax.transaction.global.timeout={seconds} - default is 180 seconds'

Example:

```
<step id="MyGlobalStep">
<properties>
<property name="javax.transaction.global.timeout" value="600"/>
</properties>
</step>
```

Batch Runtime Specification

Batch Properties Reserved Namespace

The batch runtime supports properties at job, step, partition, and artifact level. The property name prefix, 'javax.batch', is reserved for use by the batch runtime, as prescribed by this specification and future revisions of same. Applications and specification implementations must not define properties for their own use that begin with this prefix. Applications that do so risk undefined behavior.

Job Metrics

The batch runtime supports the following chunk-type step metrics:

1. readCount - the number of items successfully read.
2. writeCount - the number of items successfully written.
3. filterCount - the number of items filtered by ItemProcessor.
4. commitCount - the number of transactions committed.
5. rollbackCount - the number of transactions rolled back.
6. readSkipCount - the number of skippable exceptions thrown by the ItemReader.
7. processSkipCount - the number of skippable exceptions thrown by the ItemProcessor.
8. writeSkipCount - the number of skippable exceptions thrown by the ItemWriter.

These metrics are available through the StepExecution runtime object. See section 10.9.10 for further information on StepExecution.

Job Runtime Identifiers

Job runtime artifacts are uniquely defined at runtime with the following operational identifiers:

instanceId	Is a long that represents an instance of a job. A new job instance is created everytime a job is started with the JobOperator "start" method.
executionId	Is a long that represents the next attempt to run a particular job instance. A new execution is created the first time a job is started and everytime thereafter when an existing job execution is restarted with the JobOperator "restart" method. Note there can be no more than one executionId in the STARTED state at one time for a given job instance.

stepExecutionId	Is a long that represents the attempt to execute a particular step within a job execution.

Note instanceId, executionId, and stepExecutionId are all globally unique values within a job repository. See section 7.4 for explanation of job repository.

JobOperator

The JobOperator interface provides a set of operations to start, stop, restart, and inspect jobs. See 10.9.6 for detailed description of this interface. The JobOperator interface is accessible via a factory pattern:

```
JobOperator jobOper = BatchRuntime.getJobOperator();
```

See section 10.9.5 for details on the BatchRuntime class. === Batch Artifact Loading

All batch artifacts comprising a batch application are loadable by the following loaders in the order specified:

1. Implementation-specific loader

   The batch runtime implementation _may_provide an implementation-specific means by which batch artifacts references in a Job XML (i.e. via the 'ref=' attribute) are resolved to an implementation class and instantiated. When the batch runtime resolves a batch artifact reference to an instance the implementation-specific mechanism (if one exists) is attempted first. The loader must return an instance or null.

   An example of an implementation-specific loader might be CDI or Spring DI.

2. Archive loader

   If an implementation-specific mechanism does not exist or fails toresolve a batch artifact reference (returns null), then the batch runtime implementation must resolve the reference with an archive loader. The implementation must provide an archive loader that resolves the reference by looking up the reference in a `batch.xml` file, which maps reference name to implementation class name. The loader must return an instance or null.

   The `batch.xml` file is packaged by the developer with the application under the ‘META-INF’ directory (‘WEB-INF/classes/META-INF’ for .war files).

   See 10.7.1 for more about the `batch.xml` file.

3. Thread Context Class Loader

   If the archive loader fails to resolve a batch artifact reference (returns null), then the batch runtime implementation must resolve the reference by treating the reference as a class name and loading it through the thread context class loader. The loader must return an instance or null.

Job XML Loading

Job XML is specified by name on the JobOperator.start command (see 10.9.6) to start a job.

All Job XML references are loadable by the following loaders in the order specified:

1. implementation-specific loader

   The batch runtime implementation _must_provide an implementation-specific means by which Job XML references are resolved to a Job XML document.

   The purpose of an implementation-specific loader is to enable Job XMLloading from outside of the application archive, such as from a repository, file system, remote cache, or elsewhere.

2. archive loader

   If the implementation-specific mechanism does fails to resolve a Job XML reference, then the batch runtime implementation must resolve the reference with an archive loader. The implementation must provide an archive loader that resolves the reference by looking up the reference from the `META-INF/batch-jobs` directory.

   Job XML documents may be packaged by the developer with the application under the 'META-INF/batch-jobs' directory ('`WEB-INF/classes/META-INF`/batch-jobs' for .war files).

   See 10.7.2 for more about the `META-INF/batch-jobs`.

Application Packaging Model

The batch artifacts that comprise a batch application requiring no unique packaging. They may be packaged in a standard jar file or can be included inside any Java archive type, as supported by the target execution platform in question.E.g. batch artifacts may be included in wars, EJB jars, etc, so long as they exist in the class loader scope of the program initiating the batch jobs (i.e. using the JobOperator start method).

`META-INF/batch.xml`

A batch application may use the archive loader (see section 10.5) to load batch artifacts. The application can direct artifact loading by supplying an optional `batch.xml` file. The `batch.xml` file must be stored under the `META-INF` directory. For .jar files it is the standard `META-INF` directory. For .war files it is the `WEB-INF/classes/META-INF` directory. The format and content of the `batch.xml` file follows:

```
<batch-artifacts xmlns="http://xmlns.jcp.org/xml/ns/javaee">
 <ref id="<reference-name>" class="<impl-class-name>" />
</batch-artifacts>
```

Where:

<reference-name>	Specifies the reference name of the batch artifact. This is the value that is specified on the ref= attribute of the Job XML.
<impl-class-name>	Specifies the fully qualified class name of the batch artifact implementation.

Notes:

1. If an implementation-specific loader is used (see 10.5) any artifact it loads takes precedence over artifacts specified in `batch.xml`.
2. Use of `batch.xml` to load batch artifacts requires the availability of a zero-argument constructor (either a default constructor or an explicitly-defined, no-arg constructor).

`META-INF/batch-jobs`

A batch application may use the archive loader (see section 10.6) to load Job XML documents. The application does this by storing the Job XML documents under the `META-INF/batch-jobs` directory. For .jar files the batch-jobs directory goes under the standard `META-INF` directory. For .war files it goes under the `WEB-INF/classes/META-INF` directory. Note Job XML documents are valid only in the batch-jobs directory: sub-directories are ignored. Job XML documents stored under `META-INF/batch-jobs` are named with the convention `<name>.xml`,Where:

<name>	Specifies the name of a Job XML. This is the value that is specified on the JobOperator.start command.
.xml	Specifies required file type of a Job XML file under `META-INF/batch-jobs`.

Note if an implementation-specific loader (see 10.6) loads a Job XML document that document takes precedence over documents stored under `META-INF/batch-jobs`.

Restart Processing

The JobOperator restart method is used to restart a JobExecution. A JobExecution is eligible for restart if:

- Its batch status is STOPPED or FAILED.
- It is the most recent JobExecution.

Job Parameters on Restart

Job parameter values are not remembered from one execution to the next. All Job Parameter

substitution during job restart is performed based exclusively on the job parameters specified on that restart.

Job XML Substitution during Restart

See section 8.8.1.8 Job Restart Rule.

Execution Sequence on Restart – Overview

On the initial execution of a JobInstance, the sequence of execution is essentially:

1. Start at initial execution element
2. Execute the current execution element
3. Either:
    a. Transition to next execution element (and go to step 2. above) OR
    b. Terminate execution

On a restart, i.e. a subsequent execution of a JobInstance, the sequence of execution is similar, but the batch implementation must, in addition, determine which steps it does and does not need to re-execute.

So on a restart, the sequence of execution looks like:

1. Start at restart position
2. Decide whether or not to execute (or re-execute) the current execution element
3. Either:
    a. Transition to next execution element (and go to step 2. above) OR
    b. Terminate execution

So it follows that for restart we need: a definition of where in the job definition to begin; rules for deciding whether or not to execute the current execution element; and rules for performing transitioning, especially taking into account that all steps relevant to transitioning may not have executed on this (restart) execution. These rules are provided below.

Execution Sequence on Restart – Detailed Rules

Upon restart, the job is processed as follows:

1. Job XML Substitution is performed (see section 8.8).
2. Start by setting the current position to the restart position. The restart position is either:
    a. the execution element identified by the <stop> elements "restart" attribute if that is how the previous execution ended; else

b. the initial execution element determined the same as upon initial job start, as described in section 8.2.5 Step Sequence;

3. Determine if the current execution element should re-execute:

a. If the current execution element is a COMPLETED step that specifies allow-restart-if-complete=false, then transition based on the exit status for this step from the previous completed execution. If the transition is a next transition, then repeat step 3 here with the value of next as the new, "current" execution element. Or, if the transition is a terminating transition such as end, stop, or fail, then terminate the restart execution accordingly.

b. If the current execution element is a COMPLETED step that specifies allow-restart-if-complete=true, then re-run the step and transition based on the new exit status from the new step execution. As above, either repeat step 3 with the next execution element or terminate the new execution as the transition element

c. If the current execution element is a STOPPED or FAILED step then restart the step and transition based on the exit status from the new step execution.+

   Note if the step is a partitioned step, only the partitions that did not complete previously are restarted. This behavior may be overridden via a PartitionMapper (see section 10.8.5). Note for a partitioned step, the checkpoints and persistent user data are loaded from the persistent store on a per-partition basis (this is not a new rule, but a fact implied by the discussion of checkpoints in section 8.2.6 and the Step Context in section 9.4.1.1, which is summarized here for convenience).

d. If the current execution element is a decision, execute the decision (i.e. execute the Decider) unconditionally. The Deciders "decide" method is passed a StepExecution array as a parameter. This array will be populated with the most-recently completed StepExecution(s) for each corresponding step.E.g. some StepExecution(s) may derive from previous job executions and some from the current restart (execution). A single decision following a split could even have a mix of old, new StepExecution(s) in the same array.

e. If the current execution element is a flow, transition to the first execution element in the flow and perform step 3 with this as the current element. When restart processing of the flow has completed, then follow the same rules which apply during the original execution (see section 8.9) to transition at the flow level to the next execution element, and repeat step 3 with that element as the current element.

   Note the same rules regarding transitioning within a flow during an original execution apply during restart processing as well.

f. If the current execution element is a split, proceed in parallel for each flow in the split. For each flow, repeat step 3 with the flow element as the current element. When all flows in the split have been processed, follow the split's transition to the next execution element and repeat step 3 with that element as the current element.

PartitionMapper on Restart

When the PartitionMapper is invoked at the beginning of a step which has been executed within a previous job execution, the first and most important decision for the mapper implementor to make is whether or not to keep the previous partitions or to begin the new execution with new partition definitions.

This decision is communicated to the batch implementation via the 'partitionsOverride' property of the PartitionPlan built by the mapper, i.e. the result of PartitionPlan's getPartitionsOverride() method.

This property directs whether or not the partitions used in the previous execution of this step will or will be used (i.e. the relevant data carried forward and applied) within the current execution of this step. (As a consequence, the value of this property has no real meaning when the mapper is first called on the first execution of this step).

partitionsOverride = False

Three rules apply in the case where override is set to 'false':

Number of Partitions Must Be Same

The key idea here is that the mapper must build a partition plan with the same number of partitions that were used in the previous execution of this step. As a consequence, it is an error for the partition plan to return (via getPartitions()) a different number than the number of partitions established by the plan the last time this step was executed.

Partition Properties Populated From Current Plan

Though the number of partitions in the previous plan is persisted, the Properties[] returned by the previous PartitionPlan's getPartitionProperties() is not. On a new execution of this step, it is the current return value of PartitionPlan#getPartitionProperties() which is used to populate the pool of potential 'partitionPlan' substitutions (see section 8.8.1.4).

"Numbering" of Partitions via Partition Properties

Upon execution of this step, the batch implementation will associate each element of the Properties[] returned by PartitionPlan#getPartitionProperties() with a single partition, in order to potentially resolve 'partitionPlan' substitutions (see section 8.8.1.4) for a single partition. During the course of execution of each partition, the batch implementation will capture data such as checkpoint values, persistent user data, etc.

Upon a new execution of this step during restart, the batch implementation must ensure that a similar mapping occurs. That is, the elements of the new Properties[] returned by the PartitionPlan#getPartitionProperties() built by the mapper must be mapped to the partitions in the same order as the earlier elements of the earlier Properties[] were mapped (for resolving 'partitionPlan' substitutions).

E.g., the following must hold:

Earlier Execution:

```
partitionPlanProps[] =
mapper.getPartitionPlan().getPartitionProperties();

partitionPlanProps[0] ---maps to---> partition leaving off at
checkpoints R0, W0

partitionPlanProps[1] ---maps to---> partition leaving off at
checkpoints R1, W1
```

Current Execution:

```
newPartitionPlanProps[] =
mapper.getPartitionPlan().getPartitionProperties();

newPartitionPlanProps[0] ---maps to---> partition resuming at
checkpoints R0, W0

newPartitionPlanProps [1] ---maps to---> partition resuming at
checkpoints R1, W1
```

In the shorthand above, "maps to" simply means that the Properties object on the left is used to potentially resolve the 'partitionPlan' substitutions for the give partition, before it executes as described.

partitionsOverride = True

In this case, all partition execution data: checkpoints, persistent user data, etc. from the earlier execution are discarded, and the new PartitionPlan built by the new execution of the PartitionMapper may define either the same or a different number of partitions; the new P artitionPlan's getPartitionProperties() return value will be used to resolve 'partitionPlan' substitutions.

Supporting Classes

JobContext

JobContext.java

```
package javax.batch.runtime.context;
/**
*
* A JobContext provides information about the current
* job execution.
```

```
*
*/
import java.util.Properties;
import javax.batch.runtime.BatchStatus;
public interface JobContext
{
    /**
    * Get job name
    * *@return* value of 'id' attribute from <job>
    */
    public String getJobName();
    /**
    * The getTransientUserData method returns a transient data object
    * belonging to the current Job XML execution element.
    * *@return* user-specified type
    */
    public Object getTransientUserData();
    /**
    * The setTransientUserData method stores a transient data object into
    * the current batch context.
    * @param data is the user-specified type
    */
    public void setTransientUserData(Object data);
    /**
    * The getInstanceId method returns the current job's instance
    * id.
    * *@return* job instance id
    */
    public *long* getInstanceId();
    /**
    * The getExecutionId method returns the current job's current
    * execution id.
    * *@return* job execution id
    */
    public *long* getExecutionId();
    /**
    * The getProperties method returns the job level properties
    * specified in a job definition.
    * <p>
    * A couple notes:
    * <ul>
    * <li> There is no guarantee that the same Properties object instance
    * is always returned in the same (job) scope.
    * <li> Besides the properties which are defined in JSL within a child
    * &lt;
    properties&gt;
    element of a &lt;
    job&gt;
```

```
    element, the batch
    * runtime implementation may choose to include additional,
    * implementation-defined properties.
    * </ul>
    *
    * *@return* job level properties
    */
    public Properties getProperties();
    /**
    * The getBatchStatus method simply returns the batch status value * set
    by the batch runtime into the job context.
    * *@return* batch status string
    */
    public BatchStatus getBatchStatus();
    /**
    * The getExitStatus method simply returns the exit status value stored
    * into the job context through the setExitStatus method or null.
    * *@return* exit status string
    */
    public String getExitStatus();
    /**
    * The setExitStatus method assigns the user-specified exit status for
    * the current job. When the job ends, the exit status of the job is
    * the value specified through setExitStatus. If setExitStatus was not
    * called or was called with a null value, then the exit status
    * defaults to the batch status of the job.
    * @param status string
    */
    public void setExitStatus(String status);
}
```

StepContext

StepContext.java

```
package javax.batch.runtime.context;
import java.io.Serializable;
import java.util.Properties;
import javax.batch.runtime.BatchStatus;
import javax.batch.runtime.Metric;
/**
*
* A StepContext provides information about the current step
* of a job execution.
*
*/
public interface StepContext
```

```
{
    /**
    * Get step name
    * *@return* value of 'id' attribute from <step>
    *
    */
    public String getStepName();
    /**
    * The getTransientUserData method returns a transient data object
    * belonging to the current Job XML execution element.
    * *@return* user-specified type
    */
    public Object getTransientUserData();
    /**
    * The setTransientUserData method stores a transient data object into
    * the current batch context.
    * @param data is the user-specified type
    */
    public void setTransientUserData(Object data);
    /**
    * The getStepExecutionId method returns the current step's
    * execution id.
    * *@return* step execution id
    */
    public *long* getStepExecutionId();
    /**
    * The getProperties method returns the step
    level properties
    * specified in a job definition.
    * <p>
    * A couple notes:
    * <ul>
    * <li> There is no guarantee that the same Properties object instance
    * is always returned in the same (step) scope.
    * <li> Besides the properties which are defined in JSL within a child
    * &lt;
    properties&gt;
    element of a &lt;
    step&gt;
    element, the batch
    * runtime implementation may choose to include additional,
    * implementation-defined properties.
    * </ul>
    * *@return* step level properties
    */
    public Properties getProperties();
    /**
    * The getPersistentUserData method returns a persistent data object
```

```
    * belonging to the current step. The user data type must implement
    * java.util.Serializable. This data is saved as part of a step's
    * checkpoint. For a step that does not do checkpoints, it is saved
    * after the step ends. It is available upon restart.
    * *@return* user-specified type
    */
    public Serializable getPersistentUserData();
    /**
    * The setPersistentUserData method stores a persistent data object
    * into the current step. The user data type must implement
    * java.util.Serializable. This data is saved as part of a step's
    * checkpoint. For a step that does not do checkpoints, it is saved
    * after the step ends. It is available upon restart.
    * @param data is the user-specified type
    */
    public void setPersistentUserData(Serializable data);
    /**
    * The getBatchStatus method returns the current batch status of the
    * current step. This value is set by the batch runtime and changes as
    * the batch status changes.
    * *@return* batch status string
    */
    public BatchStatus getBatchStatus();
    /**
    * The getExitStatus method simply returns the exit status value stored
    * into the step context through the setExitStatus method or null.
    * *@return* exit status string
    */
    public String getExitStatus();
    /**
    * The setExitStatus method assigns the user-specified exit status for
    * the current step. When the step ends, the exit status of the step is
    * the value specified through setExitStatus. If setExitStatus was not
    * called or was called with a null value, then the exit status
    * defaults to the batch status of the step.
    * @param status string
    */
    public void setExitStatus(String status);
    /**
    * The getException method returns the last exception thrown from a
    * step level batch artifact to the batch runtime.
    * *@return* the last exception
    */
    public Exception getException();
    /**
    * The getMetrics method returns an array of step level metrics. These
    * are things like commits, skips, etc.
    * *@see* javax.batch.runtime.metric.Metric for definition of standard
```

```
    * metrics.
    * *@return* metrics array
    */
    public Metric[] getMetrics();
}
```

Metric

Metric.java

```
package javax.batch.runtime;
/**
*
* The Metric interface defines job metrics recorded by
* the batch runtime.
*
*/
public interface Metric
{
    public *enum* MetricType
    {
        READ_COUNT_, _WRITE_COUNT_,
        _COMMIT_COUNT_,
        _ROLLBACK_COUNT_, _READ_SKIP_COUNT_, _PROCESS_SKIP_COUNT_,
        _FILTER_COUNT_,
        _WRITE_SKIPCOUNT
    }
    /**
    * The getName method returns the metric type.
    * *@return* metric type.
    */
    public MetricType getType();
    /**
    * The getValue method returns the metric value.
    * *@return* metric value.
    */
    public *long* getValue();
}
```

PartitionPlan

PartitionPlan.java

```
package javax.batch.api.partition;
/**
*
```

```
* PartitionPlan is a helper class that carries partition processing
* information set by the *@PartitionMapper* method.
*
* A PartitionPlan contains:
* <ol>
* <li>number of partition instances </li>
* <li>number of threads on which to execute the partitions</li>
* <li>substitution properties for each Partition (which can be
* referenced using the <b><i>#
{
    partitionPlan['propertyName']
}
</i></b>
* syntax. </li>
* </ol>
*/
import java.util.Properties;
public interface PartitionPlan
{
    /**
    * Set number of partitions.
    * @param count specifies the partition count
    */
    public void setPartitions(int count);
    /**
    * Specify whether or not to override the partition
    * count from the previous job execution. This applies
    * only to step restart .
    * <p>
    * When false is specified, the
    * partition count from the previous job execution is used
    * and any new value set for partition count in the current run
    * is ignored. In addition, partition results from the previous
    * job execution are remembered, and only incomplete partitions
    * are reprocessed.
    * <p>
    * When true is specified, the partition count from the current run
    * is used and all results from past partitions are discarded. Any
    * resource cleanup or back out of work done in the previous run is the
    * responsibility of the application. The PartitionReducer artifact's
    * rollbackPartitionedStep method is invoked during restart before any
    * partitions begin processing to provide a cleanup hook.
    */
    public void setPartitionsOverride(boolean override);
    /**
    * Return current value of partition override setting.
    * *@return* override setting.
    */
```

```
    public boolean getPartitionsOverride();
    /**
    * Set maximum number of threads requested to use to run
    * partitions for this step. A value of '0' requests the batch
    * implementation to use the partition count as the thread
    * count. Note the batch runtime is not required to use
    * this full number of threads;
    it may not have this many
    * available, and may use less.
    *
    * @param count specifies the requested thread count
    */
    public void setThreads(int count);
    /**
    * Sets array of substitution Properties objects for the set of
    Partitions.
    * @param props specifies the Properties object array
    * @see PartitionPlan#getPartitionProperties()
    */
    public void setPartitionProperties(Properties[] props);
    /**
    * Gets count of Partitions.
    * *@return* Partition count
    */
    public int getPartitions();
    /**
    * Gets maximum number of threads requested to use to run
    * partitions for this step. A value of '0' requests the batch
    * implementation to use the partition count as the thread
    * count. Note the batch runtime is not required to use
    * this full number of threads;
    it may not have this many
    * available, and may use less.
    *
    * *@return* requested thread count
    */
    public int getThreads();
    /**
    * Gets array of Partition Properties objects for Partitions.
    * <p>
    * These can be used in Job XML substitution using
    * substitution expressions with the syntax:
    * <b><i>#
    {
        partitionPlan['propertyName']
    }
    </i></b>
    * <p>
```

```
    * Each element of the Properties array returned can
    * be used to resolving substitutions for a single partition.
    * In the typical use case, each Properties element will
    * have a similar set of property names, with a
    * substitution potentially resolving to the corresponding
    * value for each partition.
    *
    * *@return* Partition Properties object array
    */
    public Properties[]
    getPartitionProperties();
}
```

PartitionPlanImpl.java

```
package javax.batch.api.partition;
import java.util.Properties;
/**
* The PartitionPlanImpl class provides a basic implementation
* of the PartitionPlan interface.
*/
public class PartitionPlanImpl implements PartitionPlan
{
    *private* int partitions= 0;
    *private* boolean override= *false*;
    *private* int threads= 0;
    Properties[] partitionProperties= null;
    @Override
    public void setPartitions(int count)
    {
        partitions= count;
        // default thread count to partition count
        *if* (threads == 0) threads= count;
    }
    @Override
    public void setThreads(int count)
    {
        threads= count;
    }
    @Override
    public void setPartitionsOverride(boolean override)
    {
        *this*.override= override;
    }
    @Override
    public boolean getPartitionsOverride()
    {
```

```
        return override;
    }
    @Override
    public void setPartitionProperties(Properties[] props)
    {
        partitionProperties= props;
    }
    @Override
    public int getPartitions()
    {
        return partitions;
    }
    @Override
    public int getThreads()
    {
        return threads;
    }
    @Override
    public Properties[] getPartitionProperties()
    {
        return partitionProperties;
    }
}
```

BatchRuntime

BatchRuntime.java

```
package javax.batch.runtime;
/**
* The BatchRuntime represents the batch
* runtime environment.
*
*/
import javax.batch.operations.JobOperator;
/**
* BatchRuntime represents the JSR 352 Batch Runtime.
* It provides factory access to the JobOperator interface.
*
*/
public class BatchRuntime
{
    /**
    * The getJobOperator factory method returns
    * an instance of the JobOperator interface.
    * *@return* JobOperator instance.
    */
    public *static* JobOperator getJobOperator()
    {
        return null;
    }
}
```

BatchStatus

BatchStatus.java

```
package javax.batch.runtime;

/**
* BatchStatus enum defines the batch status values
* possible for a job.
*
*/
public enum BatchStatus
{
    STARTING_, _STARTED_, _STOPPING_,
    _STOPPED_, _FAILED_, _COMPLETED_, _ABANDONED_
}
```

JobOperator

JobOperator.java

```
package javax.batch.operations;
import java.util.List;
import java.util.Set;
import java.util.Properties;
import javax.batch.runtime.JobExecution;
import javax.batch.runtime.JobInstance;
import javax.batch.runtime.StepExecution;
/**
* JobOperator provide the interface for operating on batch jobs.
* Through the JobOperator a program can start, stop, and restart jobs.
* It can additionally inspect job history, to discover what jobs
* are currently running and what jobs have previously run.
*
* The JobOperator interface imposes no security constraints. However,
* the implementer is free to limit JobOperator methods with a security
* scheme of its choice. The implementer should terminate any method
* that is limited by the security scheme with a JobSecurityException.
*
*/
public interface JobOperator
{
    /**
    * Returns a set of all job names known to the batch runtime.
    *
    * *@return* a set of job names.
    * @throws JobSecurityException
    */
    public Set<String> getJobNames() throws JobSecurityException;
    /**
    * Returns number of instances of a job with a particular name.
    *
    * @param jobName
    * specifies the name of the job.
    * *@return* count of instances of the named job.
    * @throws NoSuchJobException
    * @throws JobSecurityException
    */
    public int getJobInstanceCount(String jobName) throws
    NoSuchJobException,
    JobSecurityException;
    /**
    * Returns all JobInstances belonging to a job with a particular name
    * in reverse chronological order.
```

```
     *
     * @param jobName
     * specifies the job name.
     * @param start
     * specifies the relative starting number (zero based) to
     * return from the
     * maximal list of job instances.
     * @param count
     * specifies the number of job instances to return from the
     * starting position of the maximal list of job instances.
     * *@return* list of JobInstances.
     * @throws NoSuchJobException
     * @throws JobSecurityException
     */
     public List<JobInstance> getJobInstances(String jobName, int start,
     int count)throws NoSuchJobException, JobSecurityException;
     /**
     * Returns execution ids for job instances with the specified
     * name that have running executions.
     *
     * @param jobName
     * specifies the job name.
     * *@return* a list of execution ids.
     * @throws NoSuchJobException
     * @throws JobSecurityException
     */
     public List<Long> getRunningExecutions(String jobName) throws
     NoSuchJobException, JobSecurityException;
     /**
     * Returns job parameters for a specified job instance. These are the
     * key/value pairs specified when the instance was originally created
     * by the start method.
     *
     * @param executionId
     * specifies the execution from which to retrieve the
     * parameters.
     * *@return* a Properties object containing the key/value job parameter
     * pairs.
     * @throws NoSuchJobExecutionException
     * @throws JobSecurityException
     */
     public Properties getParameters(*long* executionId)
     throws NoSuchJobExecutionException, JobSecurityException;
     /**
     * Creates a new job instance and starts the first execution of that
     * instance, which executes asynchronously.
     *
     * Note the Job XML describing the job is first searched for by name
```

```
    * according to a means prescribed by the batch runtime implementation.
    * This may vary by implementation. If the Job XML is not found by that
    * means, then the batch runtime must search for the specified Job XML
    * as a resource from the `META-INF/batch-jobs` directory based on the
    * current class loader. Job XML files under `META-INF/batch-jobs`
    * directory follow a naming convention of "name".xml where "name" is
    * the value of the jobXMLName parameter (see below).
    *
    * @param jobXMLName
    * specifies the name of the Job XML describing the job.
    * @param jobParameters
    * specifies the keyword/value pairs for attribute
    * substitution in the Job XML.
    * *@return* executionId for the job execution.
    * @throws JobStartException
    * @throws JobSecurityException
    */
    public *long* start(String jobXMLName, Properties jobParameters)
    throws
    JobStartException, JobSecurityException;
    /**
    * Restarts a failed or stopped job instance, which executes
    * asynchronously.
    *
    * @param executionId
    * specifies the execution to to restart. This execution
    * must be the most recent execution that ran.
    * @param restartParameters
    * specifies the keyword/value pairs for attribute
    * substitution in the Job XML.
    * *@return* new executionId
    * @throws JobExecutionAlreadyCompleteException
    * @throws NoSuchJobExecutionException
    * @throws JobExecutionNotMostRecentException,
    * @throws JobRestartException
    * @throws JobSecurityException
    */
    public *long* restart(*long* executionId, Properties
    restartParameters)
    throws JobExecutionAlreadyCompleteException,
    NoSuchJobExecutionException,
    JobExecutionNotMostRecentException,
    JobRestartException,
    JobSecurityException;
    /**
    * Request a running job execution stops. This
    * method notifies the job execution to stop
    * and then returns. The job execution normally
```

```
     * stops and does so asynchronously. Note
     * JobOperator cannot guarantee the jobs stops:
     * it is possible a badly behaved batch application
     * does not relinquish control.
     * <p>
     * Note for partitioned batchlet steps the Batchlet
     * stop method is invoked on each thread actively
     * processing a partition.
     *
     * @param executionId
     * specifies the job execution to stop.
     * The job execution must be running.
     * @throws NoSuchJobExecutionException
     * @throws JobExecutionNotRunningException
     * @throws JobSecurityException
     */
    public void stop(*long* executionId) throws
    NoSuchJobExecutionException,
    JobExecutionNotRunningException, JobSecurityException;
    /**
     * Set batch status to ABANDONED. The instance must have
     * no running execution.
     * <p>
     * Note that ABANDONED executions cannot be restarted.
     *
     * @param executionId
     * specifies the job execution to abandon.
     * @throws NoSuchJobExecutionException
     * @throws JobExecutionIsRunningException
     * @throws JobSecurityException
     */
    public void abandon(*long* executionId) throws
    NoSuchJobExecutionException,
    JobExecutionIsRunningException, JobSecurityException;
    /**
     * Return the job instance for the specified execution id.
     *
     * @param executionId
     * specifies the job execution.
     * *@return* job instance
     * @throws NoSuchJobExecutionException
     * @throws JobSecurityException
     */
    public JobInstance getJobInstance(*long* executionId) throws
    NoSuchJobExecutionException, JobSecurityException;
    /**
     * Return all job executions belonging to the specified job instance.
     *
```

```
    * @param jobInstance
    * specifies the job instance.
    * *@return* list of job executions
    * @throws NoSuchJobInstanceException
    * @throws JobSecurityException
    */
    public List<JobExecution> getJobExecutions(JobInstance instance)
    throws
    NoSuchJobInstanceException, JobSecurityException;
    /**
    * Return job execution for specified execution id
    *
    * @param executionId
    * specifies the job execution.
    * *@return* job execution
    * @throws NoSuchJobExecutionException
    * @throws JobSecurityException
    */
    public JobExecution getJobExecution(*long* executionId) throws
    NoSuchJobExecutionException, JobSecurityException;
    /**
    * Return StepExecutions for specified execution id.
    *
    * @param executionId
    * specifies the job execution.
    * *@return* step executions (order not guaranteed)
    * @throws NoSuchJobExecutionException
    * @throws JobSecurityException
    */
    public List<StepExecution> getStepExecutions(*long* jobExecutionId)
    throws NoSuchJobExecutionException, JobSecurityException;
}
```

JobInstance

JobInstance.java

```
package javax.batch.runtime;
public interface JobInstance
{
    /**
    * Get unique id for this JobInstance.
    * *@return* instance id
    */
    public *long* getInstanceId();
    /**
    * Get job name.
    * *@return* value of 'id' attribute from <job>
    */
    public String getJobName();
}
```

JobExecution

JobExecution.java

```
package javax.batch.runtime;
import java.util.Date;
import java.util.Properties;
public interface JobExecution
{
    /**
    * Get unique id for this JobExecution.
    * *@return* execution id
    */
    public *long* getExecutionId();
    /**
    * Get job name.
    * *@return* value of 'id' attribute from <job>
    */
    public String getJobName();
    /**
    * Get batch status of this execution.
    * *@return* batch status value.
    */
    public BatchStatus getBatchStatus();
    /**
    * Get time execution entered STARTED status.
    * *@return* date (time)
    */
    public Date getStartTime();
    /**
```

```
    * Get time execution entered end status: COMPLETED, STOPPED, FAILED
    * *@return* date (time)
    */
    public Date getEndTime();
    /**
    * Get execution exit status.
    * *@return* exit status.
    */
    public String getExitStatus();
    /**
    * Get time execution was created.
    * *@return* date (time)
    */
    public Date getCreateTime();
    /**
    * Get time execution was last updated updated.
    * *@return* date (time)
    */
    public Date getLastUpdatedTime();
    /**
    * Get job parameters for this execution.
    * *@return* job parameters
    */
    public Properties getJobParameters();
}
```

StepExecution

StepExecution.java

```
package javax.batch.runtime;
import java.util.Date;
import java.io.Serializable;
public interface StepExecution
{
    /**
    * Get unique id for this StepExecution.
    * *@return* StepExecution id
    */
    public *long* getStepExecutionId();
    /**
    * Get step name.
    * *@return* value of 'id' attribute from <step>
    */
    public String getStepName();
    /**
    * Get batch status of this step execution.
```

```
     * *@return* batch status.
     */
     public BatchStatus getBatchStatus();
     /**
     * Get time this step started.
     * *@return* date (time)
     */
     public Date getStartTime();
     /**
     * Get time this step ended.
     * *@return* date (time)
     */
     public Date getEndTime();
     /**
     * Get exit status of step.
     * *@return* exit status
     */
     public String getExitStatus();
     /**
     * Get persistent user data.
     * <p>
     * For a partitioned step, this returns
     * the persistent user data of the
     * <code>StepContext</code> of the "top-level"
     * or main thread (the one the <code>PartitionAnalyzer</code>, etc.
     * execute on). It does not return the persistent user
     * data of the partition threads.
     * *@return* persistent data
     */
     public Serializable
     getPersistentUserData ();
     /**
     * Get step metrics
     * *@return* array of metrics
     */
     public Metric[] getMetrics();
}
```

Batch Exception Classes

This specification defines batch exception classes in package javax.batch.operations. Note all batch exceptions are direct subclasses of base class BatchRuntimeException, which itself is a direct subclass of java.lang.RuntimeException. The following batch exception classes are defined:

1. JobExecutionAlreadyCompleteException
2. JobExecutionIsRunningException

3. JobExecutionNotMostRecentException
4. JobExecutionNotRunningException
5. JobRestartException
6. JobSecurityException
7. JobStartException
8. NoSuchJobException
9. NoSuchJobExecutionException
10. NoSuchJobInstanceException

Job Runtime Lifecycle

The following sections describe an ordered flow of artifact method invocations. Simple symbols are used to denote actions as follows:

Symbol	Meaning
<action>	An action performed by the batch runtime.
< - > method	Invocation of a batch artifact method by the batch runtime.
[method]	Optional method.
// comment	Comment to clarify behavior.
LABEL:	Label used for flow control comments.

Batch Artifact Lifecycle

All batch artifacts are instantiated prior to their use in the scope in which they are declared in the Job XML and are valid for the life of their containing scope. There are three scopes that pertain to artifact lifecycle: job, step, and step-partition.

One artifact per Job XML reference is instantiated. In the case of a partitioned step, one artifact per Job XML reference per partition is instantiated. This means job level artifacts are valid for the life of the job. Step level artifacts are valid for the life of the step. Step level artifacts in a partition are valid for the life of the partition.

No artifact instance may be shared across concurrent scopes. The same instance must be used in the applicable scope for a specific Job XML reference.

Job Repository Artifact Lifecycle

All job repository artifacts are created by the batch runtime during job processing and exist until deleted by an implementation provided means.

Job Processsing

1. <Create JobContext>
2. <Store job level properties in JobContext>
3. < - >[JobListener.beforeJob…] // thread A
4. <processs execution elements>
5. < - >[JobListener.afterJob…] // thread A

6. <Destroy JobContext>

Regular Batchlet Processsing

1. <Create StepContext>
2. <Store step level properties in StepContext>
3. < - > [StepListener.beforeStep…] // thread A
4. < - > Batchlet.process // thread A
5. // if stop issued:
6. < - >[Batchlet.stop] // thread B, StepContext is available
7. < - >[StepListener.afterStep…] // thread A
8. <Store StepContext persistent area>
9. <Destroy StepContext>

Partitioned Batchlet Processsing

1. <Create StepContext>
2. <Store step level properties in StepContext>
3. < - >[StepListener.beforeStep…] // thread A
4. < - >[PartitionReducer.beginPartitionedStep] // thread A
5. < - >[PartitionMapper.mapPartitions] // thread A
6. // per partition:
    a. < - >Batchlet.process // thread Px
    b. // if stop issued:
    c. < - >[Batchlet.stop] // thread Py, StepContext is available
    d. < - >[PartitionCollector.collectPartitionData] // thread Px
7. // when collector payload arrives:
    a. < - >[PartitionAnalyzer.analyzeCollectorData] // thread A
8. // when partition ends:
    a. < - >[PartitionAnalyzer.analyzeStatus] // thread A
9. // if rollback condition occurs:
10. < - >[PartitionReducer.rollbackPartitionedStep] // thread A
11. < - >[PartitionReducer.beforePartitionedStepCompletion] // thread A

12. < - >[PartitionReducer.afterPartitionedStepCompletion] // thread A
13. < - >[StepListener.afterStep…] // thread A
14. <Store StepContext persistent area>
15. <Destroy StepContext>

Regular Chunk Processing

1. <Create StepContext>
2. <Store step level properties in StepContext>
3. < - >[StepListener.beforeStep…] // thread A
4. [<begin transaction>]
5. < - >ItemReader.open // thread A
6. < - >ItemWriter.open // thread A
7. [<commit transaction>]
8. // chunk processing:
9. <repeat until no more items (i.e. while readItem hasn’t returned 'null') > \{
    a. <begin checkpoint interval [<begin chunk transaction>]>
    b. <repeat until checkpoint criteria reached OR readItem returns `null`> \{ …< - >ItemReader.readItem // thread A
        i. // if readItem returns non-null
            A. < - >ItemProcessor.processItem // thread A
            B. // if processItem returns non-null, <add item to writeItems buffer>
    c. }
    d. // if at least one non-null value has been successfully read inthe present chunk
        i. < - >ItemWriter.writeItems // thread A
    e. < - >[ItemReader.checkpointInfo] // thread A
    f. < - >[ItemWriter.checkpointInfo] // thread A
    g. <Store StepContext persistent area>
    h. [<commit chunk transaction>]
10. }
11. [<begin transaction>]
12. < - >ItemWriter.close // thread A

13. < - >ItemReader.close // thread A
14. [<commit transaction>]
15. < - >[StepListener.afterStep…] // thread A
16. <Store StepContext persistent area>
17. <Destroy StepContext>

Partitioned Chunk Processing

1. <Create StepContext>
2. <Store step level properties in StepContext>
3. < - >[StepListener.beforeStep…] // thread A
4. < - >[PartitionReducer.beginPartitionedStep] // thread A
5. < - >[PartitionMapper.mapPartitions] // thread A+ // per partition - on thread Px:
    a. [<begin transaction>]
    b. < - >ItemReader.open // thread Px
    c. < - >ItemWriter.open // thread Px
    d. [<commit transaction>]
    e. <repeat until no more items (i.e. while readItem hasn’t returned 'null') > \{
        i. <begin checkpoint interval [<begin chunk transaction>]>
        ii. <repeat until checkpoint criteria reached OR readItem returns 'null'> \{
            A. < - >ItemReader.readItem // thread Px
            B. // if readItem returns non-null
                I. < - >ItemProcessor.processItem // thread Px
                II. //if processItem returns non-null, <add item to writeItems buffer>
        iii. }
        iv. //if at least one non-null value has been successfully read in this partition of the present chunk
            A. < - >ItemWriter.writeItems // thread Px …< - >[ItemReader.checkpointInfo] // thread Px … < - >[ItemWriter.checkpointInfo] // thread Px
        v. <Store (partition-local) StepContext persistent area>
        vi. [<commit chunk transaction>]
        vii. < - >[PartitionCollector.collectPartitionData] // thread Px
    f. }

    g. [<begin transaction>]
    h. < - >ItemWriter.close // thread Px
    i. < - >ItemReader.close // thread Px
    j. [<commit transaction>]
6. [<begin transaction>] // thread A
7. // Actions 9-12 run continuously until all partitions end.
8. // when collector payload arrives:
    a. < - >[PartitionAnalyzer.analyzeCollectorData] // thread A
9. // when partition ends:
    a. < - >[PartitionAnalyzer.analyzeStatus] // thread A
10. // Remaining actions run after all partitions end:
11. // if rollback condition occurs:
    a. < - >[PartitionReducer.rollbackPartitionedStep] // thread A
    b. [<rollback transaction >]
12. // else not rollback
13. < - >[PartitionReducer.beforePartitionedStepCompletion] // thread A
14. [<commit transaction>] // thread A
15. < - >[PartitionReducer.afterPartitionedStepCompletion] // thread A
16. < - >[StepListener.afterStep…] // thread A
17. <Store StepContext persistent area>
18. <Destroy StepContext>

Chunk with Listeners (except RetryListener)

1. <Create StepContext>
2. <Store step level properties in StepContext>
3. < - >[StepListener.beforeStep…] // thread A
4. [<begin transaction>]
5. < - >ItemReader.open // thread A
6. < - >ItemWriter.open // thread A
7. [<commit transaction>]
8. // chunk processing:

9. <repeat until no more items (i.e. while readItem hasn’t returned 'null') > \{
    a. <begin checkpoint interval [<begin chunk transaction>]>
    b. < - >[ChunkListener.beforeChunk] // thread A
    c. <repeat until checkpoint criteria reached OR readItem returns 'null'> {
        i. < - >[ItemReadListener.beforeRead] // thread A
        ii. < - >ItemReader.readItem // thread A
        iii. < - >[ItemReadListener.afterRead] // thread A
        iv. // or:
        v. \{
        vi. < - >[ItemReadListener.onReadError] // thread A
        vii. < - >[SkipListener.onSkipReadItem] // thread A
        viii. }
        ix. // if readItem returns non-null
            A. < - >[ItemProcessListener.beforeProcess] // thread A
            B. < - >ItemProcessor.processItem // thread A
            C. < - >[ItemProcessListener.afterProcess] // thread A
            D. //if processItem returns non-null,< add item to writeItems buffer>
            E. // or:
            F. \{
            G. < - >[ItemProcessListener.onProcessError] // thread A
            H. < - >[SkipListener.onSkipProcessItem] // thread A
            I. }
        x. }
        xi. //if at least one non-null value has been successfully read in the present chunk
            A. < - >[ItemWriteListener.beforeWrite] // thread A
            B. < - >ItemWriter.writeItems // thread A
            C. < - >[ItemWriteListener.afterWrite] // thread A
            D. // or:
            E. {
            F. < - >[ItemWriteListener.onWriteError] // thread A
            G. < - >[SkipListener.onSkipWriteItems] // thread A
        xii. }

    d. < - >[ChunkListener.afterChunk] // thread A
    e. < - >[ItemReader.checkpointInfo] // thread A
    f. < - >[ItemWriter.checkpointInfo] // thread A
    g. <Store StepContext persistent area>
    h. [<commit chunk transaction>]
10. }
11. [<begin transaction>]
12. < - >ItemWriter.close // thread A
13. < - >ItemReader.close // thread A
14. [<commit transaction>]
15. < - >[StepListener.afterStep…] // thread A
16. <Store StepContext persistent area>
17. <Destroy StepContext>

Chunk with RetryListener

Note rollback processing is also depicted in this section.

1. <Create StepContext>
2. <Store step level properties in StepContext>
3. < - >[StepListener.beforeStep…] // thread A
4. [<begin transaction>]
5. < - >ItemReader.open // thread A
6. < - >ItemWriter.open // thread A
7. [<commit transaction>]
8. // chunk processing:
9. <repeat until no more items (i.e. while readItem hasn’t returned 'null') > \{
    a. S1:
    b. <begin checkpoint interval [<begin chunk transaction>]>
    c. <repeat until checkpoint criteria reached OR readItem returns 'null'> \{
        i. S2:
        ii. < - >ItemReader.readItem // thread A
        iii. // if exception
        iv. < - >[ItemReadListener.onReadErrror] // thread A

v. < - >[RetryReadListener.onRetryReadException] // thread A
vi. // if retryable exception
vii. // if no-rollback exception
viii. resume S2:
ix. // else
x. <end repeat>
xi. // else
xii. <end repeat>
xiii. S3:
xiv. // if readItem
xv. returns non-null
xvi. < - >ItemProcessor.processItem // thread A
xvii. // if exception
xviii. < - >[ItemProcessListener.onProcessErrror] // thread A
xix. < - >[RetryProcessListener.onRetryProcessException] // thread A
xx. // if retryable exception
xxi. // if no-rollback exception
xxii. resume S3:
xxiii. // else
xxiv. <end repeat>
xxv. // else
xxvi. <end repeat>
xxvii. // if processItem returns non-null, <add item to writeItems buffer>

d. }
e. // if rollback exception, execute rollback procedure (below) and resume at S1 with item-count=1
f. S4:
g. // if at least one non-null value has been successfully read in the present chunk
h. < - >ItemWriter.writeItems (buffer) // thread A
i. // if exception
j. < - >[ItemWriteListener.onWriteErrror] // thread A
k. < - >[RetryWriteListener.onRetryWriteException] // thread A
l. // if retryable exception

m. // if no-rollback exception
n. resume S4:
o. // else
p. execute rollback procedure (below) and resume S1:
q. // else execute rollback procedure (below) and resume S1:
r. < - >[ItemReader.checkpointInfo] // thread A
s. < - >[ItemWriter.checkpointInfo] // thread A
t. <Store StepContext persistent area> // thread A
u. S5:
v. [<commit chunk transaction>] // thread A
w. // if exception
x. // if retryable exception
y. // if no-rollback exception:
z. resume S5:
aa. // else
ab. execute rollback procedure (below) and resume S1:
ac. // else execute rollback procedure (below) and resume S1:

10. }
11. [<begin transaction>]
12. < - >ItemWriter.close // thread A
13. < - >ItemReader.close // thread A
14. [<commit transaction>]
15. < - >[StepListener.afterStep…] // thread A
16. <Store StepContext persistent area>
17. <Destroy StepContext>

Rollback Procedure

1. < - >ItemWriter.close // thread A
2. < - >ItemReader.close // thread A
3. [ChunkListener.onError] // thread A
4. [rollback transaction]
5. [<begin transaction>]

< - >ItemWriter.open // thread A, pass last committed checkpoint info

7. < - >ItemReader.open // thread A, pass last committed checkpoint info
8. [<commit transaction>]

Chunk with Custom Checkpoint Processing

1. <Create StepContext>
2. <Store step level properties in StepContext>
3. < - >[StepListener.beforeStep…] // thread A
4. [<begin transaction>]
5. < - >ItemReader.open // thread A
6. < - >ItemWriter.open // thread A
7. [<commit transaction>]
8. // chunk processing:
9. <repeat until no more items (i.e. while readItem hasn’t returned 'null') > \{
    a. [
    b. < - >[CheckpointAlgorithm.checkpointTimeout]] // thread A
    c. < - >[CheckpointAlgorithm.beginCheckpoint] // thread A
    d. <begin checkpoint interval [<begin chunk transaction>]>
    e.]
    f. <repeat until isReadyToCheckpoint returns 'true' OR readItem returns 'null'> \{
    g. < - >ItemReader.readItem // thread A
    h. // if readItem returns non-null
10. < - >ItemProcessor.processItem // thread A
11. //if processItem returns non-null, <add item to writeItems buffer>
    a. < - >CheckpointAlgorithm.isReadyToCheckpoint // thread A
    b. }
    c. //if at least one non-null value has been successfully read in the present chunk
    d. < - >ItemWriter.writeItems // thread A
    e. < - >[ItemReader.checkpointInfo] // thread A
    f. < - >[ItemWriter.checkpointInfo] // thread A
    g. <Store StepContext persistent area>
    h. [<commit chunk transaction>]
6.

    i. < - >[CheckpointAlgorithm.endCheckpoint] // thread A

12. }
13. [<begin transaction>]
14. < - >ItemWriter.close // thread A
15. < - >ItemReader.close // thread A
16. [<commit transaction>]
17. < - >[StepListener.afterStep…] // thread A
18. <Store StepContext persistent area>
19. <Destroy StepContext>

Split Processing

1. // For each flow:
2. <run flow> // thread Fx

Flow Processing

1. // For each split or step:
2. <run split or step> // thread Xy

Stop Processing

The JobOperator.stop operation stops a running job execution. If a step is running at the time the stop is invoked, the batch runtime takes the following actions:

Chunk Step

The job and step batch status is marked STOPPING. Note the batch runtime cannot guarantee the step actually exits. The batch runtime attempts to interrupt the read/process/write chunk processing loop. The batch runtime allows the step to finish processing the current item. This means the current item is read, processed if a processor is configured, and all currently buffered items, if any, including the current item, are written. If the batch artifacts configured on the chunk type step return to the batch runtime, as expected, the job and step batch status is marked STOPPED.

Batchlet Step

The job and step batch status is marked STOPPING. The batch runtime invokes the batchlet's stop method. Note the batch runtime cannot guarantee the batchlet actually exits. But a well behaved batchlet will. If the batchlet returns to the batch runtime, the job and step batch status is marked STOPPED.

Note for partitioned batchlet steps the Batchlet stop method is invoked on each thread actively processing a partition.

Batch XML XSD

```
<xml version="1.0" encoding="UTF-8">
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified"
 targetNamespace="http://xmlns.jcp.org/xml/ns/javaee"
 mlns:jbatch="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
 <xs:element name="batch-artifacts" type="jbatch:BatchArtifacts" />
 <xs:complexType name="BatchArtifacts">
 <xs:sequence>
 <xs:element name="ref" type="jbatch:BatchArtifactRef" minOccurs="0"
maxOccurs="unbounded" />
 </xs:sequence>
 </xs:complexType>
 <xs:complexType name="BatchArtifactRef">
 <xs:attribute name="id" use="required" type="xs:string" />
 <xs:attribute name="class" use="required" type="xs:string" />
 </xs:complexType>
 </xs:schema>
```

Job Specification Language

Jobs are described by a declarative Job Specification Language (JSL) defined by an XML schema, also known informally as Job XML.

Validation Rules

The batch runtime must perform schema validation during JobOperator start processing before the start method returns to the caller. A schema validation error results in JobStartException. The implementation has two choices for handling semantic errors in the JSL:

1. Do semantic validation during JobOperator start processing before returning to the caller. If there is a semantic validation error, the implementation must throw JobStartException.
2. Do semantic validation after job execution begins. If a semantic validation error occurs, the implementation must end the job in the FAILED state. The implementation is advised to log sufficient error information to enable problem resolution.

Typical semantic validation the batch runtime should detect and handle include, but is not limited to:

1. no executable elements
2. non-existent transitions (e.g. next="value" where "value" does not exist)
3. cycles among next values (e.g. step1:next=step2; step2:next=step1)

JSL XSD

```
<xml version="1.0" encoding="UTF-8">
 <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
 elementFormDefault="qualified"
targetNamespace="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:jsl="http://xmlns.jcp.org/xml/ns/javaee" version="1.0">
 <xs:annotation>
 <xs:documentation>
 Job Specification Language (JSL) specifies a job,
 its steps, and directs their execution.
 JSL also can be referred to as "Job XML".
 </xs:documentation>
 </xs:annotation>
 <xs:simpleType name="artifactRef">
 <xs:annotation>
 <xs:documentation>
 This is a helper type. Though it is not otherwise
 called out by this name
 in the specification, it captures the fact
```

```
 that the xs:string value refers
 to a batch artifact, across numerous
 other JSL type definitions.
 </xs:documentation>
 </xs:annotation>
 <xs:restriction base="xs:string" />
 </xs:simpleType>
 <xs:complexType name="Job">
 <xs:annotation>
 <xs:documentation>
The type of a job definition, whether concrete or
Abstract. This is the type of the root element of any JSL document.
 </xs:documentation>
 </xs:annotation>
 <xs:sequence>
 <xs:element name="properties" type="jsl:Properties"
 minOccurs="0" maxOccurs="1">
 <xs:annotation>
 <xs:documentation>
The job-level properties, which are accessible
 via the JobContext.getProperties() API in a batch
Artifact.
 </xs:documentation>
 </xs:annotation>
 </xs:element>
 <xs:element name="listeners" type="jsl:Listeners"
 minOccurs="0" maxOccurs="1">
 <xs:annotation>
 <xs:documentation>
 Note that "listeners" sequence order in XML does
 not imply order of execution by
The batch runtime, per the
 specification.
 </xs:documentation>
 </xs:annotation>
 </xs:element>
 <xs:choice minOccurs="0" maxOccurs="unbounded">
 <xs:element name="decision" type="jsl:Decision" />
 <xs:element name="flow" type="jsl:Flow" />
 <xs:element name="split" type="jsl:Split" />
 <xs:element name="step" type="jsl:Step" />
 </xs:choice>
 </xs:sequence>
 <xs:attribute name="version" use="required" type="xs:string"
fixed="1.0" />
 <xs:attribute name="id" use="required" type="xs:ID" />
 <xs:attribute name="restartable" use="optional" type="xs:string" />
 </xs:complexType>
```

```
 <xs:element name="job" type="jsl:Job">
 <xs:annotation>
 <xs:documentation>
The definition of an job, whether concrete or
Abstract. This is the
 type of the root element of any JSL document.
 </xs:documentation>
 </xs:annotation>
 </xs:element>
 <xs:complexType name="Listener">
 <xs:sequence>
 <xs:element name="properties" type="jsl:Properties"
 minOccurs="0" maxOccurs="1" />
 </xs:sequence>
 <xs:attribute name="ref" use="required" type="jsl:artifactRef" />
 </xs:complexType>
 <xs:complexType name="Split">
 <xs:sequence>
 <xs:element name="flow" type="jsl:Flow" minOccurs="0"
maxOccurs="unbounded" />
 </xs:sequence>
 <xs:attribute name="id" use="required" type="xs:ID" />
 <xs:attribute name="next" use="optional" type="xs:string" />
 </xs:complexType>
 <xs:complexType name="Flow">
 <xs:sequence>
 <xs:choice minOccurs="0" maxOccurs="unbounded">
 <xs:element name="decision" type="jsl:Decision" />
 <xs:element name="flow" type="jsl:Flow" />
 <xs:element name="split" type="jsl:Split" />
 <xs:element name="step" type="jsl:Step" />
 </xs:choice>
 <xs:group ref="jsl:TransitionElements" minOccurs="0"
maxOccurs="unbounded" />
 </xs:sequence>
 <xs:attribute name="id" use="required" type="xs:ID" />
 <xs:attribute name="next" use="optional" type="xs:string" />
 </xs:complexType>
 <xs:group name="TransitionElements">
 <xs:annotation>
 <xs:documentation>
 This grouping provides allows for the reuse of the
 'end', 'fail', 'next', 'stop' element sequences which
 may appear at the end of a 'step', 'flow', 'split' or 'decision'.
The term 'TransitionElements' does not formally appear in the spec, it
is
A schema convenience.
 </xs:documentation>
```

```
 </xs:annotation>
 <xs:choice>
 <xs:element name="end" type="jsl:End" />
 <xs:element name="fail" type="jsl:Fail" />
 <xs:element name="next" type="jsl:Next" />
 <xs:element name="stop" type="jsl:Stop" />
 </xs:choice>
 </xs:group>
 <xs:complexType name="Decision">
 <xs:sequence>
 <xs:element name="properties" type="jsl:Properties"
 minOccurs="0" maxOccurs="1" />
 <xs:group ref="jsl:TransitionElements" minOccurs="0"
maxOccurs="unbounded"
 />
 </xs:sequence>
 <xs:attribute name="id" use="required" type="xs:ID" />
 <xs:attribute name="ref" use="required" type="jsl:artifactRef" />
 </xs:complexType>
 <xs:attributeGroup name="TerminatingAttributes">
 <xs:attribute name="on" use="required" type="xs:string" />
 <xs:attribute name="exit-status" use="optional" type="xs:string" />
 </xs:attributeGroup>
 <xs:complexType name="Fail">
 <xs:attributeGroup ref="jsl:TerminatingAttributes" />
 </xs:complexType>
 <xs:complexType name="End">
 <xs:attributeGroup ref="jsl:TerminatingAttributes" />
 </xs:complexType>
 <xs:complexType name="Stop">
 <xs:attributeGroup ref="jsl:TerminatingAttributes" />
 <xs:attribute name="restart" use="optional" type="xs:string" />
 </xs:complexType>
 <xs:complexType name="Next">
 <xs:attribute name="on" use="required" type="xs:string" />
 <xs:attribute name="to" use="required" type="xs:string" />
 </xs:complexType>
 <xs:complexType name="CheckpointAlgorithm">
 <xs:sequence>
 <xs:element name="properties" type="jsl:Properties"
 minOccurs="0" maxOccurs="1" />
 </xs:sequence>
 <xs:attribute name="ref" use="required" type="jsl:artifactRef" />
 </xs:complexType>
 <xs:complexType name="ExceptionClassFilter">
 <xs:sequence>
 <xs:element name="include" minOccurs="0" maxOccurs="unbounded">
 <xs:complexType>
```

```
 <xs:sequence />
 <xs:attribute name="class" use="required" type="xs:string" />
 </xs:complexType>
 </xs:element>
 <xs:element name="exclude" minOccurs="0" maxOccurs="unbounded">
 <xs:complexType>
 <xs:sequence />
 <xs:attribute name="class" use="required" type="xs:string" />
</xs:complexType>
 </xs:element>
 </xs:sequence>
 </xs:complexType>
 <xs:complexType name="Step">
 <xs:sequence>
 <xs:element name="properties" type="jsl:Properties" minOccurs="0"
maxOccurs="1" />
 <xs:element name="listeners" type="jsl:Listeners"
 minOccurs="0" maxOccurs="1">
 <xs:annotation>
 <xs:documentation>
 Note that "listeners" sequence order in XML does
 not imply order of execution by
The batch runtime, per the
 specification.
 </xs:documentation>
 </xs:annotation>
 </xs:element>
 <xs:choice minOccurs="0" maxOccurs="1">
 <xs:element name="batchlet" type="jsl:Batchlet" />
 <xs:element name="chunk" type="jsl:Chunk" />
 </xs:choice>
 <xs:element name="partition" type="jsl:Partition"
 minOccurs="0" maxOccurs="1" />
 <xs:group ref="jsl:TransitionElements"

minOccurs="0" maxOccurs="unbounded" />
 </xs:sequence>
 <xs:attribute name="id" use="required" type="xs:ID" />
 <xs:attribute name="start-limit" use="optional" type="xs:string" />
 <xs:attribute name="allow-start-if-complete" use="optional"
 type="xs:string" />
 <xs:attribute name="next" use="optional" type="xs:string" />
 </xs:complexType>
 <xs:complexType name="Batchlet">
 <xs:sequence>
 <xs:element name="properties" type="jsl:Properties"
 minOccurs="0" maxOccurs="1" />
 </xs:sequence>
```

```
 <xs:attribute name="ref" use="required" type="jsl:artifactRef" />
</xs:complexType>
 <xs:complexType name="Chunk">
 <xs:sequence>
 <xs:element name="reader" type="jsl:ItemReader" />
 <xs:element name="processor" type="jsl:ItemProcessor"
 minOccurs="0" maxOccurs="1" />
 <xs:element name="writer" type="jsl:ItemWriter" />
 <xs:element name="checkpoint-algorithm" type="jsl:CheckpointAlgorithm"
minOccurs="0" maxOccurs="1" />
 <xs:element name="skippable-exception-classes"
type="jsl:ExceptionClassFilter" minOccurs="0" maxOccurs="1" />
 <xs:element name="retryable-exception-classes"
type="jsl:ExceptionClassFilter"
 minOccurs="0" maxOccurs="1" />
 <xs:element name="no-rollback-exception-classes"
type="jsl:ExceptionClassFilter"
 minOccurs="0" maxOccurs="1" />
 </xs:sequence>
 <xs:attribute name="checkpoint-policy" use="optional"
type="xs:string">
 <xs:annotation>
 <xs:documentation>
 Specifies the checkpoint policy that governs
 commit behavior for this chunk.
 Valid values are: "item" or
 "custom". The "item" policy means the
 chunk is checkpointed after a
 specified number of items are
 processed. The "custom" policy means
The chunk is checkpointed
According to a checkpoint algorithm
 implementation. Specifying
 "custom" requires that the
 checkpoint-algorithm element is also
 specified. It is an optional
Attribute. The default policy is
 "item". However, we chose not to define
A schema-specified default for this attribute.
 </xs:documentation>
 </xs:annotation>
 </xs:attribute>
 <xs:attribute name="item-count" use="optional" type="xs:string">
 <xs:annotation>
 <xs:documentation>
 Specifies the number of items to process per chunk
 when using the item
 checkpoint policy. It must be valid XML integer.
```

```
 It is an optional
Attribute. The default is 10. The item-count
Attribute is ignored
 for "custom" checkpoint policy. However, to
 make it easier for implementations to support JSL inheritance
 we abstain from defining a schema-specified default for this
Attribute.
 </xs:documentation>
 </xs:annotation>
 </xs:attribute>
 <xs:attribute name="time-limit" use="optional" type="xs:string">
 <xs:annotation>
 <xs:documentation>
 Specifies the amount of time in seconds before
 taking a checkpoint for the
 item checkpoint policy. It must be valid
 XML integer. It is an
 optional attribute. The default is 0, which
 means no limit. However, to
 make it easier for implementations to
 support JSL inheritance
 we abstain from defining a schema-specified
 default for this attribute.
 When a value greater than zero is
 specified, a checkpoint is taken when
 time-limit is reached or
 item-count items have been processed,
 whichever comes first. The
 time-limit attribute is ignored for
 "custom" checkpoint policy.
 </xs:documentation>
 </xs:annotation>
 </xs:attribute>
 <xs:attribute name="skip-limit" use="optional" type="xs:string">
 <xs:annotation>
 <xs:documentation>
 Specifies the number of exceptions a step will
 skip if any configured
 skippable exceptions are thrown by chunk
 processing. It must be a
 valid XML integer value. It is an optional
Attribute. The default
 is no limit.
 </xs:documentation>
 </xs:annotation>
 </xs:attribute>
 <xs:attribute name="retry-limit" use="optional" type="xs:string">
 <xs:annotation>
```

```
 <xs:documentation>
 Specifies the number of times a step will retry if
Any configured retryable
 exceptions are thrown by chunk processing.
 It must be a valid XML
 integer value. It is an optional attribute.
The default is no
 limit.
 </xs:documentation>
 </xs:annotation>
 </xs:attribute>
 </xs:complexType>
 <xs:complexType name="ItemReader">
 <xs:sequence>
 <xs:element name="properties" type="jsl:Properties"
 minOccurs="0" maxOccurs="1" />
 </xs:sequence>
 <xs:attribute name="ref" use="required" type="jsl:artifactRef" />
 </xs:complexType>
 <xs:complexType name="ItemProcessor">
 <xs:sequence>
 <xs:element name="properties" type="jsl:Properties"
 minOccurs="0" maxOccurs="1" />
 </xs:sequence>
 <xs:attribute name="ref" use="required" type="jsl:artifactRef" />
 </xs:complexType>
 <xs:complexType name="ItemWriter">
 <xs:sequence>
 <xs:element name="properties" type="jsl:Properties"
 minOccurs="0" maxOccurs="1" />
 </xs:sequence>
 <xs:attribute name="ref" use="required" type="jsl:artifactRef" />
 </xs:complexType>
 <xs:complexType name="Property">
 <xs:attribute name="name" type="xs:string" use="required" />
 <xs:attribute name="value" type="xs:string" use="required" />
 </xs:complexType>
 <xs:complexType name="Properties">
 <xs:sequence>
 <xs:element name="property" type="jsl:Property" maxOccurs="unbounded"
minOccurs="0" />
 </xs:sequence>
 <xs:attribute name="partition" use="optional" type="xs:string" />
 </xs:complexType>
 <xs:complexType name="Listeners">
 <xs:sequence>
 <xs:element name="listener" type="jsl:Listener" maxOccurs="unbounded"
minOccurs="0" />
```

```
</xs:sequence>
</xs:complexType>
<xs:complexType name="Partition">
<xs:sequence>
<xs:choice minOccurs="0" maxOccurs="1">
<xs:element name="mapper" type="jsl:PartitionMapper" />
<xs:element name="plan" type="jsl:PartitionPlan" />
</xs:choice>
<xs:element name="collector" type="jsl:Collector"
minOccurs="0" maxOccurs="1" />
<xs:element name="analyzer" type="jsl:Analyzer" minOccurs="0"
maxOccurs="1" />
<xs:element name="reducer" type="jsl:PartitionReducer"
minOccurs="0" maxOccurs="1" />
</xs:sequence>
</xs:complexType>
<xs:complexType name="PartitionPlan">
<xs:sequence>
<xs:element name="properties" type="jsl:Properties"
minOccurs="0" maxOccurs="unbounded" />
</xs:sequence>
<xs:attribute name="partitions" use="optional" type="xs:string" />
<xs:attribute name="threads" use="optional" type="xs:string" />
</xs:complexType>
<xs:complexType name="PartitionMapper">
<xs:sequence>
<xs:element name="properties" type="jsl:Properties"
minOccurs="0" maxOccurs="1" />
</xs:sequence>
<xs:attribute name="ref" use="required" type="jsl:artifactRef" />
</xs:complexType>
<xs:complexType name="Collector">
<xs:sequence>
<xs:element name="properties" type="jsl:Properties"
minOccurs="0" maxOccurs="1" />
</xs:sequence>
<xs:attribute name="ref" use="required" type="jsl:artifactRef" />
</xs:complexType>
<xs:complexType name="Analyzer">
<xs:sequence>
<xs:element name="properties" type="jsl:Properties"
minOccurs="0" maxOccurs="1" />
</xs:sequence>
<xs:attribute name="ref" use="required" type="jsl:artifactRef" />
</xs:complexType>
<xs:complexType name="PartitionReducer">
<xs:sequence>
<xs:element name="properties" type="jsl:Properties"
```

```
  minOccurs="0" maxOccurs="1" />
  </xs:sequence>
  <xs:attribute name="ref" use="required" type="jsl:artifactRef" />
  </xs:complexType>
  </xs:schema>
```

Credits

Section 7 Domain Language of Batch, was adapted from Spring Batch Reference Documentation:

http://static.springsource.org/spring-batch/trunk/reference/html-single/index.html

Change Log

Version 1.0 Revision A - Maintenance Release

Issues List

Following these links will show each original issue on our official spec issues tracking list. In most cases the bug report contains the complete text of the spec delta or addition, but not in every single case.

5389	In Sec. 10.7.1, should we have said we require a "no-arg" explicit or implicit constructor rather than a "default constructor"
4827	SPEC: Misspoke on collector role on exit status
5490	Clarify JobContext/StepContext properties; fix TCK to not depend on writable Properties
5431	ItemProcessListener#onProcessError has javadoc from ItemProcessListener#afterProcess
5498	Add "mark FAILED" to BatchStatus state transitions
5370	Spec is unclear whether JobOperator methods may/must execute synchronously or not (with TCK implications)
5583	CheckpointAlgorithm needs to specify timeunit (seconds) and other javadoc fixes
5372	Evaluation order of multiple transition elements
5691	"Looping" should be clarified
5690	Flow/Split transitioning & termination not fully defined
5374	Details of exception handling (by container)
4830	8.6.1 Transition Next Element
4865	SPEC Partition Plan example confusing
5533	stop/end/fail exit-status should affect job exit status, not step (as claimed in spec).
5780	Spec should clarify StepExecution values passed to Decider on a restart

5373	Co-existence of transition elements with @next attribute PLUS behavior if no transition element @on is matched
5375	Spec contradicts itself when talking about uninitialized exit status (TCK assumes 'null')
4866	SPEC Partition Properties example has a invalid tag
5746	@Inject @BatchProperty should work for job level properties
5911	Clarify partition restart processing, PartitionPlan properties, and persistent user data for partitioned steps.
5873	Clarify when CheckpointAlgorithm#beginCheckpoint is invoked
5919	Spec doesn’t fully describe PartitionPlan override and the use of PartitionMapper
5875	When the first readItem() in a chunk return 'null', is this a zero-item chunk or is this not a new chunk after all
5403	Spec unclear on skipping part of an Exception hierarchy