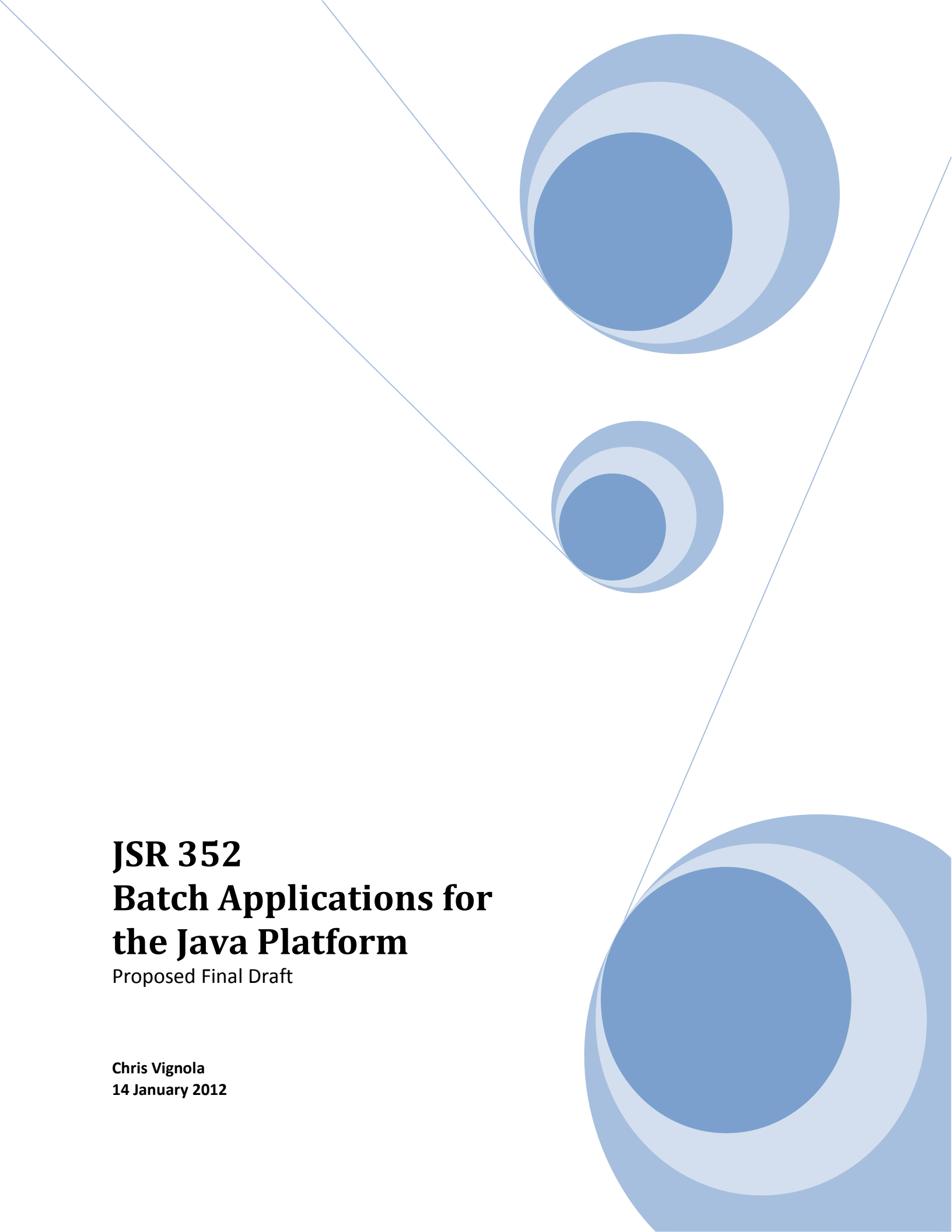


JSR 352
Batch Applications for the Java Platform

Proposed Final Draft

Chris Vignola

14 January 2012

Acknowledgements

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

- Kevin Conner
- Tim Fanelli
- 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.

1 Table of Contents

2 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.

3 Applicability of Specification

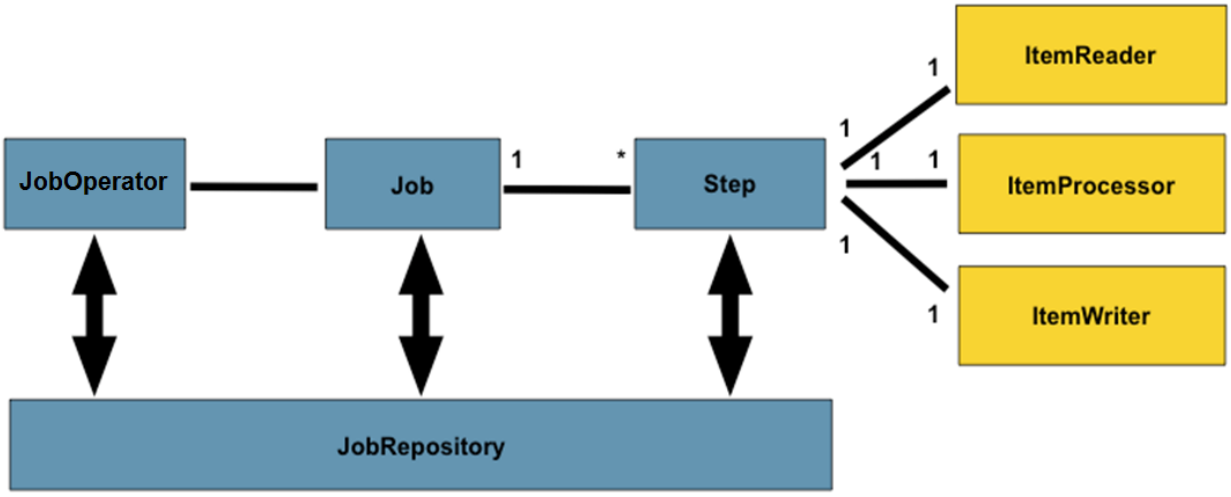
This specification applies to Java SE and Java EE environments. It requires Java 6 or higher.

4 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:

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

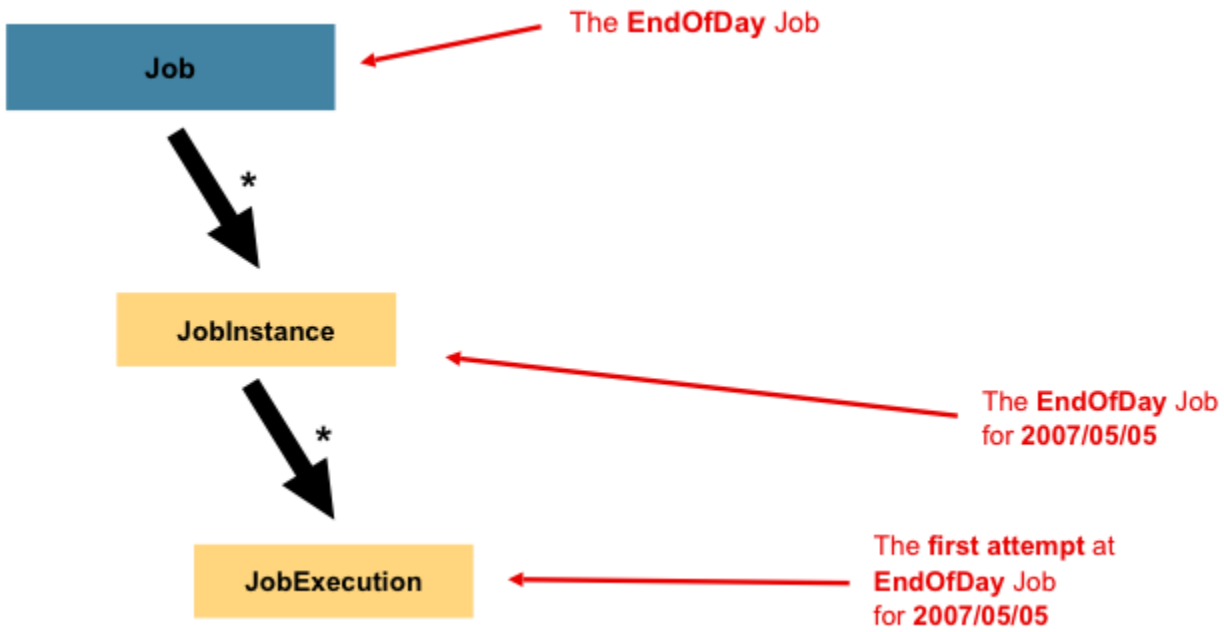
The diagram below is a simplified version of the batch reference architecture that has been used for decades. It provides an overview of the components that make up the domain language of batch processing. This architecture framework is a blueprint that has been proven through decades of implementations 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.



The diagram above highlights the key concepts that make up the domain language of batch. A Job has one to many steps, which has exactly 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).

4.1 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:



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

4.1.1 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 used. Using a new JobInstance will mean 'start from the beginning' and using an existing instance will generally mean 'start from where you left off'.

4.1.2 JobParameters

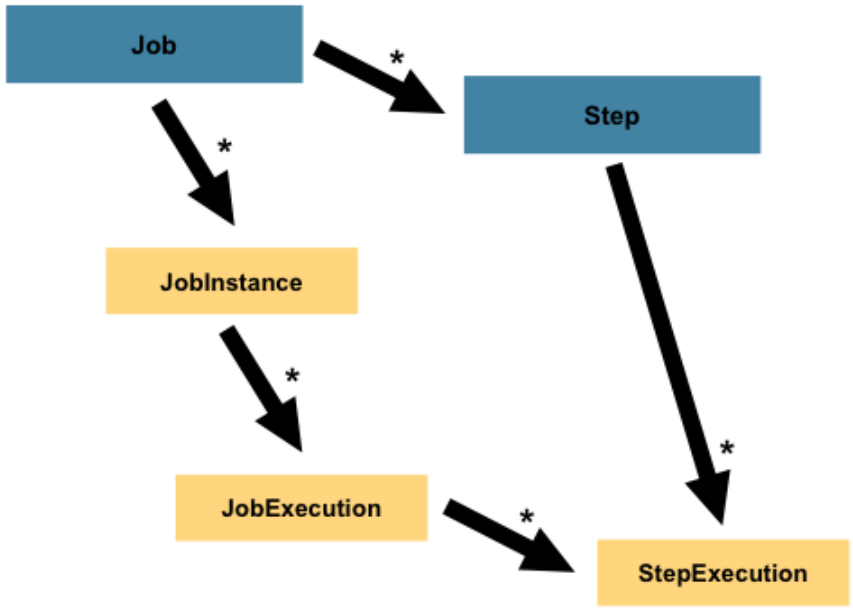
Job parameters can be specified each time a job is started. Job parameters are simply properties in the form of keyword=value. The value is type String. The JobOperator start operation support the specification of job parameters. See section 7.3 for further details on JobOperator.

4.1.3 JobExecution

A JobExecution refers to the technical concept of a single attempt to run a Job. An execution may end in failure or success, but the JobInstance corresponding to a given execution will not be considered complete unless the execution completes successfully. Each time is job is started or restarted, a new JobExecution is created, belonging to the same JobInstance. Note a completed JobExecution cannot be restarted.

4.2 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:



4.2.1 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.

4.3 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 7.3 for more details about JobOperator.

4.4 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 7.8.11, 7.8.12, 7.8.13, respectively.

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

4.5 Item Reader

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 6.1.1.1 for more details about ItemReaders.

4.6 Item Writer

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 6.1.1.3 for more details about ItemWriters.

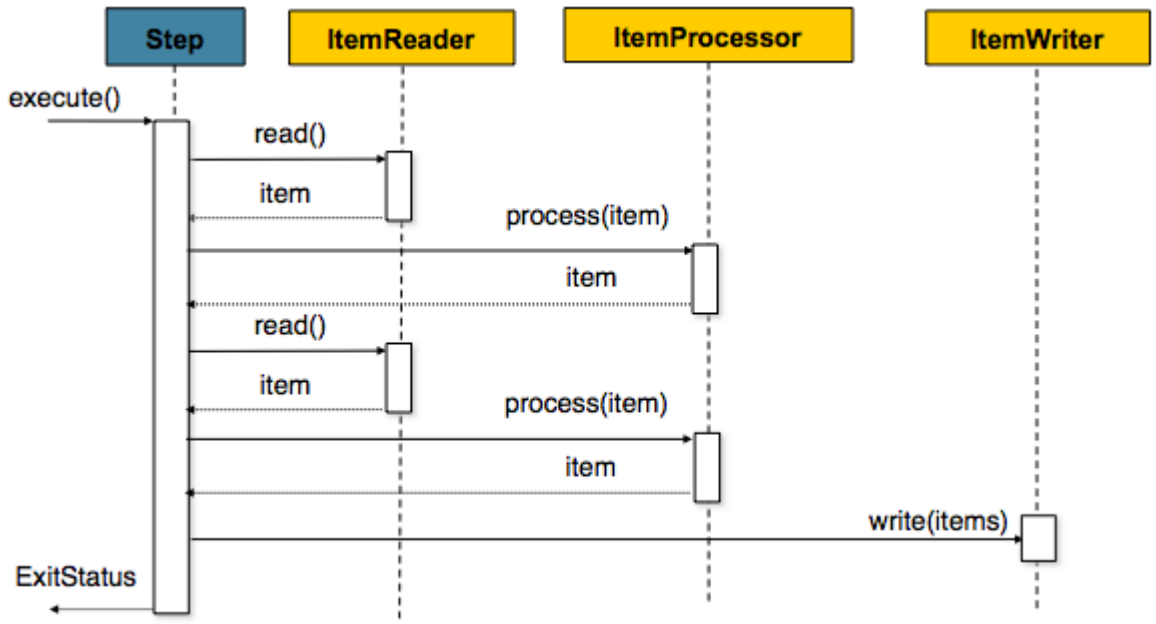
4.7 Item Processor

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 6.1.1.2 for more details about ItemProcessors.

4.8 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.



4.9 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. Ideally, this should be adjustable operationally and not explicitly controlled by the batch application itself.

5 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".

5.1 Job

The 'job' element identifies a job.

Syntax:

<job id="{*name*}" restartable="*{true|false}*" abstract ="*{true|false}*" parent="*{name}*">

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.*
Abstract	Specifies whether or not this job is an abstract job. Abstract jobs are used for job inheritance through the parent attribute. The elements of an abstract job are combined with the elements of the inheriting job. It must specify *true* or *false*. This is an optional attribute. The default is *false.*
Parent	Specifies the id value of a job definition to be incorporated into the current job. It must be a valid XML string value. This is an optional attribute. The default is no parent.

5.1.1 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.

The lifecycle of a job level listener is the life of the job. A job level listener is created at the start of the job and is deleted at the end of the job.

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

Syntax:

<listener ref="{*name*}">

Where:

Ref	Specifies the name of a batch artifact.

5.1.2 Job Level Exception Handling

Any unhandled exception thrown during job processing causes the job to terminate with a batch status of FAILED. The optional JobListener batch artifact may be used to interpose on any exception thrown by a job. The exceptions visible to the JobListener include both exceptions thrown by non-step job-level batch artifacts, which includes listeners and decisions, as well as exceptions thrown by the batch runtime.

5.1.3 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. Step level properties are available through the Job Context runtime object. See section 6.4 for further information about Job Context.

Syntax:

<properties>
    <property name="{*property-name*}" value="{ *name-value*}" save-as="{*output-property-name*}"/>
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

5.2 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.

Syntax:

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

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 reached, 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. Note: next elements cannot be specified such that a loop occurs among steps.
abstract	Specifies whether or not this step is an abstract step. Abstract step are used for step inheritance through the parent attribute. The elements of an abstract step are combined with the elements of the inheriting step. This is an optional attribute. The default is false.
parent	Specifies the id value of a step definition to be used by the current step. This is an optional attribute. The default is no parent.

5.2.1 Chunk

The 'chunk' element identifies a chunk type step. It is a child element of the step element. A chunk is a type of step. A chunk implements the reader-processor-writer pattern of batch. A chunk runs in the scope of a transaction. Transaction management is configurable. See section **Error! Reference source not found.** for further details on configuring transactions. A chunk may be configured so the batch runtime periodically checkpoints the progress of the step by committing the current transaction and starting a new transaction scope. 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}"
        buffer-items="{true|false}"
        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. It is an optional attribute. The default policy is "item".
item-count	Specifies the number of items to process before taking a checkpoint for 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 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.
buffer-items	Specifies whether items are buffered until it is time to take a checkpoint. It must be the value *'true'* or *'false'*. It is an optional attribute. The default is true. When items are buffered, a single call to the item writer is made to write the items immediately before the next checkpoint is taken.
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.

5.2.1.1 Reader

The 'reader' element specifies the item reader for a chunk step. It is a child element of the 'chunk' element. Zero or one reader element may be specified on a chunk step.

Syntax:

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

Where:

ref	Specifies the name of a batch artifact.

5.2.1.1.1 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*}" save-as="{*output-property-name*}"/>
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

5.2.1.2 Processor

The 'processor' element specifies the item processor for a chunk step. It is a child element of the 'chunk' element. Zero or one processor element may be specified on a chunk step.

Syntax:

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

Where:

ref	Specifies the name of a batch artifact.

5.2.1.2.1 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*}" save-as="{*output-property-name*}"/>
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

5.2.1.3 Writer

The 'writer' element specifies the item writer for a chunk step. It is a child element of the 'chunk' element. Zero or one writer element may be specified on a chunk step.

Syntax:

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

Where:

ref	Specifies the name of a batch artifact.

5.2.1.3.1 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}" save-as="{output-property-name}" />
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

5.2.1.4 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 reader, processor, and writer artifacts by configuring exceptions to skip or to retry. 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, and writer batch artifacts of a chunk type step. The total number of skips is set by the skip-limit attribute on the chunk element. See section 5.2.1 for details on the chunk element.

The optional Skip Listener batch artifact can be configured to the step. A Skip Listener can receive control for a skip in the reader, processor, or writer. See section 6.2.7 for details on the Skip Listener batch artifact.

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. 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. 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.

5.2.1.4.1 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. The total number of retry attempts is set by the retry-limit attribute on the chunk element. Exceptions thrown from any of the batch artifacts, reader, processor, or writer contribute to the limit tally. See section 5.2.1 for details on the chunk element.

The optional Retry Listener batch artifact can be configured on the step. A Retry Listener receives control before the operation that threw the original exception is retried. See section 6.2.8 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. 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.The exclude child element is optional. However, when specified, the class attribute is required.

5.2.1.5 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 The default retry behavior can be overridden by configuring the no-rollback-exception-classes element. See section 5.2.1.6 for more information on specifying no-rollback exceptions.

5.2.1.6 Preventing Rollback During Retry

The no-rollback-exception-classes element specifies a list of exceptions that override the default behavior of rollback for exceptions. This element is a child element of the chunk element. If an exception is thrown the default behavior is to rollback and end the job. If an exception is specified as a no-rollback exception, then no rollback occurs and the current operation is retried. RetryListeners, if configured, are invoked. See section 6.2.8 for details on the Retry Listener batch artifact.

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. 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.The exclude child element is optional. However, when specified, the class attribute is required.

5.2.1.7 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 6.1.1.4 for further information about custom checkpoint algorithms.

Syntax:

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

Where:

ref	Specifies the name of a batch artifact.

5.2.1.7.1 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*}" save-as="{*output-property-name*}"/>
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

5.2.2 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 6.1.2 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.

5.2.2.1 Batchlet Exception Handling

When any batch artifact that is part of a batchlet type step throws an exception to the Batch Runtime, the job execution ends with a batch status of FAILED.

5.2.2.2 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*}" save-as="{*output-property-name*}"/>
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

5.2.3 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 Step Context runtime object. See section 6.4 for further information about Step Context.

Syntax:

<properties>
    <property name="{*property-name*}" value="{ *name-value*}" save-as="{*output-property-name*}"/>
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

5.2.4 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

The lifecycle of a step level listener is the lifecycle of the step to which the listener is defined. The listener is created at the start of the step and is deleted at the end of the step.

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

Syntax:

<listener ref="{*name*}">

If more than one listener is specified:

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

Where:

ref	Specifies the name of a batch artifact.

5.2.4.1 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*}" save-as="{*output-property-name*}"/>
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

Example:

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

5.2.5 Step Sequence

The first step, flow, or split defines the first step (flow or split) to execute for a given Job XML. The 'next' attribute on the step, flow, or split defines what executes next. The next attribute may specify a step, flow, split, or decision. The next attribute is supported on step, flow, and split elements. Steps may alternatively use the "next" *element* to specify what executes next. The next attribute and next element may not be specified together in the same step.

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.

Multiple next elements may be specified for a given step. When multiple next elements are specified, they are ordered from most specific to least specific. The current exit status value is then compared against each next element until a match is found. When the next element is specified, all possible exit status values must be accounted for. If an exit status occurs that does not match one of the next elements, the job ends in the failed state.

5.2.6 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 .../>
```

```
        <partition .../>
...</step>
```

5.2.6.1 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 6.5.1 for further details on partition mapper.

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"/>
                ...
</step>
```

5.2.6.2 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 6.5.1 for further information on partition mapper.

Syntax:

<properties partition="*partition-number*">
    <property name="{*property-name*}" value="{ *name-value*}" save-as="{*output-property-name*}"/>

Where:

partition-number	Specifies the logical partition number to which the specified properties apply. This must be a positive integer value, starting at 0.
name	Specifies a unique property name. 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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

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>
    </properties>
</partition
```

5.2.6.3 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 6.5.1 for further information. Note a 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>
```

5.2.6.3.1 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}" save-as="{output-property-name}"/>
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

5.2.6.4 Partition Reducer

A partitioned step executes in the context of a partition reducer. A partition reducer provides a kind of unit of work demarcation around the execution of the collection of threads. Programmatic interception of the partition reducer lifecycle is possible by providing a partition reducer. The PartitionReducer element specifies a reference to a PartitionReducer batch artifact; see section 6.5.2 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>
```

5.2.6.4.1 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}" save-as="{output-property-name}"/>
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

5.2.6.5 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 step; it is invoked once at the end of step for batchlet type steps. A collector returns a Java Externalizable object, which is delivered to the step's Partition Analyzer. See section 6.5.4 for further information about the Partition Analyzer. The collector element specifies a reference to a PartitionCollector batch artifact; see section 6.5.3 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>
```

5.2.6.5.1 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*}" save-as="{*output-property-name*}"/>
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

5.2.6.6 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 6.5.4 for further information.

Syntax:

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

Where:

ref	Specifies the name of a batch artifact.

Example:

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

5.2.6.6.1 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*}" save-as="{*output-property-name*}"/>
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

5.2.7 Step Exception Handling

Any unhandled exception thrown during step processing causes the step and job to terminate with a batch status of FAILED. The optional StepListener batch artifact may be used to interpose on any exception thrown by a step. The exceptions visible to the StepListener include both exceptions thrown by step-level batch artifacts as well as exceptions thrown by the batch runtime.

5.3 Flow

A flow defines a sequence of steps 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. The steps within a flow may only transition among themselves; steps in a flow may not transition to elements outside of the flow. Besides steps, a flow may contain flow, decision, and split elements See section 5.6 for more on decisions. See section 5.4 for more on splits.

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 elements cannot be specified such that a loop occurs among steps.

5.4 Split

A split defines a set of flows that execute concurrently. See section 5.3 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, split, decision, or another flow. The flows within a split may only transition among themselves; flows in a split may not transition to elements outside of the split. A split may not contain a split, nor may it contain steps that are not part of a flow.

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 elements cannot be specified such that a loop occurs among steps.

5.5 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".

Each step, flow, split, and job ends 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. By default, the exit status is the same as the batch status. Note if a batch artifact explicitly sets the exit status that overrides the default. The batch and exit status values are available in the JobContext and StepContext runtime objects. 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 stop processing has commenced by the batch runtime.
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 will finish execution under the following conditions:

1. A job-level execution element (step, flow, or split) that does not specify a "next" attribute finishes execution. In this case, the batch status is set to COMPLETED.

2. A step throws an unresolved exception. In this case, the batch status is set to FAILED. 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 split 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. it is initially the same as the batch and exit status of the last execution element (step, flow, split) to run

2. batch and exit status can be overridden by a decision element

3. batch and exit status for the job can be overridden by a job listener as the job ends. This overrides all else.

5.5.1 Batch and Exit Status for Steps

Step batch status is set by the batch runtime at the conclusion of the step. 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 6.4 for further information about the StepContext object.

Step termination may also be configured in the Job XML using the fail, end, and stop elements. These elements can also set the exit status. Note that setting the exit status programmatically through the StepContext object overrides setting the exit status through any of these element types. The following sections explain more more about these element types.

5.5.1.1 Fail Element

The fail element is used to terminate the job at the current step. The batch status is set to FAILED.

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 step. It must be a valid XML string value. This is an optional attribute. If not specified, the exit status is unchanged.

Example:

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

5.5.1.2 End Element

The end element is used to terminate the job at the current step. The batch status is set to COMPLETED.

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 step. It must be a valid XML string value. This is an optional attribute. If not specified, the exit status is unchanged.

Example:

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

5.5.1.3 Stop Element

The stop element is used to terminate the job at the current step. The stop element is only considered if step batch status is COMPLETED. If the stop element matches the exit status, the batch status is then set to STOPPED. The stop element may also optionally set the job-level step, flow, or split at which the job will restart once restarted.

<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 step. It must be a valid XML string value. This is an optional attribute. If not specified, the exit status is unchanged.
restart	Specifies the job-level step, flow, or split on which to restart when the job is restarted. It must be a valid XML string value. This is an optional attribute. By default it is the current step.

Example:

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

5.5.2 Batch and Exit Status for Flows

A flow finishes when its final step finishes. The final step in a flow is a step with no "next" attribute. A flow may also finish when any of its steps finishes due to an unhandled exception, or due to an end, fail, or stop element. A flow also has a batch and exit status. The batch and exit status of a flow is the batch and exit status of the last step to execute. The batch and exit status of a flow is available in the FlowContext. See section 6.4.1.1 for FlowContext.

5.5.3 Batch and Exit Status for Splits

A split finishes when its final flow finishes. A split may also finish when any of its flows finishes due to an unhandled exception, or due to an end, fail, or stop element by any of its constituent steps. All other still-running parallel flows end with STOPPED batch status. A split also has a batch and exit status. The batch and exit status of a split is based on the outcome of the individual flows that make up the split as follows:

- if all flows have a batch status of COMPLETED, then the batch status of the split is COMPLETED
- if any flow has a batch status of FAILED, then the batch status of the split is FAILED.

- if no flow has a batch status of FAILED, but at least one flow has a batch status of STOPPED, then the batch status of the split is STOPPED.

The batch and exit status of a split is available in the SplitContext. See section 6.4.1.1 for SplitContext.

5.6 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 split or from a job-level step or flow. A decision must supply a decider batch artifact (see section 6.6). The decider's purpose is to decide the next transition. The decision uses stop, fail, end, and next elements to select the next transition. The decider may set a new exit status to facilitate the transition choice.

The decider has access to JobContext and to the context of last job-level execution element that finished. That means a StepContext is available if the decision was transitioned to from a step; a FlowContext if from a flow; and a SplitContext if from a split. The context objects hold the batch and exit status values for the last execution element that finished. The decider may use this context information to make its decision.

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>
```

5.6.1 Fail Element

The fail element is a child element of the decision element. A decision element may contain any number of fail elements. The fail element is used to terminate the job at the current decision. The batch status is set to FAILED.

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 exit status for the job. It must be a valid XML string value. This is an optional attribute. If not specified, the job exit status is unchanged.

Example:

```
<decision id="AfterFlow1" ref="MyDecider" >
        <fail on="FAILED" exit-status="DO NOT RESTART"/>
</decision>
```

5.6.2 End Element

The end element is a child element of the decision element. A decision element may contain any number of end elements. The end element is used to terminate the job at the current decision. The batch status is set to COMPLETED.

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 exit status for the job. It must be a valid XML string value. This is an optional attribute. If not specified, the job exit status is unchanged.

Example:

```
<decision id="AfterFlow1" ref="MyDecider" >
        <end on="COMPLETED" exit-status="EARLY COMPLETION"/>
</decision>
```

5.6.3 Stop Element

The stop element is a child element of the decision element. A decision element may contain any number of stop elements. The stop element is used to terminate the job at the current decision. The batch status is set to STOPPED. The stop element may also optionally set the job-level step, flow, or split at which the job will restart once restarted.

<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 exit status is unchanged.
restart	Specifies the job-level step, flow, or split on which to restart when the job is restarted. It must be a valid XML string value. This is an optional attribute. By default it is the last non-COMPLETED execution element (step, flow, split) to execute - i.e. the execution element that transitioned to this decision element. If there is no last non-COMPLETED execution element and restart is not specified, then there is no restart point for this job and when it gets restarted it will be marked COMPLETED.

Example:

```
<decision id="AfterFlow1" ref="MyDecider" >
        <stop on="COMPLETED" exit-status="RESTART" restart="Step2"/>
</decision>
```

5.6.4 Next Element

The next element is a child element of the decision element. A decision element may contain any number of next elements. The next element is used to transition the current decision to the next execution element. <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 job-level step, flow, or split 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:

```
<decision id="AfterFlow1" ref="MyDecider" >
        <next on="*" to="Step2" />
</decision>
```

5.6.5 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}" save-as="{output-property-name}"/>
</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.
save-as	Specifies the job-unique property name with which this property's value will be saved to the job repository at end of scope.

5.6.6 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.

5.7 Job XML Substitution

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

<attribute-value> ::= ' " ' <value-expression> ' " '

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

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

<operator1> ::= "jobParameters" "[" <single-quoted-string-literal> "]"

<operator2> ::= "jobProperties" "[" <single-quoted-string-literal> "]"

<operator3> ::= "systemProperties" "[" <single-quoted-string-literal> "]"

<operator4> ::= "partitionPlan" "[" <single-quoted-string-literal> "]"

<operator5> ::= "saved" "[" <single-quoted-string-literal> "]"

<single-quoted-string-literal> ::= " ' " <string-literal> " ' "

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

Hiearchy Rule

#{jobProperties['some.property']} resolved as follows:

1. check enclosing nesting level and use 'some.property' from that level if found
2. repeat step 1 until 'some.property' found or no further nesting levels
3. if some.property not found result is null

Examples:

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

  infile.name="in.txt"

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

  infile.name= value of infile.name job parameter

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

  infile.name= value of infile.name system property

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

  infile.name= value of infile.name job property

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

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

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

  infile.name = value of infile.name job parameter or "in.txt" if infile.name job parameter is null

<property name="infile.name" value="#{saved['step1.infile.name']}" />

infile.name= value of step1.infile.name "save as" property value from another step.
Example in context and in combination:

```
<step>
	<properties>
		<!-- set property infile.name with infile.name job parameter or "in.txt" -->
		<property name="infile.name" value="#{jobParameters['infile.name']}?:in.txt" />

		<!-- set property infile.path with  infile.path job parameter or "/tmp" -->
		<property name="infile.path"
value="#{jobParameters['infile.path']}?:#{systemProperties['file.separator']}tmp" />
	</properties>
	<batchlet ... >
		<properties>
			<!-- set property infile.name based on property values of infile.path and
infile.name - default is /tmp/in.txt -->
			<property name="infile.name"
value="#{jobProperties[infile.path]}#{systemProperties['file.separator']}#{jobProperties['infile.name']}" />
		</properties>
	</batchlet>
</step>
```

5.8 Job XML Inheritance

Job XML supports inheritance of jobs and steps. The following rules apply:

1. Jobs can inherit jobs:
    1. Parent job steps, flows, and splits are not inherited.
    2. The child's job element overrides the parent's.

2. Steps can inherit steps:
    1. The child's step element overrides the parent's.
3. For all other elements:
    1. Non-intersecting elements are merged.
    2. Non-intersecting attributes on intersecting elements are merged.
    3. Intersecting attributes on intersecting elements result in child overrides parent attribute.
    4. Lists support merge=true|false attribute.

Example:

parent-job.xml:

```
<job id="step.auditor">
        <listeners>
                <listener ref="StepAuditor"/>
        </listeners>
</job>
```

child-job.xml:

```
<job id="Job1" parent="step.auditor"/>
        <step id="step1">
                <chunk reader="MyReader" processor="MyProcessor" writer="MyWriter/>
        </step>
</job>
```

When "Job1" is started, the effective Job XML:

```
<job id="Job1"/>
        <listeners>
                <listener ref="StepAuditor"/>
        </listeners>
        <step id="step1">
                <chunk reader="MyReader" processor="MyProcessor" writer="MyWriter/>
        </step>
</job>
```

Not only are the child's elements combined with that of the parent's, but the attributes contained by those child elements are also combined, as illustrated in the following example:

parent-step.xml:

```
<job id="Parent-Step"/>
        <step id="MyProcessor" abstract="true">
                <chunk processor="MyProcessor">
                        <properties>
                                <property name="audit" value="true"/>
                        </properties>
                </chunk>
        </step>

</job>
```

Note: abstract="true" must be specified if the parent element does not include all required elements and attributes.

child-job.xml:

```
<job id="Job2" />
        <step id="step1" parent="MyProcessor">
                <chunk reader="MyReader" writer="MyWriter/>
        </step>
</job>
```

When "Job2" is started, the effective Job XML:

```
<job id="Job2"/>
        <step id="step1">
                <chunk reader="MyReader" processor="MyProcessor" writer="MyWriter>
                        <properties>
                                <property name="audit" value="true"/>
                        </properties>
                </chunk>
        </step>
</job>
```

Note: a single Job XML file may contain

5.8.1 Merging Lists

When list elements are included in the inheritance process, there is a choice of overriding the parent list or merging with it. All list elements support this capability. List elements are <properties> and <listeners>.

Example:

parent-step.xml:

```
<job id="Parent-Step"/>
        <step id="step.auditor" abstract="true">
                <step id="MyProcessor">
                        <chunk processor="MyProcessor">
                                <properties>
                                        <property name="audit" value="true"/>
                                </properties>
                        </chunk>
                </step>
        </step>

</job>
```

child-job.xml:

```
<job id="Job3" />
        <step id="step1" parent="MyProcessor">
                <chunk reader="MyReader" writer="MyWriter>
                        <properties merge="true">
                                <property name="infile" value="/tmp/input.txt" target="reader"/>
                                <property name="outfile" value="/tmp/output.txt" target="writer"/>
                        </properties>
                </chunk>
        </step>
</job>
```

When "Job3" is started, the effective Job XML:

```
<job id="Job3"/>
        <step id="step1">
                <chunk reader="MyReader" processor="MyProcessor" writer="MyWriter>
                        <properties>
                                <property name="audit" value="true"/>
```

```
                    <property name="infile" value="/tmp/input.txt" target="reader"/>
                    <property name="outfile" value="/tmp/output.txt" target="writer"/>
                </properties>
            </chunk>
        </step>
</job>
```

Note: a single Job XML document may contain one non-abstract job definition and zero or more abstract job definitions.

5.8.2 XML Inheritance and Substitution

Inheritance is resolved first, then substitution.

5.9 Saved Batch Properties

There exists a persistent job-level properties object for each job instance into which a job execution may store property end state values. The end state of a property is the value of that property at the end of scope in which the property is defined.

The end state of a property defined at job level or to a job level batch artifact is the value of that property at end of job execution. The end state of a property defined at step level or to a step level batch artifact is the value of that property at end of step execution.

Saved batch properties support the use case where the end state value of a property from one step is used as a substitution input to a subsequent step, including across restarts. It also supports the use case where a job level end state property value is available across job restarts.

Any batch property can be a saved property by using the save-as attribute on the property element - e.g.

<property name="MyProperty" value="SomeInitialValue" save-as="MySavedPropertyName" />

saves the final value of property "MyProperty" to the Job Repository for the current JobInstance with the name "MySavedPropertyName" at end of defining scope.

Elsewhere in the same Job XML, the end state value of "MyProperty" can be obtained through substitution via the Job XML substitution expression:

#{saved['MySavedPropertyName']}

Saved batch properties are stored dynamically during job execution. Similarly, saved batch property substitution references are dynamically resolved at runtime. This means it is the dynamic, runtime value of the property that is saved and substituted - not the static value of the property known at the time the job is first started.

6 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. The @BatchProperty and @BatchContext qualifiers work with @Inject to enable applications access to job and runtime information. See sections 6.3.1 and 6.4.1, respectively, for further information on these two annotations.

6.1 Steps

A batch step is either chunk or batchlet.

6.1.1 Chunk

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

6.1.1.1 ItemReader Interface

An ItemReader is used to read a stream of 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:

```
package javax.batch.api;

import java.io.Externalizable;
/**
 *
 * ItemReader defines the batch artifact that reads from a
 * stream of items for chunk processing.
 *
 * @param <T> specifies the item type returned by this reader.
 */
public interface ItemReader <T> {

      /**
       * 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 instructs the reader where to reposition the
       * stream upon job restart. A checkpoint value of null
       * means reposition from the start of stream or rely on
       * an application managed means of determining whether to
       * position for start or restart. The persistent area of
       * the StepContext may be used to implement application
       * managed stream repositioning.
       *
       * @param checkpoint specifies the last checkpoint
       * @throws Exception is thrown for any errors.
       */
      public void open(Externalizable checkpoint) throws Exception;
      /**
       * The close method marks the end of use of the item
       * stream. The reader is free to do any cleanup
       * necessary on the stream.
       * @throws Exception is thrown for any errors.
       */
      public void close() throws Exception;
      /**
       * The readItem method returns the next item from the
       * stream.  It returns null to indicate end of stream.
       * @return next item or null
       * @throws Exception is thrown for any errors.
       */
      public T readItem() throws Exception;
      /**
       * The checkpointInfo method returns the current
       * checkpoint position for this reader. It is
```

```
         * called before a chunk checkpoint is committed.
         * @return checkpoint data
         * @throws Exception is thrown for any errors.
         */
        public Externalizable checkpointInfo() throws Exception;

}


package javax.batch.api;

import java.io.Externalizable;

/**
 * The AbstractItemReader provides default implementations
 * of optional methods.
 *
 * @param <T> specifies the item type read by the
 * ItemReader.
 */
public abstract class AbstractItemReader<T> implements ItemReader<T> {
        /**
         * Optional method.
         *
         * Implement this method if the ItemReader 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(Externalizable checkpoint) throws Exception { }
        /**
         * Optional method.
         *
         * Implement this method if the ItemReader requires
         * any close time processing.
         * The default implementation does nothing.
         *
         * @throws Exception (or subclass) if an error occurs.
         */
        @Override
        public void close() throws Exception { }
        /**
         * Required method.
         *
         * Implement read logic for the ItemReader in this
         * method.
         *
         * @return next item or null
         * @throws Exception (or subclass) if an error occurs.
         */
        @Override
        public abstract T readItem() throws Exception;
```

```
	/**
	 * Optional method.
	 *
	 * Implement this method if the ItemReader supports
	 * checkpoints.
	 * The default implementation returns null.
	 *
	 * @return checkpoint data
	 * @throws Exception (or subclass) if an error occurs.
	 */
	@Override
	public Externalizable checkpointInfo() throws Exception {
		return null;
	}
}
```

6.1.1.2 ItemProcessor Interface

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

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

```
package javax.batch.api;

/**
 * ItemProcessor is used in chunk processing
 * to operate on an input item and produce
 * an output item.
 *
 */
public interface ItemProcessor <T, R> {

	/**
	 * 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. If processItem
	 * returns null, no value is passed onto the chunk's
	 * item writer.  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 R processItem(T item) throws Exception;

}
```

6.1.1.3 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:

```
package javax.batch.api;

import java.io.Externalizable;
import java.util.List;

/**
 *
 * ItemWriter defines the batch artifact that writes to a
 * stream of items for chunk processing.
 *
 * @param <T> specifies the item type returned by this reader.
 */

public interface ItemWriter <T> {

      /**
       * 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 instructs the writer where to reposition the
       * stream upon job restart. A checkpoint value of null
       * means reposition from the start of stream or rely on
       * an application managed means of determining whether to
       * position for start or restart. The persistent area of
       * the StepContext may be used to implement application
       * managed stream repositioning.
       *
       * @param checkpoint specifies the last checkpoint
       * @throws Exception is thrown for any errors.
       */
      public void open(Externalizable checkpoint) throws Exception;
      /**
       * The close method marks the end of use of the item
       * stream. The writer is free to do any cleanup
       * necessary on the stream.
       * @throws Exception is thrown for any errors.
       */
      public void close() throws Exception;
```

```
	/**
	 * The writeItems method writes a list of item
	 * for the current chunk to the item writer
	 * stream.
	 * @param items specifies the list of items to write.
	 * @throws Exception is thrown for any errors.
	 */
	public void writeItems(List<T> items) throws Exception;

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

}


package javax.batch.api;

import java.io.Externalizable;
import java.util.List;

/**
 * The AbstractItemWriter provides default implementations
 * of optional methods.
 *
 * @param <T> specifies the item type written by the
 * ItemWriter.
 */
public abstract class AbstractItemWriter<T> implements ItemWriter<T> {

	/**
	 * Optional method.
	 *
	 * Implement 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(Externalizable checkpoint) throws Exception { }
	/**
	 * Optional method.
	 *
	 * Implement 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 { }
        /**
         * Required method.
         *
         * 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<T> items) throws Exception;
        /**
         * Optional method.
         *
         * Implement 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 Externalizable checkpointInfo() throws Exception {
              return null;
        }

}
```

6.1.1.4 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:

```
package javax.batch.api;

/**
 * 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 checkpoint transaction begins. This
method returns an integer value,
        * which is the timeout value that will be used for the next checkpoint
        * transaction.  This method is useful to automate the setting of the
        * checkpoint timeout based on factors known outside the job
        * definition.
        *
        * @param timeout specifies the checkpoint timeout value for the   *
next checkpoint interval.
        * @return the timeout interval 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.
        * @throws Exception thrown for any errors.
        */
       public void beginCheckpoint() throws Exception;
       /**
        * The isReadyToCheckpoint method is invoked by
        * the batch runtime after each item read 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
        * current checkpoint ends.
        * @throws Exception thrown for any errors.
        */
       public void endCheckpoint() throws Exception;

}

package javax.batch.api;

/**
 * The AbstractCheckpointAlgorithm provides default
 * implementations of optional methods.
 */
public abstract class AbstractCheckpointAlgorithm implements
             CheckpointAlgorithm {
       /**
        * Optional method.
        *
        * Implement this method if the CheckpointAlgorithm
        * establishes a checkpoint timeout.
        * The default implementation returns 0, which means
        * maximum permissible timeout allowed by
        * runtime environment.
        *
```

```
         * @return the timeout interval to use for the next
         * checkpoint interval
         * @throws Exception (or subclass) if an error occurs.
         */
        @Override
        public int checkpointTimeout(int timeout) throws Exception {
              return 0;
        }
        /**
         * Optional method.
         *
         * Implement this method for the CheckpointAlgorithm
         * to do something before a checkpoint begins.
         * The default implementation does nothing.
         *
         * @throws Exception (or subclass) if an error occurs.
         */
        @Override
        public void beginCheckpoint() throws Exception {}
        /**
         * Required method.
         *
         * This method implements the logic 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;
        /**
         * Optional method.
         *
         * Implement this method for the CheckpointAlgorithm
         * to do something after a checkpoint ends.
         * The default implementation does nothing.
         *
         * @throws Exception (or subclass) if an error occurs.
         */
        @Override
        public void endCheckpoint() throws Exception {}
}
```

6.1.2 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:

```
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;
}
```

```
package javax.batch.api;
/**
 * The AbstractBatchlet provides default
 * implementations of optional methods.
 */
public abstract class AbstractBatchlet implements Batchlet {
	/**
	 * Required method.
	 *
	 * 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;

        /**
         * Optional method.
         *
         * Implement 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 {}
}
```

Programming Guideline

A well designed batchlet stops gracefully when the JobOperator.stop operation is invoked. See section 8.13 for further information about stop processing.

6.2 Listeners

Use Listeners to interpose on batch execution.

6.2.1 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:

```
package javax.batch.api;

/**
 * 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;
	/**
	 * The onException method receives control
	 * when an exception is thrown by job
	 * processing.
	 * @param ex is the exception thrown.
	 */
	public void onException(Exception ex);
}
```

```
package javax.batch.api;
/**
 * The AbstractJobListener provides default
 * implementations of optional methods.
 */
public abstract class AbstractJobListener implements JobListener {
	/**
	 * Optional method.
	 *
	 * Implement 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 {}
	/**
	 * Optional method.
	 *
	 * Implement 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 {}
	/**
	 * Optional method.
	 *
```

```
         * Implement this method if the JobListener
         * will do something when an exception is
         * thrown during job processing.
         * @param ex is the exception thrown.
         */
        @Override
        public void onException(Exception ex) {}
}
```

6.2.2 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:

```
package javax.batch.api;

/**
 * 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;
        /**
         * The onException method receives control
         * when an exception is thrown during
         * step processing.
         * @param ex is the exception thrown.
         */
        public void onException(Exception ex);
}

package javax.batch.api;
/**
 * The AbstractStepListener provides default
 * implementations of optional methods.
 */
public abstract class AbstractStepListener implements StepListener {
        /**
```

```
         * Optional method.
         *
         * Implement 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 {}
        /**
         * Optional method.
         *
         * Implement 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 {}
        /**
         * Optional method.
         *
         * Implement this method if the StepListener
         * will do something when an exception is
         * thrown during step processing.
         * @param ex is the exception thrown.
         */
        @Override
        public void onException(Exception ex) {}
}
```

6.2.3 ChunkListener Interface

A chunk listener can receive control at beginning and end of chunk processing and before and after a chunk checkpoint is taken. The ChunkListener interface may be used to implement an ChunkListener batch artifact:

```
package javax.batch.api;

/**
 * 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 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;

}

package javax.batch.api;
/**
 * The AbstractChunkListener provides default
 * implementations of optional methods.
 */
public abstract class AbstractChunkListener implements ChunkListener {
        /**
         * Optional method.
         *
         * Implement 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 {}
        /**
         * Optional method.
         *
         * Implement 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 {}
}
```

6.2.4 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:

```
package javax.batch.api;

/**
 * ItemReadListener intercepts item reader
 * processing.
 *
 * @param <T> specifies the type processed by
 * an item reader.
 */
public interface ItemReadListener<T> {

      /**
       * 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(T 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;
}

package javax.batch.api;
/**
 * The AbstractItemReadListener provides default
 * implementations of optional methods.
 *
 * @param <T> specifies the item type read by the
 * ItemReader paired with this ItemReadListener.
 */
public abstract class AbstractItemReadListener<T> implements
            ItemReadListener<T> {
      /**
       * Optional method.
       *
       * Implement 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 {}
        /**
         * Optional method.
         *
         * Implement 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(T item) throws Exception {}
        /**
         * Optional method.
         *
         * Implement 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 {}
}
```

6.2.5 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:

```
package javax.batch.api;

/**
 * ItemProcessListener intercepts item processing.
 *
 * @param <T> specifies the type processed by
 * an item processor.
 * @param <R> specifies the type returned by
 * an item processor.
 */
public interface ItemProcessListener <T, R> {

        /**
         * 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(T 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 about to be processed.
         * @param result specifies the item to pass to the item writer.
         * @throws Exception if an error occurs.
         */
        public void afterProcess(T item, R result) throws Exception;
        /**
         * The onProcessError a method receives control after an item
* processor processItem method
         * throws an exception.  This method receives the exception and the
         * input item.
         * @param item specifies the item about to be processed.
         * @param ex specifies the exception thrown by the item processor.
         * @throws Exception
         */
        public void onProcessError(T item, Exception ex) throws Exception;

}

package javax.batch.api;
/**
 * The AbstractItemProcessListener provides default
 * implementations of optional methods.
 *
 * @param <T> specifies the item type read by the
 * ItemProcessor paired with this ItemProcessListener.
 *
 * @param <R> specifies the item type returned by the
 * ItemProcessor paired with this ItemProcessListener.
 */
public abstract class AbstractItemProcessListener<T, R> implements
             ItemProcessListener<T, R> {
        /**
         * Optional method.
         *
         * Implement 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(T item) throws Exception {}
        /**
         * Optional method.
         *
```

```
         * Implement 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(T item, R result) throws Exception {}
        /**
         * Optional method.
         *
         * Implement 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(T item, Exception ex) throws Exception {}
}
```

6.2.6 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:

```
package javax.batch.api;

import java.util.List;

/**
 * ItemWriteListener intercepts item writer
 * processing.
 *
 * @param <T> specifies the type processed by
 * an item reader.
 */
public interface ItemWriteListener <T> {

        /**
         * 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
```

```
         * reader as an input.
         * @param items specifies the items about to be
         * written.
         * @throws Exception is thrown if an error occurs.
         */
        public void beforeWrite(List<T> 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 reader as an input.
         * @param items specifies the items about to be
         * written.
         * @throws Exception is thrown if an error occurs.
         */
        public void afterWrite(List<T> 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 reader as an input.
         * @param items specifies the items about to be
         * written.
         * @param ex specifies the exception thrown by the item
         * writer.
         * @throws Exception is thrown if an error occurs.
         */
        public void onWriteError(List<T> items, Exception ex) throws Exception;
}

package javax.batch.api;

import java.util.List;
/**
 * The AbstractItemWriteListener provides default
 * implementations of optional methods.
 *
 * @param <T> specifies the item type written by the
 * ItemWriter paired with this ItemWriteListener.
 */
public abstract class AstractItemWriteListener<T> implements
            ItemWriteListener<T> {
    /**
         * Optional method.
         *
         * Implement this method if the ItemWriteListener
         * will do something before the items are written.
         * The default implementation does nothing.
         *
         * @param items specifies the items about to be
         * written.
         * @throws Exception (or subclass) if an error occurs.
         */
        @Override
        public void beforeWrite(List<T> items) throws Exception {}
```

```
    /**
       * Optional method.
       *
       * Implement this method if the ItemWriteListener
       * will do something after the items are written.
       * The default implementation does nothing.
       *
       * @param items specifies the items about to be
       * written.
       * @throws Exception (or subclass) if an error occurs.
       */
      @Override
      public void afterWrite(List<T> items) throws Exception {}
    /**
       * Optional method.
       *
       * Implement this method if the ItemWriteListener
       * will do something when the ItemWriter writeItems
       * method throws an exception.
       * The default implementation does nothing.
       *
       * @param items specifies the items about to be
       * written.
       * @param ex specifies the exception thrown by the item
       * writer.
       * @throws Exception (or subclass) if an error occurs.
       */
      @Override
      public void onWriteError(List<T> items, Exception ex) throws Exception
{}
}
```

6.2.7 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:

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

package javax.batch.api;
/**
 * SkipProcessListener intercepts skippable
 * itemProcess exception handling.
 *
 * @param <T> specifies the item type processed
 * by the ItemProcessor.
 */
public interface SkipProcessListener <T> {
	/**
	 * 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(T item, Exception ex) throws Exception;
}

package javax.batch.api;

import java.util.List;
/**
 * SkipWriteListener intercepts skippable
 * itemWriter exception handling.
 *
 * @param <T> specifies the item type processed
 * by the itemWriter.
 */
public interface SkipWriteListener <T> {
	/**
	 * 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<T> items, Exception ex) throws
Exception;
}
```

6.2.8 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:

```
package javax.batch.api;
/**
 * RetryReadListener intercepts retry processing for
 * an ItemReader.
 */
public interface RetryReadListener <T> {
      /**
       * 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;
}

package javax.batch.api;
/**
 * RetryProcessListener intercepts retry processing for
 * an ItemProcessor.
 *
 * @param <T> specifies the type of item processed
 * by the ItemProcessor paired with this listener.
 */
public interface RetryProcessListener <T> {
      /**
       * 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(T item, Exception ex) throws
Exception;
}

package javax.batch.api;
import java.util.List;
```

```
/**
 * RetryWriteListener intercepts retry processing for
 * an ItemWriter.
 *
 * @param <T> specifies the type of item written
 * by the ItemWriter paired with this listener.
 */
public interface RetryWriteListener <T> {
      /**
       * 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<T> items, Exception ex) throws
Exception;
}
```

6.3 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. However batch properties values can be formed from other properties according to Job XML Substitution Rules. See section 5.7 for further information on substitution.

6.3.1 @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..

Syntax:

package: javax.batch.annotation

@Inject @BatchProperty(name="<property-name>") String <field-name>[=<default-initializer>];

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.
<default-initializer>	is the Java initializer for this field. This is the property field default value if no value is specified for this property on this artifact in the Job XML.

Example:

```
public class MyItemReaderImpl {
        @BatchProperty String fname="/tmp/input.txt";
}
```

Behavior:

When the batch runtime instantiates the batch artifact (item reader in this example), it assigns the value of the named property provided on the job initiation to the corresponding @BatchProperty field. If no value is provided in the job the Java default sets the field value.

6.4 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.

6.4.1 @BatchContext

The @BatchContext annotation identifies a class field injection as a batch context. Such a class field is initialized with a context object according to type by the batch runtime. Batch contexts are runtime objects available during job execution. They provide a communication mechanism between the runtime and the batch artifacts of a batch application.

Syntax:

package: javax.batch.annotation

@BatchContext <context-type> <field-name>;

Where:

<context-type>	is the context type. Supported types are: JobContext , StepContext, FlowContext, and SplitContext
<field-name>	is the field name of the context.

Example:

```
public class MyJobListenerImpl {
        @BatchContext  JobContext  jctxt;
}
```

See section 7.8.2 for definition of JobContext class.

```
public class MyStepListenerImpl {
        @BatchContext  StepContext  sctxt;
}
```

See section 7.8.3 for definition of StepContext class.

6.4.1.1 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 @BatchContext annotated 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.

3. FlowContext

   There is one FlowContext per flow execution. It exists for the life of the flow.

4. SplitContext

   There is one SplitContext per split execution. It exists for the life of the split. There is one SplitContext for the parent split/thread.

6.5 Parallelization

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

1. 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 more detailed control over parallel processing:

   a. PartitionMapper provides a programmatic means for calculating the number of partitions and unique properties for each.

   b. PartitionReducer provides a unit of work demarcation around partition processing.

   c. PartitionCollector provides a means for merging interrim results from individual partitions.

   d. PartitionAnalyzer provides a means to gather interrim and final results from individual partitions for single point of control processing and decision making.

2. Concurrent:

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

6.5.1 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 an PartitionMapper batch artifact:

```
package javax.batch.api;

import javax.batch.api.parameters.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;
}
```

Note the PartitionMapper is not invoked during job restart. The Batch Runtime remembers the partition plan from the first JobExecution and uses it on all subsequent JobExecutions for the same JobInstance.

See section 7.8.7 for details on the PartitionPlan result value type.

6.5.2 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:

```
package javax.batch.api;

/**
 * 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;
}

package javax.batch.api;
/**
 * The AbstractBatchlet provides default
 * implementations of optional methods.
 */
public abstract class AbstractPartitionReducer implements PartitionReducer {
        /**
         * Optional method.
         *
         * Implement this method to take action before
         * partitioned step processing begins.
         *
         * @throws Exception is thrown if an error occurs.
         */
        @Override
        public void beginPartitionedStep() throws Exception {}
        /**
         * Optional method.
         *
         * Implement 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 {}
        /**
         * Optional method.
         *
         * Implement 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 {}
        /**
         * Optional method.
         *
         * Implement 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 {}

}
```

6.5.3 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 6.5.4 for further information about the PartitionAnalyzer. The PartitionCollector interface may be used to implement an PartitionCollector batch artifact:

```
package javax.batch.api;

import java.io.Externalizable;
/**
 * 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>
        * @return an Externalizable object to pass to the
        * PartitionAnalyzer.
        * @throws Exception is thrown if an error occurs.
        */
       public Externalizable collectPartitionData() throws Exception;
}
```

6.5.4 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:

```
package javax.batch.api;

import java.io.Externalizable;
/**
 * 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
       * Externalizable 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(Externalizable 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(String batchStatus, String exitStatus) throws
Exception;
}
```

```
package javax.batch.api;

import java.io.Externalizable;
```

```
/**
 * The AbstractPartitionAnalyzer provides default
 * implementations of optional methods.
 */
public abstract class AbstractPartitionAnalyzer implements PartitionAnalyzer
{
      /**
       * Optional method.
       *
       * Implement 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(Externalizable data) throws Exception
{}
      /**
       * Optional method.
       *
       * Implement 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(String batchStatus, String exitStatus)
                  throws Exception {}
}
```

6.6 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:

```
package javax.batch.api;

import javax.batch.runtime.context.BatchContext;

/**
 * Decider determines batch exit status and to influence
 * 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.
 *
 * @param <T> specifies the type of the transient data in the
 * batch context.
```

```
 */
public interface Decider <T> {
      /**
       * The decide method receives control during job processing to
       * set exit status between a step, split, or flow. The return
       * value updates the current job execution's exit status.
       * @param ctx specifies the last execution element's batch context.
       * It may be a StepContext, FlowContext, or SplitContext.
       * @return updated job exit status
       * @throws Exception is thrown if an error occurs.
       */
      public String decide(BatchContext<T> ctx) throws Exception;
}
```

6.7 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>
```

7 Batch Runtime Specification

7.1 Job Metrics

The batch runtime supports the following step-level metrics:

1. readCount - the number of items that have been successfully read
2. writeCount - the number of items that have been successfully written
3. commitCount - the number transactions that have been committed for this execution
4. rollbackCount - the number of times the business transaction controlled by the Step has been rolled back.
5. readSkipCount - the number of times read has failed, resulting in a skipped item.
6. processSkipCount - the number of times process has failed, resulting in a skipped item.
7. filterCount - the number of items that have been 'filtered' by the ItemProcessor.
8. writeSkipCount - the number of times write has failed, resulting in a skipped item.

These metrics are available through the StepExecution runtime object. See section 7.8.13 for further information on StepExecution.

7.2 Job Identifiers

Jobs 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 instance is restarted with the JobOperator "restart" method. Note there can be no more than one executionId in the STARTED state at one time.
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.

7.3 JobOperator

The JobOperator interface provides a set of operations to start, stop, restart, and inspect jobs. See 7.8.10 for detailed description of this interface. The JobOperator interface is accessible via a factory pattern:

JobOperator jobOper= BatchRuntime.getJobOperator();

See section 7.8.9 for details on the BatchRuntime class.

7.4 Classloader Scope

The batch runtime and the batch artifacts it loads execute in the class loader scope of the their invoker. This means if a typical Java main program starts a job through the JobOperator interface, the batch runtime and the batch artifacts it loads are loaded by the JVM system class loader. If a Java EE application starts a job, the Java EE application class loader hierarchy does the loading. This same principle applies to other arrangements, in which the job initiating environment supplies its own unique class loader configuration.

7.5 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.

2. archive loader

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

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 7.7.1 for more about the batch.xml file.

7.6 Job XML Loading

Job XML is referenced in two cases:

1. on the JobOperator.start command (see 7.8.10) to start a job.
2. from within a Job XML to resolve inheritance (see 5.8).

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.

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 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 directory (WEB-INF/classes/META-INF for .war files).

   See 7.7.2 for more about the META-INF/batch-jobs.

7.7 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).

7.7.1 META-INF/batch.xml

A batch application may use the archive loader (see section 7.5) to load batch artifacts. The application does this by supplying a 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://jcp.org.batch/jsl">
   <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.

Note if an implementation-specific loader is used (see 7.5) any artifact it loads takes precedence over artifacts specified in batch.xml.

7.7.2 META-INF/batch-jobs

A batch application may use the archive loader (see section 7.6) 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 it is the standard META-INF directory. For .war files it is the WEB-INF/classes/META-INF directory.

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 or on the parent= attribute of a Job XML.
.xml	specifies required file type of a Job XML file under META-INF/batch-jobs.

Note if an implementation-specific loader (see 7.6) loads a Job XML document that document takes precedence over documents stored under META-INF/batch-jobs.

7.8 Supporting Classes

7.8.1 BatchContext

```
package javax.batch.runtime.context;

import java.util.List;

/**
 * Base class for all batch context types.
 *
 */
public interface BatchContext <T> {
      /**
       * The getId method returns the context id. This
       * is value of the id attribute from the Job
       * XML execution element corresponding to this
       * context  type.
       * @return id string
       */
      public String getId();
      /**
       * The getTransientUserData method returns a transient data object
       * belonging to the current Job XML execution element.
       * @return user-specified type
       */
      public T getTransientUserData();
      /**
       * The setTransientUserData method stores a transient data object into
       * the current batch context.
       * @param data is the user-specified type
       */
      public void setTransientUserData(T data);
      /**
       * The getBatchContexts method returns a list of BatchContexts
       * corresponding to a compound Job XML execution element,
       * either a split or a flow. The batch context of a compound
       * execution element contains a list of batch contexts of
       * the execution elements contained within that compound
       * execution element.  For example, if this batch context
       * belongs to a split, the list of batch contexts is the
       * flow contexts belonging to the flows in that split; if
       * this batch context belongs to a flow, the list of batch
```

```
         * contexts may contain a combination of split and step
         * batch contexts. For regular execution elements (e.g.
         * job, step) this method returns null.
         * @return list of BatchContexts
         */
        public List<BatchContext<T>> getBatchContexts();

}
```

7.8.2 JobContext

```
package javax.batch.runtime.context;
/**
  *
  * JobContext is the class field type associated with the @JobContext
  * annotation. A JobContext provides information about the current
  * job execution.
  *
  * @see javax.batch.annotation.context.JobContext
  */
import java.util.Properties;
import javax.batch.runtime.context.BatchContext;

public interface JobContext <T> extends BatchContext <T> {

        /**
         * 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.
         * @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 String 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);
}
```

7.8.3 StepContext

```
package javax.batch.runtime.context;
/**
  *
  * StepContext is the class field type associated with the @StepContext
  * annotation. A StepContext provides information about the current step
  * of a job execution.
  *
  * @see javax.batch.annotation.context.StepContext
  */
import java.io.Externalizable;
import java.util.Properties;
import javax.batch.runtime.context.BatchContext;
import javax.batch.runtime.metric.Metric;

public interface StepContext <T,P extends Externalizable> extends
BatchContext <T> {

        /**
         * 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.
         * @return job 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.Externalizable. 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 P getPersistentUserData();
    /**
     * The setPersistentUserData method stores a persistent data object
     * into the current step. The user data type must implement
     * java.util.Externalizable. 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(P 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 String 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();
}
```

7.8.4 FlowContext

```
package javax.batch.runtime.context;
/**
  *
  * FlowContext is a class field type associated with the @BatchContext
  * annotation. A FlowContext provides information about the current
  * Flow execution.
  *
  * @see javax.batch.annotation.context.BatchContext
  */

public interface FlowContext <T> extends BatchContext <T> {}
```

7.8.5 SplitContext

```
package javax.batch.runtime.context;
/**
  *
  * SplitContext is a class field type associated with the @BatchContext
  * annotation. A SplitContext provides information about the current
  * Split execution.
  *
  * @see javax.batch.annotation.context.BatchContext
  */
import javax.batch.runtime.context.BatchContext;
import java.util.List;
public interface SplitContext <T> extends BatchContext <T> {
      /**
       * The getFlowResults method returns a list of results for each
       * step that has completed in the split.
       * @return list of FlowResults
       */
      List<FlowResults> getFlowResults();
}
```

7.8.6 FlowResults

```
package javax.batch.runtime.context;
/**
 * The FlowResults object identifies a
 * flow and its batch and exit status.
 * It is used by the SplitContext.
 *
 */
public interface FlowResults {
      /**
       * The getFlowId method returns
```

```
         * the value of the flow element's
         * id attribute.
         * @return flow id
         */
        String getFlowId();
        /**
         * The getBatchStatus method
         * returns the batch status
         * value of the flow.
         * @return batch status
         */
        String getBatchStatus();
        /**
         * The getExitStatus method
         * returns the exit status
         * value of the flow.
         * @return exit status
         */
        String getExitStatus();
}
```

7.8.7 Metric

```
package javax.batch.runtime;
/**
  *
  * The Metric interface defines job metrics recorded by
  * the batch runtime.
  *
  */
public interface Metric {
      public enum MetricName {READCOUNT, WRITECOUNT, COMMITCOUNT,
             ROLLBACKCOUNT, READSKIPCOUNT, PROCESSSKIPCOUNT, FILTERCOUNT,
             WRITESKIPCOUNT}
        /**
         * The getName method returns the metric name. The following names
         * are defined: "readCount", "writeCount", "commitCount",
         * "rollbackCount", "readSkipCount", "processSkipCount", "filterCount",
         * "writeSkipCount"
         * @return metric name.
         */
        public MetricName getName();
        /**
         * The getValue method returns the metric value.
         * @return metric value.
         */
        public long getValue();
}
```

7.8.8 PartitionPlan

```
package javax.batch.api.parameters;
/**
 *
 * 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 </li>
 * </ol>
 *
 * @see javax.batch.annotation.parallel.PartitionMapper
 */
import java.io.Externalizable;
import java.util.Properties;

public interface PartitionPlan extends Externalizable {

/**
 * Set number of partitions. * @param count specifies the partition count
 */
public void setPartitions(int count) ;

/**
 * Set number of threads. Defaults to zero, which means
 * thread count is equal to partition count.
 * @param count specifies the 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
 */
public void setPartitionProperties(Properties[] props) ;

/**
 * Gets count of Partitions.
 * @return Partition count
 */
public int getPartitionCount() ;

/**
 * Gets count of threads.
 * @return thread count
 */
public int getThreadCount() ;


/**
 * Gets array of Partition Properties objects for Partitions.
 * @return Partition Properties object array
 */
```

```
public Properties[] getPartitionProperties() ;

}
```

7.8.9 BatchRuntime

```
package javax.batch.runtime;

/**
 * The BatchRuntime represents the batch
 * runtime environment.
 *
 */
import javax.batch.runtime.JobOperator;

public class BatchRuntime {

      /**
       * The getJobOperator factory method returns
       * an instance of the JobOperator interface.
       * Repeated calls to this method returns the
       * same instance.
       * @return JobOperator instance.
       */
      public static JobOperator getJobOperator() {...}

}
```

7.8.10 JobOperator

```
package javax.batch.operations;

import java.util.List;
import java.util.Set;
import java.util.Properties;
import javax.batch.operations.exception.JobExecutionIsRunningException;
import javax.batch.operations.exception.JobExecutionNotRunningException;
import javax.batch.operations.exception.JobExecutionAlreadyCompleteException;
import javax.batch.operations.exception.JobExecutionNotMostRecentException;
import javax.batch.operations.exception.JobRestartException;
import javax.batch.operations.exception.JobStartException;
import javax.batch.operations.exception.NoSuchJobException;
import javax.batch.operations.exception.NoSuchJobExecutionException;
import javax.batch.operations.exception.NoSuchJobInstanceException;
import javax.batch.runtime.JobExecution;
import javax.batch.runtime.JobInstance;
import javax.batch.runtime.StepExecution;

public interface JobOperator {

      /**
```

```
         * Returns a set of all job names known to the batch runtime.
         *
         * @return a set of job names.
         */
        Set<String> getJobNames();

        /**
         * 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
         */
        int getJobInstanceCount(String jobName) throws NoSuchJobException;

        /**
         * Returns all JobInstances belonging to a job with a particular name.
         *
         * @param jobName
         *            specifies the job name.
         * @param start
         *            specifies the relative starting number 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
         */
        List<JobInstance> getJobInstances(String jobName, int start, int count)
                    throws NoSuchJobException;

        /**
         * Returns JobInstances for all running jobs across all instances of a
job
         * with a particular name.
         *
         * @param jobName
         *            specifies the job name.
         * @return a list of JobInstances.
         * @throws NoSuchJobException
         */
        List<JobInstance> getRunningInstances(String jobName) throws
NoSuchJobException;

        /**
         * Returns all JobExecutions belonging to a particular job instance.
         *
         * @param instanceId
         *            specifies the job instance.
         * @return List of JobExecutions.
         * @throws NoSuchJobInstanceException
         */
```

```
      List<JobExecution> getExecutions(JobInstance instance) throws
NoSuchJobInstanceException;

      /**
       * 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 instance
       *            specifies the job instance.
       * @return a Properties object containing the key/value job parameter
pairs.
       * @throws NoSuchJobExecutionException
       */
      Properties getParameters(JobInstance instance)
                  throws NoSuchJobExecutionException;

      /**
       * Creates a new job instance and starts the first execution of that
       * instance.
       *
       * 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 of the new job instance.
       * @throws JobStartException
       */
      long start(String jobXMLName, Properties jobParameters) throws
JobStartException;

      /**
       * Restarts a failed or stopped job instance.
       *
       * @param executionId
       *            specifies the execution to to restart. This execution
       *            must be the most recent execution that ran.
       * @return new executionId
       * @throws JobExecutionAlreadyCompleteException
       * @throws NoSuchJobExecutionException
       * @throws JobExecutionNotMostRecentException,
       * @throws JobRestartException
       */
```

```
        long restart(long executionId) /* exception if restart older execution
*/
                    throws JobExecutionAlreadyCompleteException,
                    NoSuchJobExecutionException,
                    JobExecutionNotMostRecentException,
                    JobRestartException;

        /**
         * 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.
         *
         * @param executionId
         *            specifies the job execution to stop.
         *            The job execution must be running.
         * @throws NoSuchJobExecutionException
         * @throws JobExecutionNotRunningException
         */
        void stop(long executionId) throws NoSuchJobExecutionException,
                    JobExecutionNotRunningException;

        /**
         * Set batch status to ABANDONED.  The instance must have
         * no running execution.
         *
         * @param instanceId
         *            specifies the job instance to abandon.
         * @throws NoSuchJobInstanceException
         * @throws JobExecutionIsRunningException
         */
        void abandon(JobInstance instance) throws NoSuchJobInstanceException,
                    JobExecutionIsRunningException;


        /**
         * Return the job instance for the specified execution id.
         *
         * @param executionId
         *            specifies the job execution.
         * @return job instance
         * @throws NoSuchJobExecutionException
         */
        JobInstance getJobInstance(long executionId) throws
NoSuchJobExecutionException;

        /**
         * Return all job executions belonging to the specified job instance.
         *
         * @param jobInstance
         *            specifies the job instance.
         * @return list of job executions
```

```
         * @throws NoSuchJobInstanceException
         */
        List<JobExecution> getJobExecutions(JobInstance instance) throws
NoSuchJobInstanceException;

        /**
         * Return job execution for specified execution id
         *
         * @param executionId
         *            specifies the job execution.
         * @return job execution
         * @throws NoSuchJobExecutionException
         */
        JobExecution getJobExecution(long executionId) throws
NoSuchJobExecutionException;

        /**
         * Return StepExecutions for specified execution id.
         *
         * @param executionId
         *            specifies the job execution.
         * @param stepExecutionId
         *            specifies the step belonging to that execution
         * @return step execution
         * @throws NoSuchJobExecutionException
         */
        List<StepExecution> getStepExecutions(long executionId) throws
NoSuchJobExecutionException;

}
```

7.8.11 JobInstance

```
package javax.batch.runtime;

public interface JobInstance {
        /**
         * Get id for this JobInstance.
         * @return instance id
         */
        public long getId();
        /**
         * Get job name
         * @return value of 'id' attribute from <job>
         */
        public String getJobName();

        /**
         * Get instance id for this job instance
         * @return instance id
         */
        public long getInstanceId();
```

```
}
```

7.8.12 JobExecution

```
package javax.batch.runtime;

import java.util.Date;
import java.util.Properties;

public interface JobExecution {
	/**
	 * Get id for this JobExecution.
	 * @return execution id
	 */
	public long getId();
	/**
	 * Get batch status of this execution.
	 * @return batch status value.
	 */
	public String getStatus();
	/**
	 * 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();

}
```

7.8.13 StepExecution

```
package javax.batch.runtime;

import java.util.Date;

public interface StepExecution<P> {
	/**
	 * Get id for this StepExecution.
	 * @return StepExecution id
	 */
	public long getId();
	/**
	 * Get batch status of this step execution.
	 * @return batch status.
	 */
	public String getStatus();
	/**
	 * 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 user persistent data
	 * @return persistent data
	 */
	public P getUserPersistentData();
	/**
	 * Get step metrics
	 * @return array of metrics
	 */
	public Metric[] getMetrics();

}
```

7.8.14 Batch Exception Classes

This specification defines the following exception classes:

1. JobExecutionAlreadyCompleteException
2. JobExecutionIsRunningException
3. JobExecutionNotMostRecentException
4. JobExecutionNotRunningException
5. JobRestartException
6. JobStartException
7. NoSuchJobException
8. NoSuchJobExecutionException
9. NoSuchJobInstanceException

8 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.

8.1 Batch Artifact Lifecycle

All batch artifacts are instantiated in the scope in which they are declared in the Job XML and exist for the life of their containing scope. One artifact per Job XML reference is instantiated. This means job level artifacts exist for the life of the job. Step level artifacts exist for the life of the step.

8.2 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.

8.3 Job Processsing

1. <Create JobContext>
2. <Apply substitutions to job level elements>
3. <Store job level properties in JobContext>
4. <->[JobListener.beforeJob...] // thread A
5. <processs execution elements>
6. <->[JobListener.afterJob...] // thread A
7. <Store save-as properties to repository>
8. <Destroy JobContext>

8.4 Regular Batchlet Processsing

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

8.5 Partitioned Batchlet Processsing

1. <Create StepContext>
2. <Apply substitutions (except #{partitionPlan}) to step level elements>
3. <Store step level properties in StepContext>
4. <->[StepListener.beforeStep...] // thread A
5. <->[PartitionReducer.beginPartitionedStep] // thread A
6. <->[PartitionMapper.mapPartitions] // thread A

7. // per partition:
    a. <Apply #{partitionPlan} substitutions to step level elements>
    b. <->Batchlet.process // thread Px
    c. // if stop issued:
    d. <->[Batchlet.stop] // thread Py, StepContext is available
    e. <->[PartitionCollector.collectPartitionData] // thread Px
8. // when collector payload arrives:
9. <->[PartitionAnalyzer.analyzeCollectorData] // thread A
10. // when partition ends:
11. <->[PartitionAnalyzer.analyzeStatus] // thread A
12. // if rollback condition occurs:
13. <->[PartitionReducer.rollbackPartitionedStep] // thread A
14. <->[PartitionReducer.beforePartitionedStepCompletion] // thread A
15. <->[PartitionReducer.afterPartitionedStepCompletion] // thread A
16. <->[StepListener.afterStep...] // thread A
17. <Store StepContext persistent area>
18. <Store save-as properties to repository>
19. <Destroy StepContext>

8.6 Regular Chunk Processing

1. <Create StepContext>
2. <Apply substitutions to step level elements>
3. <Store step level properties in StepContext>
4. <->[StepListener.beforeStep...] // thread A
5. [<begin transaction> EE only]
6. <->ItemReader.open // thread A
7. <->ItemWriter.open // thread A
8. [<commit transaction> EE only]
9. // chunk processing:
10. <repeat until no more items> {
    a. <begin checkpoint [<begin transaction> EE only]>
    b. <repeat until commit criteria reached> {
        i. <->ItemReader.readItem // thread A
        ii. <->ItemProcessor.processItem // thread A
        iii. // if buffering
        iv. <add item to buffer>
        v. else <->ItemWriter.writeItems(1) // thread A
    c. }
    d. // if buffering
    e. <->ItemWriter.writeItems // thread A
    f. <->[ItemReader.checkpointInfo] // thread A
    g. <->[ItemWriter.checkpointInfo] // thread A
    h. <commit checkpoint (commit transaction)>
11. }
12. [<begin transaction> EE only]
13. <->ItemWriter.close // thread A
14. <->ItemReader.close // thread A
15. [<commit transaction> EE only]
16. <->[StepListener.afterStep...] // thread A
17. <Store StepContext persistent area>
18. <Store save-as properties to repository>
19. <Destroy StepContext>

8.7 Partitioned Chunk Processing

1. <Create StepContext>
2. <Apply substitutions to step level elements>
3. <Store step level properties in StepContext>
4. <->[StepListener.beforeStep...] // thread A
5. <->[PartitionReducer.beginPartitionedStep] // thread A
6. <->[PartitionMapper.mapPartitions] // thread A
   // per partition - on thread Px:
   a. <Apply #PartitionPlan substitutions to step level elements>
   b. [<begin transaction> EE only]
   c. <->ItemReader.open // thread Px
   d. <->ItemWriter.open // thread Px
   e. [<commit transaction> EE only]
   a. <repeat until no more items> {
      i. <begin checkpoint [<begin transaction> EE only]>
      ii. <repeat until commit criteria reached> {
         1. <->ItemReader.readItem // thread Px
         2. <->ItemProcessor.processItem // thread Px
         3. // if buffering
         4. <add item to buffer>
         5. else <->ItemWriter.writeItems(1) // thread Px
      iii. }
      iv. // if buffering
      v. <->ItemWriter.writeItems // thread Px
      vi. <->[ItemReader.checkpointInfo] // thread Px
      vii. <->[ItemWriter.checkpointInfo] // thread Px
      viii. <commit checkpoint (commit transaction)>
      ix. <->[PartitionCollector.collectPartitionData] // thread Px
   f. }
   g. [<begin transaction> EE only]
   h. <->ItemWriter.close // thread Px
   i. <->ItemReader.close // thread Px
   j. [<commit transaction> EE only]
7. [<begin transaction> EE only] // thread A
8. // Actions 9-12 run continuously until all partitions end.
9. // when collector payload arrives:
10. <->[PartitionAnalyzer.analyzeCollectorData] // thread A
11. // when partition ends:
12. <->[PartitionAnalyzer.analyzeStatus] // thread A
13. // Remaining actions run after all partitions end:
14. // if rollback condition occurs:
15. <->[PartitionReducer.rollbackPartitionedStep] // thread A
16. [<rollback transaction EE only>]

17. // else not rollback
18. <->[PartitionReducer.beforePartitionedStepCompletion] // thread A
19. [<commit transaction> EE only]
20. <->[PartitionReducer.afterPartitionedStepCompletion] // thread A
21. <->[StepListener.afterStep...] // thread A
22. <Store StepContext persistent area>
23. <Store save-as properties to repository>
24. <Destroy StepContext>

8.8 Chunk with Listeners (except RetryListener)

1. <Create StepContext>
2. <Apply substitutions to step level elements>
3. <Store step level properties in StepContext>
4. <->[StepListener.beforeStep...] // thread A
5. [<begin transaction> EE only]
6. <->ItemReader.open // thread A
7. <->ItemWriter.open // thread A
8. [<commit transaction> EE only]
9. // chunk processing:
10. <repeat until no more items> {
    a. <begin checkpoint [<begin transaction> EE only]>
    b. <->[ChunkListener.beforeChunk] // thread A
    c. <repeat until commit criteria reached> {
        i. <->[ItemReadListener.beforeRead] // thread A
        ii. <->ItemReader.readItem // thread A
        i. <->[ItemReadListener.afterRead] // thread A
        ii. // or:
        iii. {
        iv. <->[ItemReadListener.onReadError] // thread A
        v. <->[SkipListener.onSkipReadItem] // thread A
        vi. }
        vii. <->[ItemProcessListener.beforeProcess] // thread A
        viii. <->ItemProcessor.processItem // thread A
        ix. <->[ItemProcessListener.afterProcess] // thread A
        x. // or:
        xi. {
        xii. <->[ItemProcessListener.onProcessError] // thread A
        xiii. <->[SkipListener.onSkipProcessItem] // thread A
        xiv. }
        xv. // if buffering
        xvi. <add item to buffer>
        iii. else
        iv. <->[ItemWriteListener.beforeWrite] // thread A
        v. <->ItemWriter.writeItems (1) // thread A
        vi. <->[ItemWriteListener.afterWrite] // thread A
        vii. // or:
        viii. {
        ix. <->[ItemWriteListener.onWriteError] // thread A
        x. <->[SkipListener.onSkipWriteItems] // thread A
        xvii. }
    d. }
    e. // if buffering
    f. <->[ItemWriteListener.beforeWrite] // thread A

g. <->ItemWriter.writeItems // thread A
h. <->[ItemWriteListener.afterWrite] // thread A
i. // or:
j. {
k. <->[ItemWriteListener.onWriteError] // thread A
l. <->[SkipListener.onSkipWriteItems] // thread A
m. }
n. <->[ChunkListener.afterChunk] // thread A
o. <->[ItemReader.checkpointInfo] // thread A
p. <->[ItemWriter.checkpointInfo] // thread A
q. <commit checkpoint (commit transaction)>

11. }
12. [<begin transaction> EE only]
13. <->ItemWriter.close // thread A
14. <->ItemReader.close // thread A
15. [<commit transaction> EE only]
16. <->[StepListener.afterStep...] // thread A
17. <Store StepContext persistent area>
18. <Store save-as properties to repository>
19. <Destroy StepContext>

8.9 Chunk with RetryListener

1. <Create StepContext>
2. <Apply substitutions to step level elements>
3. <Store step level properties in StepContext>
4. <->[StepListener.beforeStep...] // thread A
5. [<begin transaction> EE only]
6. <->ItemReader.open // thread A
7. <->ItemWriter.open // thread A
8. [<commit transaction> EE only]
9. // chunk processing:
10. <repeat until no more items> {
    a. S1:
    b. <begin checkpoint [<begin transaction> EE only]>
    c. S2:
    d. <repeat until commit-interval reached> {
        i. <->ItemReader.readItem // thread A
        ii. // if retryable exception
        iii. <->[RetryReadListener.onRetryReadException] // thread A
        iv. <end repeat>
        v. <->ItemProcessor.processItem // thread A
        vi. // if retryable exception
        vii. <->[RetryProcessListener.onRetryProcessException] // thread A
        viii. <end repeat>

ix. // if buffering
x. <add item to buffer>
xi. // else
xii. <->ItemWriter.writeItems (1) // thread A
xiii. // if retryable exception
xiv. <->[RetryWriteListener.onRetryWriteException] // thread A
xv. <end repeat>

e. }
f. // if rollback exception, rollback checkpoint and resume at S1 else resume S2
g. // if buffering
h. <->ItemWriter.writeItems // thread A
i. // if retryable exception
j. <->[RetryWriteListener.onRetryWriteException] // thread A
k. <end repeat>
l. <->[ItemReader.checkpointInfo] // thread A
m. <->[ItemWriter.checkpointInfo] // thread A
n. <commit checkpoint (commit transaction)>

11. }
12. [<begin transaction> EE only]
13. <->ItemWriter.close // thread A
14. <->ItemReader.close // thread A
15. [<commit transaction> EE only]
16. <->[StepListener.afterStep...] // thread A
17. <Store StepContext persistent area>
18. <Store save-as properties to repository>
19. <Destroy StepContext>

8.10 Chunk with Custom Checkpoint Processing

1. <Create StepContext>
2. <Apply substitutions to step level elements>
3. <Store step level properties in StepContext>
4. <->[StepListener.beforeStep...] // thread A
5. [<begin transaction> EE only]
6. <->ItemReader.open // thread A
7. <->ItemWriter.open // thread A
8. [<commit transaction> EE only]
9. // chunk processing:
20. <repeat until no more items> {
    a. [
    b. <->[CheckpointAlgorithm.checkpointTimeout]
    c. <begin checkpoint [<begin transaction> EE only]>
    a.]
    b. <repeat until commit criteria reached> {
        i. <->ItemReader.readItem // thread A
        ii. <->ItemProcessor.processItem // thread A
        iii. // if buffering
        iv. <add item to buffer>
        v. else <->ItemWriter.writeItems(1) // thread A
        vi. <->CheckpointAlgorithm.isReadyToCheckpoint // thread A
    c. }
    d. // if buffering
    e. <->ItemWriter.writeItems // thread A
    f. <->[ItemReader.checkpointInfo] // thread A
    g. <->[ItemWriter.checkpointInfo] // thread A
    d. <->[CheckpointAlgorithm.beginCheckpoint] // thread A
    e. <commit checkpoint (commit transaction)>
    h. <->[CheckpointAlgorithm.endCheckpoint] // thread A
10. }
11. [<begin transaction> EE only]
12. <->ItemWriter.close // thread A
13. <->ItemReader.close // thread A
14. [<commit transaction> EE only]
15. <->[StepListener.afterStep...] // thread A
16. <Store StepContext persistent area>
17. <Store save-as properties to repository>
18. <Destroy StepContext>

8.11 Split Processing

1. <Create SplitContext>
2. // For each flow:
3. <run flow> // thread Fx
4. <->[SplitCollector.collectSplitData] // thread A
5. <->[SplitAnalyzer.analyzeCollectorData] // thread A
6. <->[SplitAnalyzer.analyzeStatus] // thread A
7. <Destroy SplitContext>

8.12 Flow Processing

1. <Create FlowContext>
2. // For each split or step:
3. <run split or step> // thread Xy
4. <Destroy FlowContext>

8.13 Stop Processing

The JobOperator.stop operation stops a running job execution. If a step is running at the time the stop is invoked, the step stops with the following behavior:

Chunk Step

The job and step batch status is marked STOPPING. 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. 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. The job and step batch status is marked STOPPED.

9 Job Specification Language (Job XML) XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
 Copyright 2012 International Business Machines Corp.

 See the NOTICE file distributed with this work for additional information
 regarding copyright ownership. Licensed under the Apache License,
 Version 2.0 (the "License"); you may not use this file except in compliance
 with the License. You may obtain a copy of the License at

  http://www.apache.org/licenses/LICENSE-2.0

 Unless required by applicable law or agreed to in writing, software
 distributed under the License is distributed on an "AS IS" BASIS,
 WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 See the License for the specific language governing permissions and
 limitations under the License.
-->
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
        targetNamespace="http://batch.jsr352/jsl" xmlns:jsl="http://batch.jsr352/jsl" xmlns:jaxb="http://java.sun.com/xml/ns/jaxb"
        jaxb:version="2.0">

        <xs:simpleType name="artifactRef">
                <xs:restriction base="xs:string" />
        </xs:simpleType>

        <xs:complexType name="Job">
                <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: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="id" use="required" type="xs:string" />
                <xs:attribute name="parent" use="optional" type="xs:string" />
                <xs:attribute name="abstract" use="optional" type="xs:string" />
                <xs:attribute name="restartable" use="optional" type="xs:string" />
        </xs:complexType>

        <xs:element name="job" type="jsl:Job" />
        <!-- step, flow, split are valid top level elements for inheritance snippets -->
        <xs:element name="flow" type="jsl:Flow" />
        <xs:element name="split" type="jsl:Split" />
        <xs:element name="step" type="jsl:Step" />

        <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="properties" type="jsl:Properties" minOccurs="0" maxOccurs="1" />
```

```
                <xs:element name="listeners" type="jsl:Listeners" minOccurs="0" maxOccurs="1" />
                <xs:element name="flow" type="jsl:Flow" minOccurs="0" maxOccurs="unbounded"/>
                <xs:group ref="jsl:ControlElements" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="id" use="required" type="xs:string" />
        <xs:attribute name="next" use="optional" type="xs:string" />
        <xs:attribute name="abstract" use="optional" type="xs:string" />
        <xs:attribute name="parent" use="optional" type="xs:string" />
</xs:complexType>

<xs:complexType name="Flow">
        <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:choice minOccurs="0" maxOccurs="unbounded">
                        <xs:element name="decision" type="jsl:Decision" />
                        <xs:element name="step" type="jsl:Step" />
                        <xs:element name="split" type="jsl:Split" />
                </xs:choice>
                <xs:group ref="jsl:ControlElements" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="id" use="required" type="xs:string" />
        <xs:attribute name="next" use="optional" type="xs:string" />
        <xs:attribute name="abstract" use="optional" type="xs:string" />
        <xs:attribute name="parent" use="optional" type="xs:string" />
</xs:complexType>

<xs:group name="ControlElements">
        <!-- I guess no point being strict if you don't want to group like elements. -->
        <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:ControlElements" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="id" use="required" type="xs:string" />
        <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="optional" type="jsl:artifactRef" />
</xs:complexType>

<xs:complexType name="ExceptionClassFilter">
        <xs:sequence>
                <xs:element name="include" minOccurs="0" maxOccurs="1">
                  <xs:complexType>
                    <xs:sequence/>
                    <xs:attribute name="class" use="required" type="xs:string"/>
                  </xs:complexType>
              </xs:element>
                <xs:element name="exclude" minOccurs="0" maxOccurs="1">
                  <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:choice minOccurs="0" maxOccurs="1"> <!--  TODO validate a merged Step has at least one -->
                        <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:ControlElements" minOccurs="0" maxOccurs="unbounded" />
        </xs:sequence>
        <xs:attribute name="id" use="required" type="xs:string" />
        <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:attribute name="abstract" use="optional" type="xs:string" />
        <xs:attribute name="parent" 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>

<!--  required attributes were made optional to allow validated snippets to be loaded for inheritance -->
<xs:complexType name="Chunk">
        <xs:sequence>
                <xs:element name="reader" type="jsl:ItemReader" minOccurs="0" maxOccurs="1" />
                <xs:element name="processor" type="jsl:ItemProcessor" minOccurs="0" maxOccurs="1" />
                <xs:element name="writer" type="jsl:ItemWriter" minOccurs="0" maxOccurs="1" />
                <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" default="item" />
                <xs:attribute name="item-count" use="optional" type="xs:string" default="10" />
                <xs:attribute name="time-limit" use="optional" type="xs:string" default="0" />
                <!--  Note that 'buffer-items' is a boolean, but we use a string to allow for property substitution. -->
                <xs:attribute name="buffer-items" use="optional" type="xs:string" />
                <xs:attribute name="skip-limit" use="optional" type="xs:string" />
                <xs:attribute name="retry-limit" use="optional" type="xs:string" />
        </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>
                  <!--  We could have a useful "empty" list case when a child wants to "unset" parent properties by doing merge="false"
with an
                      empty list.  -->
                        <xs:element name="property" type="jsl:Property" maxOccurs="unbounded" minOccurs="0" />
                </xs:sequence>
                <xs:attribute name="merge" use="optional" type="xs:string" />
                <xs:attribute name="partition" use="optional" type="xs:string" />
        </xs:complexType>

        <xs:complexType name="Listeners">
                <xs:sequence>
                  <!--  We could have a useful "empty" list case when a child wants to "unset" parent listeners by doing merge="false"
with an
                      empty list.  -->
                        <xs:element name="listener" type="jsl:Listener" maxOccurs="unbounded" minOccurs="0" />
                </xs:sequence>
                <xs:attribute name="merge" use="optional" type="xs:string" />
        </xs:complexType>

        <xs:complexType name="Partition">
                <xs:sequence>
                  <xs:element name="mapper" type="jsl:PartitionMapper" minOccurs="0" maxOccurs="1" />
```

```
                <xs:element name="plan" type="jsl:PartitionPlan" minOccurs="0" maxOccurs="1" />
                        <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>
```

10 Credits

Section 4 Domain Language of Batch, was adapted from Spring Batch Reference Documentation:

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