



**IBM WebSphere Liberty**  
**Java Batch**  
**Technical Overview**

© 2016, IBM Corporation

## Topics to be Discussed

- **Brief Overview of Batch Processing**  
Including background on Java Batch evolution
- **Overview of JSR 352**  
A review of the key elements of the standard
- **IBM Implementation and Extensions**  
A review of how JSR 352 is implemented by IBM, including extensions to the standard that provide additional operational features and benefits

# ***Batch Processing ...***

## **... and what led up to Java Batch**

# Batch Processing Has Been Around a Very Long Time



A picture from the 1960s, and batch processing pre-dated this by several decades, or even centuries, depending on what is considered a “computer”

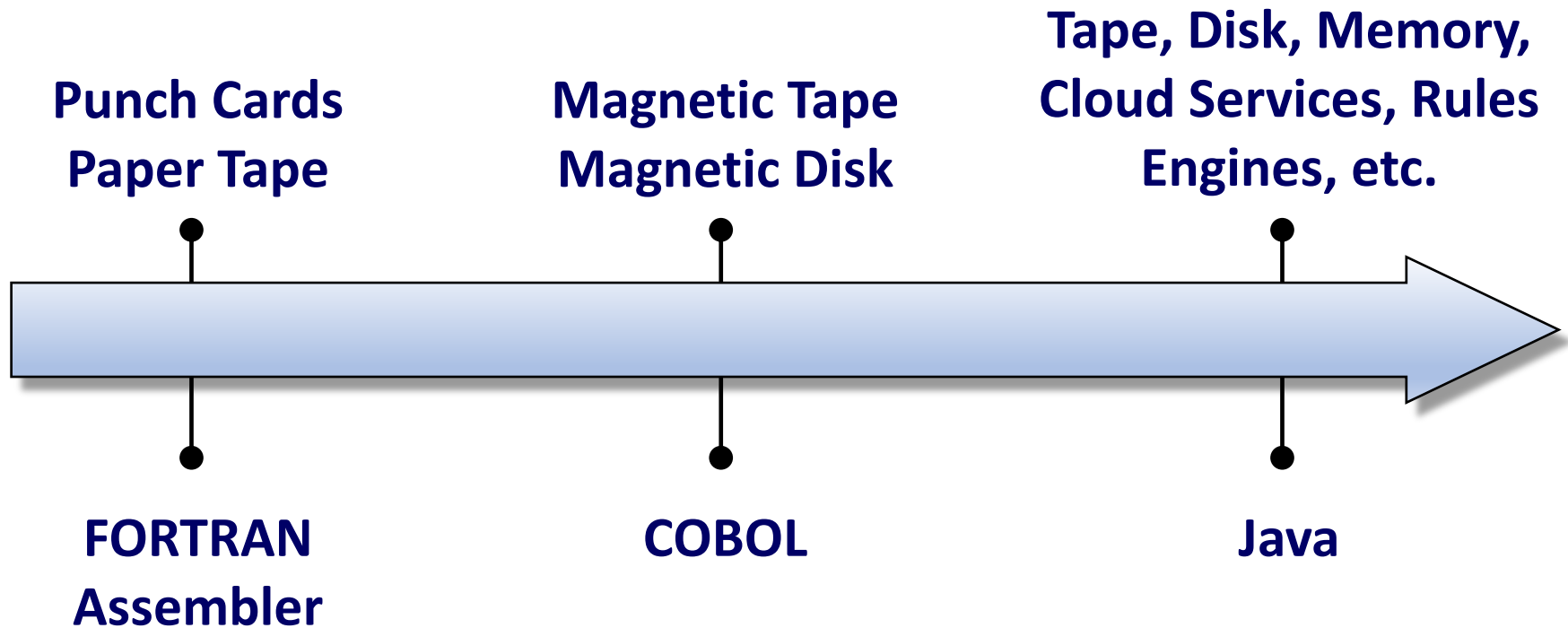
**There has long been a need to process large amounts of data to arrive at results from the data**

**There continues to be the same need today**

**It is unlikely the need to do processing in batch will go away any time soon**

**The need persists, the approach has evolved ...**

# Evolution: Data Storage and Programming Languages



**Change is driven by need. So what is driving the trend towards Java for batch processing?**

# Things Creating Push to Java for Batch

## Desire to Modernize Batch Processes

Motivation behind this takes many forms – new business needs; some update to an existing batch program is needed and it's seen as a good opportunity to re-write in Java; separate business logic into rules engine for more agile processing

## Availability of Java Skills

Particularly relative to other skills such as COBOL.

## z/OS: Ability to Offload to Specialty Engines

Workload that runs on z/OS specialty engines (zAAP, zIIP) is not counted towards CPU-based software charges.

## Can Java Run as Fast as Compiled Code?

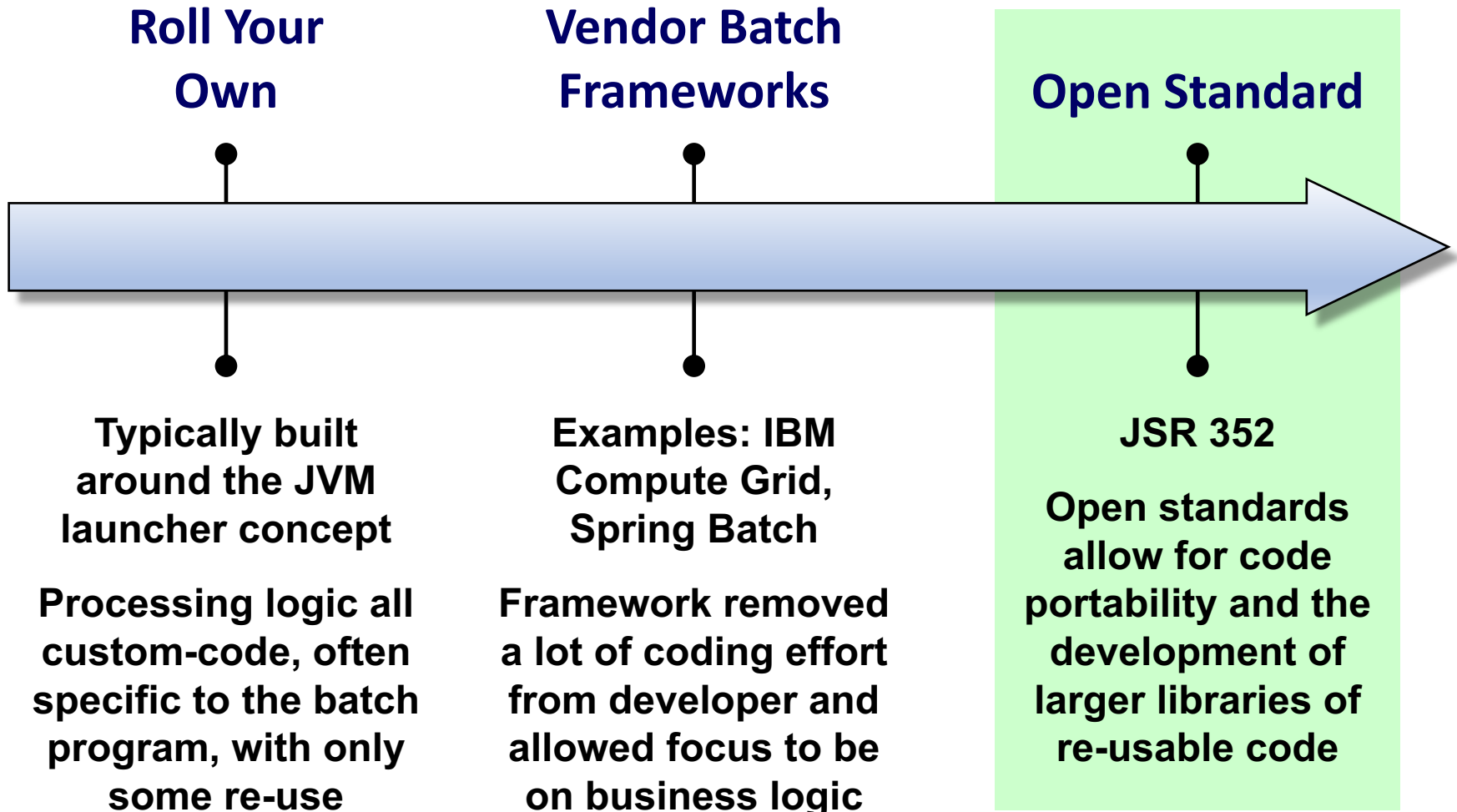
### Comparably ... and *sometimes* faster\*:

- Batch processing is by its nature iterative, which means Java classes prone to being Just-in-Time (JIT) compiled at runtime
- Java JIT compilers are getting very good at optimizing JIT'd code
- z/OS: System z processor chips have instructions specifically designed to aid JIT-compiled code
- COBOL that has not been compiled in a long time is operating with less-optimal compiled code that does not take specific advantage of chip instructions



Results vary, depending on many factors. This is not a promise of performance results.

# The Evolution of Java Batch ...





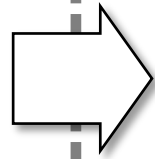
# ***Overview of JSR 352***

## The Process of Creating an Open Standard

### Formation of Working Group



Individuals working on the challenges independent of one another

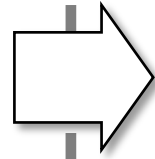


The group works to create a vision and a document of the proposed specification.

After review and acceptance, it becomes a published specification.

IBM led this group, with involvement from people from several other companies.

### Initial Release of Standard Specification

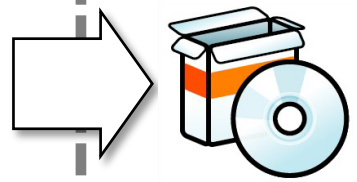


<https://jcp.org/en/jsr/detail?id=352>

The specification details the requirements and interfaces.

The JSR 352 specification was released in May 2013, and has been accepted as a component of the Java EE 7 specification as well.

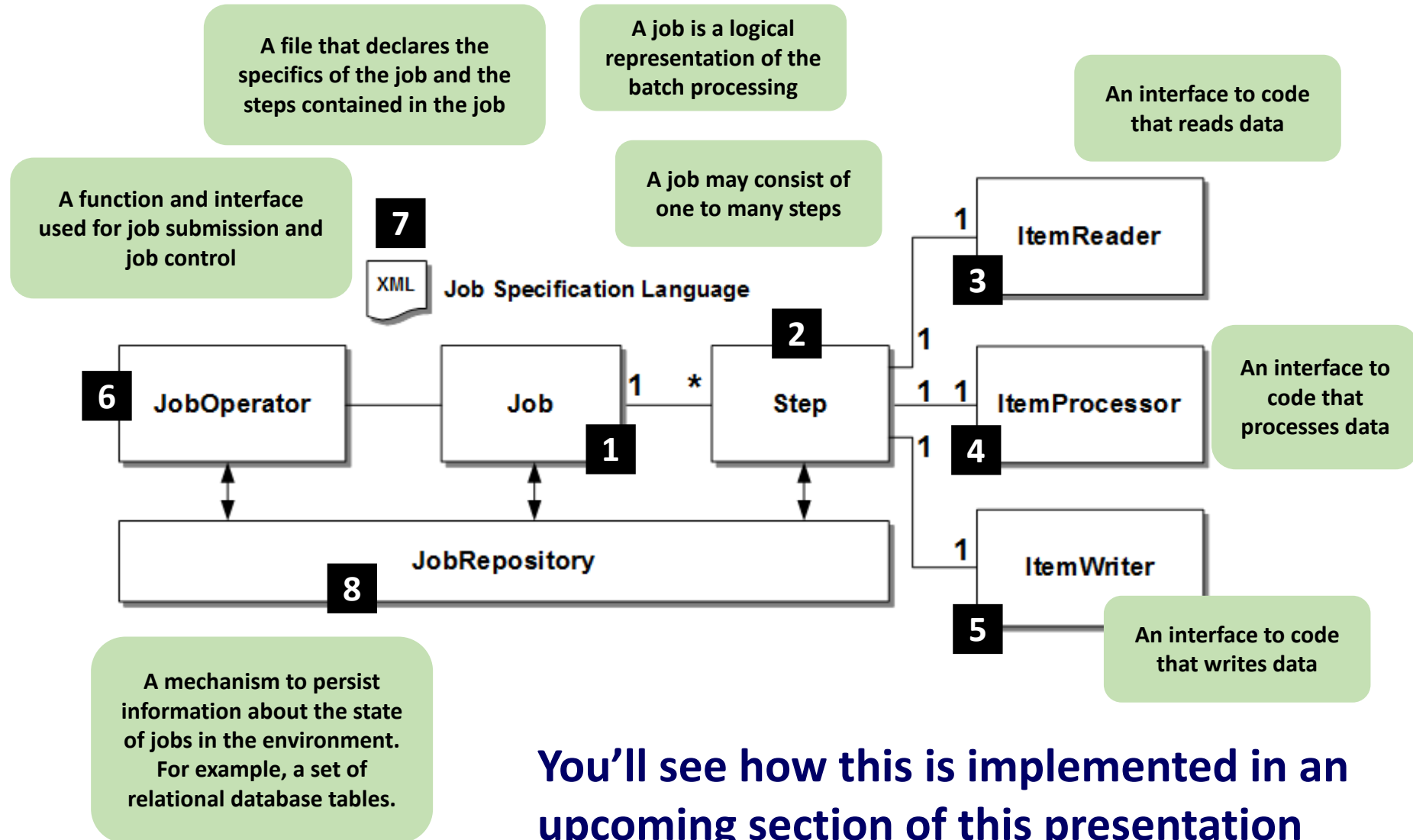
### Release of Vendor Implementations



Vendors release products and provide extensions for additional value

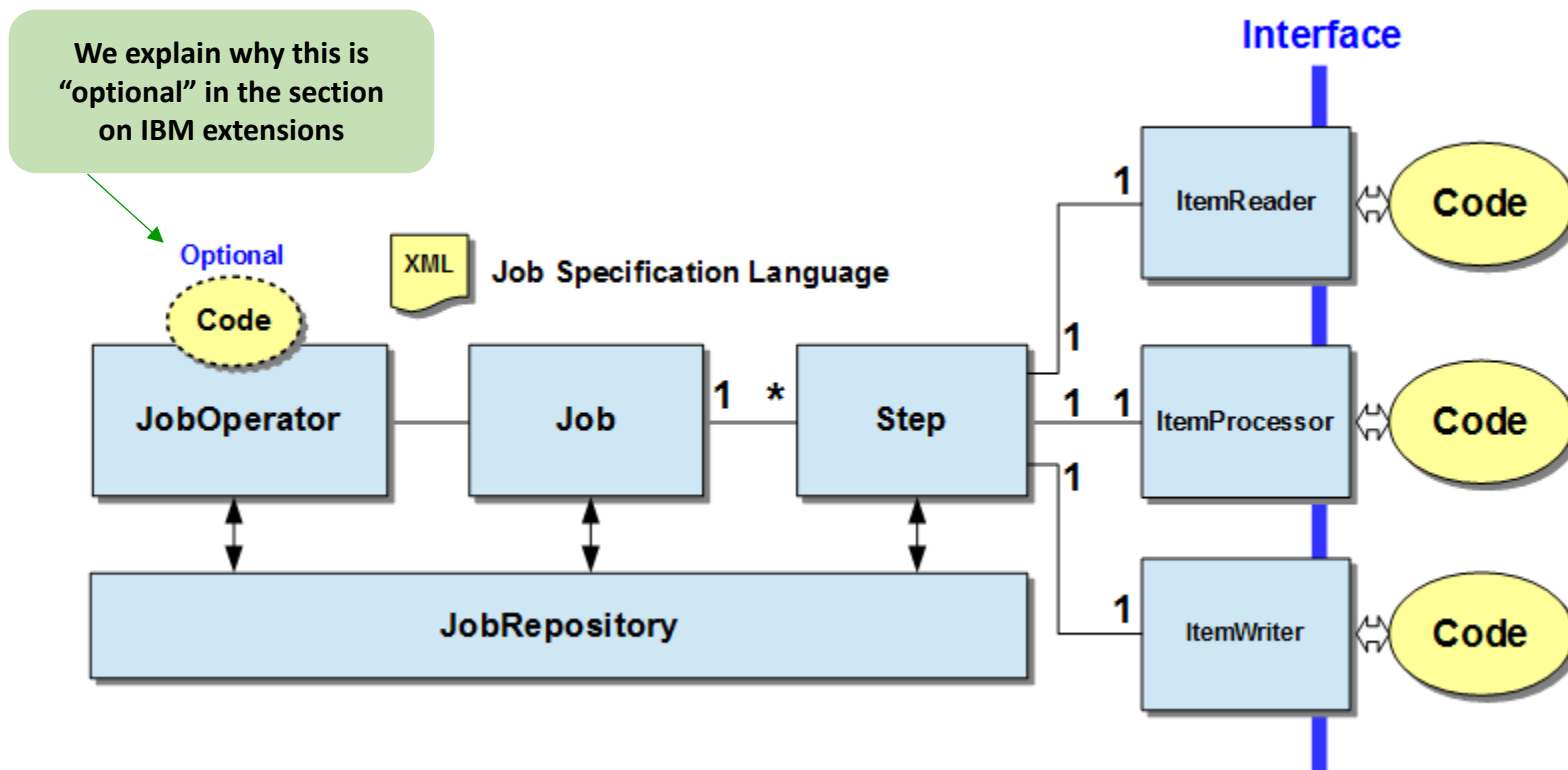


## The JSR 352 Diagram to Describe the Architecture



You'll see how this is implemented in an upcoming section of this presentation

# How Much of that Picture Do / Have to Code?




## It turns out ... relatively little

Much of the processing is handled by the vendor implementation of the JSR 352 standard. Your code sits behind standard interfaces and is called by the JSR 352 runtime.

# Job Step Types – Chunk and Batchlet

---


## Chunk Step



Job Step

- What we typically think of as a “batch job” – an iterative loop through data, with periodic commits of data written out during processing
  - This involves the `ItemReader`, `ItemProcessor` and `ItemWriter` interfaces shown earlier.
- 

## Batchlet Step

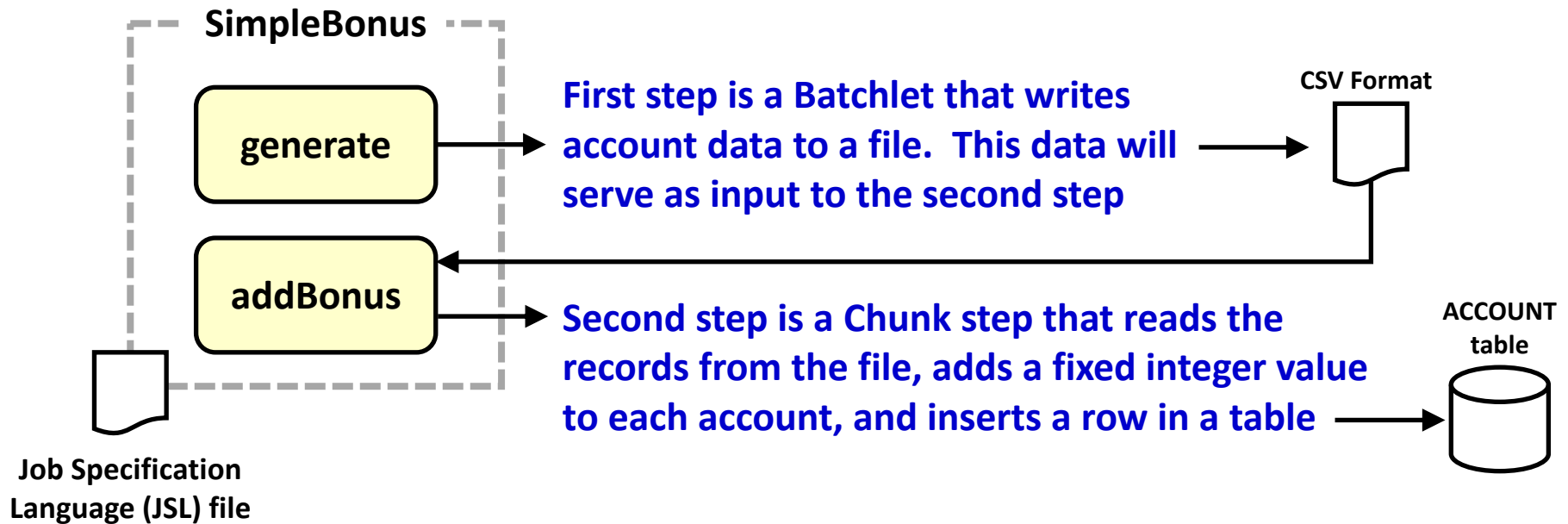


Job Step

- A job step with much less structure ... it is called, it runs and does whatever is in the code, and ends
- This job step type is useful for operations that are not iterative in nature, but may take some time ... a large file FTP, for example
- This is also useful for encapsulating existing Java `main()` programs into the JSR 352 model

**A multi-step job may consist of either ... or both**

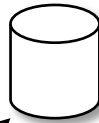
# High-Level Example ... to Illustrate the Key Concepts



**Not real-world, but useful to illustrate essential JSR 352 concepts. What does packaging look like?**

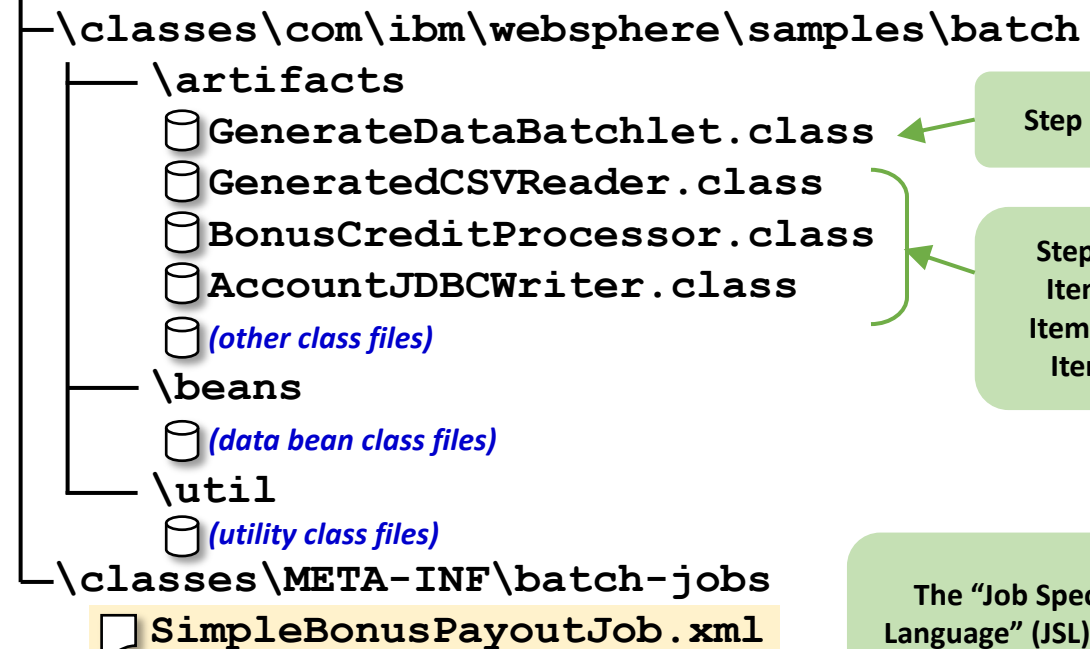
# A Peek Inside the Sample Application WAR file

Application Developer



BonusPayout-1.0.war

WEB-INF



The “How to write JSR 352 applications” topic is important, but outside the scope of this overview discussion.

The “Job Specification Language” (JSL) file, which we’ll look at next ...

This deploys into the Liberty Profile server’s /dropins directory, or pointed to with <application> tag like any other application



# JSL: Job Specification Language, Part 1

```
<?xml version="1.0" encoding="UTF-8"?>
<job id="SimpleBonusPayoutJob">
```

Properties are a way to get values into your batch job. They can be specified in the JSL as shown, and overridden at submission time using IBM's REST interface (shown later)

```
<properties>
  <property name="numRecords" value="#{jobParameters['numRecords']}?:1000;" />
  <property name="chunkSize" value="#{jobParameters['chunkSize']}?:100;" />
  <property name="dsJNDI" value="#{jobParameters['dsJNDI']}?:java:comp/env/jdbc/BonusPayoutDS;" />
  <property name="bonusAmount" value="#{jobParameters['bonusAmount']}?:100;" />
  <property name="tableName" value="#{jobParameters['tableName']}?:BONUSPAYOUT.ACCOUNT;" />
</properties>
```



```
<step id="generate" next="addBonus">
  <batchlet ref="com.ibm.websphere.samples.batch.artifacts.GenerateDataBatchlet">
    <properties>
      <property name="numRecords" value="#{jobProperties['numRecords']}" />
    </properties>
  </batchlet>
</step>
:
```

*(second part on next chart)*

The first step is defined as a Batchlet. The Java class file that implements the Batchlet is indicated. The property to tell the Batchlet how many records to create is specified.

**The job specification is taking shape. What about the second step? That's shown next ...**

# JSL: Job Specification Language, Part 2

*(first part on previous chart)*

```

:
<step id="addBonus">
  <chunk item-count="{jobProperties['chunkSize']}">
    <reader ref="com.ibm.websphere.samples.batch.artifacts.GeneratedCSVReader"/>
    <processor ref="com.ibm.websphere.samples.batch.artifacts.BonusCreditProcessor">
      <properties>
        <property name="bonusAmount" value="{jobProperties['bonusAmount']}" />
      </properties>
    </processor>
    <writer ref="com.ibm.websphere.samples.batch.artifacts.AccountJDBCWriter">
      <properties>
        <property name="dsJNDI" value="{jobProperties['dsJNDI']}" />
        <property name="tableName" value="{jobProperties['tableName']}" />
      </properties>
    </writer>
  </chunk>
</step>
</job>
  
```

The second step is defined as a Chunk step. The “chunkSize” (commit interval) is a property from earlier.

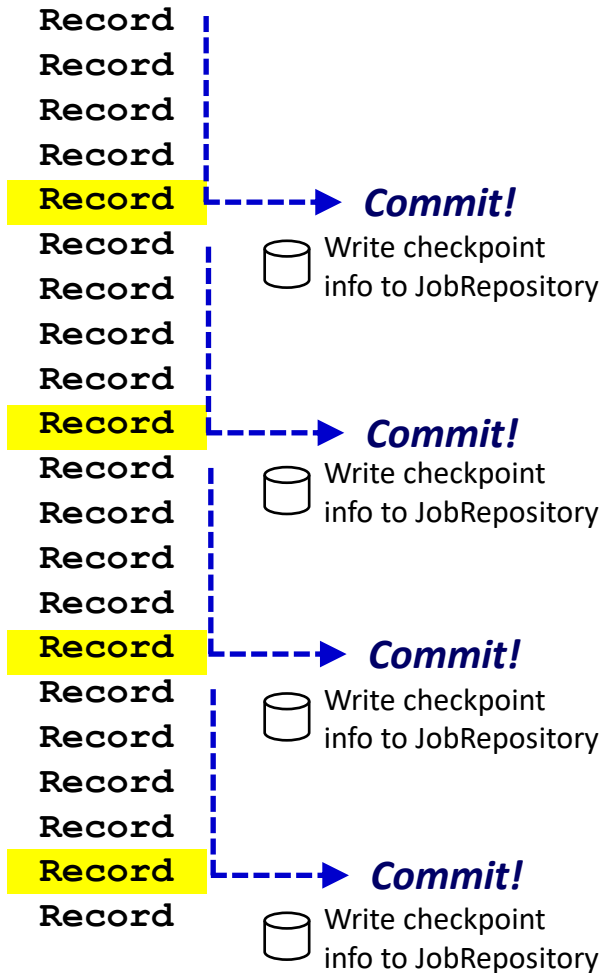
The “reader,” “processor” and “writer” Java classes are specified

A property on the processor provides the integer bonus to add to each account. Properties on the writer indicate how to reach the database and what table to use

**Summary: the JSR 352 runtime provides the infrastructure to run batch jobs; this JSL tells it what Java classes to use and other details related to the operation of the job**

# Checkpoint/Restart

```
<chunk item-count="5" />
```



Interval specified by `item-count` on `<chunk>` element for step in JSL

You may externalize with a property in the JSL, allowing you to pass the interval in at submission

Container wraps a transaction around update processing and commits at the specified interval

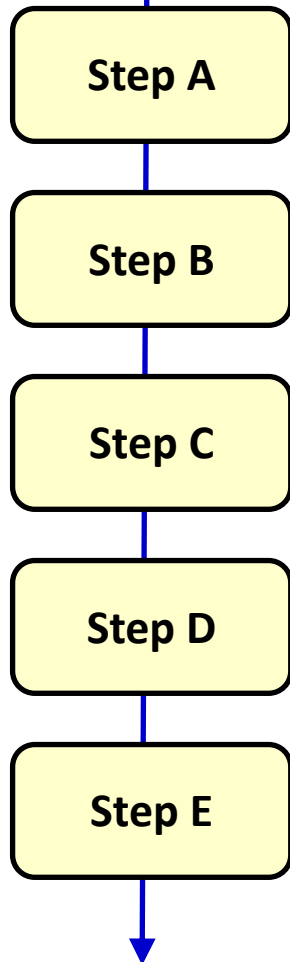
Container persists last-good commit point, and in the event of restart will pick up at last-good commit



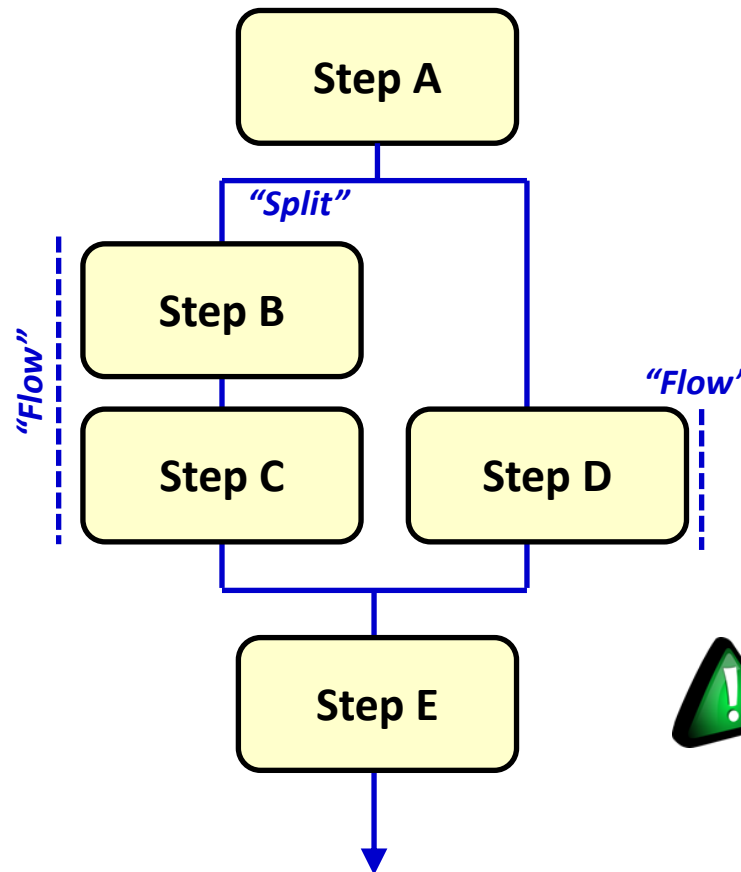
***This is a function of the JSR 352 container. Your code does not need to handle any of this.***

# Split/Flow Processing

Simple sequential step processing ...



... or, you may organize the steps to execute like this:



You specify in the JSL the way you want the splits and flows to process

Split steps will process on separate threads in the execution JVM

Step execution may be defined as conditional on previous step completion

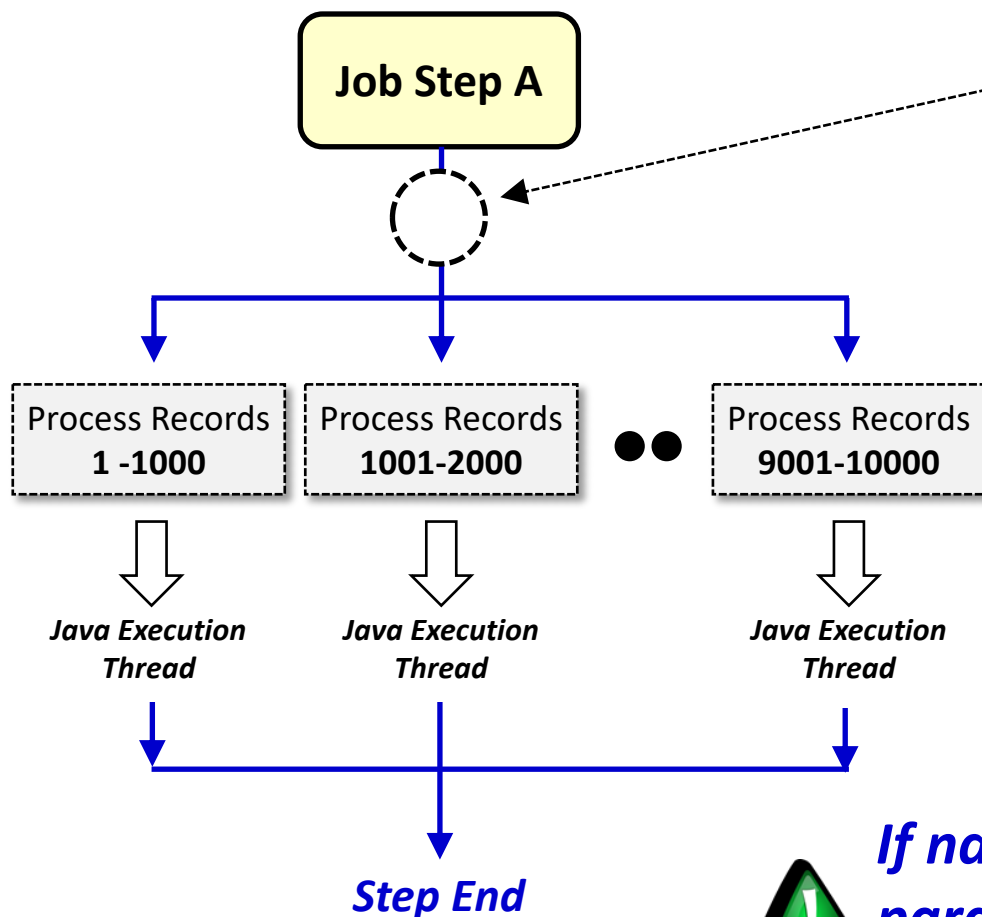
*Keep steps logically organized within a single job, but process in splits and flows if needed*



Same job steps, but job on right organized to run with splits and flow

# Step Partitions

Finer-grained parallel processing than splits-flows ... this is *within* a job step:



You may partition based on record ranges (passed-in parameters) or by indicating number of partitions (your code determines records ranges accordingly)

The container then dispatches execution on separate threads within the JVM with the specified properties for different data ranges.

When all partitions end, step ends



***If nature of job step lends itself to parallel processing, then partition across execution threads***

# Listeners

Think of “listeners” as “exits” -- points during execution of batch processing where your code would get control to do whatever you wish to do for that event at that time:

*Each is an  
implementable interface:*

*Receives control ...*

## JobListener

*... before and after a job execution runs, and if exception thrown*

## StepListener

*... before and after a step runs, and if exception thrown*

## ChunkListener

*... at the beginning and the end of chunk, and if exception thrown*

## ItemReadListener

*... before and after an item is read by an item reader, and if exception thrown*

## ItemProcessListener

*... before and after an item is processed by an item processor, and if exception*

## ItemWriteListener

*... before and after an item is written by an item writer, and if exception*

## SkipListener

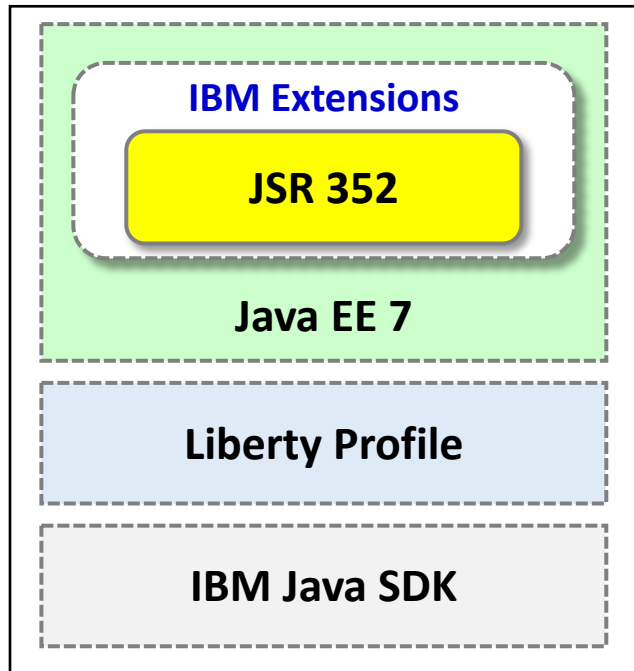
*... when a skippable exception is thrown from an item reader, processor, or writer*

## RetryListener

*... when a retryable exception is thrown from an item reader, processor, or writer*

# ***IBM Implementation And Extensions***

# Built on Liberty Profile as the Java Runtime Server



All Platforms Supported By Liberty Profile

## Liberty Profile 8.5.5.6 and above

- IBM's fast, lightweight, composable server runtime
- Dynamic configuration and application updates

## JVM Stays Active Between Jobs

- Avoids the overhead of JVM initialize and tear down for each job

## IBM Extensions to JSR 352

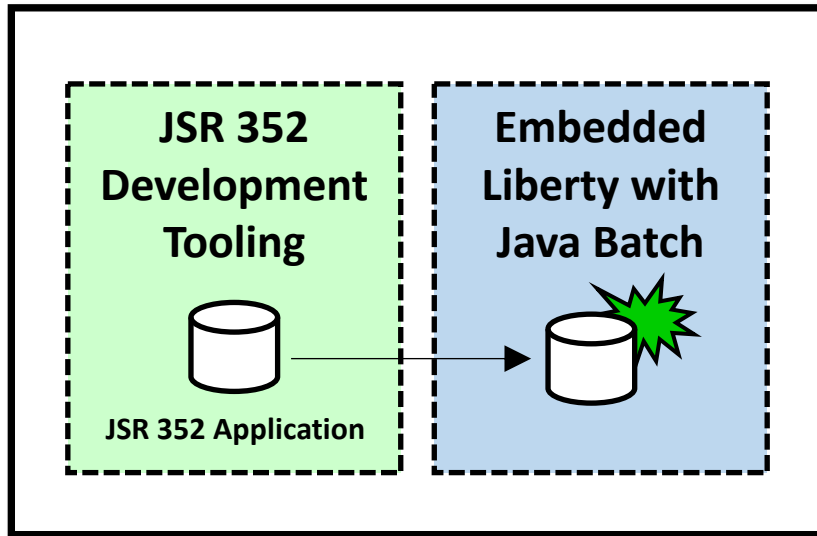
- JSR 352 is largely a *programming* standard
- IBM extensions augment this with valuable *operational* functions
- Includes:
  - Job logs separated by job execution
  - REST interface to JobOperator
  - Command line client for job submission
  - Integration with enterprise scheduler functions
  - Multi-JVM support: dispatcher and endpoint servers provide a distributed topology for batch job execution
  - Inline JSL (8.5.5.7)
  - Batch events (8.5.5.7)

# WebSphere Liberty Java Batch

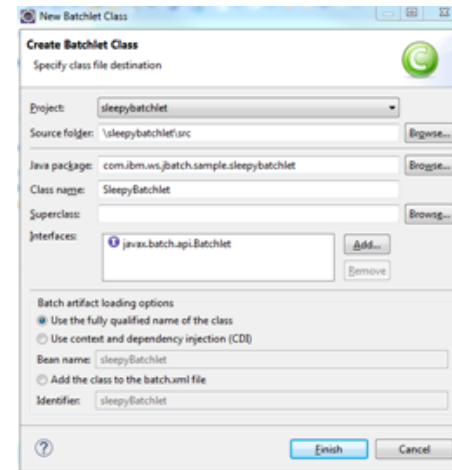


# Integration with WebSphere Developer Tools (WDT)

## Workstation Eclipse Platform



## Eclipse-based JSR 352 tooling



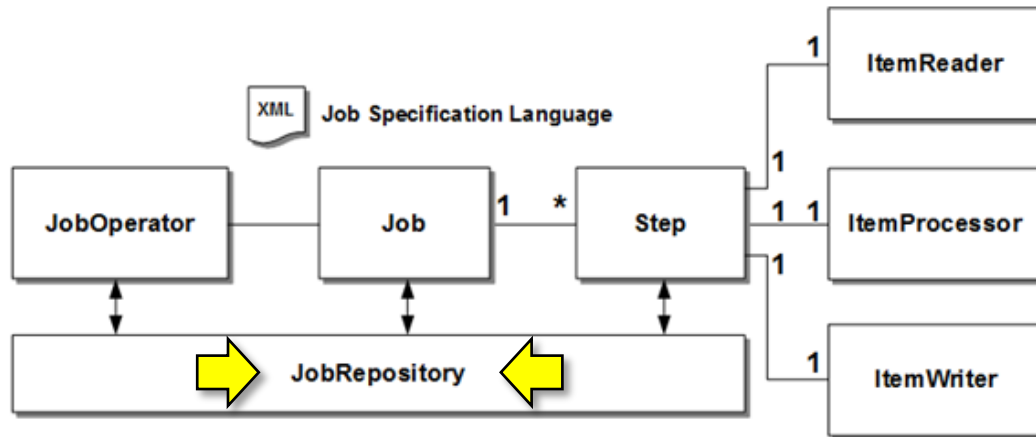
Understands the JSR 352 requirements, helps you build the implementation classes, and creates the JSL

Embedded Liberty allows you to deploy and run your application all within your Eclipse framework

TechDoc: <http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102639>

wasDev: <https://developer.ibm.com/wasdev/docs/creating-simple-java-batch-application-using-websphere-developer-tools/>

# JobRepository Implementation



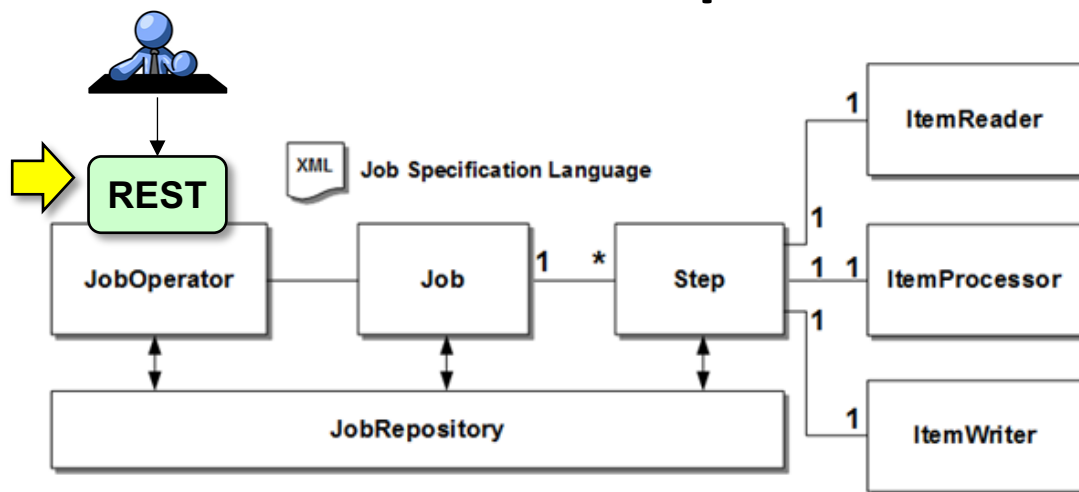
The JSR 352 standard calls for a JobRepository to hold job state information, but it does not spell out implementation details

## IBM WebSphere Liberty Batch provides three options for this:

1. **An in-memory JobRepository**  
For development and test environments where job state does not need to persist between server starts
2. **File-based Derby JobRepository**  
For runtime environments where a degree of persistence is desired, but a full database product is not needed
3. **Relational database product JobRepository**  
For production and near-production environments where a robust database product is called for

**Table creation is automatic. Relatively easy to drop one set of tables and re-configure to use a different data store.**

# REST Interface to JobOperator



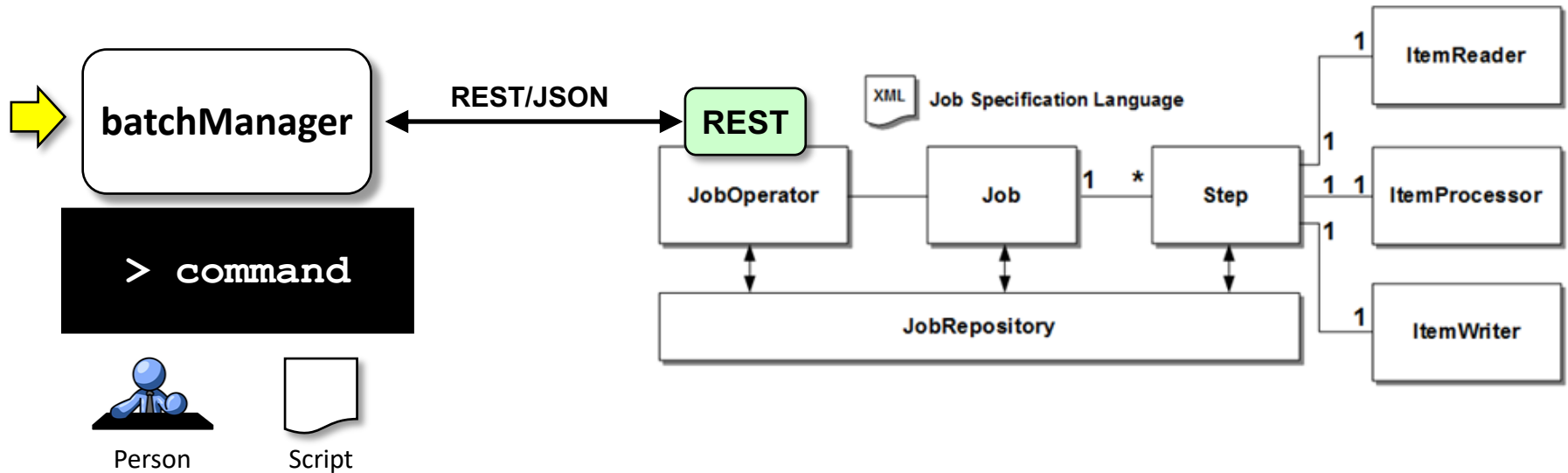
The JSR 352 standard calls for a *JobOperator interface*, but leaves to vendors to implement function to handle external requests for job submission, control and monitoring

## The IBM WebSphere Liberty Batch REST interface provides:

1. **A RESTful interface for job submission, control and monitoring**  
Job submission requests may come from outside the Liberty Profile runtime
2. **Security model for authentication and authorization**  
Authorization is role-based: administrator, submitter, monitor
3. **JSON payload carries the specifics of the job to be submitted**  
With information such as the application name, the JSL file name, and any parameters to pass in

**This permits the remote submission and control of jobs; it provides a way to integrate with external systems such as schedulers**

# Command Line Client to REST Interface

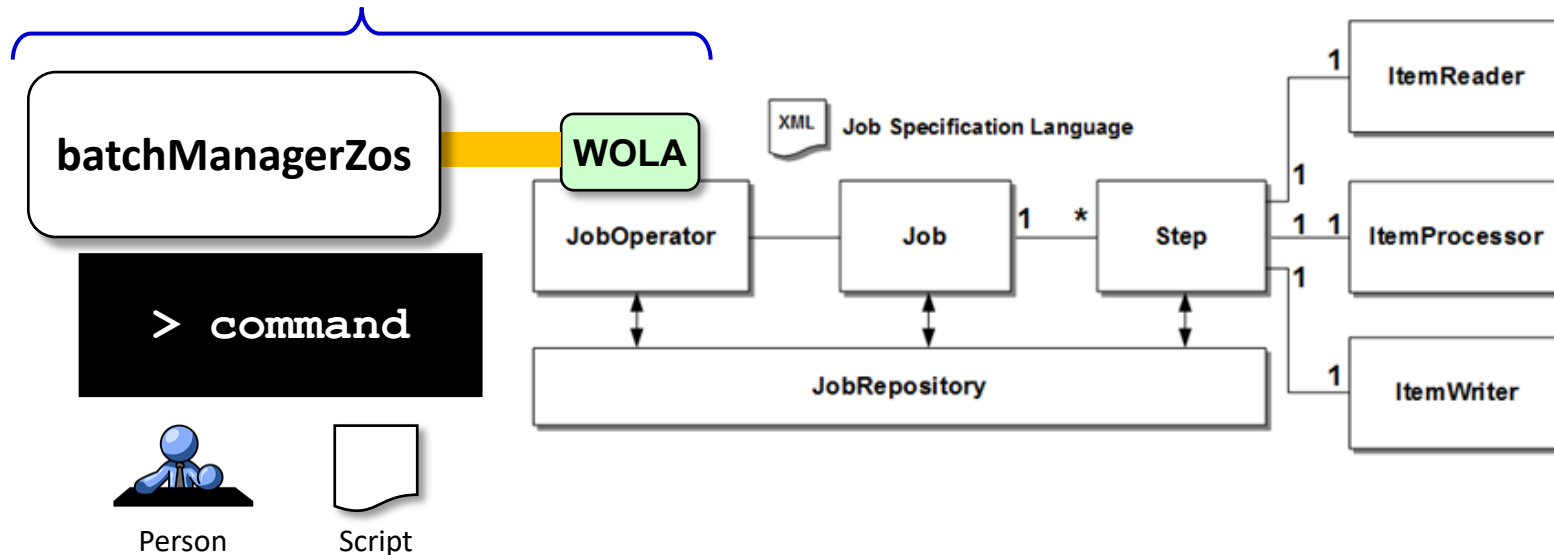


## The batchManager command line interface client provides:

1. **A way to submit, monitor and control jobs remotely using a command line interface**  
On the same system, or a different system ... different OS ... doesn't matter: TCP/IP and REST/JSON
2. **Uses the REST interface on the IBM Java Batch server**  
Which means the same security model is in effect: SSL, authentication, role-based access
3. **External schedulers can use this to submit and monitor job completion**  
batchManager parameters allow the script to "wait" for Java to complete. Parameters allow for discovery of job log information, and a mechanism to retrieve the job log for archival if desired.

# z/OS: Native Program Command Line Interface

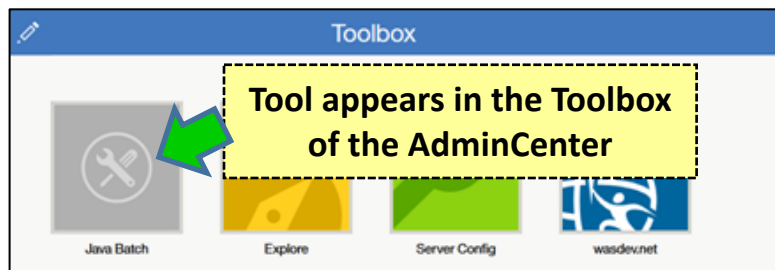
*Same LPAR, cross-memory*



## Same batchManager command line function, but ...

1. **Not a Java client, so do not need to spin up a JVM for each invocation**  
Saves the CPU associated with initiating the JVM, and when there's a lot of jobs this can be significant
2. **Cross-memory**  
Very low latency, and since no network then no SSL and management of certificates
3. **Same access security model**  
Once the WOLA connection is established, the same "admin," "submitter" and "monitor" roles apply

# AdminCenter Java Batch Tool **16.0.0.4**



## Graphical interface to list jobs and their status, and view job logs

Java Batch

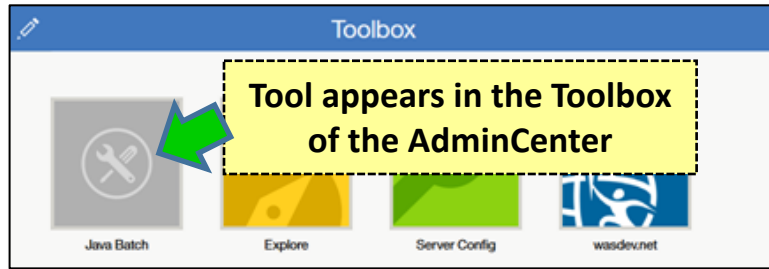
Batch Job Name	Instance ID	Application Name	Job Execution ID	Batch Status	Exit Status	Instance State
sleepy-batchlet	157	SleepyBatchletSample-1.0	157	Completed	COMPLETED	Completed
sleepy-batchlet	156	SleepyBatchletSample-1.0	156	Completed	COMPLETED	Completed
sleepy-batchlet		batchletSample-1.0				Completed
sleepy-batchlet		batchletSample-1.0				Completed
sleepy-batchlet		batchletSample-1.0				Completed
sleepy-batchlet		batchletSample-1.0				Completed
sleepy-batchlet		batchletSample-1.0				Completed
sleepy-batchlet	149	SleepyBatchletSample-1.0				Completed
sleepy-batchlet	148	SleepyBatchletSample-1.0				Completed

The Job Repository is queried and a list of jobs is provided, including their status.

```
[10/21/16 12:10:03:327 GMT] com.ibm.ws.batch.JobLogger
-----
Started invoking execution for a job
JobInstance id = 157
JobExecution id = 157
Job Name = sleepy-batchlet
Job Parameters = {}
-----
[10/21/16 12:10:03:327 GMT] com.ibm.ws.batch.JobLogger
-----
Completed invoking execution for a job
JobInstance id = 157
JobExecution id = 157
Job Name = sleepy-batchlet
Job Parameters = {}
Job Batch Status = COMPLETED, Job Exit Status = COMPLETED
-----
```

You can display the job log from the tool as well.

# AdminCenter Java Batch Tool **17.0.0.1**



## Update to include Job 'STOP' and Job 'RESTART'

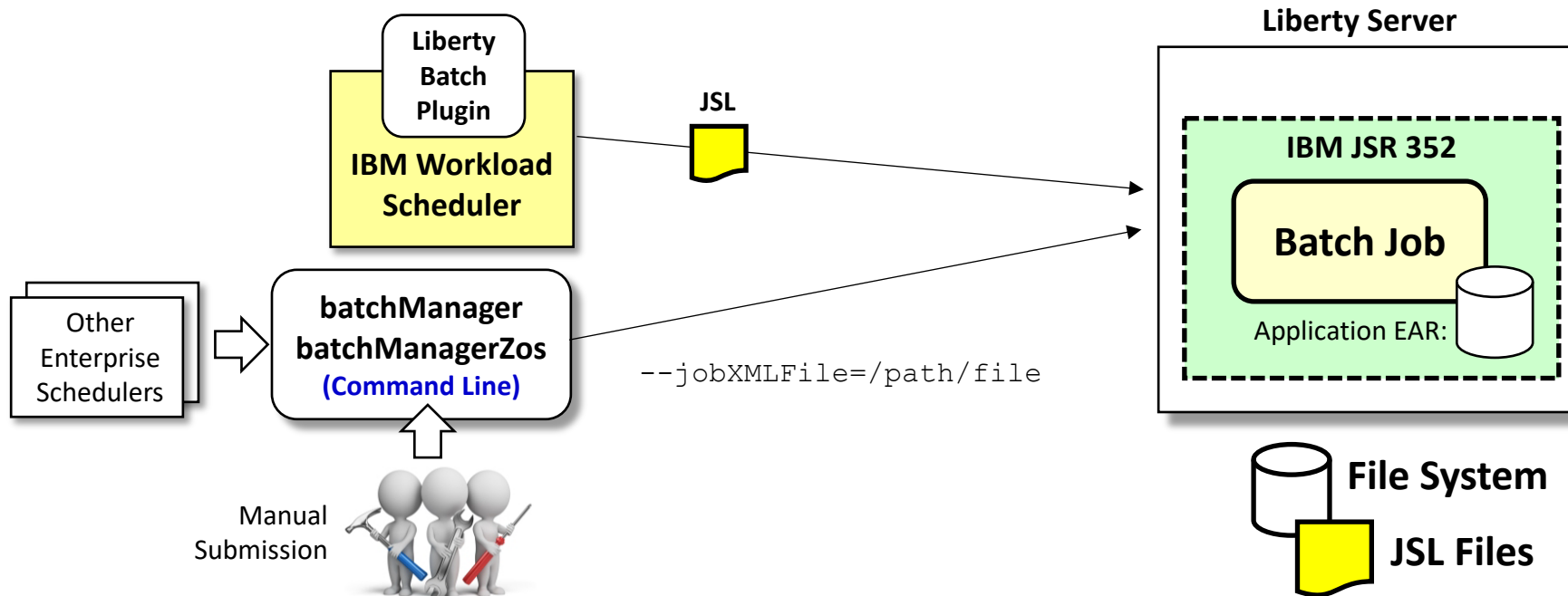
Java Batch

Instance ID	JES Job Name	Application Name	Submitter	Instance State	Actions
123	WLPADM	SleepyBatchletSample-1.0	bob	Dispatched	
122	DEMO2	SleepyBatchletSample-1.0	bob	Completed	Restart
121	DEMO2	SleepyBatchletSample-1.0	bob	Completed	Stop

The "Actions" button now allows you to act upon a job -- either Stop a running job, or Restart a job that is in a restartable state

# Inline JSL 8.5.5.7

Provides a way to maintain Job Specification Language (JSL) file *outside* the batch job application package file

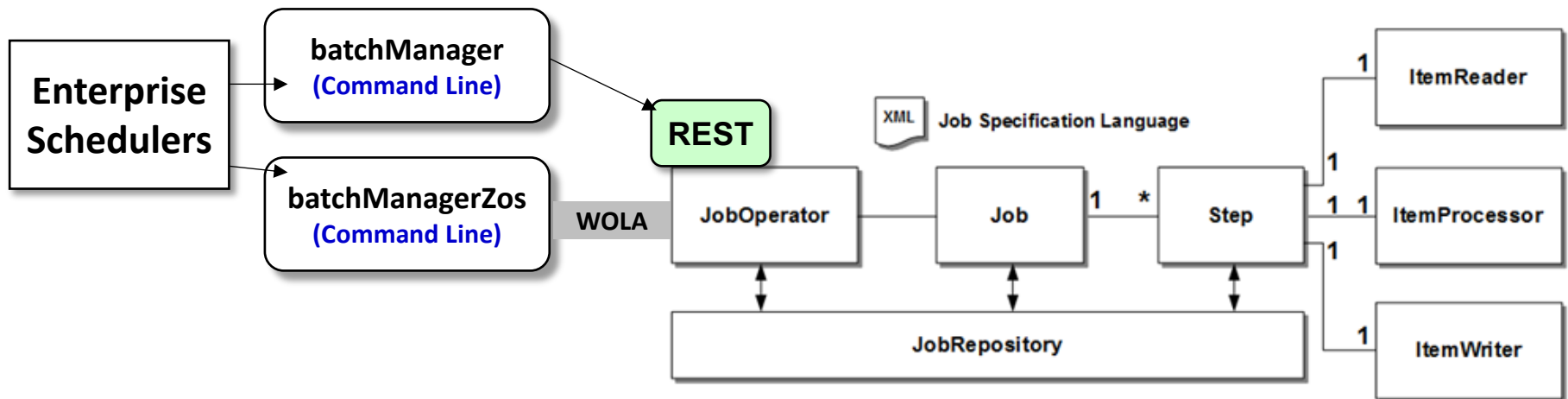


## Provides flexibility in where you maintain JSL files:

1. Package JSL in application EAR or maintain outside EAR and point to at submission
2. Can use from command line utilities with `--jobXMLFile=` parameter
3. IBM Workload Scheduler can be configured to pass in the JSL file to use



# Integration with Enterprise Schedulers

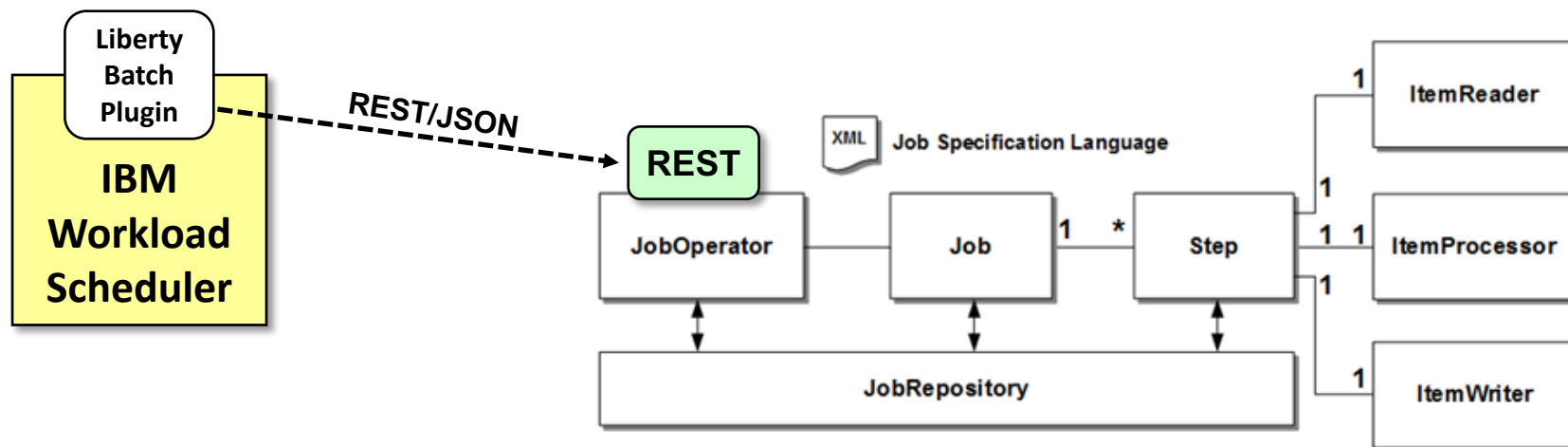


## The `batchManager` and `batchManagerZos` utilities provide this

1. `batchManager` is a command line interface that integrates with REST interface  
This can be used on z/OS or on other platforms, same LPAR or across-LPARs
2. `batchManagerZos` is same command line interface but uses cross-memory WOLA  
Used on the same LPAR as the batch server, it is very fast because of cross-memory WOLA

Submit jobs, check status of jobs, retrieve job logs

# IBM Workload Scheduler Integration

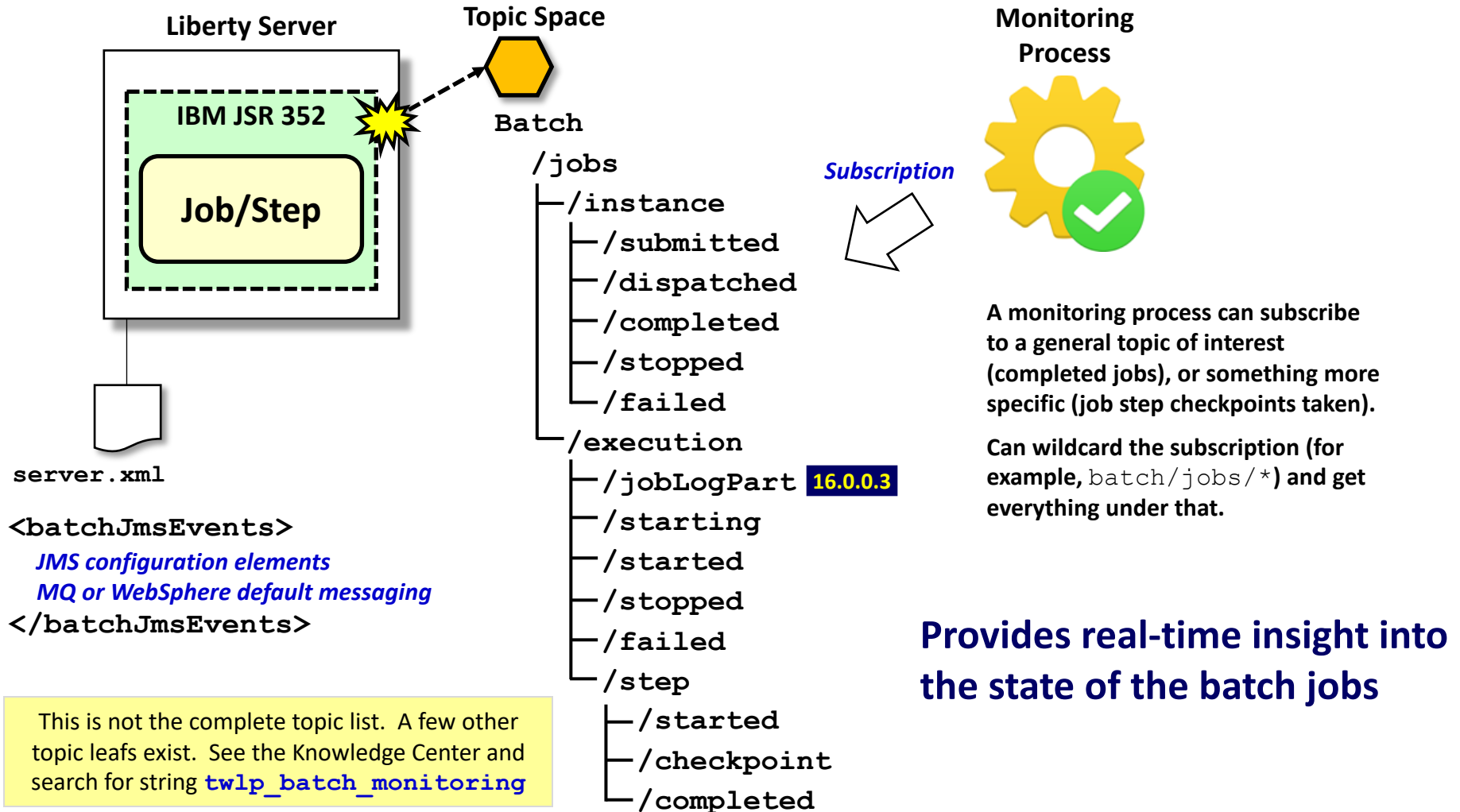


## IWS can integrate *directly* with the REST interface of IBM JSR 352

1. Eliminates the need for the command line interface utilities  
Simplicity of design ... Command line interfaces may be used by other enterprise schedulers
2. Can be used by IBM Liberty Batch on z/OS or on distributed operating systems
3. Supports IBM's inline JSL file function
4. A recorded demonstration can be seen here: <http://youtu.be/VF5TyZN-MP0>

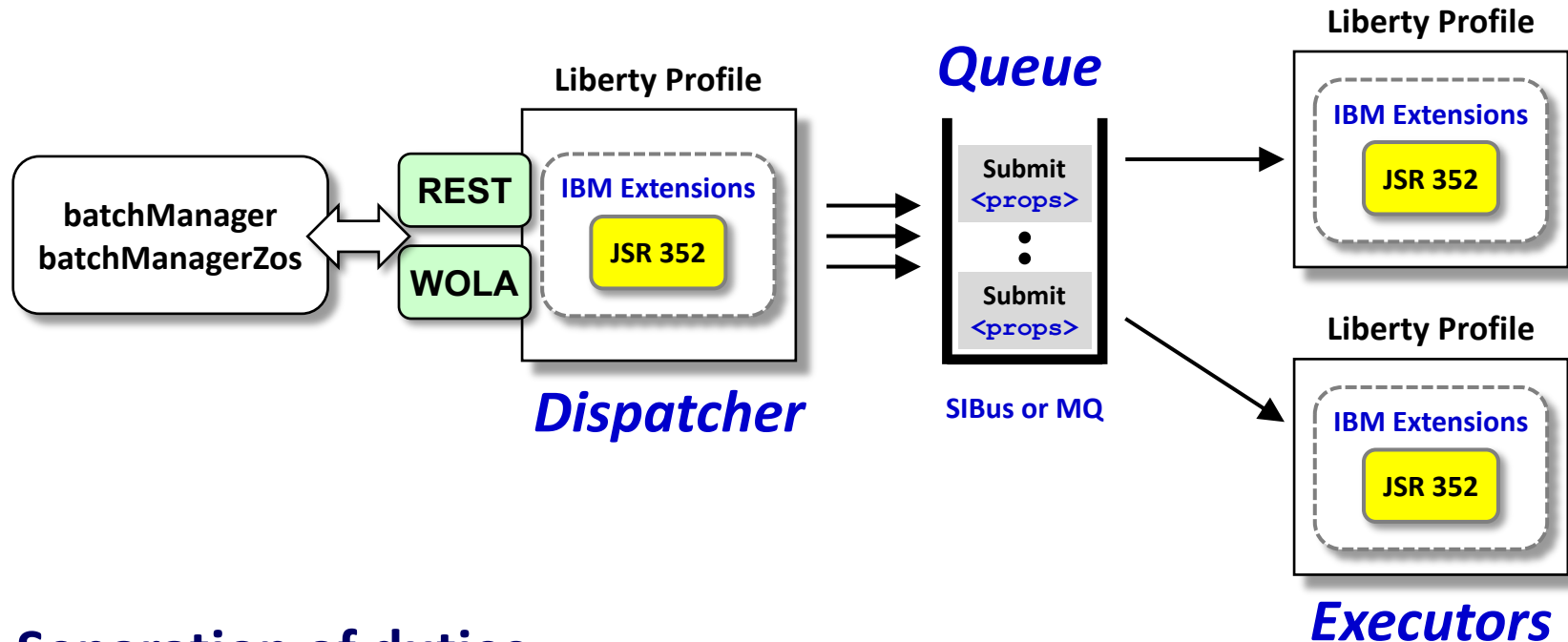
# Batch Events 8.5.5.7

Emit messages to a JMS topic space at key events during the batch job lifecycle:



This is not the complete topic list. A few other topic leaves exist. See the Knowledge Center and search for string [twlp\\_batch\\_monitoring](#)

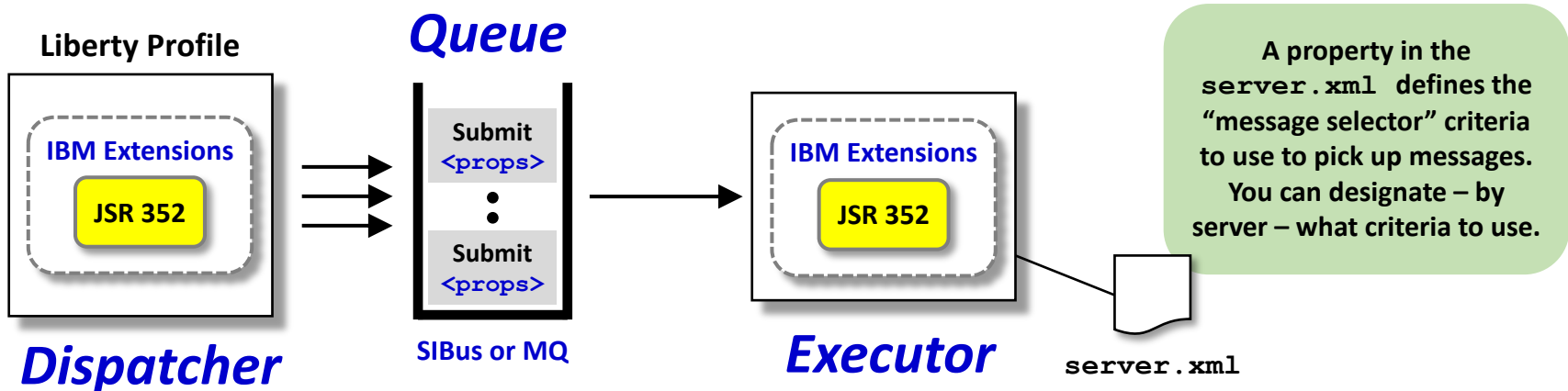
# Multi-JVM Support: Job Dispatchers, End-Points



## Separation of duties ...

- 1. Server designated as dispatchers handle job requests, and places them on JMS queue**  
 The endpoints listen on the JMS queues and pick up the job submission request based on criteria you set to indicate which jobs to pick up (*more on that next chart*)
- 2. Endpoint servers run the batch jobs**  
 Deploy the batch jobs where most appropriate; co-locate some batch jobs and others have their own server
- 3. JMS queues (either Service Integration Bus or MQ) serve as integration between two**  
 This provides a mechanism for queuing up jobs prior to execution

# Multi-JVM Support: Get Jobs Based on Endpoint Criteria



```
... messageSelector="com_ibm_ws_batch_applicationName = 'BatchJobA'"
```

**1**

```
... messageSelector="com_ibm_ws_batch_applicationName = 'BatchJobA'
                    OR com_ibm_ws_batch_applicationName = 'BatchJobB'"
```

**2**

```
... messageSelector="com_ibm_ws_batch_applicationName = 'BatchJobA'
                    AND com_ibm_ws_batch_myProperty = 'myValue'"
```

**3**

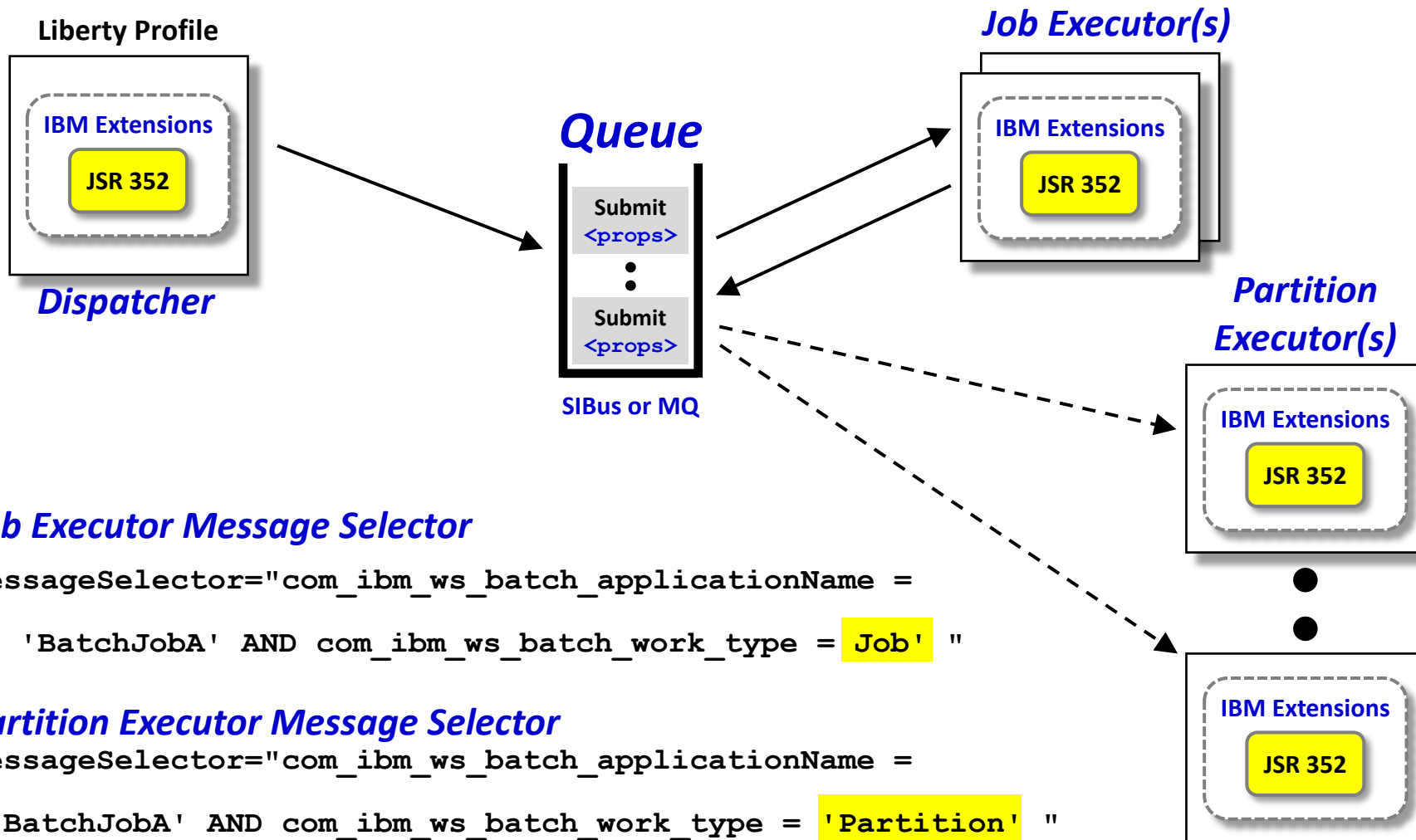
Submit jobs and have them run only when intended server starts and picks up the submission request

Have jobs run in intended servers based on selection criteria of your choice

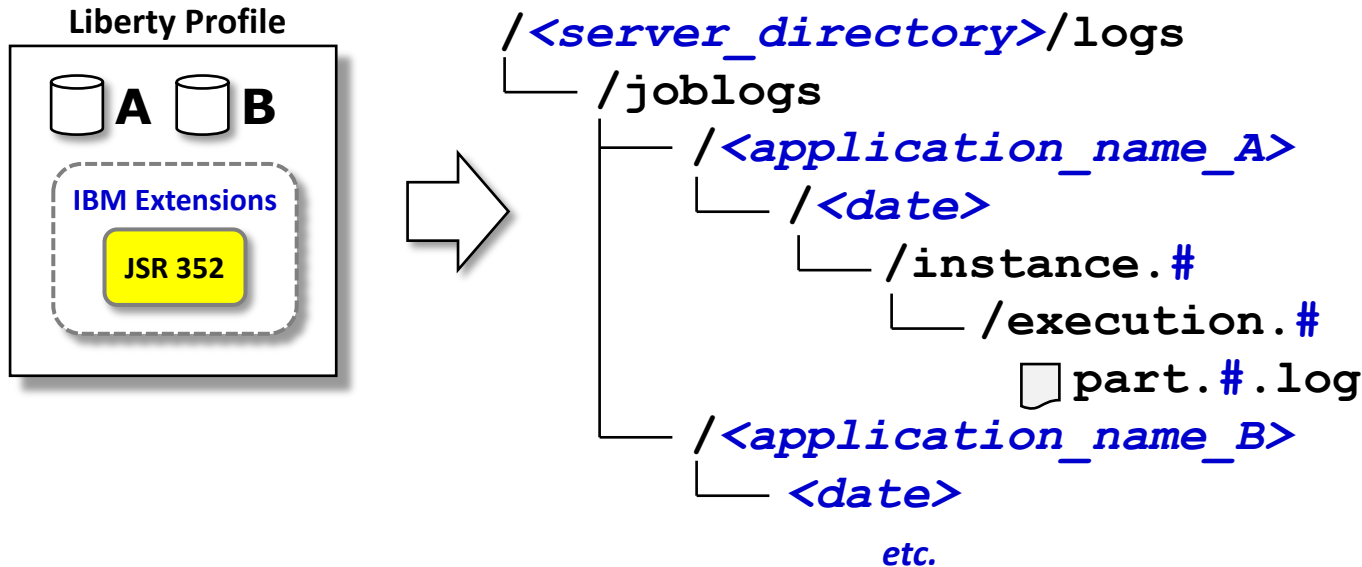
Not limited to system, not limited to platform ... may span systems and platforms

# Multi-JVM Support: Partitions 8.5.5.8

This is an extension of the earlier “Step Partition” feature, but here across separate JVMs



# Job Logging



## Job logs separate from the server log, separate from each other

1. Each job's logs are kept separate by application name, date, instance and execution
2. The IBM JSR 352 REST interface has a method for discovery and retrieval of job logs  
This is accessible through the batchManager command line interface as well. This is how job log retrieval and archival can be achieved if needed.
3. Also publish the logs to the `jobLogPart` topic (as noted earlier) as each log part closes or on a timer basis **16.0.0.4**

# SMF 120.12 Records for Java Batch 16.0.0.3

Records written at end of each step and at end of job

## SMF 120.12 record sections:

Standard Header	1 / record
Subsystem Section	1 / record
Identification Section	1 / record
Completion Section	1 / record
Processor Section	1 / record
Accounting Section	0 - n / record
USS Section	1 / record

## Noteworthy fields in the SMF 120.12 record:

**Record Type** - step end / job end

**Server identification** - which Liberty ran the job

**Job identification** - job, step, execution id, app name, etc.

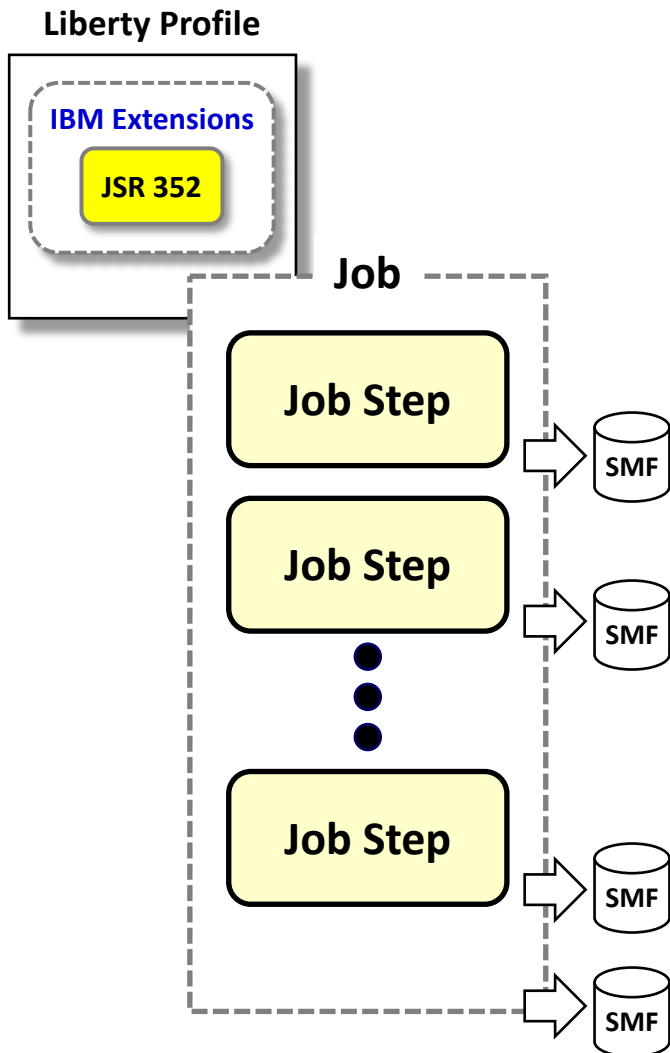
**Timestamps** - job submit, start, end; each step start / end

**JES Job Identifiers** - batchManagerZos JES jobname/ID

**Exit Status** - job or step completion code

**CPU times** - total CPU by job / step; GP and zIIP

**Accounting** - useful for accounting / chargeback





## Liberty Profile server.xml

```

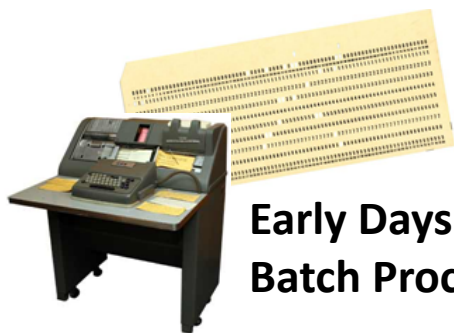
:
<featureManager>
  <feature>servlet-3.1</feature>
  <feature>batch-1.0</feature>
  <feature>batchManagement-1.0</feature>
</featureManager>
:
<batchPersistence jobStoreRef="BatchDatabaseStore" />
<databaseStore id="BatchDatabaseStore"
  dataSourceRef="batchDB" schema="JBATCH" tablePrefix="" />
:
  
```

### Relatively simple updates to server.xml ...

1. The `batch-1.0` feature enables the JSR 352 core functionality
2. The `batchManagement-1.0` feature enables the REST interface, job logging, and the ability to configure the multi-JVM support.
3. The `<batchPersistence>` element provides information about where the JobRepository is located

**Some details left out of this chart, of course ... but the key point is that configuring the support is based on updates to server.xml**

## Overall Summary



Early Days of Batch Processing

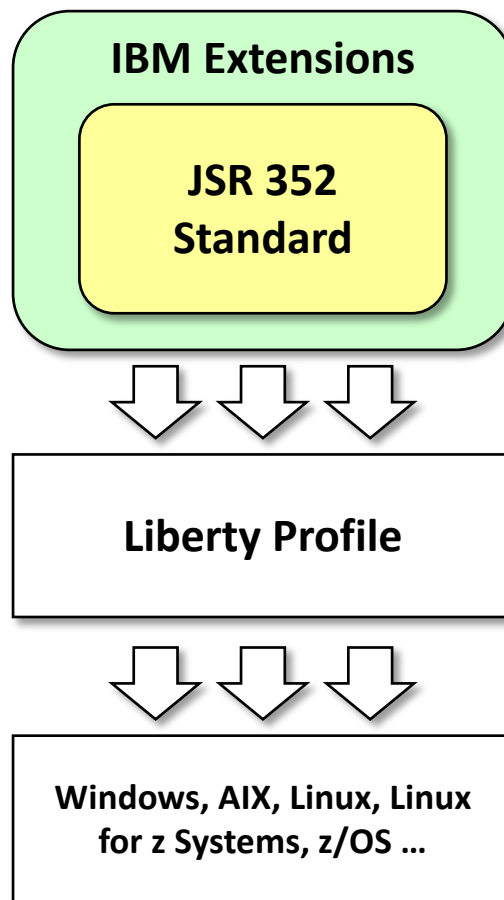


Over time ...

### Modernization

### Java

### JSR 352



Multiple JobRepository

Job logging

REST interface

Command line client

z/OS: native client

Multi-JVM capability

# IBM WebSphere Liberty Java Batch

## Other Documentation

### 8.5.5 Knowledge Center

[http://www-01.ibm.com/support/knowledgecenter/SSAW57\\_8.5.5/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/twlp\\_container\\_batch.html](http://www-01.ibm.com/support/knowledgecenter/SSAW57_8.5.5/com.ibm.websphere.wlp.nd.multiplatform.doc/ae/twlp_container_batch.html)

### The Techdoc for this presentation

<http://www.ibm.com/support/techdocs/atmastr.nsf/WebIndex/WP102544>

- Overview presentation
- Video (in case your access to YouTube is blocked)
- Quick Start Guide
- Detailed step-by-step implementation guide

### Other Techdocs related to Java Batch:

**Job Classification:** <http://www-03.ibm.com/support/techdocs/atmastr.nsf/WebIndex/WP102600>

**Batch Events:** <http://www-03.ibm.com/support/techdocs/atmastr.nsf/WebIndex/WP102603>

**Batch Topologies:** <http://www-03.ibm.com/support/techdocs/atmastr.nsf/WebIndex/WP102626>

**REST Interface:** <http://www-03.ibm.com/support/techdocs/atmastr.nsf/WebIndex/WP102632>

**Using DFSORT and IDCAMS:** <http://www-03.ibm.com/support/techdocs/atmastr.nsf/WebIndex/WP102636>

**Batch Migration:** <http://www-03.ibm.com/support/techdocs/atmastr.nsf/WebIndex/WP102638>

**Lab Materials:** <http://www-03.ibm.com/support/techdocs/atmastr.nsf/WebIndex/WP102639>

**Data Set Contention:** <http://www-03.ibm.com/support/techdocs/atmastr.nsf/WebIndex/WP102667>

**Batch SMF:** <http://www-03.ibm.com/support/techdocs/atmastr.nsf/WebIndex/WP102668>

### YouTube Video

<https://youtu.be/tRhKTMb-5lo>

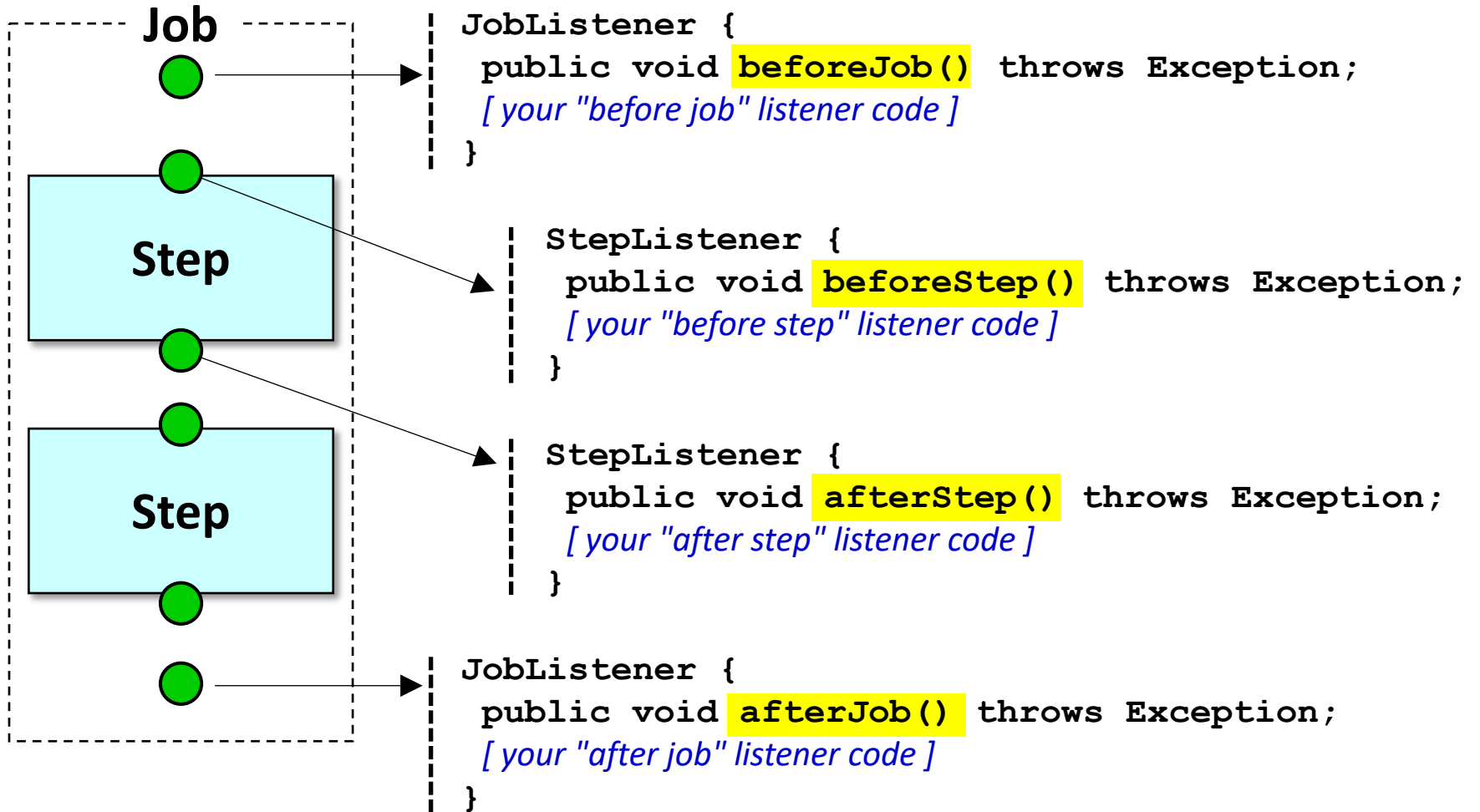
### Github Repository (for examples and other links)

<https://github.com/WASdev/sample.batch.bonuspayout/wiki/WebSphereLibertyBatchLinks>

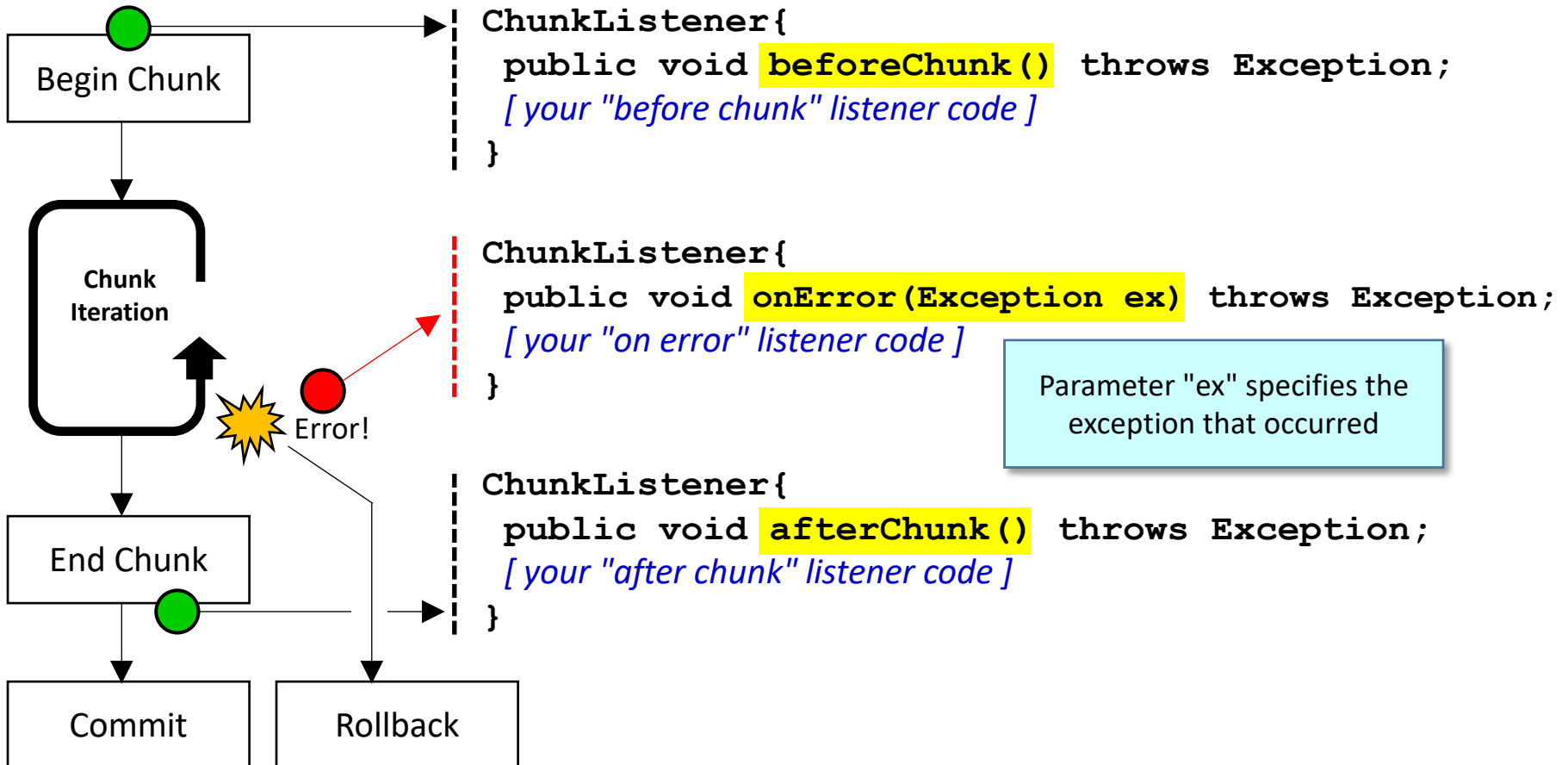
# Charts providing additional detail

# Listeners

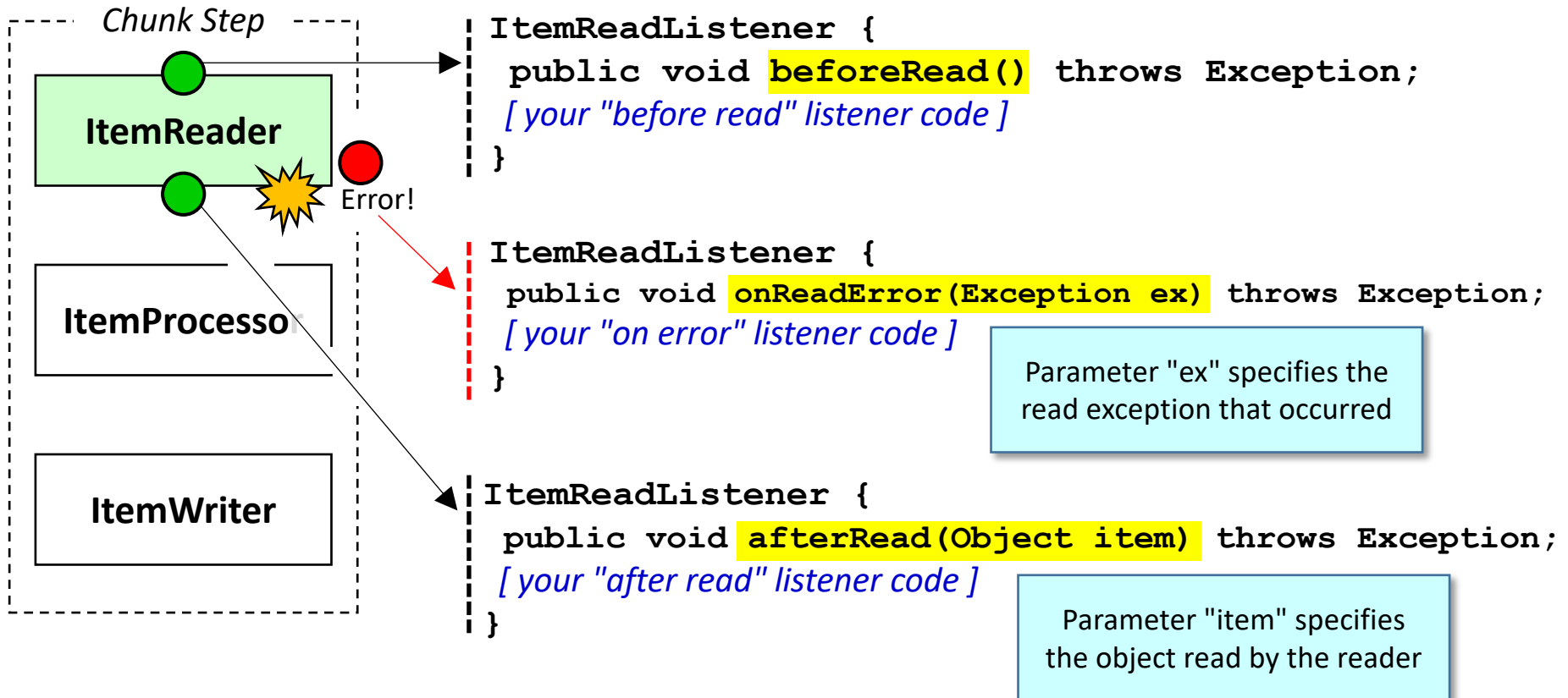
# Job and Step Listeners



# Chunk Listener

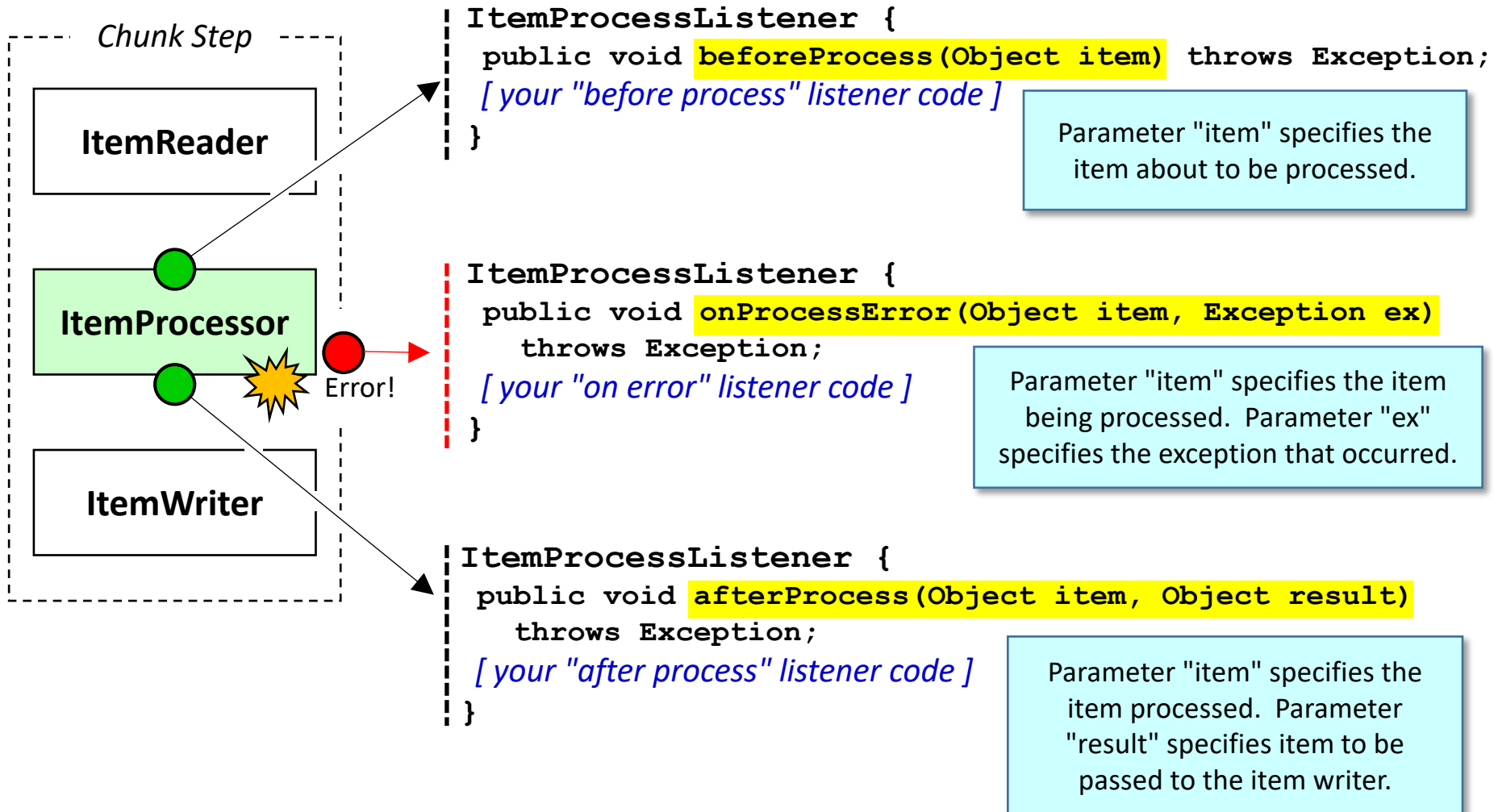


# ItemRead Listener

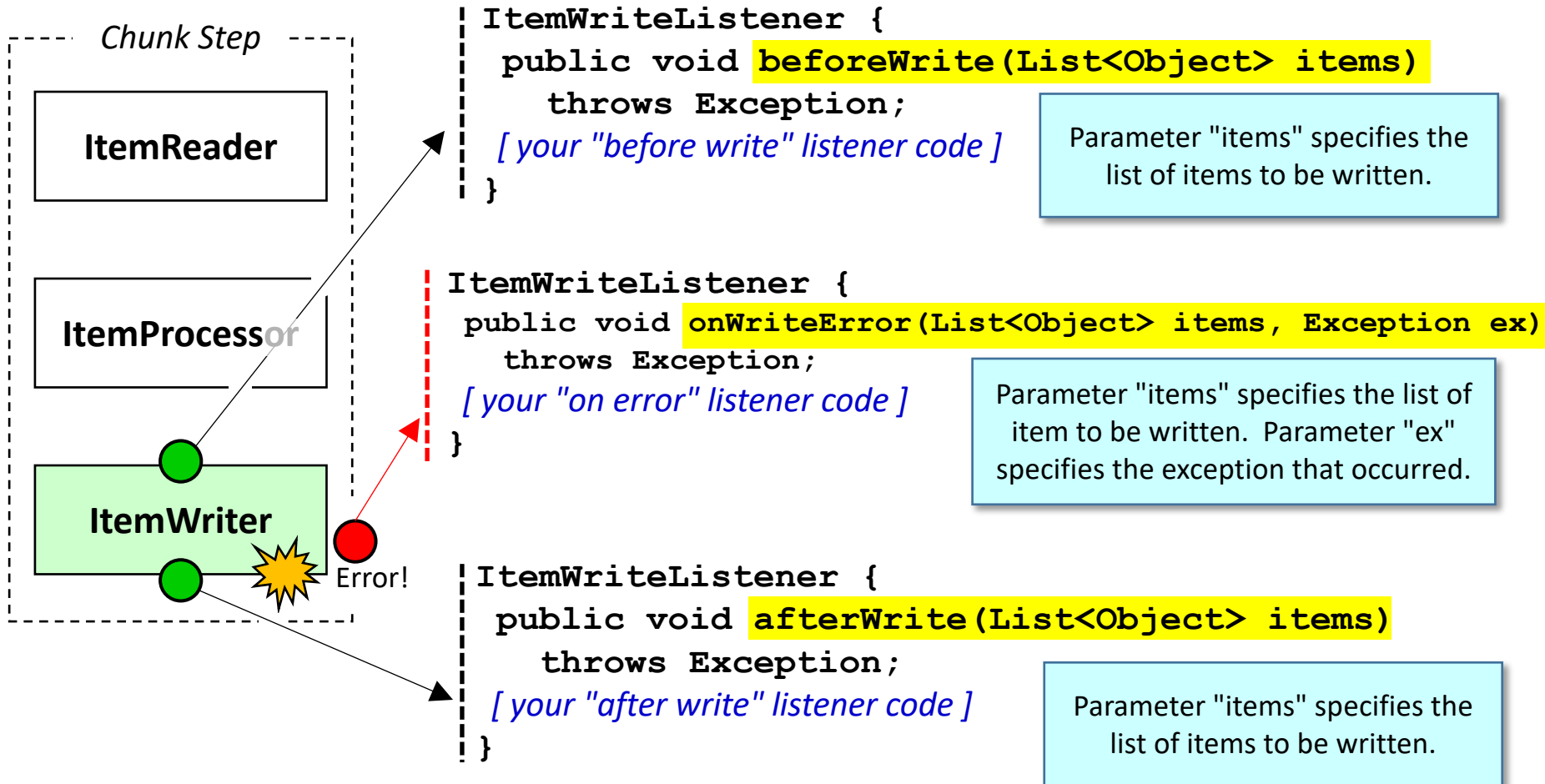




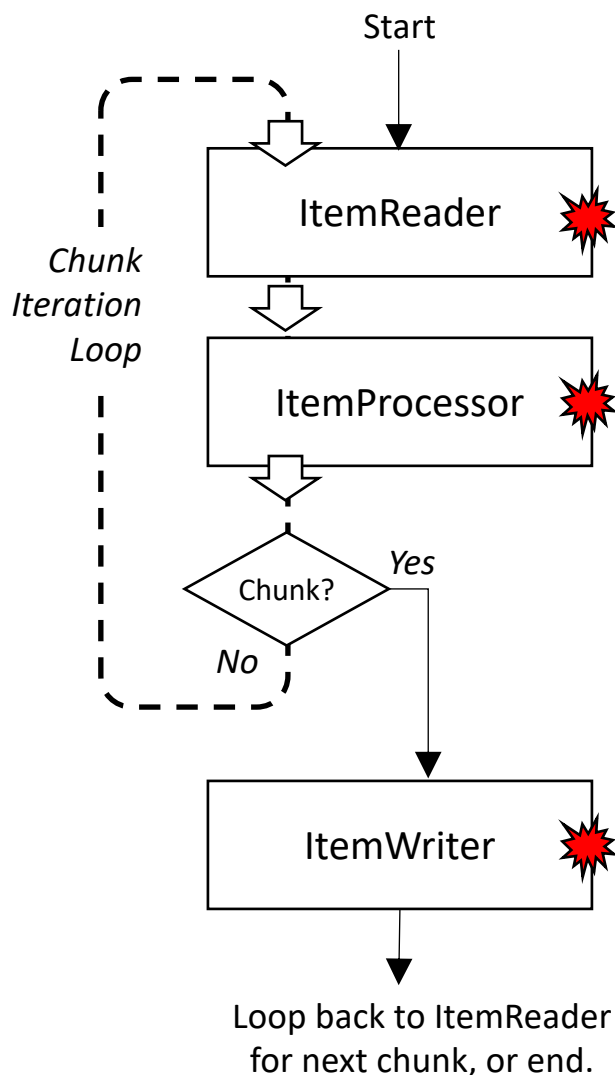
# ItemProcess Listener



# ItemWrite Listener



# Overview of Skip Exception Processing



Exceptions may occur in all three batch artifacts: *read, process, or write*.

In the absence of any "skippable exception" definition, an *unhandled* exception thrown results in the termination of the step.

You can define what types of unhandled exceptions can be "skipped"; that is, ignored and processing continues. You may define what is skipped for the chunk step in the JSL. Example:

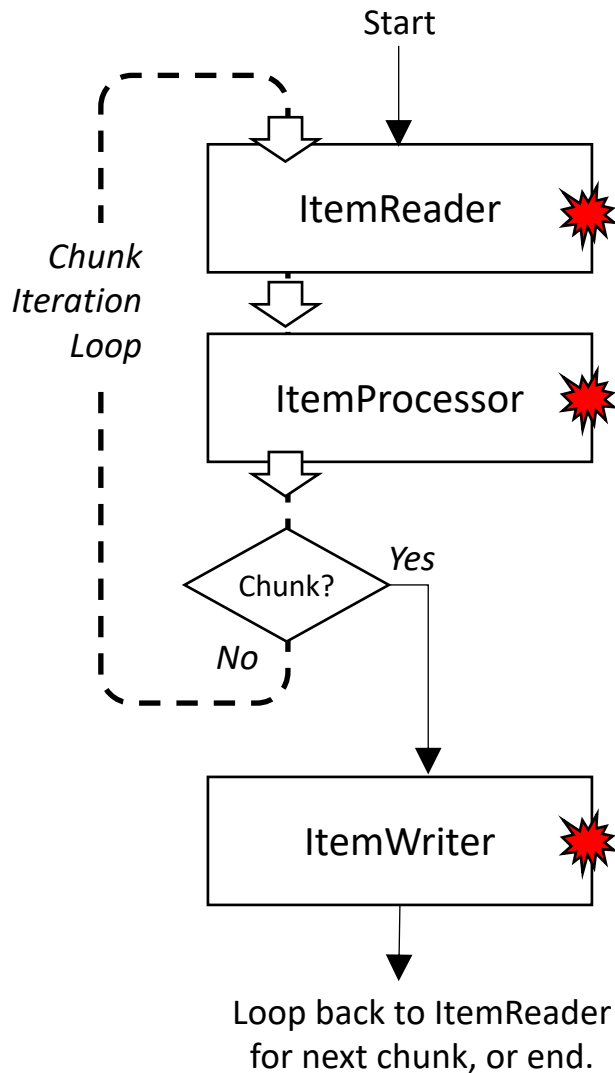
```
<skippable-exception-classes>
  <include class="java.lang.Exception"/>
  <exclude class="java.io.FileNotFoundException"/>
</skippable-exception-classes>
```

This would skip *all exceptions* except `java.io.FileNotFoundException` (along with any subclasses of `java.io.FileNotFoundException`).

Good practice: skip based on your own class names, never general Java exception classes.

The number of skips for a step may be limited by the `<skip-limit>` JSL element. Default is no limit.

# What Happens With a Skipped Exception?



**Skipped read -- the container calls ItemReader again and the next item is read.**

If you implement a SkipRead listener, you can capture information about the skipped record for processing later.

**Skipped process -- the container loops and calls ItemReader again. The next item is read and processed.**

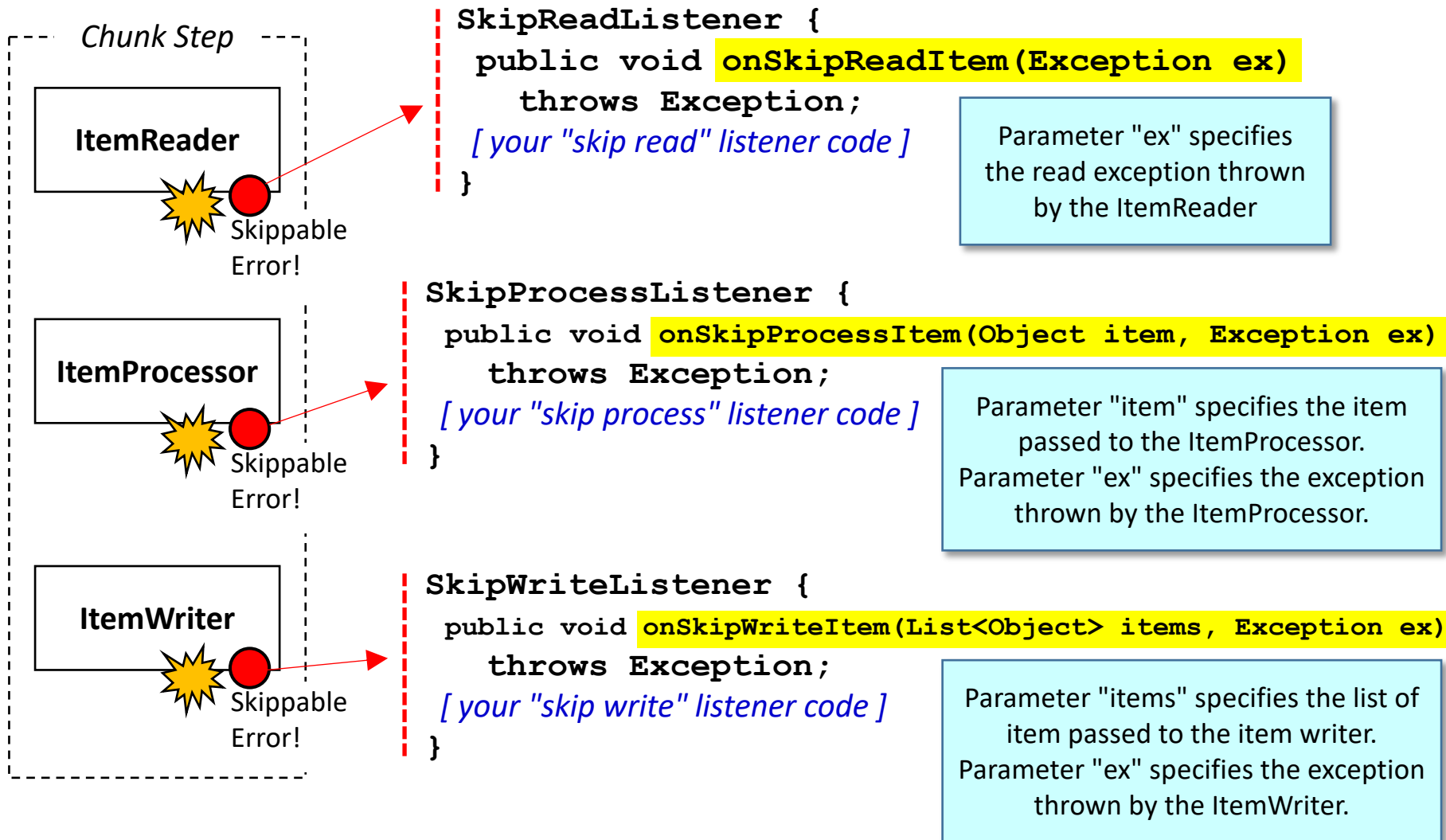
If you implement a SkipProcess listener, you can capture information about the item that wasn't processed.

**Skipped write -- the container commits the transaction with whatever was (or was not) written. The container starts a *new chunk* and calls the ItemReader.**

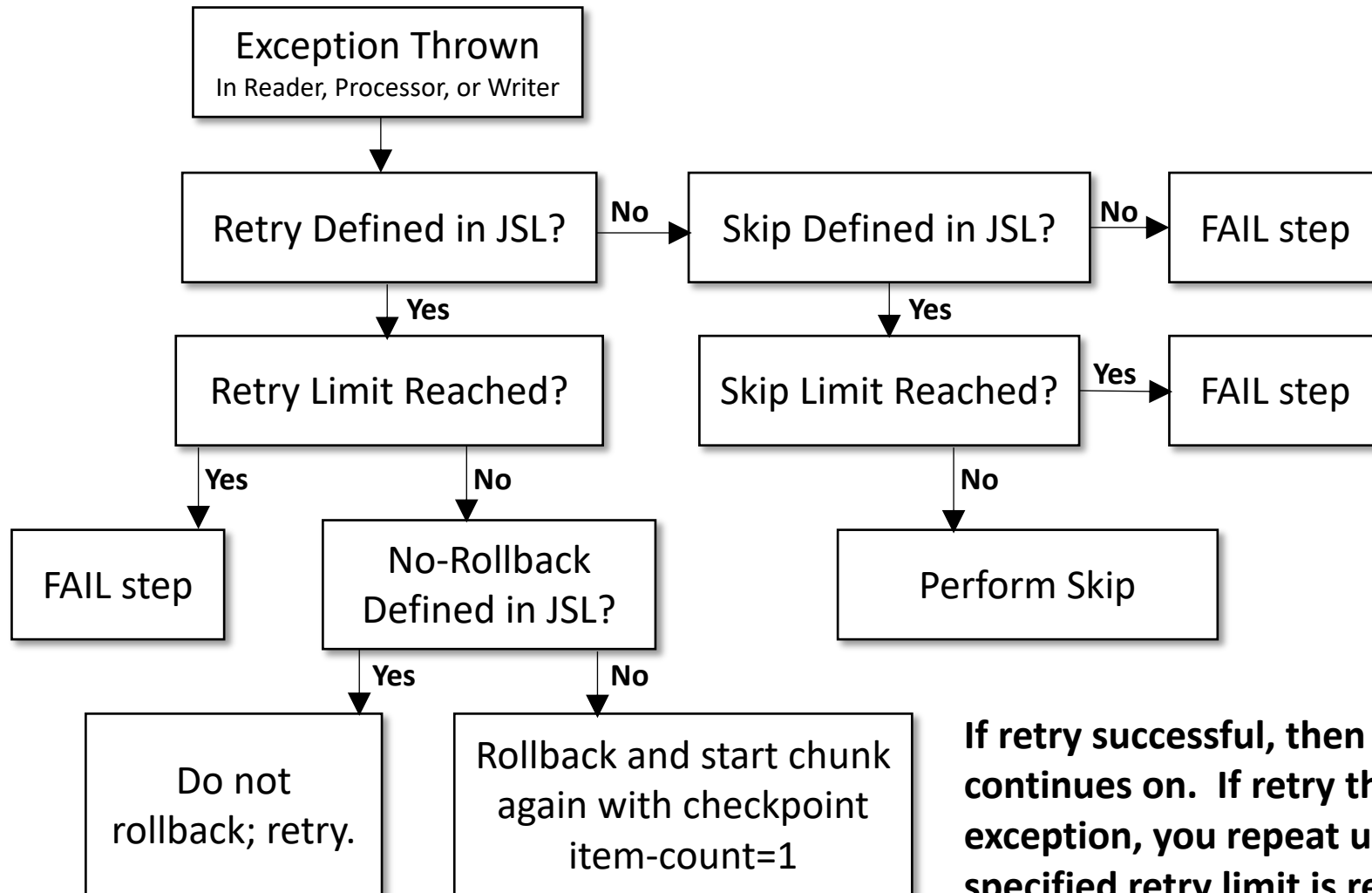
If you implement a SkipWrite listener, you can capture information about the list of items passed to the writer.

# Read, Process, and Write Skip Listeners

**Important!** Skip listeners only called if the exception is defined in JSL as "skippable."



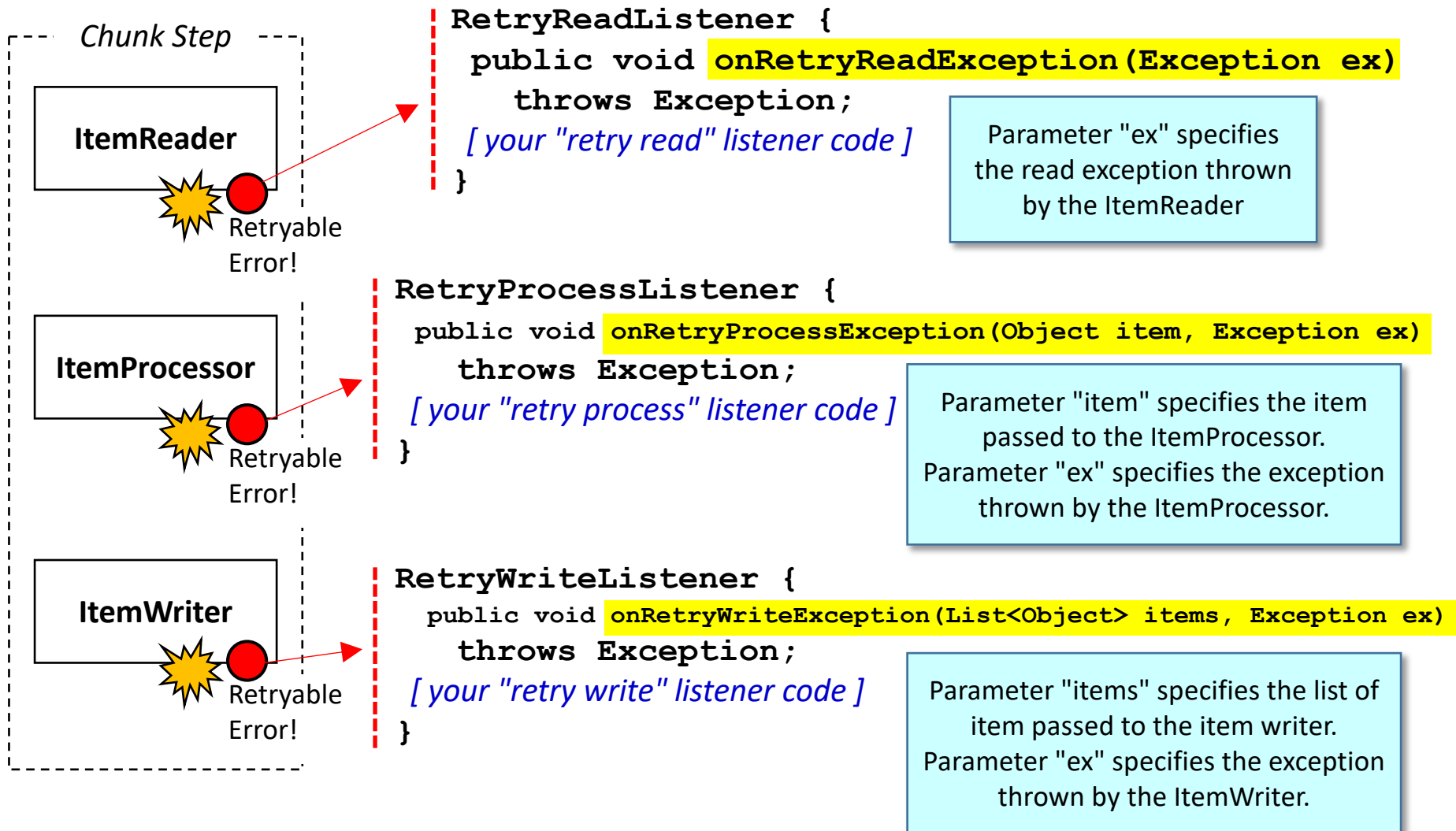
# Overview of Retry Exception Processing



**If retry successful, then the step continues on. If retry throws exception, you repeat until specified retry limit is reached**

# Read, Process, and Write Retry Listeners

**Important!** Retry listeners only called if the exception is defined in JSL as "retryable."

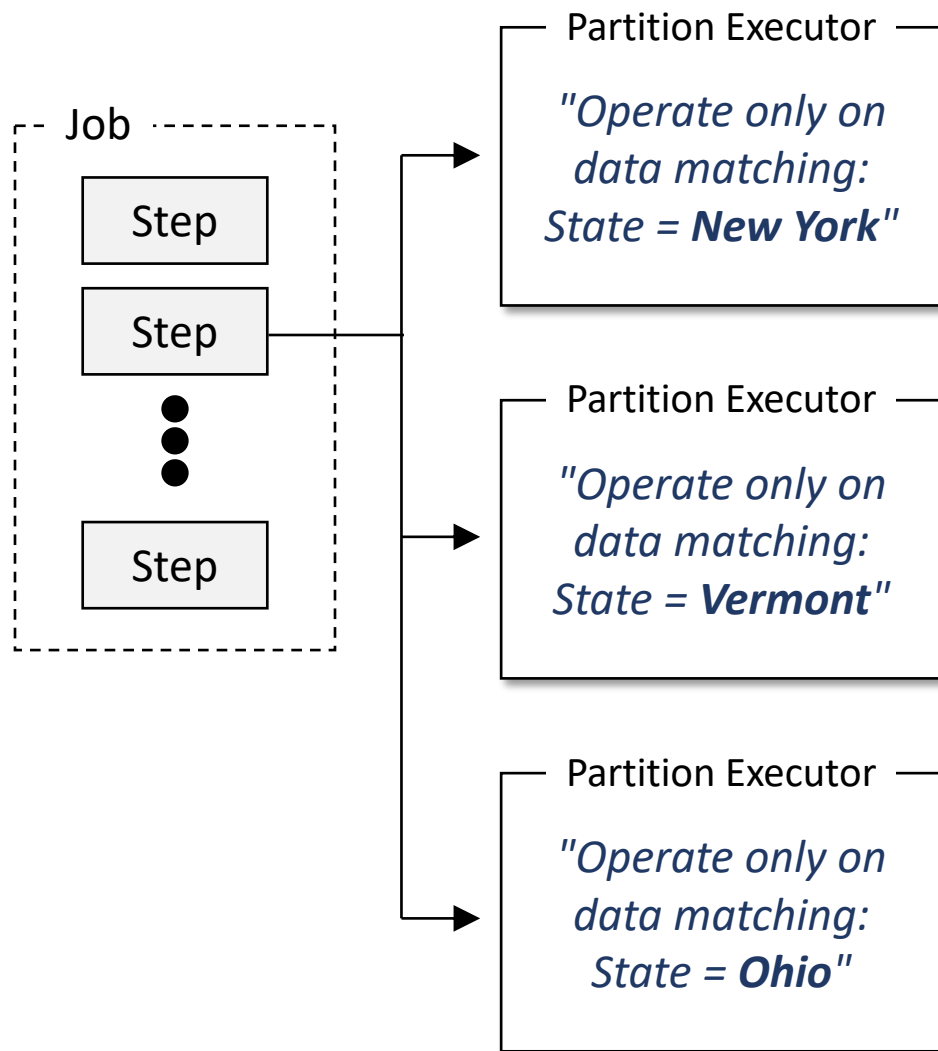




# Partitioning



# Example: Partitions to Operate on Data by State Name



**Two ways to accomplish this:**

- 1. Static values in JSL**
- 2. Using a partition mapper**

**We're showing three partitions, but you could have any number of partitions based on your data processing needs**

# Partitions: "Fixed"\* Definition in the JSL

JSL

```

<step id="Step1">
  <chunk item-count="10000">
    <reader ref="com.ibm.ws390.batch.Reader">
    </reader>
    <processor ref="com.ibm.ws390.batch.Processor">
      <properties >
        <property name="State" value="#{partitionPlan['State']}" />
      </properties>
    </processor>
    <writer ref="com.ibm.ws390.batch.Writer">
    </writer>
  </chunk>
  <partition>
    <plan partitions="3">
      <properties partition="0" />
      <property name="State" value="New York" />
    </properties>
    <properties partition="1" />
      <property name="State" value="Vermont" />
    </properties>
    <properties partition="2" />
      <property name="State" value="Ohio" />
    </properties>
  </plan>
</partition>
  :

```

The property is defined to the processor, and is indicated to come from a Partition Plan, which in this JSL is a fixed <plan>

The number of partitions is set to a fixed value of 3.

The first partition is given a property of "New York." It works on records related to New York only.

The other two partitions are given properties of "Vermont" and "Ohio." They work on their respective data records.

# Partitions: Using the Partition Mapper

JSL

```

<step id="Step1">
  <chunk item-count="10000">
    <reader ref="com.ibm.ws390.batch.Reader">
    </reader>
    <processor ref="com.ibm.ws390.batch.Processor">
      <properties >
        <property name="State" value="#{partitionPlan['State']}" />
      </properties>
    </processor>
    <writer ref="com.ibm.ws390.batch.Writer">
    </writer>
  </chunk>

  <partition>
    <mapper ref="com.ibm.ws390.batch.StateNameBasedPartitionMapper">

      <properties >
        <property name="States" value="#{jobParameters['States']}" />
      </properties>

    </mapper>
  </partition>
  
```

The property is defined to the processor, and down in the <partition> section a <mapper> is defined.

You write this class to provide the parameter string you need for your partitions.

*Partitions = 3*  
*Array of String Values*

"New York" , "Vermont" , "Ohio"

## Partition Mapper Specifics

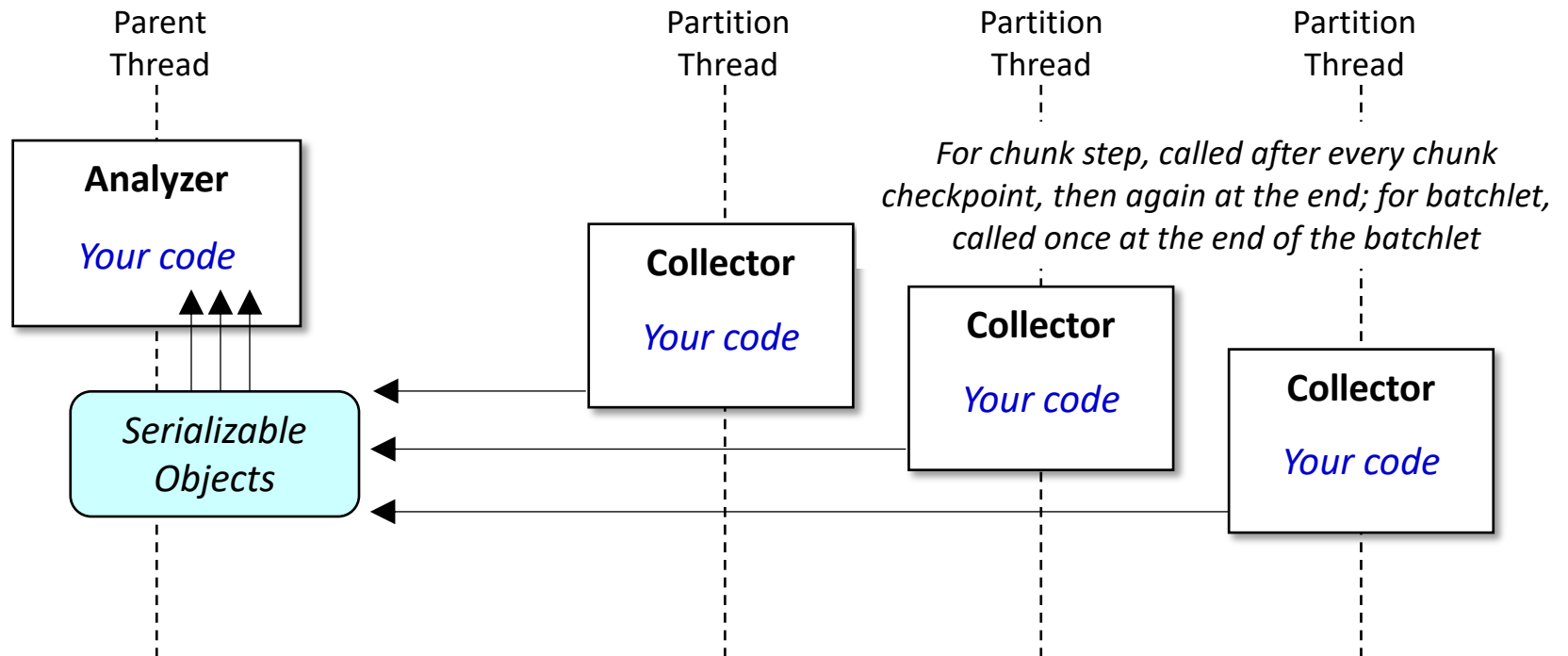
JSL

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

```
package javax.batch.api.partition;  
import javax.batch.api.partition.PartitionPlan;  
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;  
}
```

Returns the PartitionPlan. A sample implementation of the PartitionPlan is provided as PartitionPlanImpl in the JSR-352 specification.

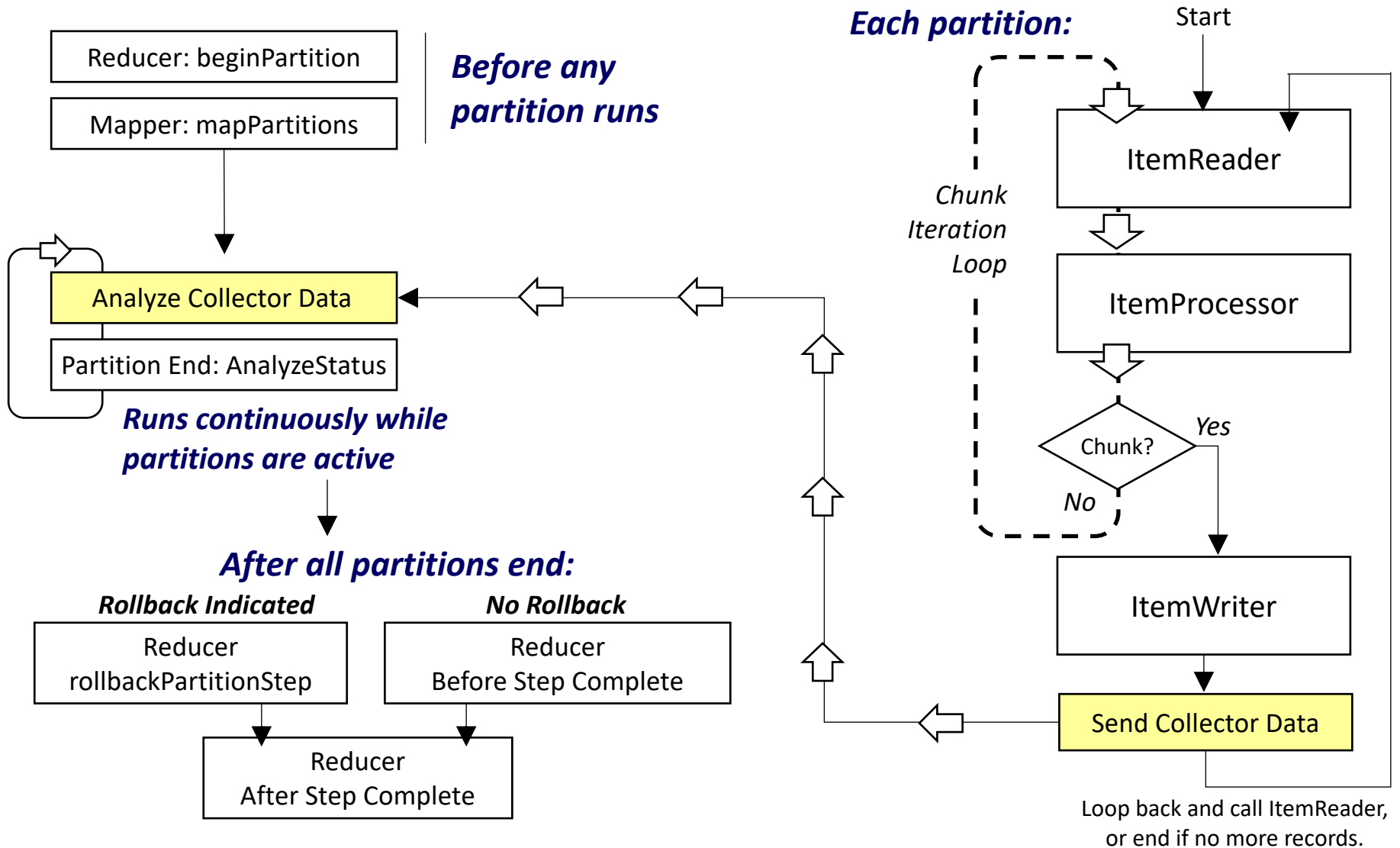
# Overview: Partition Collector and Analyzer



**These are optional interfaces you may implement if you wish to collect information from each partition during execution.**

**The collector runs in each partition and passes data over to the analyzer, which runs in the parent thread.**

# Overview: Chunk Step with Partitions



## Batch Contexts - Job Context

```
getJobName ()
```

```
getInstanceId ()
```

```
getExecutionId ()
```

```
getProperties ()
```

```
getBatchStatus ()
```

```
setTransientUserData (Object data)
```

```
getTransientUserData ()
```

```
setExitStatus (String status)
```

```
getExitStatus ()
```

This data is not persisted to the job repository; it does not exist past the life of the job.

This information stays local to the JVM in which it is set.

This is useful for passing information between steps in a job.

## Batch Contexts - Step Context

`getStepName()`

`getStepExecutionId()`

`getProperties()`

`getBatchStatus()`

`getException()`

`getMetrics()`

`setTransientUserData(Object data)`

`getTransientUserData()`

`setPersistentUserData(Serializable data)`

`getPersistentUserData()`

`setExitStatus(String status)`

`getExitStatus()`

The "step transient" data is different from the "job transient" data.

The "step persistent" data is stored in the job repository at each checkpoint, or at the end of a batchlet step.

For partitions each partition gets its own unique step context, so you can not communicate across partitions this way.

The step context can be a good way to communicate among the components of a step, such as between the reader and the reader listener or a skip listener.