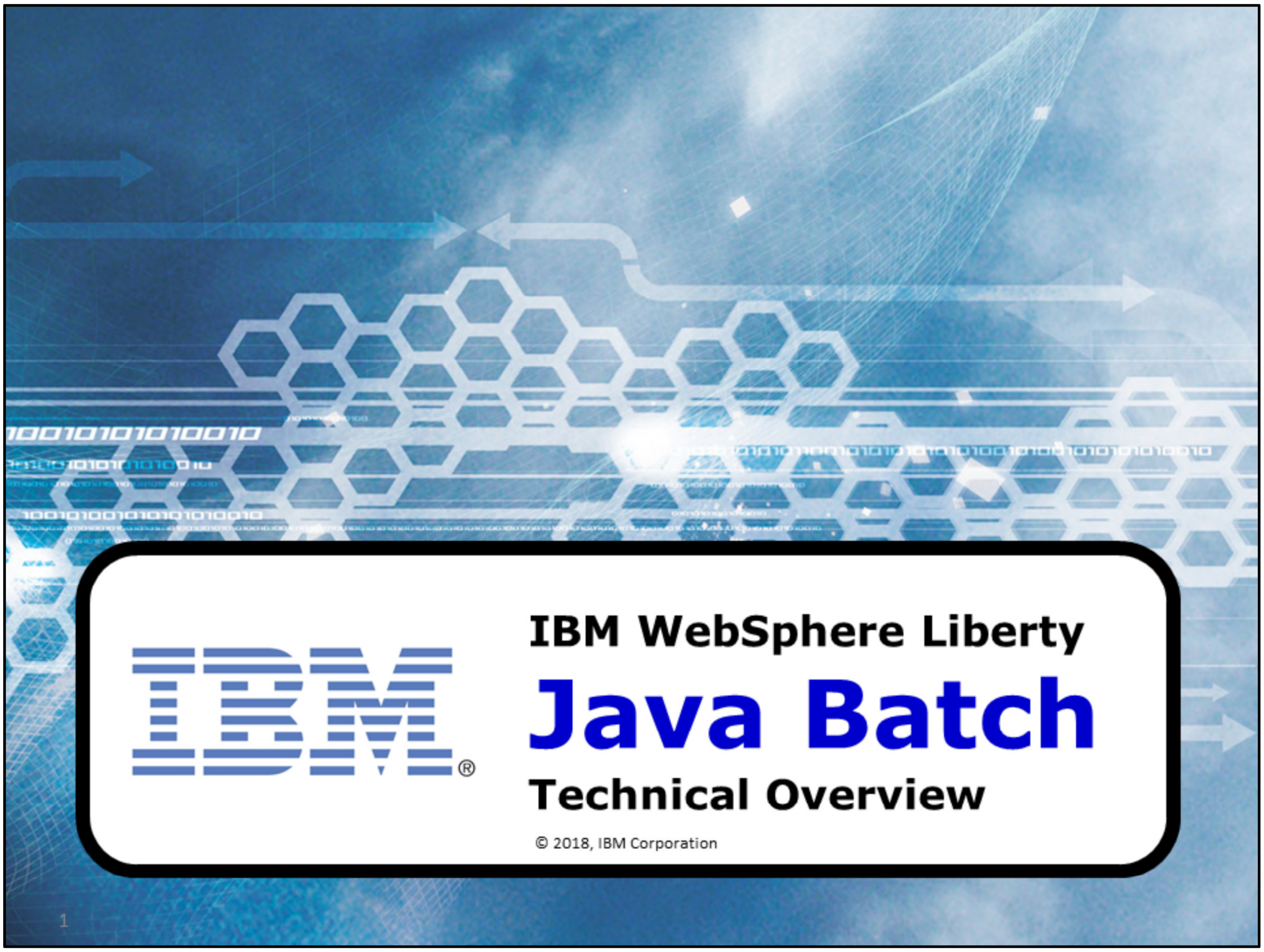
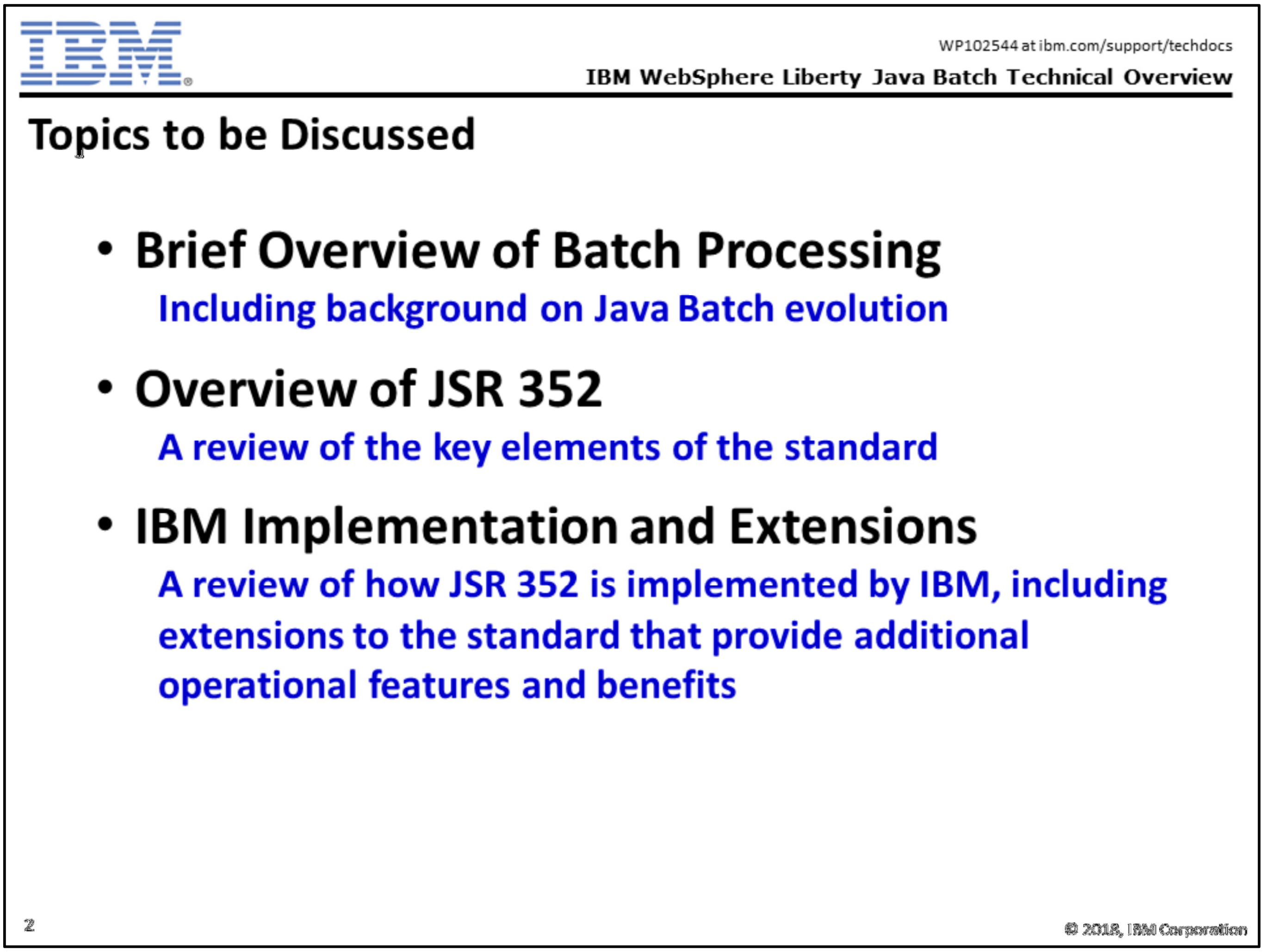


IBM WebSphere Liberty

Java Batch

Technical Overview

© 2018, IBM Corporation



Topics to be Discussed

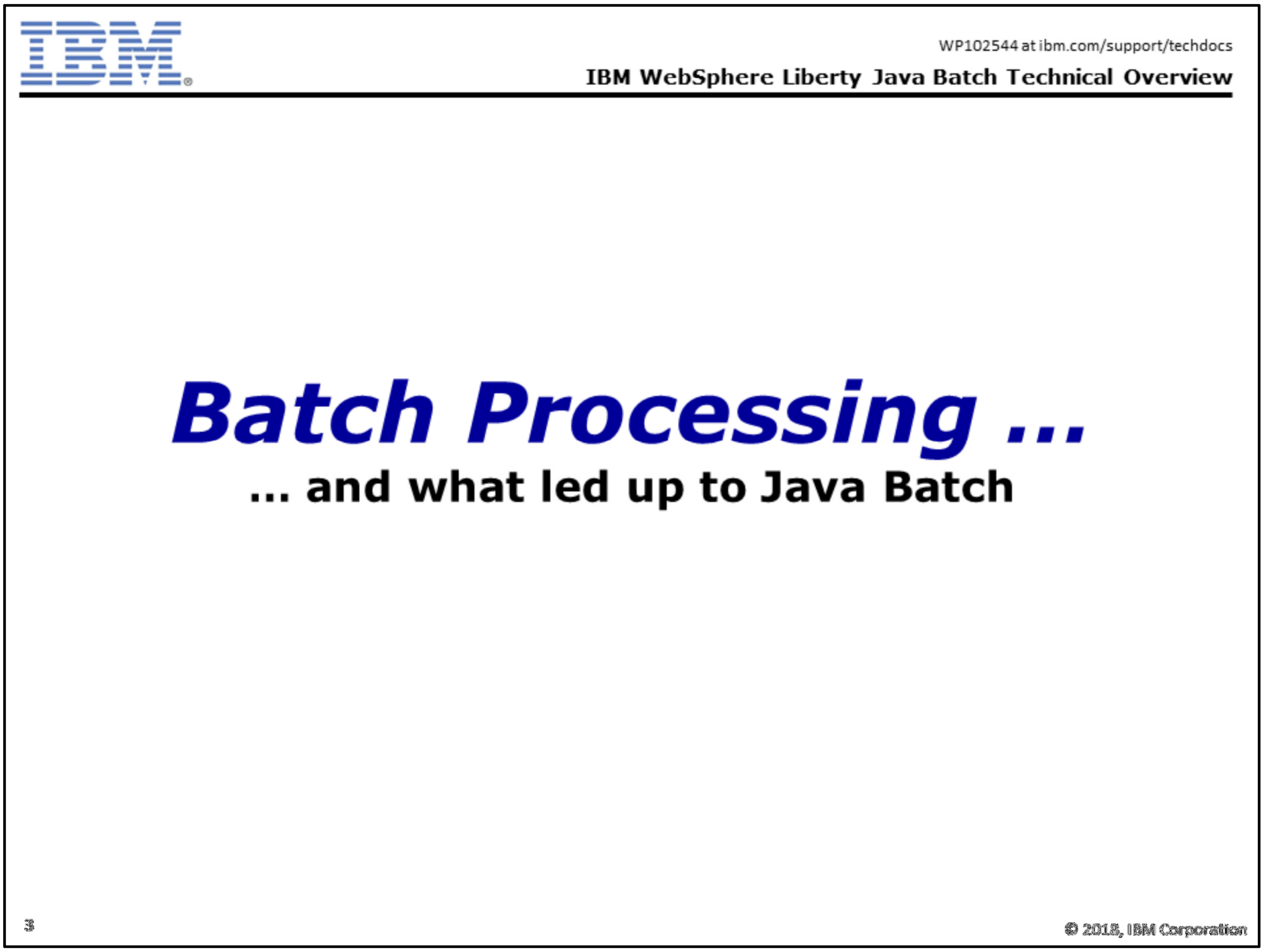
- **Brief Overview of Batch Processing**
  Including background on Java Batch evolution
- **Overview of JSR 352**
  A review of the key elements of the standard
- **IBM Implementation and Extensions**
  A review of how JSR 352 is implemented by IBM, including extensions to the standard that provide additional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

© 2018, IBM Corporation

This presentation will cover three areas of discussion:

1. A brief overview of batch processing … as a way to set context and provide some background on the evolution of batch processing using Java. That evolution led to the development of an open standard for Java batch processing, which was …
2. JSR 352, the open standard specification for Java batch processing. In this section we will offer a review of the essential elements of the standard. This will help you understand what the standard provides and does not provide. What the standard does not provide can be provided by vendors as …
3. JSR 352 extensions, which IBM has developed as a way to offer additional value above and beyond what the standard itself requires.

Like any "overview" presentation, this can't cover every detail. But what it can do is provide a good understanding of the framework of Java batch processing, JSR 352 Java batch processing, and using IBM's implementation of JSR 352 and the extensions IBM provides.



Batch Processing …

… and what led up to Java Batch

© 2018, IBM Corporation

WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

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 …

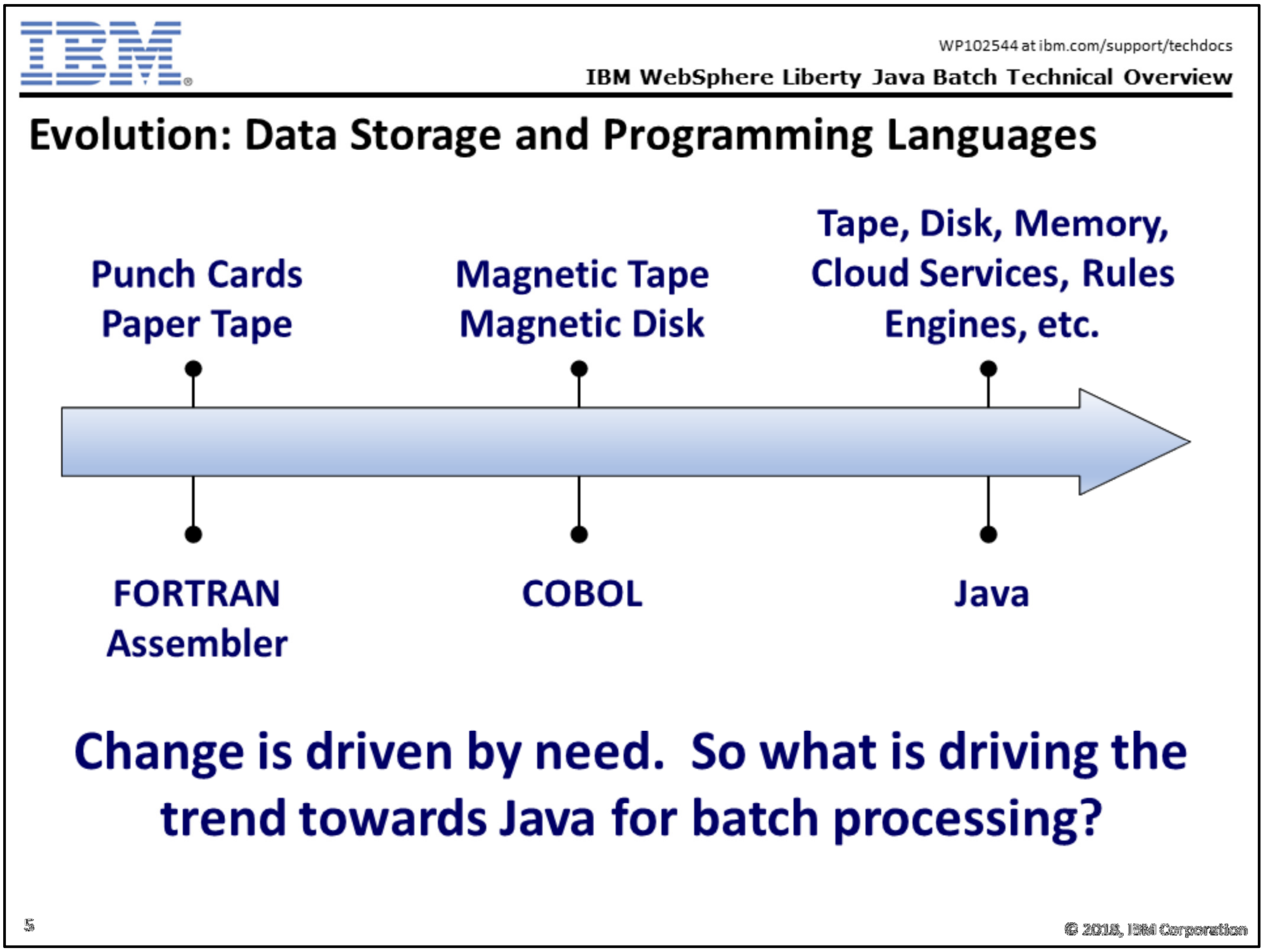
4

© 2018, IBM Corporation

When we set out to design this presentation we wanted to start out with *some* basic level-setting about batch processing, but not dwell on it too much as we expected most would already have a core understanding of it. So we start this by stating the obvious – batch has been around a *long* time. The picture in the chart was from the about mid-1960s, but batch pre-dated that by a fair amount. Punch cards as a means of holding data were used with mechanical computers in the late 1890's and the first half of the 20th century.

Even though the world of real-time and online processing has advanced quite a bit, the need to do data processing in *bulk* (another way of referring to batch) continues to this day. It's not likely to ever go away. Some work, by its very nature, is better done in batches.

What has changed is the approach. That has evolved over time as different technologies emerged.



Evolution: Data Storage and Programming Languages

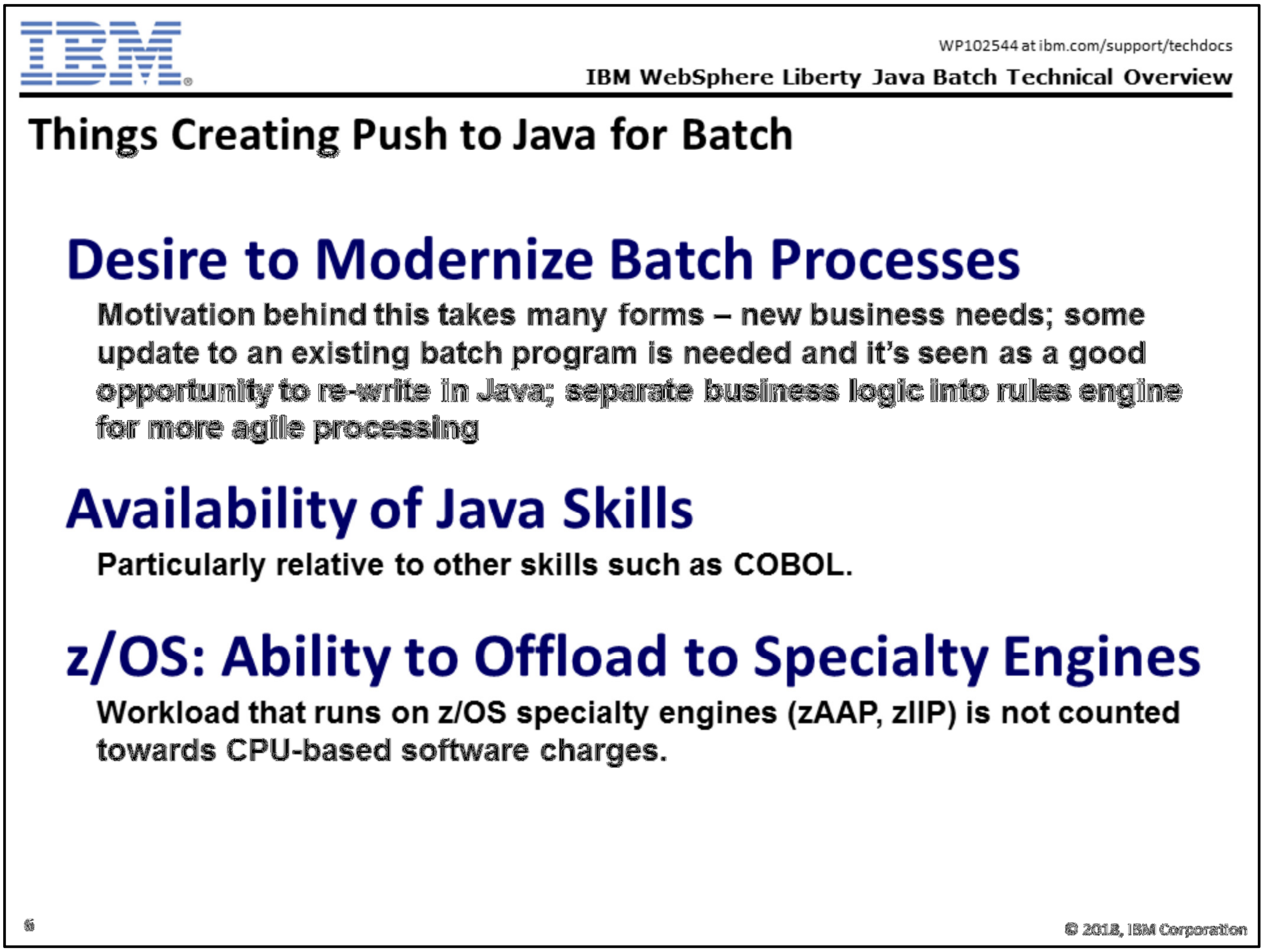
Change is driven by need. So what is driving the trend towards Java for batch processing?

© 2018, IBM Corporation

This is a somewhat simplistic chart illustrating the change in batch processing approaches over time. The two key areas of change involved the storage medium used, and the programming language employed.

In this day and age the source of data as input to batch processing can be wherever the data resides. Requirements around latency and batch window sizes might limit this somewhat, but the point holds: where in past days the storage medium was limited, now it is not.

The programming languages used evolved over time based on what was available (FORTRAN, Assembler), what was easy and wide-spread (COBOL), and what is becoming more and more wide-spread (Java). People do not change just for the sake of change; change is driven by some need, and the direction of change is towards a solution to the need. We explore what's driving the change to Java on the next chart.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

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.

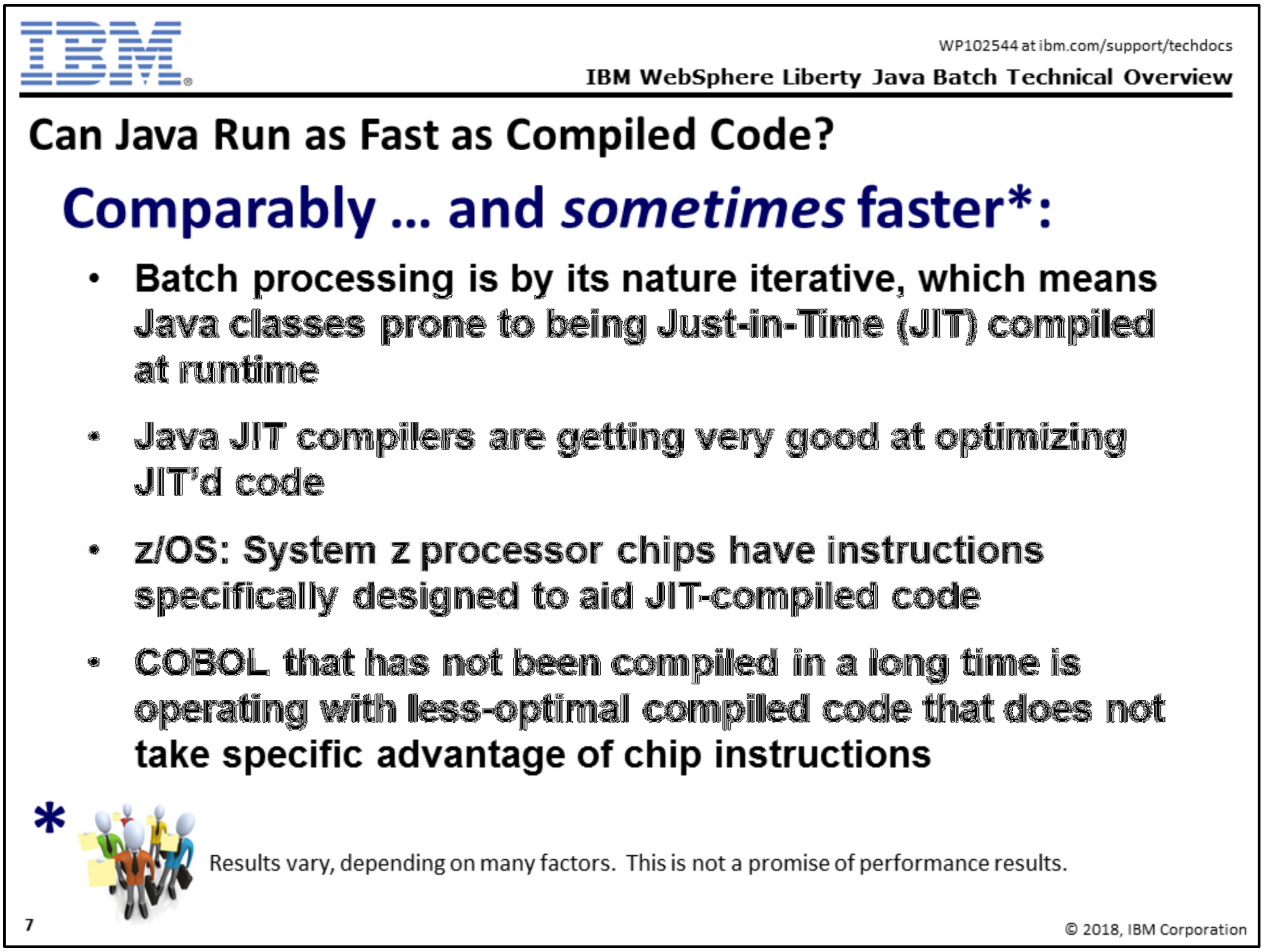
6

© 2018, IBM Corporation

This chart lists three things that served as drivers for change to use Java for batch processing:

- **Modernization** – this is more than a buzzword; the motivation behind "modernizing" batch processing is driven by a need to be more responsive to the needs of the business. It used to take *months* to plan for and implement a change; that is no longer acceptable. As change requirements are identified, many are seeing this as an opportunity to re-engineer to Java as part of the process. Further, there is an increasing desire to separate business rules to be executed by a rules engine apart from the batch processing. All this is with the goal in mind of being more responsive to the needs of the business.
- **Skill availability** – as Java becomes a more and more prevalent programming language, the skills available to program in Java become greater. The inverse is true as well: as COBOL becomes less prevalent, the skills there become less available. There's a lot of COBOL programs in existence, and the need for good COBOL programmers will not disappear overnight; still, when an opportunity to re-engineer a batch process presents itself, many are opting to do that using Java rather than COBOL.
- **Specialty engine offload** – on z/OS, specialty engines (zAAP originally, then zIIP, and now with the z13 only zIIP) provide the ability to offload certain types of work to processors where the accounting for software license charges do not apply. Java is one such workload. By offloading Java workload to zIIP engines, the general processors (GPs) are left to use for traditional work such as CICS, DB2 and COBOL processing.

Do you see other motivators to use Java for batch? If so, note those on this chart. The more reasons given, the greater the case made for the point of this chart – valid reasons exist, and those reasons are behind the general movement to Java for batch processing.



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.

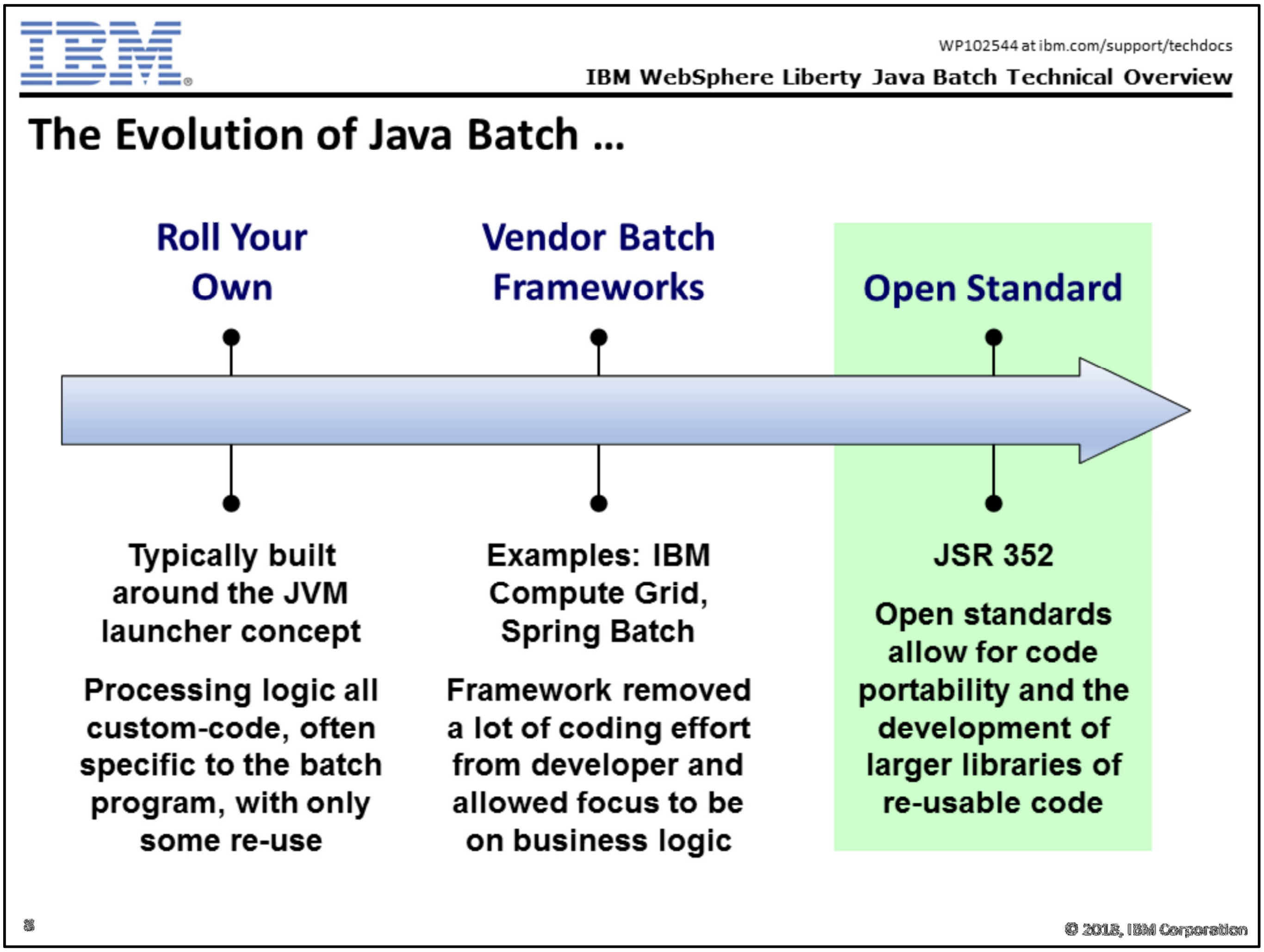
© 2018, IBM Corporation

One question that comes up often is whether Java can perform as well as compiled code for batch processing. The answer is … it can perform comparably, and perhaps better, depending on various factors.

Note: this is where we direct your attention to the asterisk in the lower-left of the chart … performance results vary, and your results may be different. There are just too many factors that contribute to the overall performance of a complex system. Nothing on this chart implies a promise or a guarantee.

How can we suggest an interpreted language such as Java could possibly compare to a compiled language like COBOL? Much of the argument focuses on the Java Just-in-Time (JIT) compilers, which turn interpreted class files into compiled code. The JIT compilers work by watching for code that is being executed over and over again, then compiling those classes. Batch processing is by its nature repetitive, so the JIT compilers fairly quickly recognize the code as candidates for being JIT'd. Further, the JIT compilers are getting better and better and compiling very efficient code. Further still, on platforms such as IBM z Systems the processor has instructions that were put on the chip expressly for the purpose of assisting the JIT'd code to run faster. Because the code is compiled "real time" (meaning: when the JIT engine recognizes the code should be compiled), an up-to-date compiler is used. Contrast that with some COBOL modules that haven't been compiled in years, or maybe decades. How efficient were those compilers? How much did those compilers understand about the modern chips and the modern instructions?

This is why we say Java batch can perform comparably.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

The Evolution of Java Batch …

Roll Your Own

Typically built around the JVM launcher concept

Processing logic all custom-code, often specific to the batch program, with only some re-use

Vendor Batch Frameworks

Examples: IBM Compute Grid, Spring Batch

Framework removed a lot of coding effort from developer and allowed focus to be on business logic

Open Standard

JSR 352

Open standards allow for code portability and the development of larger libraries of re-usable code

8

© 2018, IBM Corporation

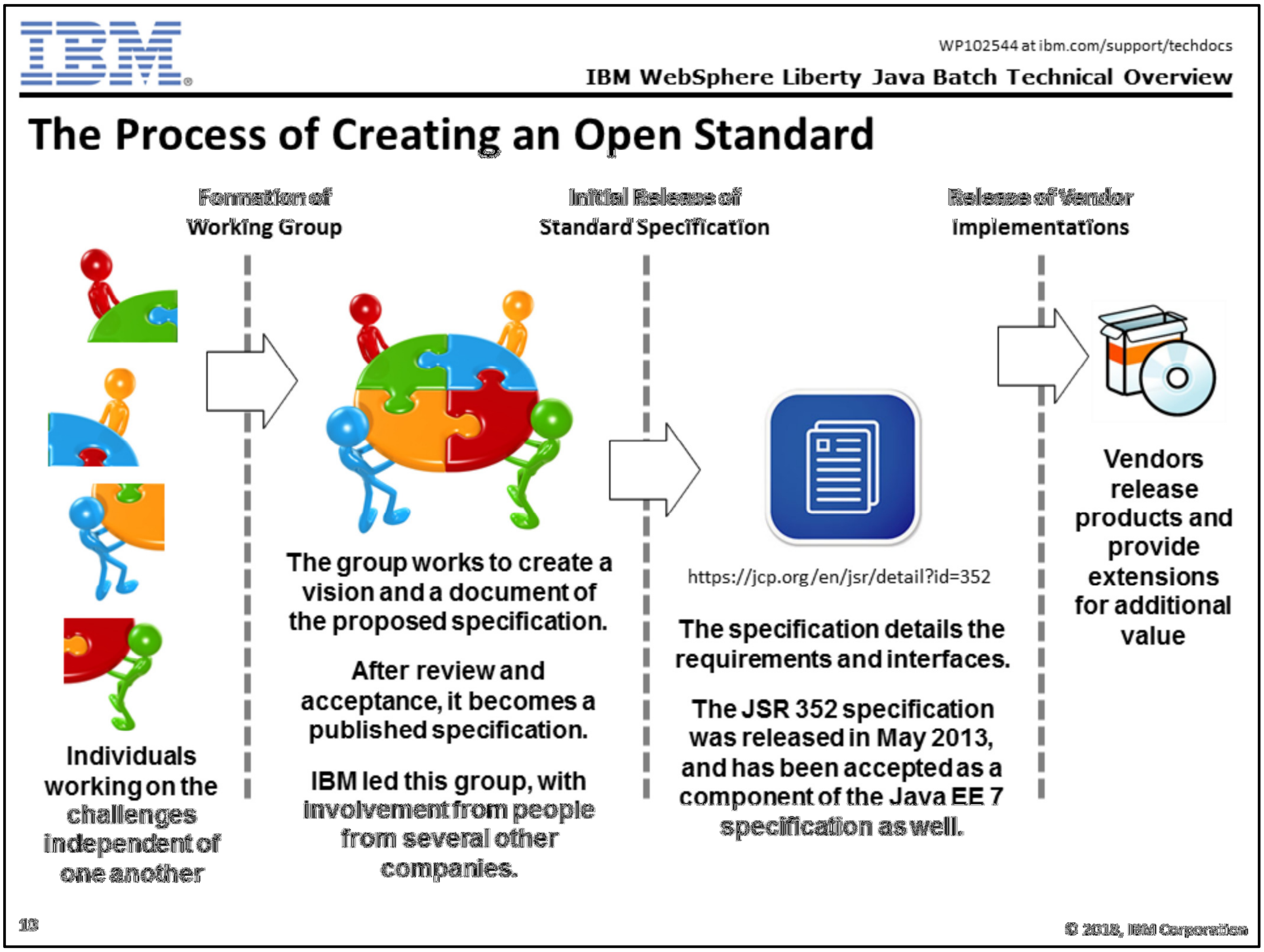
Java has been around for close to 20 years now, so the use of Java for batch processing is hardly new. In the early days the batch programs were mostly “roll your own,” meaning the programmer wrote everything, with only a little use of frameworks, and not much re-use. It was sufficiently effective to satisfy the business needs at the time.

Later various vendor frameworks emerged – programming frameworks and execution runtime environments. These took a great deal of the programming effort off the developer and allowed them to focus on the business logic.

Eventually a movement emerged to create an open standard around Java batch. This was driven by a desire to have a common programming model. Standards encourage wider adoption and the creation of libraries of re-usable code. The open standard that resulted is known as JSR 352. We’ll take a look at that next.

Overview of JSR 352

© 2018, IBM Corporation



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

The Process of Creating an Open Standard

Formation of Working Group

Initial Release of Standard Specification

Release of Vendor Implementations

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.

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.

Vendors release products and provide extensions for additional value

13

© 2018, IBM Corporation

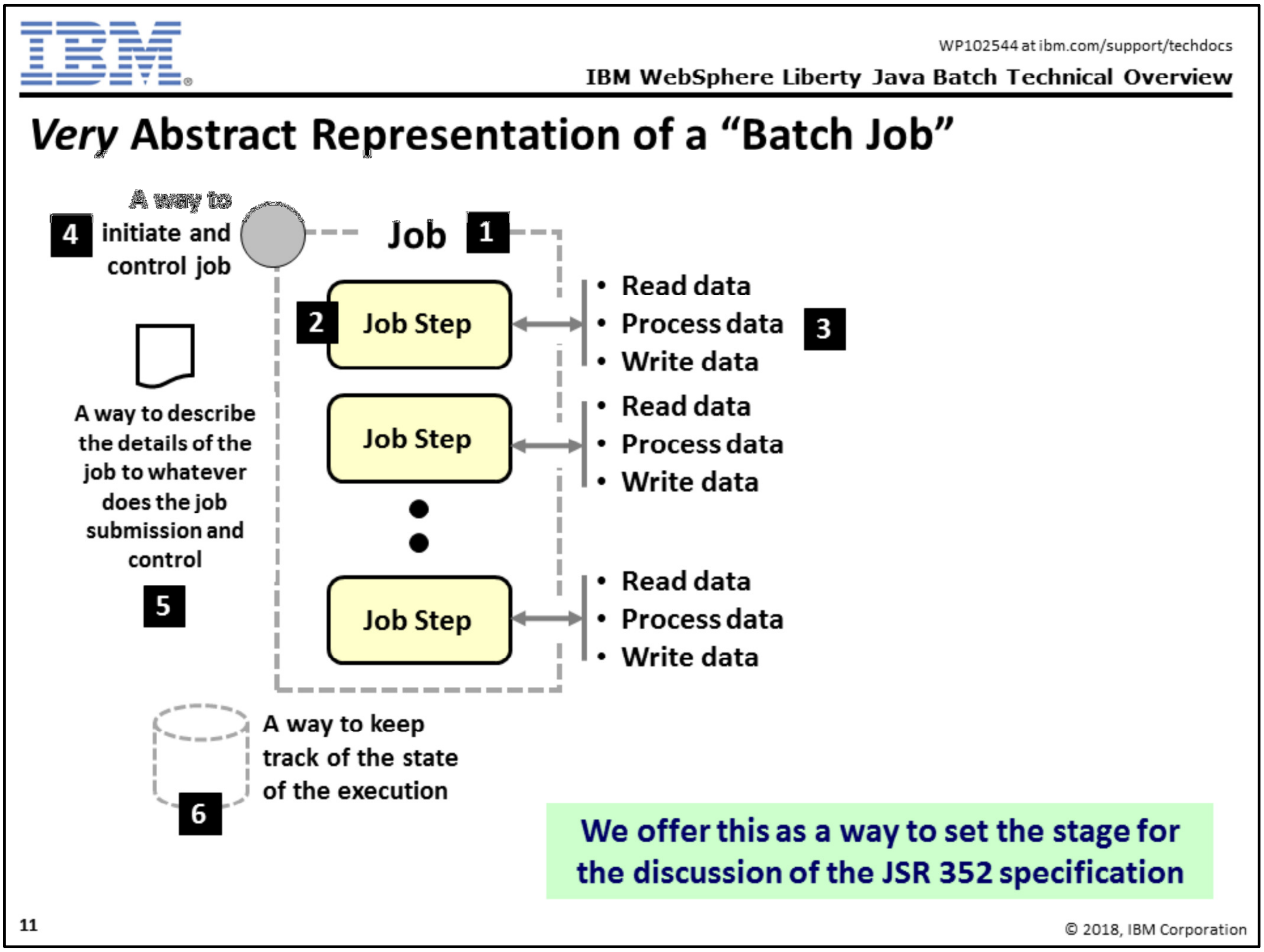
The development of an open standard comes about when a group of people, working individually to solve some challenge, come together in an effort to share ideas and agree on a common approach. When a group comes together they form a working group, and the group works on defining and agreeing to a standard.

That's what happened with JSR 352. The group that formed was led by IBM, and had involvement from people representing several different companies. The result was the creation and adoption of a standard – JSR 352 – in May of 2013. That standard specification can be found at the URL shown on the chart. The JSR 352 standard has been accepted as a component of the broader Java EE 7 standard. That means that any platform claiming Java EE 7 compliance must demonstrate compliance with the JSR 352 standard, along with all the other standards that make up Java EE 7.

At that point vendors market their products to those seeking to use the functionality.

Note: IBM wrote the reference implementation and the test cases to verify compliance with the standard. The reference implementation was what went into Liberty Profile to provide the core compliance with the standard.

In addition to providing what the specification calls for, vendors may offer *extensions* to the standard to provide functional value. That is what we will discuss later in this presentation – the extensions IBM has made to the JSR 352 standard to address functional areas the standard does not address.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

Very Abstract Representation of a "Batch Job"

A way to initiate and control job
4
Job
1
2
Job Step
Job Step
Job Step
• Read data
• Process data
• Write data
3
• Read data
• Process data
• Write data
• Read data
• Process data
• Write data
A way to describe the details of the job to whatever does the job submission and control
5
A way to keep track of the state of the execution
6

We offer this as a way to set the stage for the discussion of the JSR 352 specification

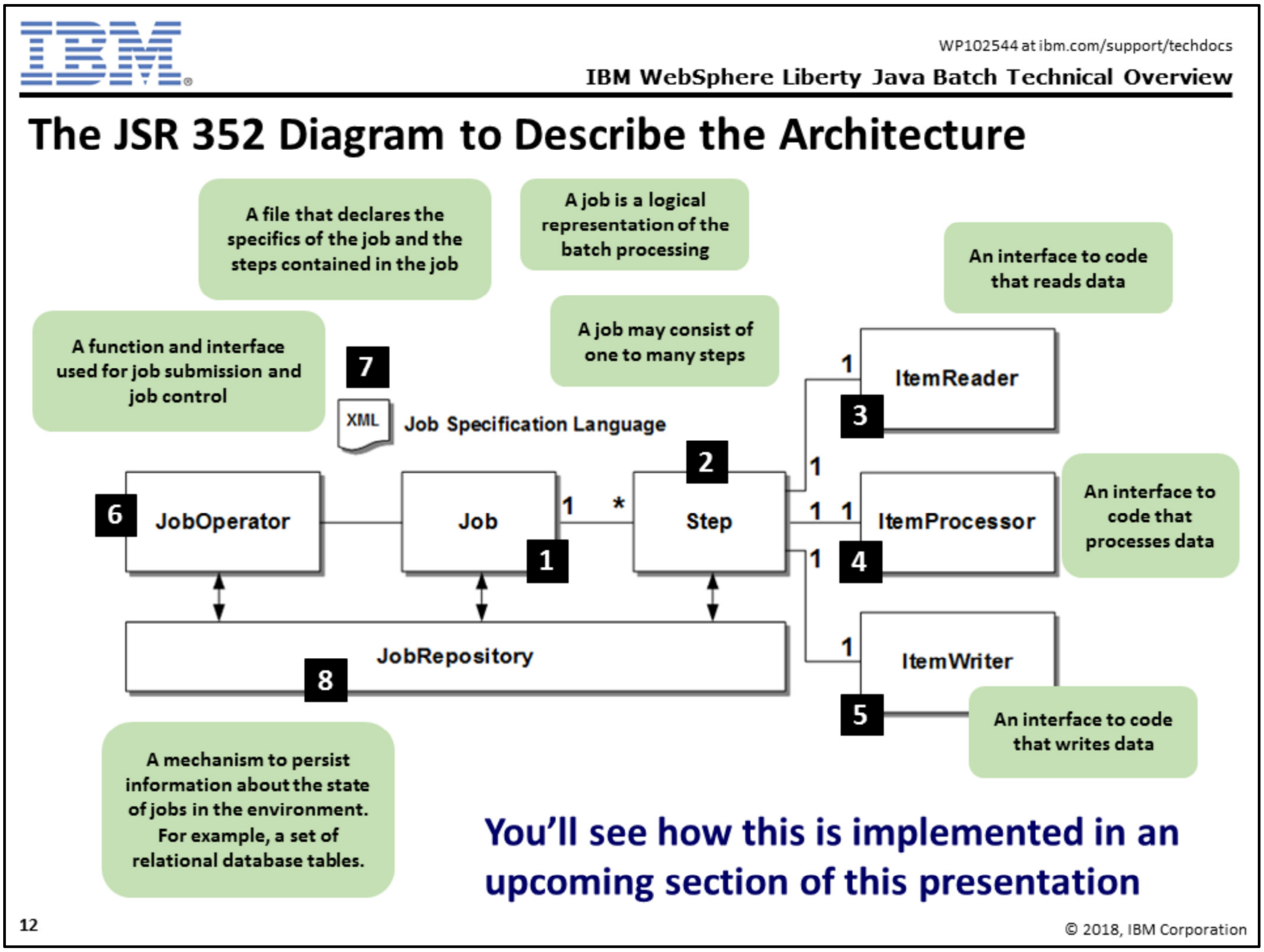
11

© 2018, IBM Corporation

We start with a *very* abstract representation of a batch job. We do this to begin to explain what the JSR 352 standard provides. The notes below correspond with the numbered blocks in the chart:

1. A job is a logical collection of processing that is performed when the job is submitted for execution.
2. A job will contain between 1 and *n* steps. Job steps perform specific batch processing within what the batch job is trying to accomplish.
3. At a very high level, a job step typically *reads* some data, *processes* the data, and then *writes* the data.
4. For a job to be run, a function must be in place to accept a command to initiate the running of the job, and to provide a way to determine when the job has completed.
5. The function described in **4** is going to need to understand the details of the job being submitted, so there must be some means of describing the job.
6. Finally, if we have any hope of having this environment survive outages, some *repository* needs to be present to keep track of jobs, their state, and their completion results.

With that we're ready to introduce the diagram that is part of the JSR 352 specification and begin the process of explaining the IBM JSR 352 implementation.



The JSR 352 Diagram to Describe the Architecture

A file that declares the specifics of the job and the steps contained in the job

A job is a logical representation of the batch processing

An interface to code that reads data

A function and interface used for job submission and job control

A job may consist of one to many steps

An interface to code that processes data

An interface to code that writes data

A mechanism to persist information about the state of jobs in the environment. For example, a set of relational database tables.

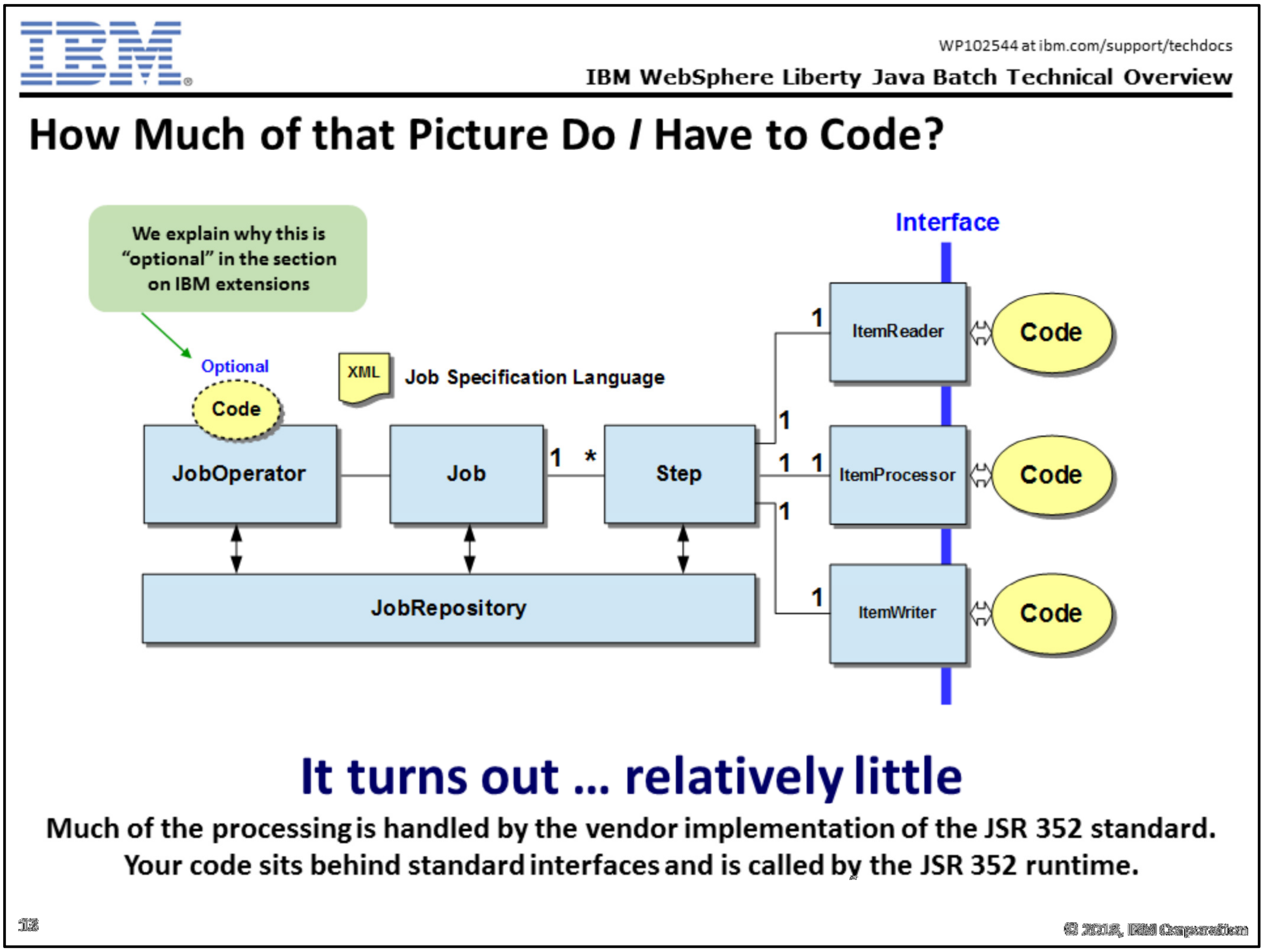
You'll see how this is implemented in an upcoming section of this presentation

© 2018, IBM Corporation

This is the diagram that is in the JSR 352 specification document. The green text boxes were added for this presentation to describe what each element represents. The numbered blocks on the chart correspond to the notes below.

In truth, the diagram offered by the JSR 352 specification document is not revolutionary; in fact, the JSR 352 document rather clearly states that the diagram is generic and applies to batch processing down through the years. It's useful to go over this diagram because many of the elements of the JSR 352 specification map directly to this.

1. A job represents a collection of processes that comprise the batch processing to be done.
2. Jobs are comprised of 1 or more steps. Steps represent specific processing within the overall job.
3. Each step has one instance of an "ItemReader." This is what reads the data for the step. In JSR 352, the ItemReader is an interface behind which *your code that reads the data for step resides*. (Your code is identified in the JSL, which is block **7** above.)
4. Each step has one instance of an "ItemProcessor." This is what processes the data read by the ItemReader. Again, this is an interface; your code resides behind this interface.
5. Each step has one instance of an "ItemWriter." This is what writes the data to wherever your code indicates – a file, a database, whatever.
6. The JobOperator is what provides the ability to submit, monitor and control the execution of jobs.
7. The Job Specification Language (JSL) file is an XML file that describes the job to be run. It provides the details about the job – the steps, the Java to be run for each ItemReader, ItemProcessor and ItemWriter, job properties to control job execution, etc.
8. The JobRepository is a data store the JSR 352 runtime uses to maintain information about job state. For example: job status, the last good checkpoint, etc.



How Much of that Picture Do *I* Have to Code?

It turns out … relatively little

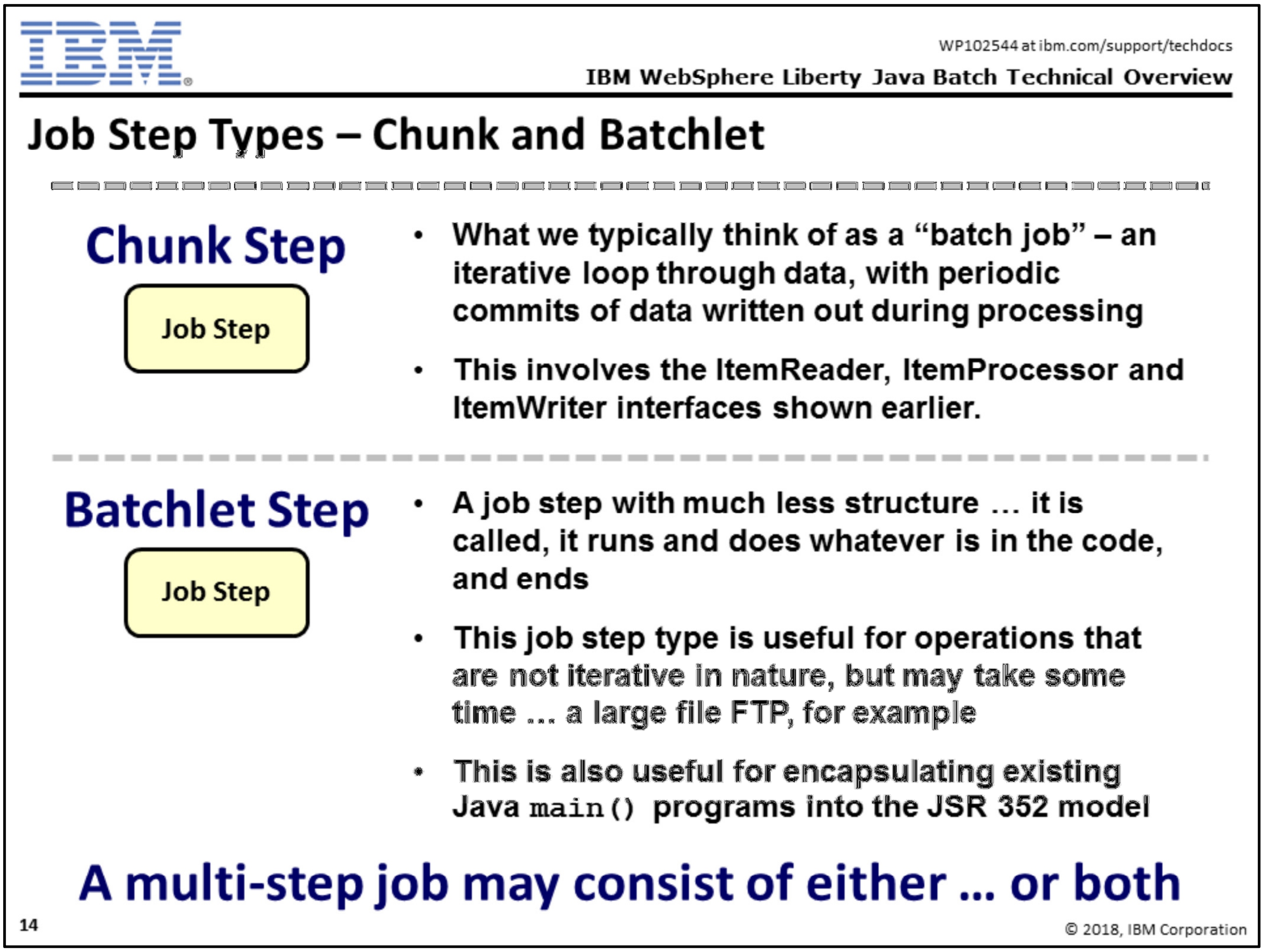
Much of the processing is handled by the vendor implementation of the JSR 352 standard. Your code sits behind standard interfaces and is called by the JSR 352 runtime.

© 2018, IBM Corporation

The JSR 352 diagram presented on the previous chart may leave you with the impression a batch programmer has a lot of work to do when writing a batch program for JSR 352. It turns out, that's not really the case. Much of the diagram from the previous chart represents functional components a JSR 352 runtime implements. The Java developer is responsible for a small slice of that.

The picture above shows the diagram from before, with the boxes in light blue representing what the JSR 352 runtime provides, and the elements in light yellow the things the batch programmer provides. They are:

- The Java code that implements the ItemReader for a job step. The ItemReader is what reads in data from wherever the batch developer determines is needed for that step. It could be a file, it could be a database, it could be a web service … wherever the data resides, the batch programmer implements the read pattern in the ItemReader class file that is specified for a job step. (It is specified in the JSL, which we talk about below.)
- The Java code that implements the ItemProcessor for a job step. This is what does the processing on the data. This is the business logic for the batch processing step.
- The Java code that implements the ItemWriter for a job step. This is what writes the data out to wherever the batch programmer indicates. The JSR 352 runtime controls the frequency of the writes based on the "chunk interval," which you can think of like a "commit interval."
- The Job Specification Language (JSL) file is an XML file that tells the JSR 352 runtime environment the specifics for the job. For example, it is in the JSL that you indicate details about each step, including the Java class files that represent the ItemReader, ItemProcessor and ItemWriter for the job step.
- Finally, the JSR 352 specification calls for a JobOperator, but *something* has to invoke the JobOperator to submit the job. That "something" could be provided by the vendor (in the case of IBM, that's the REST interface we'll talk about in a bit). Or it could be a little bit of customer code the developer writes to invoke the JobOperator interface to submit and control the job. That's why this is listed as "optional" – whether it's needed or not is dependent on who or what provides the function to invoke the JobOperator.



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

14

© 2018, IBM Corporation

JSR 352 defines two types of job step implementations – the "Chunk step" and the "Batchlet step."

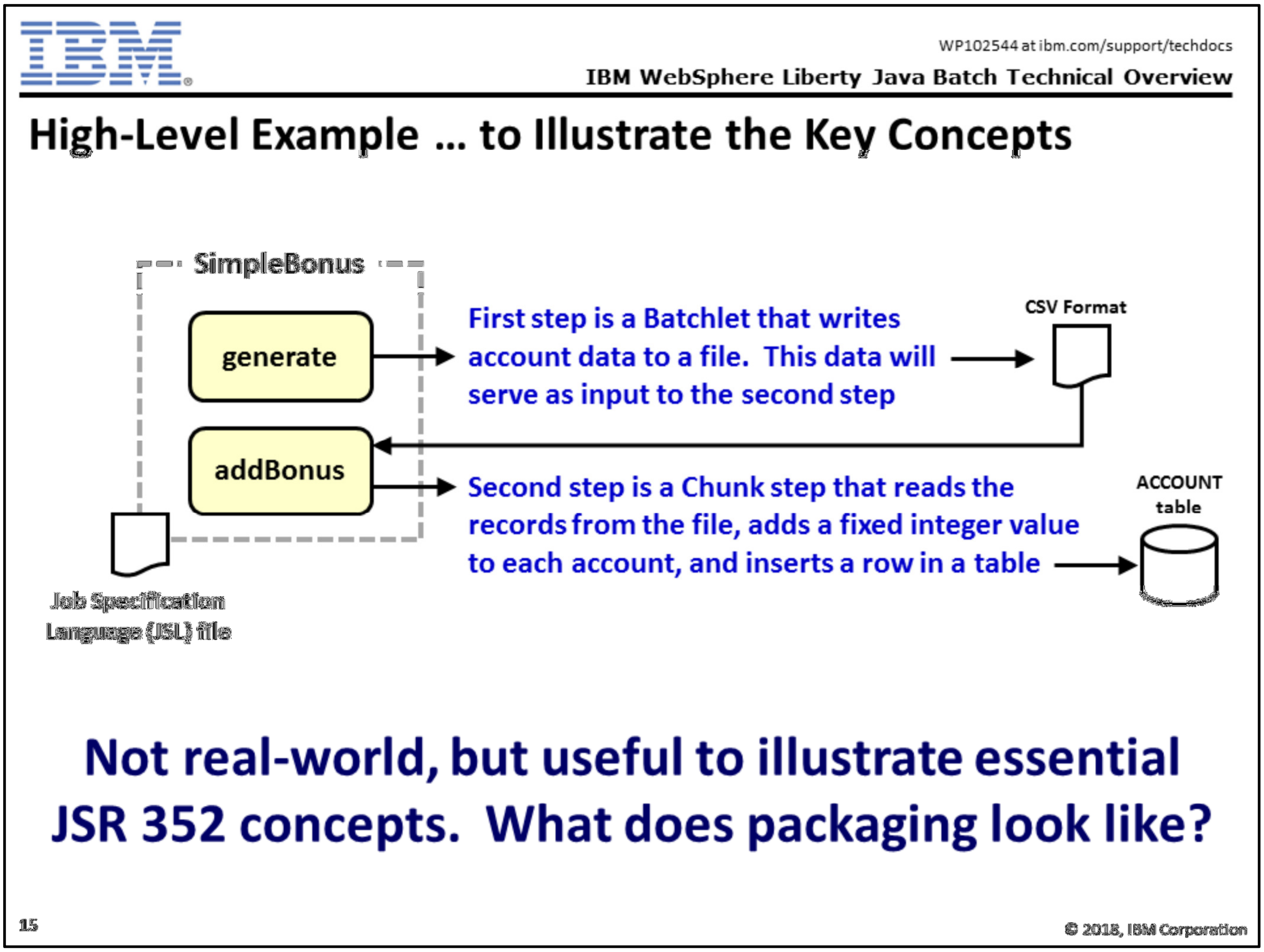
The chunk step is what we just covered when discussing the JSR 352 specification diagram. It is what we typically think of when we think of "batch processing." This step type requires an ItemReader, ItemProcessor and ItemWriter to be implemented.

The "batchlet" job step type is a bit simpler … it is implemented with a single Java class file. The JSR 352 runtime invokes that class file and the batchlet step runs. It does whatever the batchlet class is written to do. When it finishes the runtime sees that and moves on to the next step.

Batchlet job steps are useful for job tasks that are not necessarily loop / iterative in nature. For example, a step that FTPs a large file is one you would want to invoke and have it process until complete. Trying to implement an FTP step using the Chunk-style ItemReader, ItemProcessor, ItemWriter model would be challenging. But with the batchlet step model it is far easier – the class implementation for the batchlet step is called, it does the FTP and it returns.

The batchlet model is also handy for re-hosting existing Java `main()` programs. Those may be batch programs written many years ago, before some of the vendor frameworks were available. Rather than re-write those programs from scratch, you can modify them to fit within the JSR 352 batchlet job step model and run them as part of JSR 352 processing.

A job can consist of steps written as chunk or batchlet. A multi-step job may consist of both. Any given job step must be either chunk or batchlet, however. A given job step can't be a mix of both.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

High-Level Example … to Illustrate the Key Concepts

SimpleBonus

generate

First step is a Batchlet that writes account data to a file. This data will serve as input to the second step

CSV Format

addBonus

Second step is a Chunk step that reads the records from the file, adds a fixed integer value to each account, and inserts a row in a table

ACCOUNT table

Job Specification Language (JSL) file

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

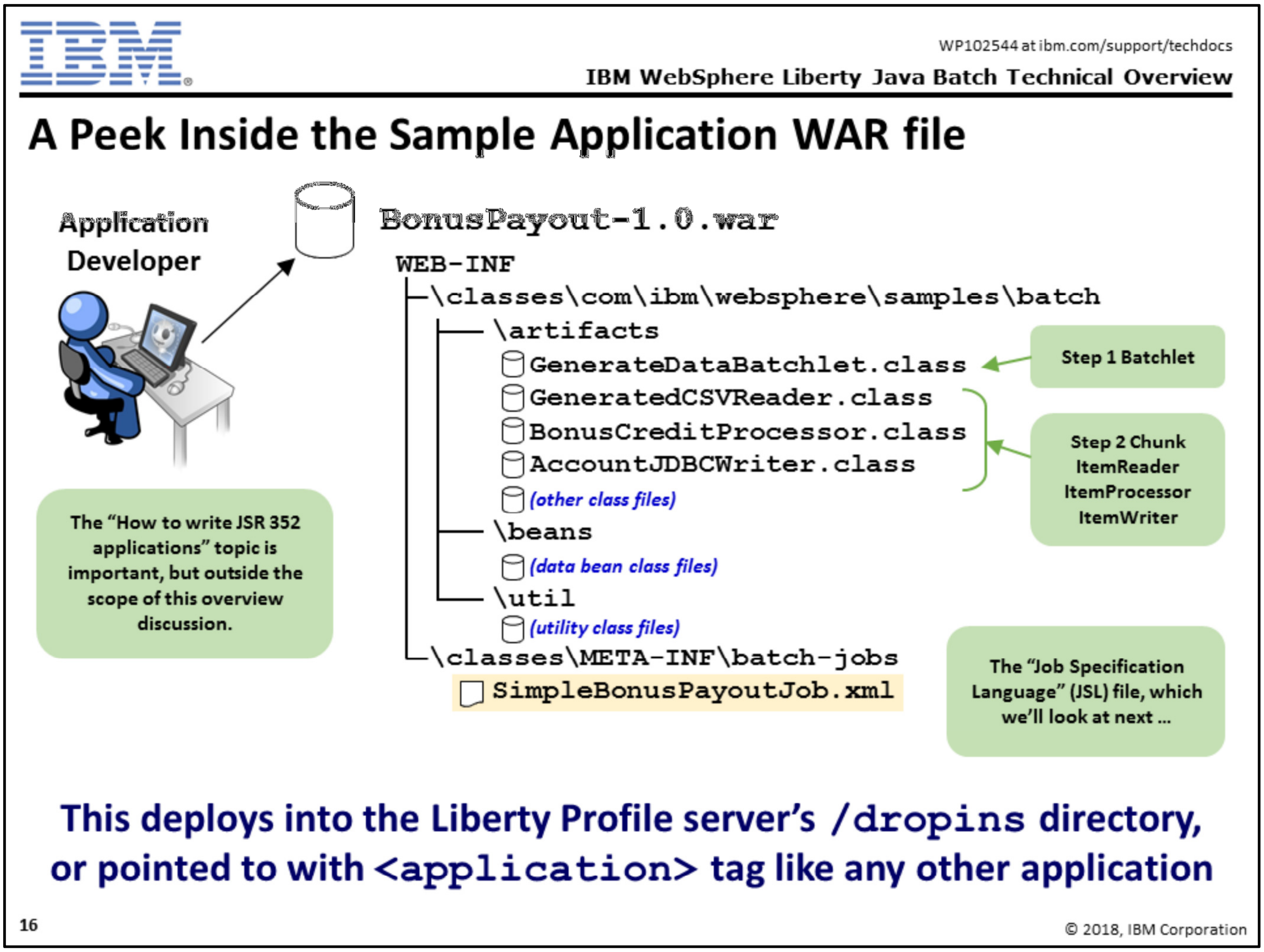
15

© 2018, IBM Corporation

To illustrate some of this we'll use one of the samples provided with IBM's JSR 352 implementation. This sample application is based on a hypothetical account balance model. The purpose of the sample is to illustrate reading account balances in, adding a fixed "bonus" value to each account balance, and then writing the updated account data to a database file.

This sample is implemented as two job steps. The first job step – "generate" is a batchlet step that creates a file with account data written in CSV (comma-separated value) format. That file is then used as input to the second job step – "addBonus." The "addBonus" step reads from the file created in the first step, adds the fixed bonus amount to each account, and writes it out to a database table.

It's a very simple approximation of a banking or other account-oriented batch job. It's simple enough to follow and it shows the essential elements of JSR 352 batch processing. Let's now take a look at what the packaging of this application looks like, then we'll take a look at the Job Specification Language (JSL) file for it.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

A Peek Inside the Sample Application WAR file

```
BonusPayout-1.0.war
 WEB-INF
  ├\classes\com\ibm\websphere\samples\batch
  │  ├── \artifacts
  │  │     GenerateDataBatchlet.class
  │  │     GeneratedCSVReader.class
  │  │     BonusCreditProcessor.class
  │  │     AccountJDBCWriter.class
  │  │     (other class files)
  │  ├── \beans
  │  │     (data bean class files)
  │  └── \util
  │        (utility class files)
  └\classes\META-INF\batch-jobs
       SimpleBonusPayoutJob.xml
```

Step 1 Batchlet

Step 2 Chunk
ItemReader
ItemProcessor
ItemWriter

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

16

© 2018, IBM Corporation

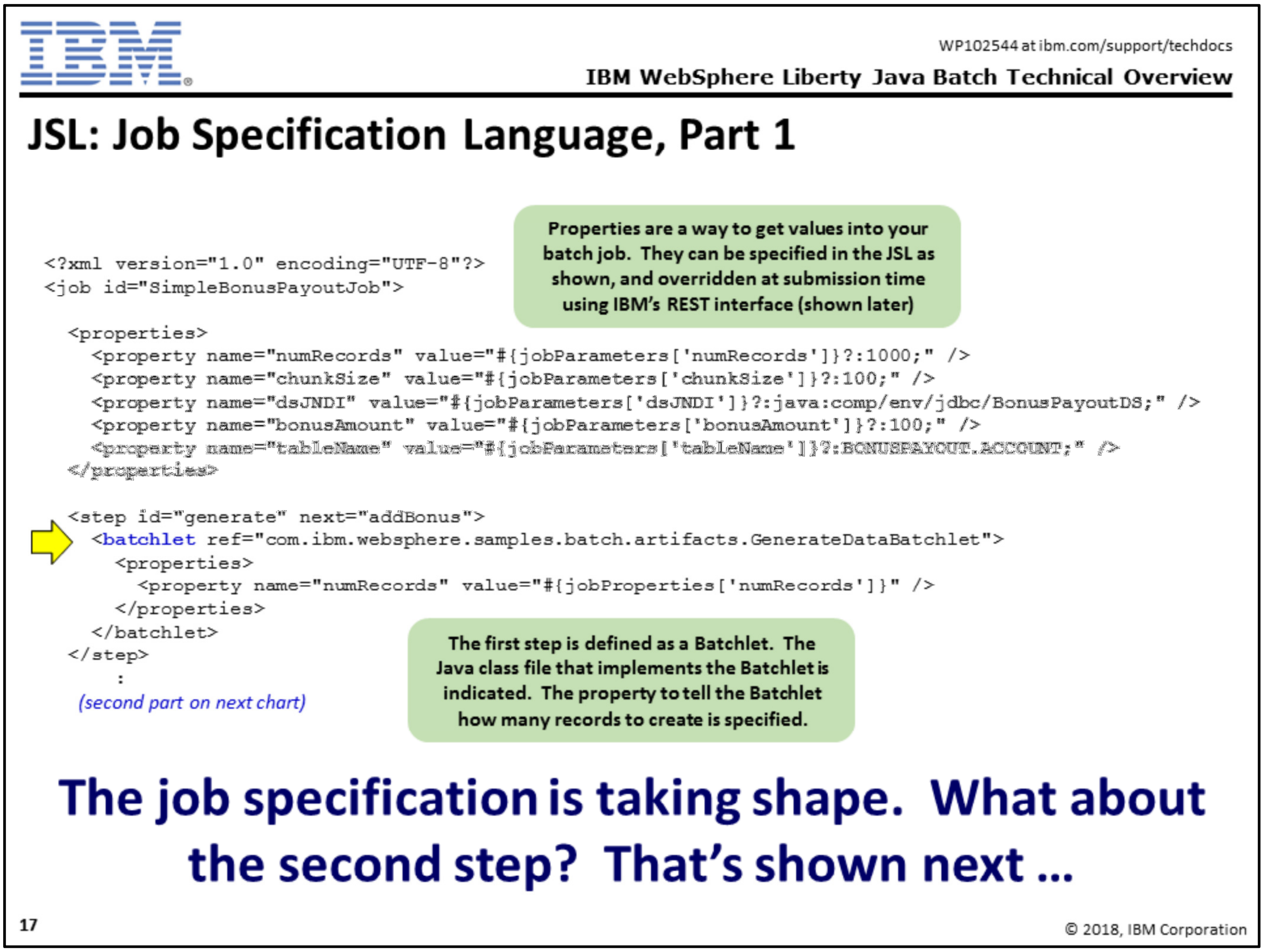
The sample batch application is packaged in a WAR file format. Inside the WAR file are the Java class files that implement the two steps of the job, some data beans and utility class files, and a the Job Specification Language file.

Note: the topic of *writing* JSR 352 application is an important topic, but it falls outside the scope of what this document is designed to do. Most Java programmers will find writing JSR 352 applications fairly easy.

In the WAR file there is a Java package in which the Java class files reside. Under the `\artifacts` directory the class files that implement the two steps can be found – one class for the batchlet step, and three classes for the chunk step. Why three? Because a chunk step must implement an ItemReader, ItemProcessor and ItemWriter. That's what you see in the chart – three classes, each indicating the role it servers: "reader," "processor," and "writer."

The JSL file is found under the `META-INF` directory. We'll take a look at that file in the upcoming charts.

This WAR file is deployed into a Liberty Profile server environment like any application is deployed – it may be placed in the `/dropins` directory and be detected and loaded dynamically, or it may be placed wherever you'd like and pointed to with an `<application>` element in the server.xml. The key point here is that when packaged the JSR 352 batch application is really no different than any other application from a deployment point of view.



JSL: Job Specification Language, Part 1

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

  <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)

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)

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 …

© 2018, IBM Corporation

Let's now take a look at the Job Specification Language (JSL) file for this sample application. We'll do this in two parts. The first part shown here covers the job properties and the details of the first step.

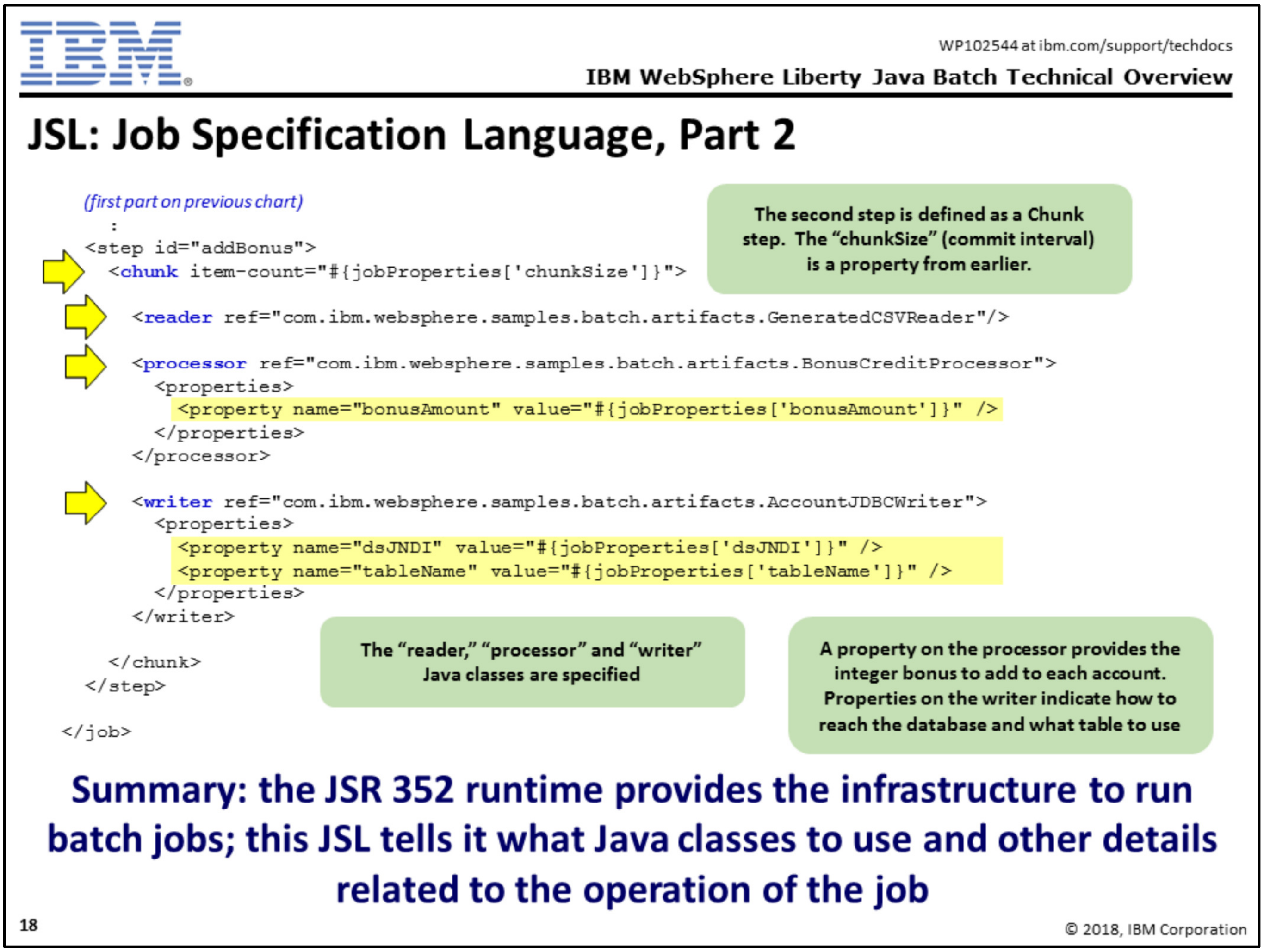
Job properties provide a way to define values for things and have those values apply to the steps in the job. For this sample job there are certain values that can be specified – such as the number of account records the first step will generate and write to the file. Or the "chunk" (commit interval) size to use when writing the records to the database table in the second step. Also things like where the database table is and how to reach it.

Job properties can be overridden at the time of job submission. Or the default values in the JSL file can apply. This is an operational choice left to the people operating the runtime environment. The sample application provides default, and in the absence of overrides provided at job submission time, the defaults will apply.

The first step is specified. It is identified as a `<batchlet>` step, and the `ref=` names the Java class file that implements the batchlet. This job step has one property – and that's the number of records to write out. It inherits the value for `numRecords` from the properties section for the job. If the `numRecords` value is overridden at job submission time, the value supplied at submission would be used.

We can see this taking shape. There's really not much special about this … it's just an XML file that spells out the details of the job so the JSR 352 runtime can make sense of the things and know what to do.

Note: those familiar with z/OS Job Control Language (JCL) files should spot the *conceptual* similarities. Different syntax of course, but conceptually they are very similar.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

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

18

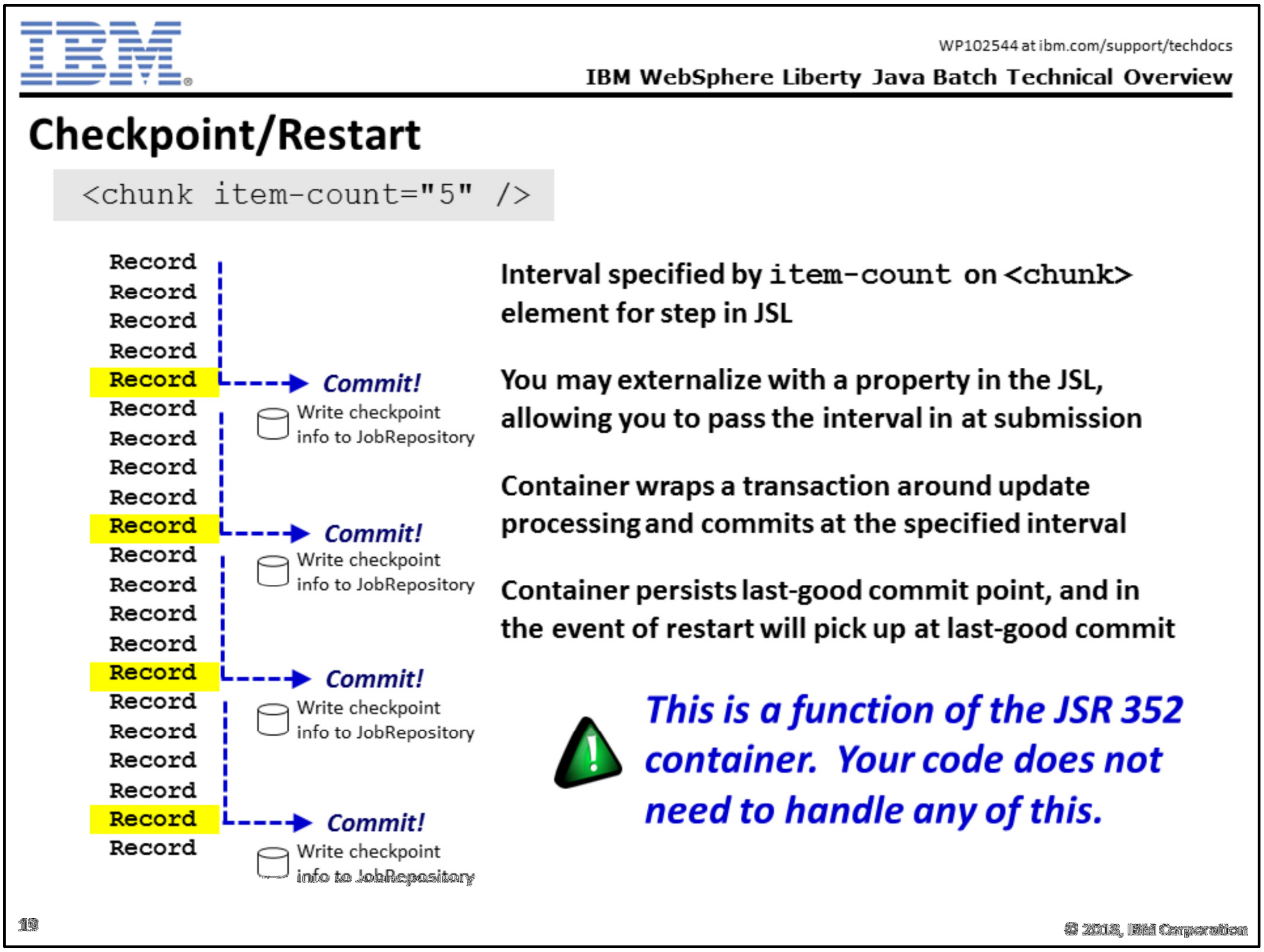
© 2018, IBM Corporation

This is the second part of the JSL file. This shows the second step, which is the chunk step type. It is identified as a chunk step type with the `<chunk>` element. Notice also that the `<chunk>` element specifies the property `chunkSize`, which you can think of as the commit interval. It's the interval at which the ItemWriter is called to write out the data that has been processed for that interval.

The first thing we see is that the step has three distinct elements – a `<reader>`, a `<processor>`, and a `<writer>` element. That corresponds directly to the diagram from earlier where we showed the ItemReader, ItemProcessor and ItemWriter requirement for a chunk job step for JSR 352. Each element specifies the Java class file that implements the reader, processor or writer. We saw in the packaging chart earlier how those three class files were part of the WAR file.

One of the properties we saw earlier was `bonusAmount`. That is an integer value that is used by the ItemProcessor and added to the account balance for each account. That is really the "business logic" for this sample application – read in account data, *add a bonus*, and write the results to a database.

The pieces of the puzzle come together – the JSR 352 runtime provides what this application requires to run. The JSL file tells the runtime about the job – the class files to load, the properties to use. The Liberty Profile server provides the rest – the JVM in which to operate, the function to access the database, etc.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

Checkpoint/Restart

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

Record
Record
Record
Record
Record → Commit! Write checkpoint info to JobRepository
Record
Record
Record
Record
Record → Commit! Write checkpoint info to JobRepository
Record
Record
Record
Record
Record → Commit! Write checkpoint info to JobRepository
Record
Record
Record
Record
Record → Commit! Write checkpoint info to JobRepository
Record

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.

19

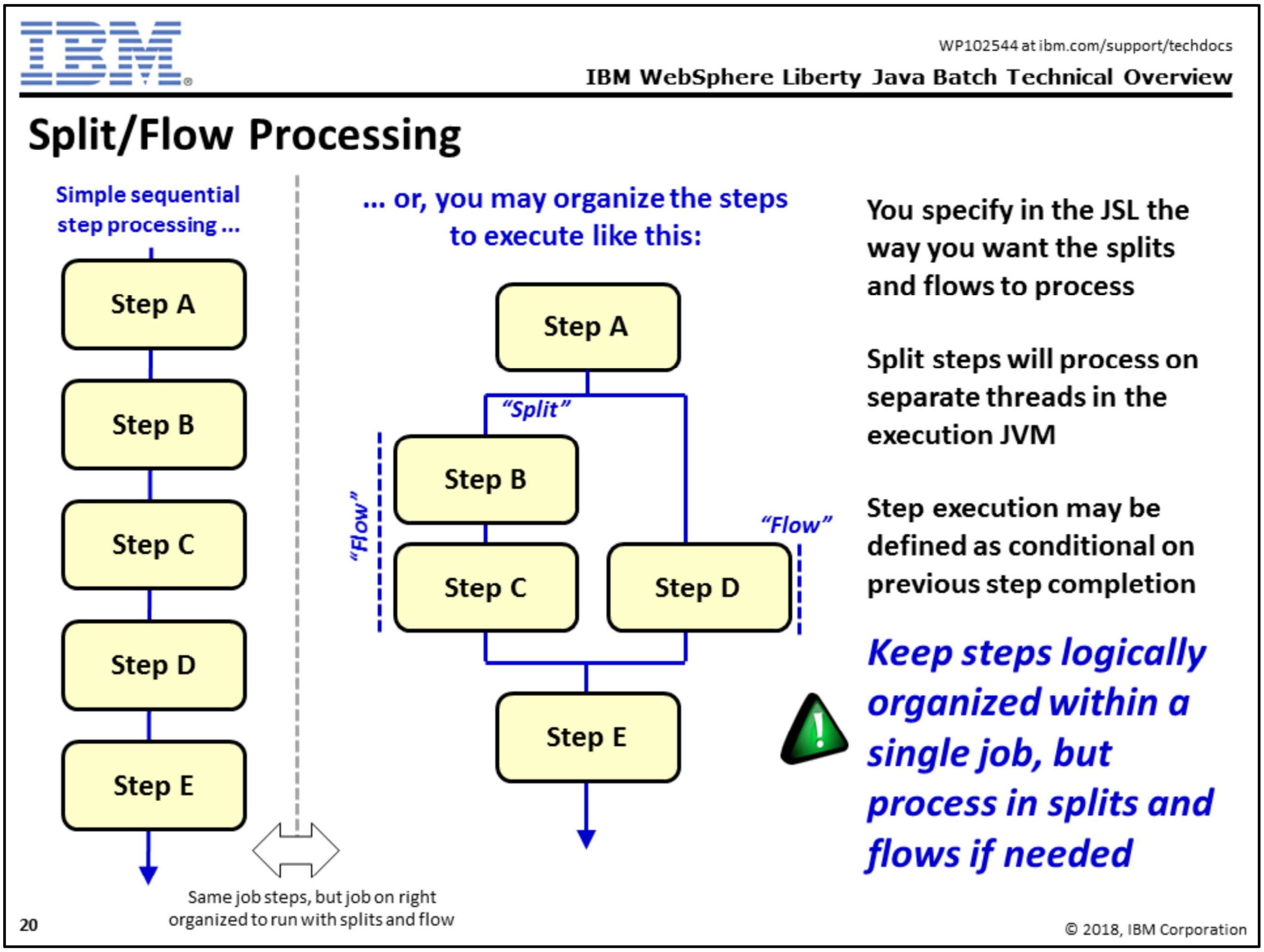
© 2018, IBM Corporation

The JSR 352 concept of "chunk" processing implies processing some number of records, then committing the updates made up to this point. This is standard checkpoint processing. The difference from historic batch processing is this checkpoint processing is handled by the JSR 352 container, and not your batch code.

The interval is specified with `item-count` on the `<chunk>` element in the job specification language (JSL) file. The chart is showing a checkpoint interval of 5, which means every five records the container will do the commit processing. This includes persisting the checkpoint information about the "last known good" commit interval. This is used when the step is restarted and the container needs to pick up where it left off last.

If some exception occurs in the middle of a chunk interval, then the container performs a rollback.

The item-count value can be hard-coded as shown, or it can be defined as a job property and passed in at the time of job submission. In either case, the container receives the item-count value and processes the job step with that interval in mind.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

Split/Flow Processing

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

20

© 2018, IBM Corporation

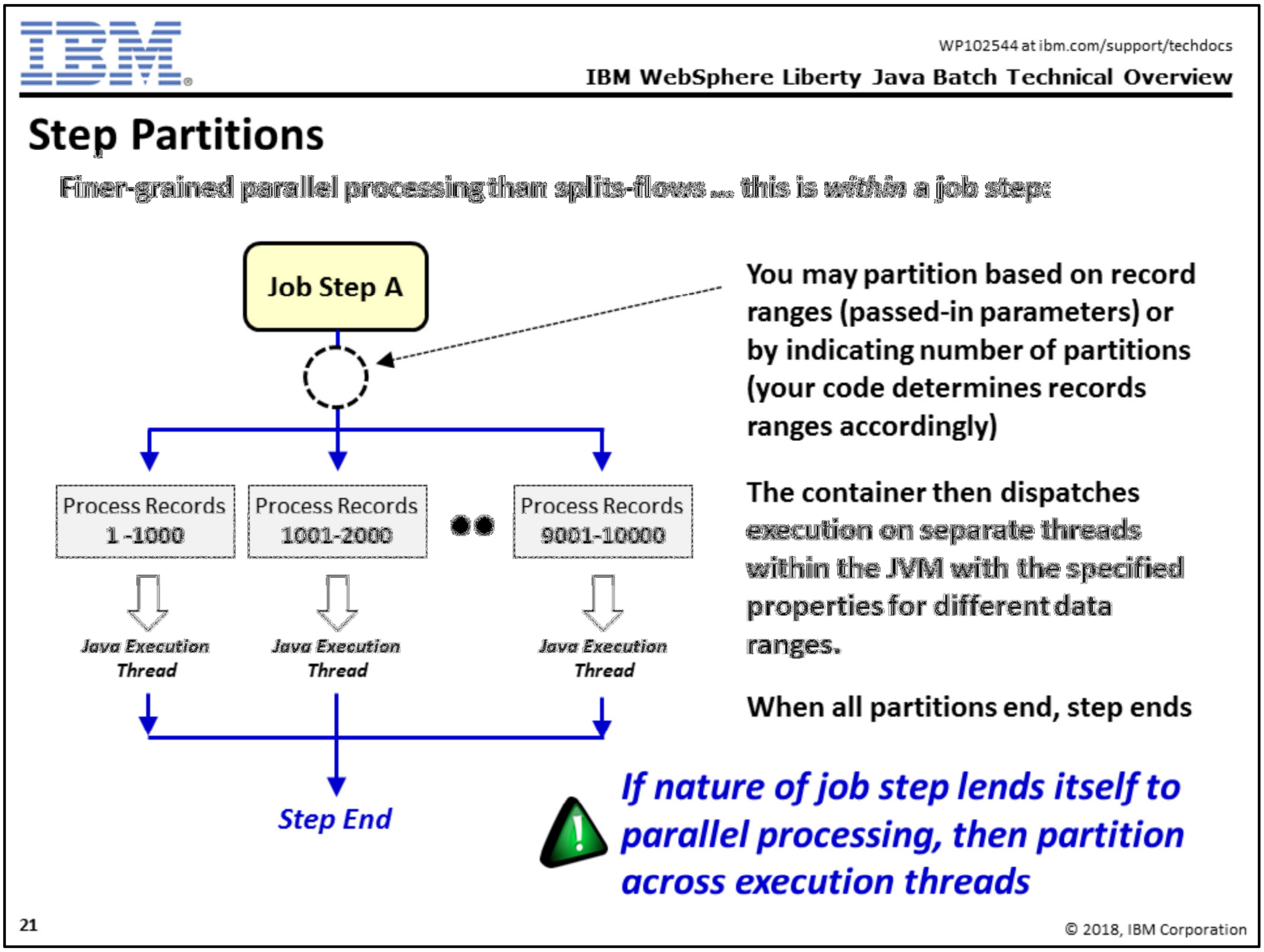
Another capability of the JSR 352 specification is the ability to organize job steps into "splits" and "flows."

Imagine a simple sequential processing of five job steps, as illustrated by the picture on the left in the chart. That's how batch processing is frequently done. Steps are run in the order indicated.

Now imagine the ability to designate the step processing contain a "split" where steps are then run concurrently, and the ability to designate several steps into a logical unit of execution called a "flow." The diagram on the right in the chart above illustrates this. The definition for splits and flows is contained within the JSL for the submitted job. We're not showing the details for that here.

When a job has a split (as shown in the simple example above), the container will dispatch the processing onto separate JVM threads. That allows concurrent processing of steps.

Note: as the architect of your batch processing, it is up to you to understand what steps require sequential processing and what steps may be run concurrently. The point here is that the JSR 352 specification permits the definition of splits and flows, which gives you the flexibility to organize your job processing according to how you see it best being performed.



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

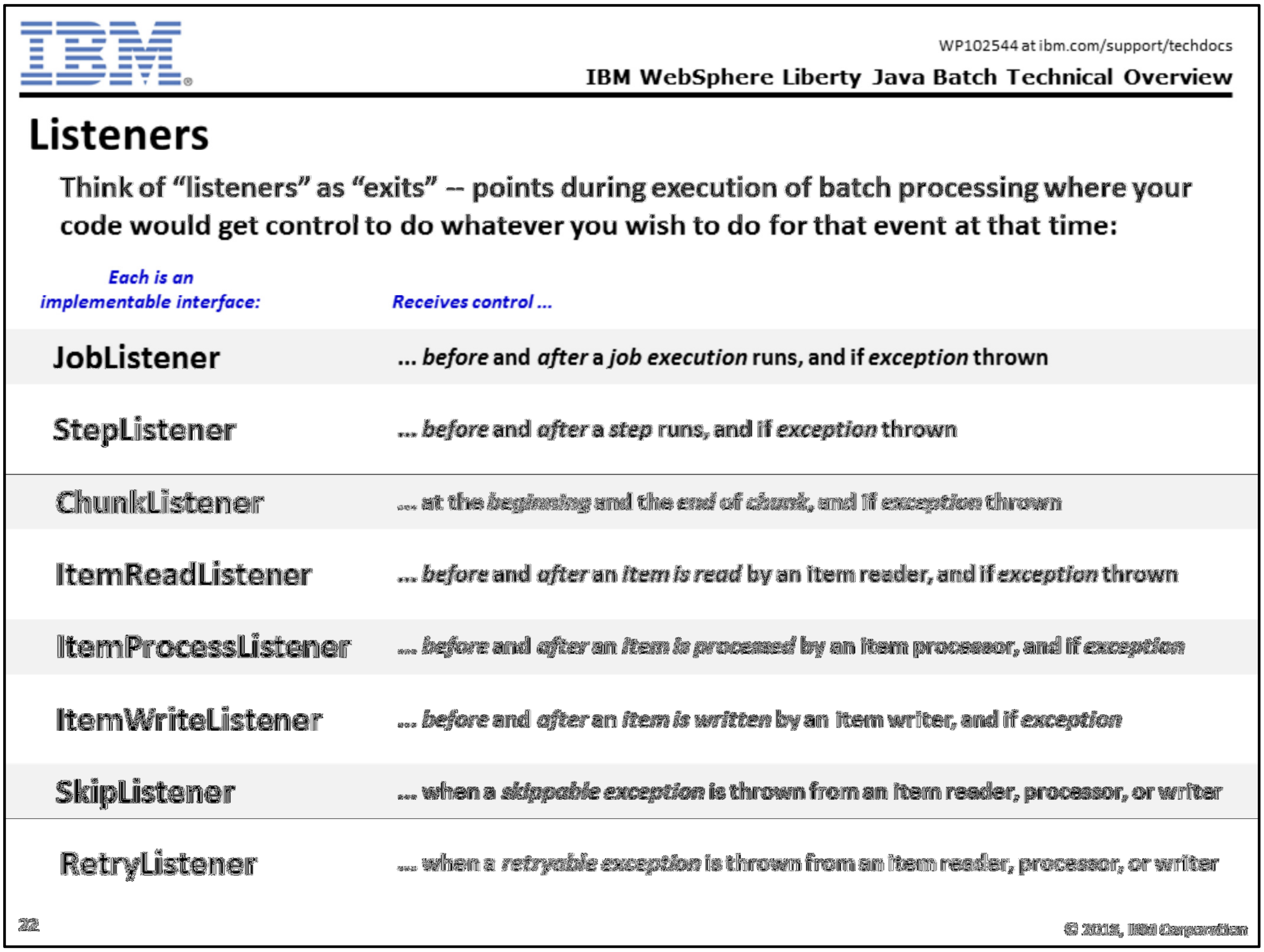
If nature of job step lends itself to parallel processing, then partition across execution threads

© 2018, IBM Corporation

Now imagine you have a job step in which you, as architect of the batch processing, know can be further partitioned to run in parallel. In JSR 352 language this is known as *partitioning*. It is concurrent processing *within* a job step.

To do this, something has to indicate the data ranges each concurrent partition is going to act upon. Your batch code is written to take as input either specified data ranges, or to calculate the data ranges based on a specified number of partitions. The JSR 352 container then dispatches execution across separate threads within the JVM, and each partition then runs concurrently.

For step processing where the organization of the data permits this, partitioning can result in reduced execution time. That is the value of parallel processing -- concurrent execution permits a reduced elapsed time, given sufficient resources to execute in parallel.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

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:

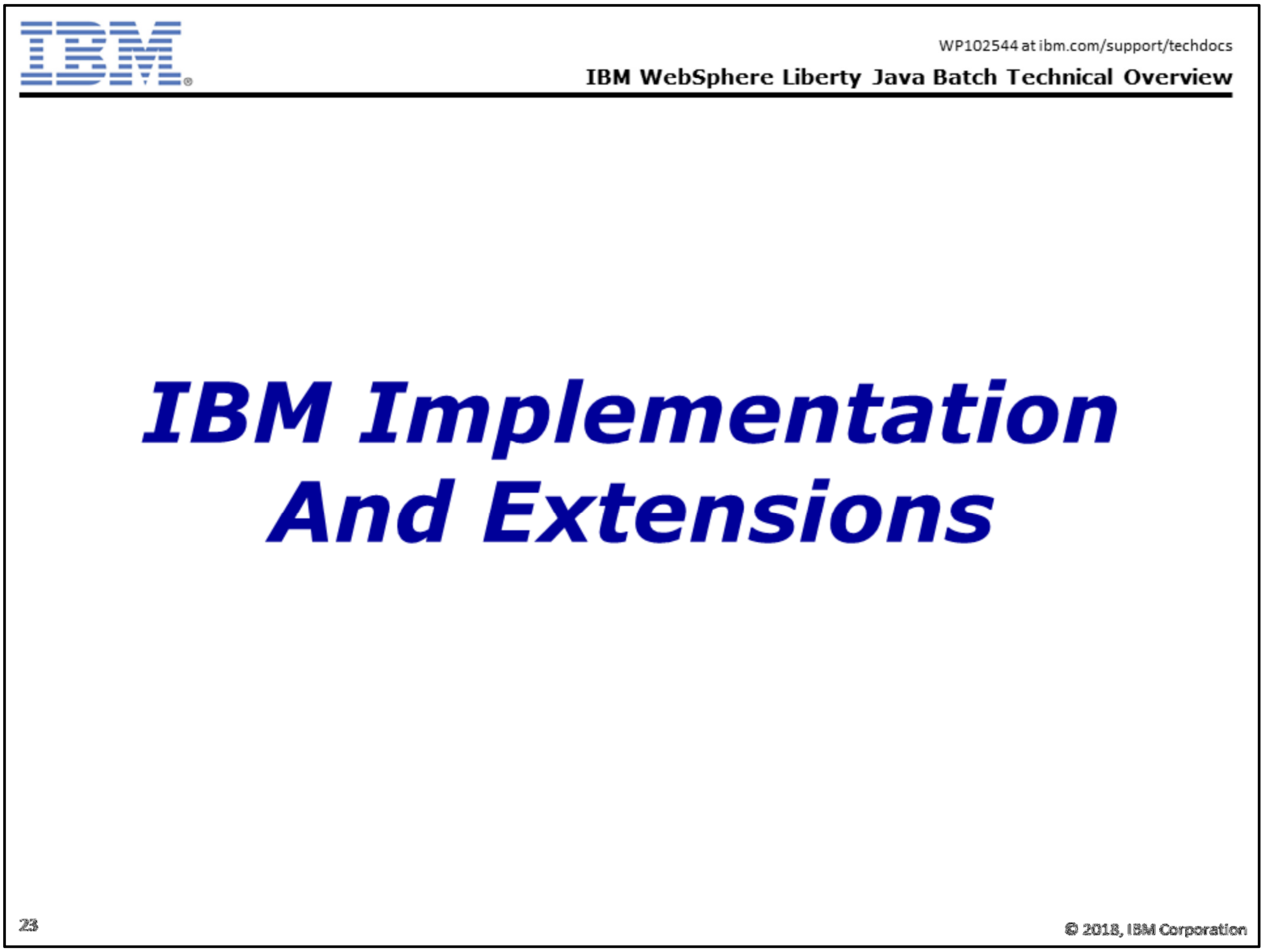
Each is an implementable interface:	*Receives control ...*
JobListener	... *before* and *after* a *job execution* runs, and if *exception* thrown
StepListener	... *before* and *after* a *step* runs, and if *exception* thrown
ChunkListener	... at the *beginning* and the *end of chunk*, and if *exception* 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

22

© 2015, IBM Corporation

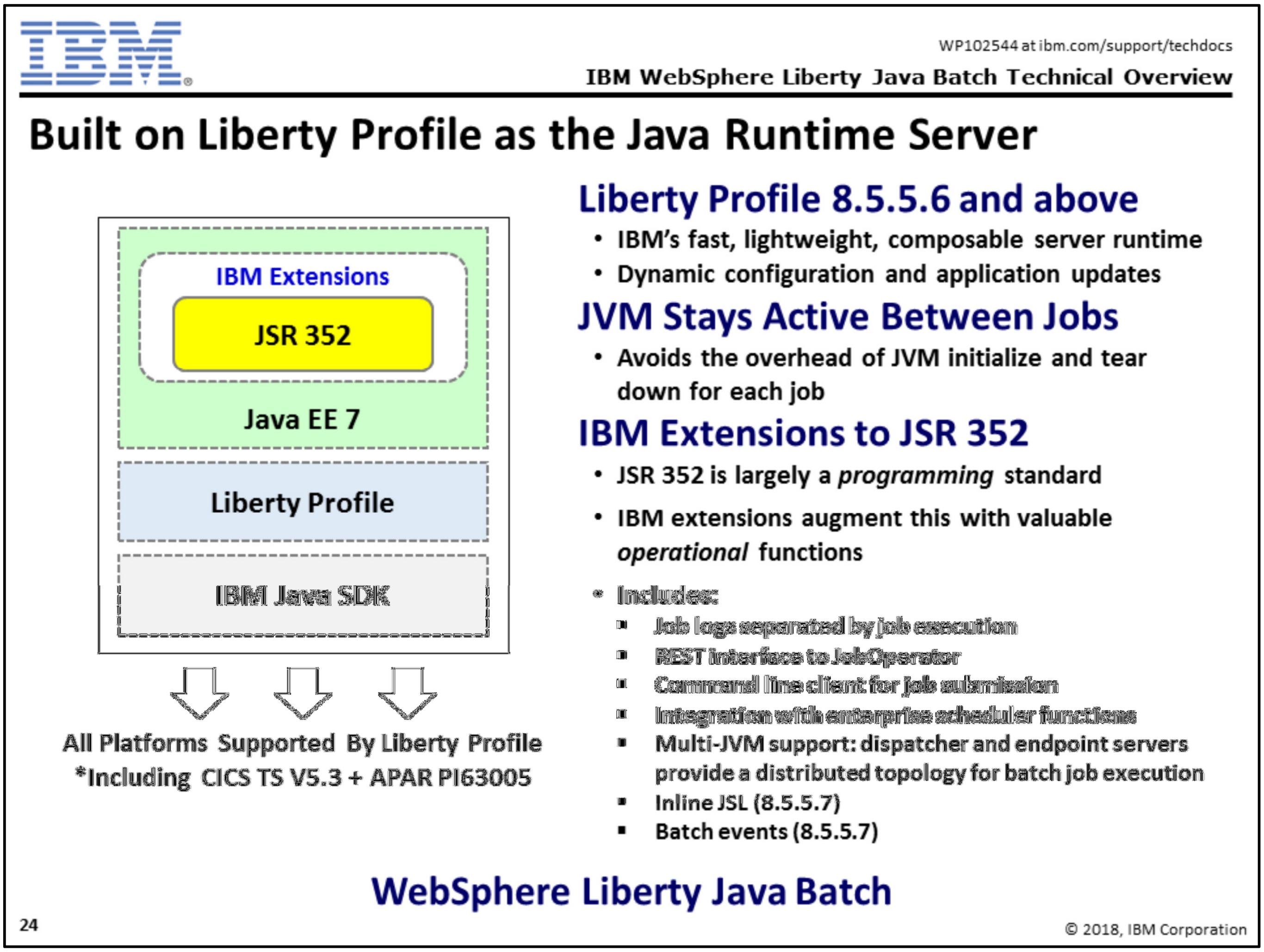
Finally, we have *listeners*. These are like "exits" in that they provide points during job and step execution where the container will turn control over to code you provide so that code can execute. What your listener code does is up to you. It is based on processing you wish to do at that time.

There are 8 listeners provided with the JSR 352 implementation. They are shown on the chart above. For example, the first one -- JobListener -- provides an exit point before the job begins step execution, and after (as well as calling your listener code if an exception is thrown). StepListener provides a callout exit before and after steps. And the list goes on ... begin and end of a chunk, begin and end of an itemRead operation, etc.



IBM Implementation And Extensions

© 2018, IBM Corporation



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

Built on Liberty Profile as the Java Runtime Server

IBM Extensions

JSR 352

Java EE 7

Liberty Profile

IBM Java SDK

All Platforms Supported By Liberty Profile
*Including CICS TS V5.3 + APAR PI63005

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

24

© 2018, IBM Corporation

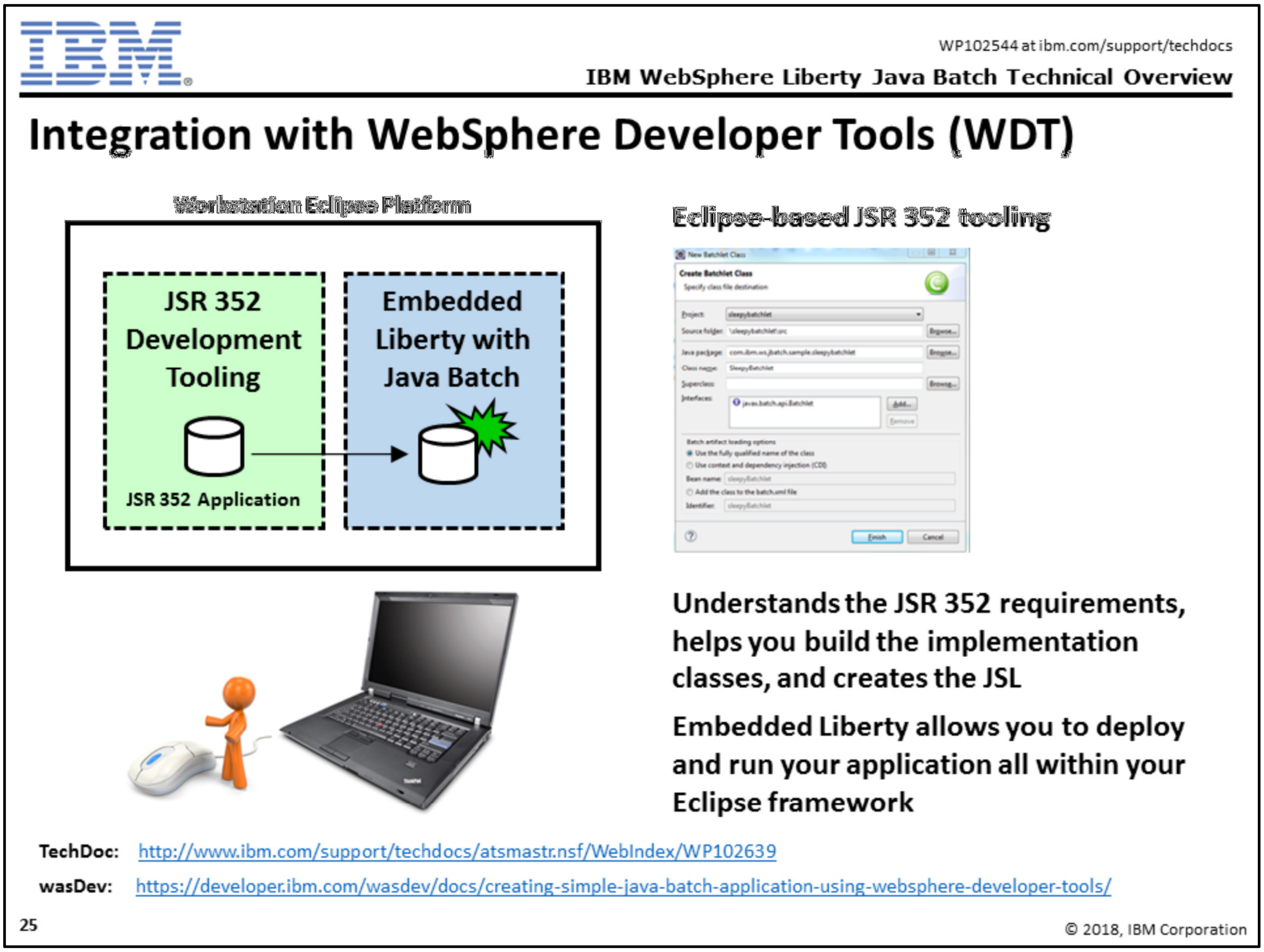
The JSR 352 implementation provided by IBM is built on Liberty Profile (all platforms). It is released along with the update of Liberty Profile to support the Java EE 7 standard, which contains the JSR 352 support as noted earlier. The first release with the Java EE 7 and JSR 352 support is 8.5.5.6.

Liberty Profile provides a good platform for running batch jobs because it is designed to be composable (you configure only those functions you need), which means it is also lightweight (only the memory needed for the functions you configure).

When batch jobs execute in a persistent server model – meaning, the server and its JVM stay up and active even when batch jobs are not executing – then the cost of initializing the JVM and tearing it down for every batch job is eliminated. That's a key consideration when the number of batch jobs being executed is a larger and larger number. A few batch jobs a day is likely not an issue, but when you get into hundreds or thousands of batch jobs a day, the cost of starting and stopping the JVM each time adds up. Better to leave the JVM active and execute the batch jobs in the same JVM.

Note: this is where Liberty Profile's dynamic nature becomes an asset. You do not need to stop and restart the Liberty Profile server when you deploy a new JSR 352 batch application. You do not need to stop and restart the server with most configuration changes. You can avoid the cost of server stops and restarts for many (if not most) changes you may need to make to your Liberty Profile configuration.

The JSR 352 standard is primarily a programming interface standard, which means many of the operational considerations are left unaccounted for by the standard. This is where vendors, such as IBM, are free to extend the standard with their own function. This is what IBM has done in a number of key areas. The rest of this presentation will focus on those extensions and what they provide.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

Integration with WebSphere Developer Tools (WDT)

Workstation Eclipse Platform

JSR 352 Development Tooling

JSR 352 Application

Embedded Liberty with Java Batch

Eclipse-based JSR 352 tooling

New Batchlet Class

Create Batchlet Class

Specify class file destination

Project: sleepybatchlet

Source folder: \sleepybatchlet\src

Java package: com.ibm.ws.jbatch.sample.sleepybatchlet

Class name: SleepyBatchlet

Superclass:

Interfaces: javax.batch.api.Batchlet

Batch artifact loading options

● Use the fully qualified name of the class

○ Use context and dependency injection (CDI)

Bean name: sleepyBatchlet

○ Add the class to the batch.xml file

Identifier: sleepyBatchlet

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

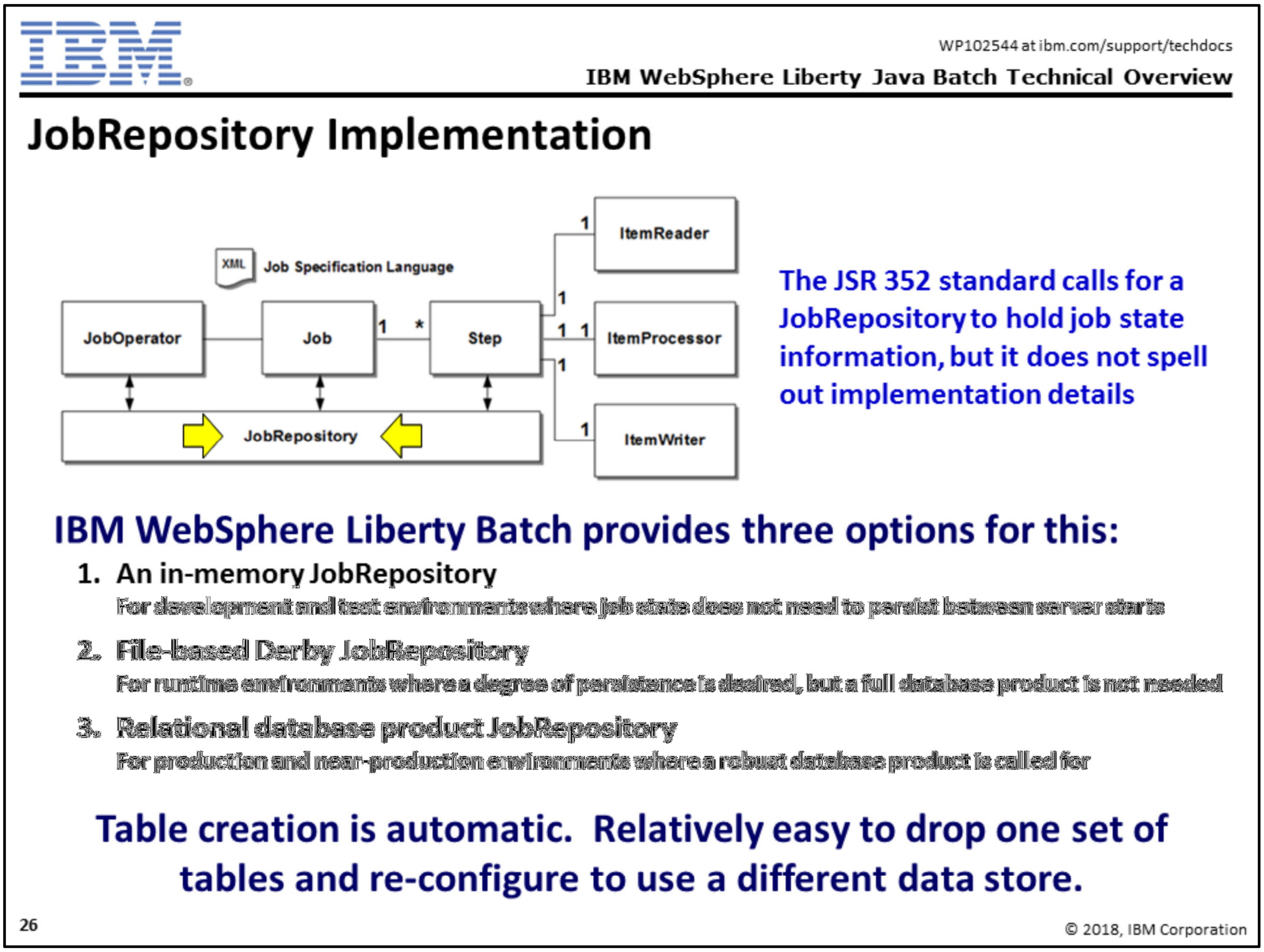
wasDev: https://developer.ibm.com/wasdev/docs/creating-simple-java-batch-application-using-websphere-developer-tools/

25

© 2018, IBM Corporation

WebSphere Developer Tools (WDT) is a plugin to Eclipse that helps you develop and test JSR 352 applications. It provides the Eclipse-based view of developing an application, with wizards and assistance in creating the job steps and the JSL to describe the job. The result is an application package.

WDT also has the ability to host a Liberty runtime in which you can deploy and test your Java batch application. WDT is aware of this embedded Liberty, and is capable of deploying applications to the directory for Liberty to detect and load automatically. Further, WDT makes use of the REST interface of the IBM JSR 352 runtime to submit the job and run it. (We explore the REST interface in a few charts. For now, understand that WDT makes use of one of the IBM extensions to the JSR 352 runtime to run the batch job.)


Job Specification Language
XML
JobOperator
Job
1
*
Step
1
ItemReader
1
1 1
ItemProcessor
1
1
ItemWriter
JobRepository

WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

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.

26

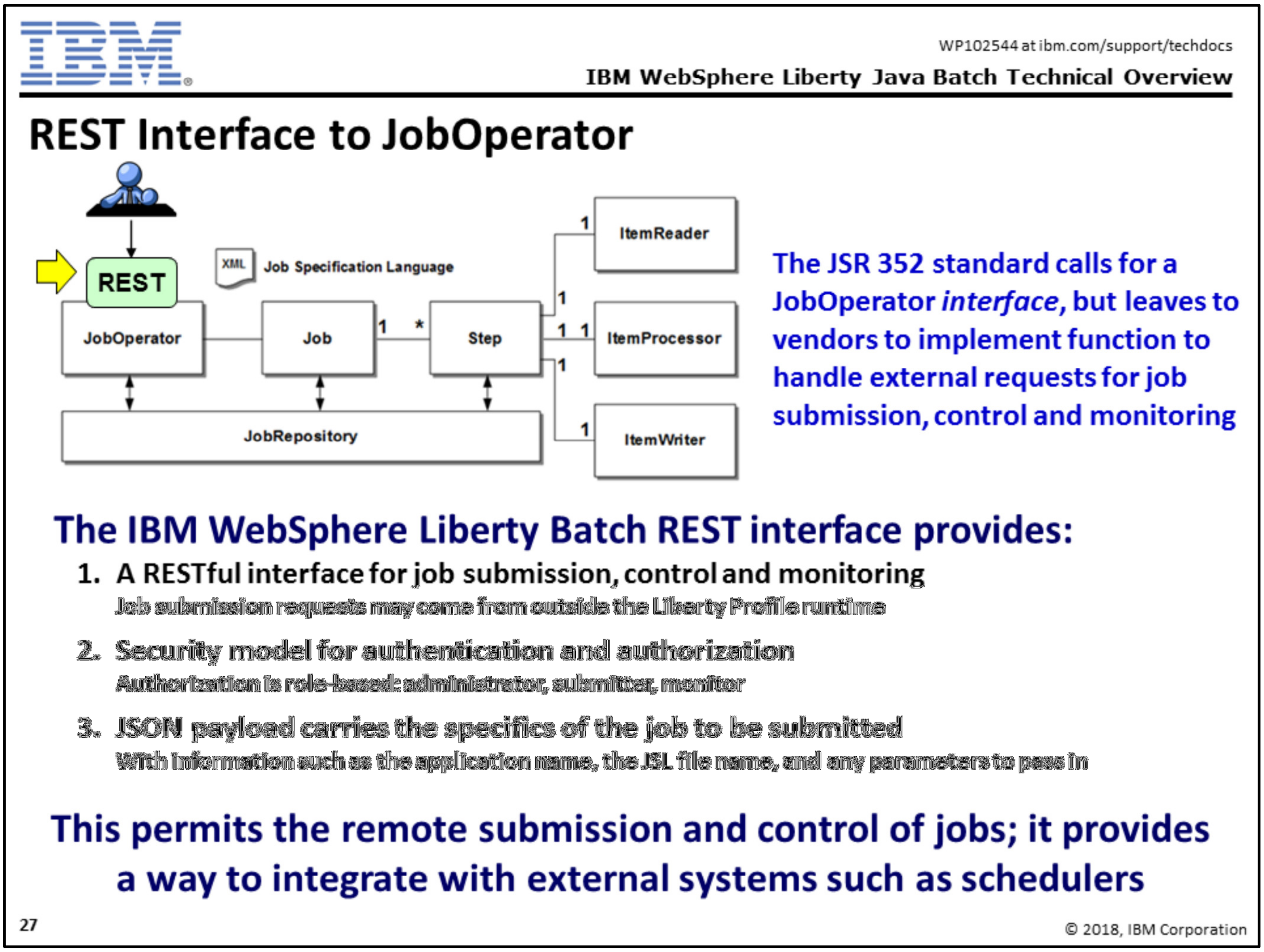
© 2018, IBM Corporation

As we noted earlier, the JSR 352 specification calls for a JobRepository, which is a mechanism to maintain information about the state of batch jobs. The specification does not spell out the details of how that is to be implemented, just that one must exist to be JSR 352 compliant.

IBM is providing three mechanisms for this JobRepository. Which you use is really a function of what you need, based on what you're using the Liberty Profile JSR 352 runtime for. If your use-case is ad hoc development and testing, and you don't require the information to persist across server starts, then an in-memory JobRepository is available. This makes configuring the JobRepository as simple as possible.

The next level of robustness is the file-based relational Derby database. This will provide data persistence across server restarts. It's something you can set up and use without having to involve your database administrators because creating the database is a simple matter of running a Java command. The database manifests as a set of UNIX files. It works. It's great for more advanced testing.

Finally, the next step up is a true relational database product such as IBM DB2. When the JSR 352 runtime environment is being used for production or near-production use-cases, then this is likely what you want to use.



REST Interface to JobOperator

REST
XML
Job Specification Language
JobOperator
Job
1
*
Step
1
ItemReader
1
1
ItemProcessor
1
1
ItemWriter
JobRepository

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

© 2018, IBM Corporation

The JSR 352 specification calls for a JobOperator function. But that function is an interface that needs to be invoked by some other code to do job submission, job monitoring and job control. The standard specification does not spell out what that code must be … that is left to developers or vendors to implement the user interface function that drives JobOperator.

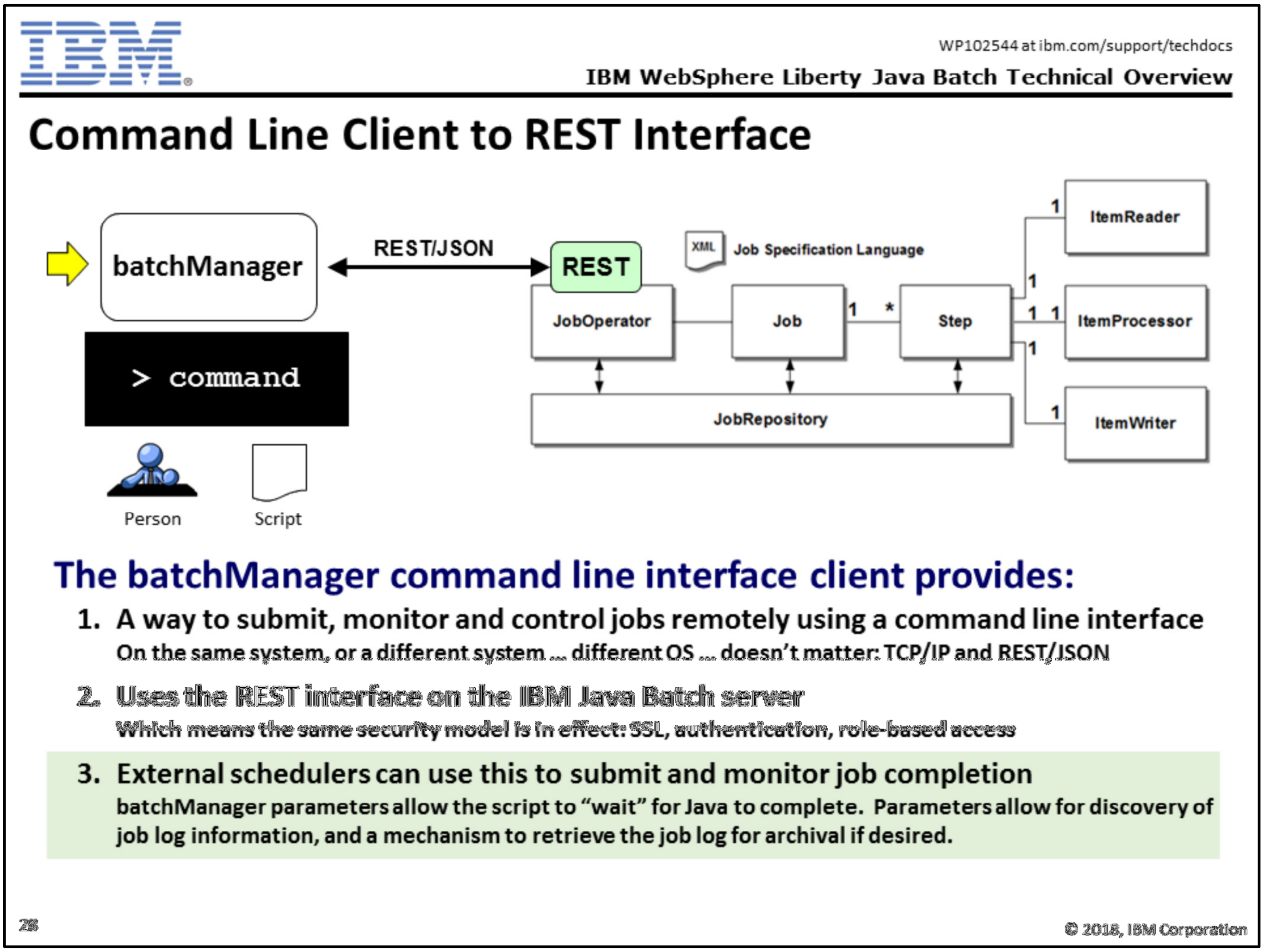
IBM provides a RESTful interface on the front of JobOperator. This provides a way to remotely submit and control batch jobs that run in this JSR 352 environment.

Note: this also serves as the foundation for the function used to integrate external schedulers. That function is a command line interface that uses the REST interface on the back end to submit and manage jobs. The command line interface (known as "batchManager") can be invoked by a scheduler. The batchManager then uses REST/JSON to submit the job. More on this in an upcoming chart.

JSON is passed with the REST call to provide information about the job to be submitted. The JSON is interpreted by the IBM REST interface function, and that's used to invoke the JobOperator methods to submit the job.

Note: an alternative is for the developer to provide their own submit / monitor / control function that uses the JobOperator interface. The IBM sample "SleepyBatchlet" does just that – it packages a servlet that takes a user's standard HTTP URL and submits and manages jobs. The REST interface eliminates the need for developers to spend time on a submission / control mechanism. That allows the developer to focus more time on implementing the batch business logic.

The REST provides three levels of security: transport security in the form of an encrypted SSL connection; access security in the form of authentication; and authorization security in the form of role-based access to functions of the JSR 352 runtime.



Command Line Client to REST Interface

batchManager

REST/JSON

REST

> command

Person

Script

XML Job Specification Language

JobOperator

Job

1 *

Step

1

1 1

1

ItemReader

ItemProcessor

ItemWriter

1

1

JobRepository

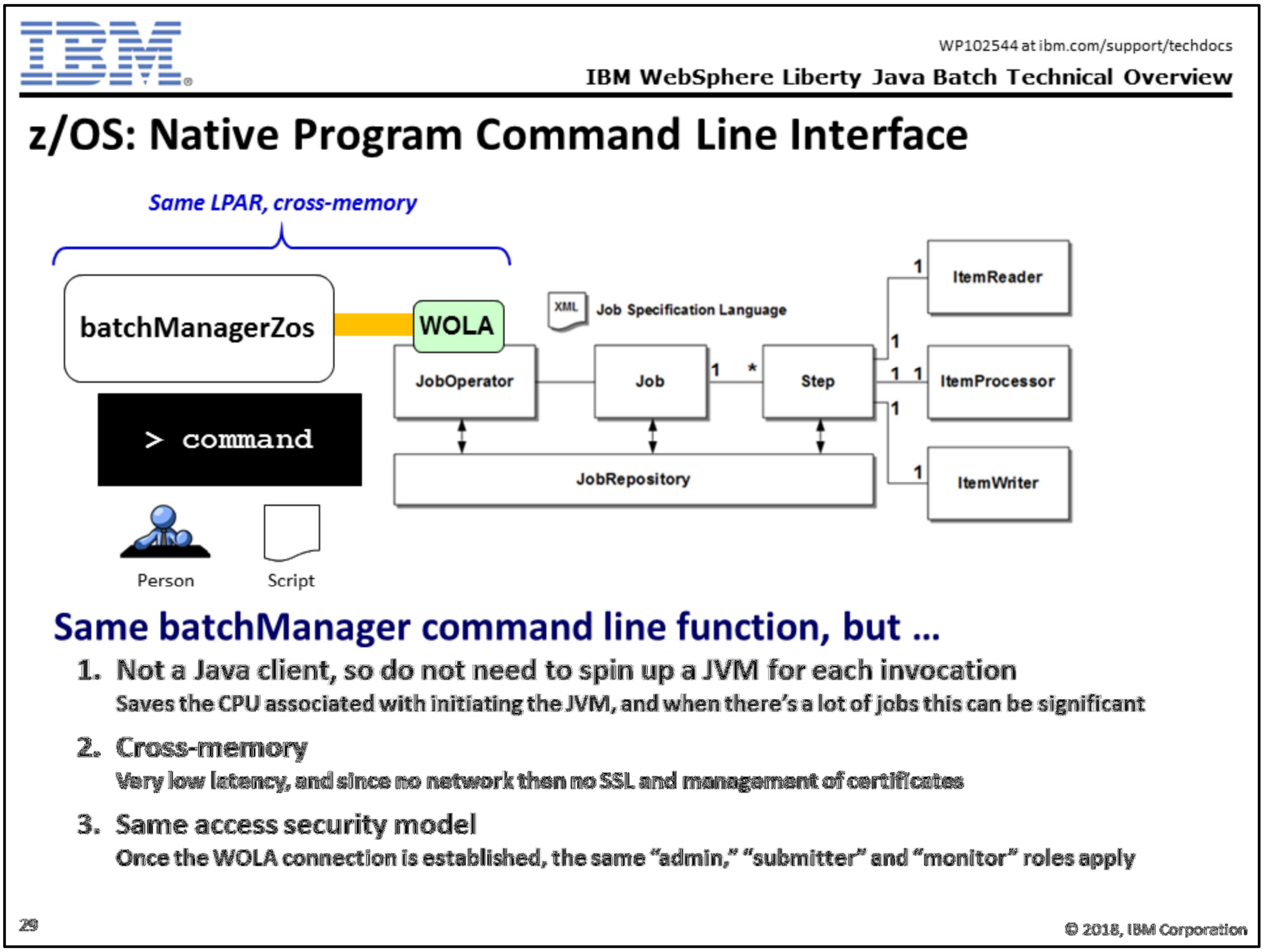
The batchManager command line interface client provides:

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.

© 2018, IBM Corporation

The batchManager function is a command line client that uses REST/JSON and the REST interface to submit and monitor jobs. The batchManager function eliminates the need for you to construct a REST client; the batchManager is the REST client with a command line interface.

The batchManager function provides a way to integrate with external schedulers. Those schedulers are already equipped to run shell scripts or submit programs. A shell script can be written to invoke the command line interface offered by batchManager. A "–wait" parameter can be used to have the script delay completion until the Java batch job in JSR 352 completes. When the Java batch job completes, the "—wait" parameter will release and the script ends. That tells the scheduler the job has completed.



z/OS: Native Program Command Line Interface

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

© 2018, IBM Corporation

Another variation on the batchManager command line interface is one for z/OS only … it's called batchManagerZos and it uses the cross-memory WOLA function to access the Liberty Profile server and invoke the JobOperator. It does not use the REST interface.

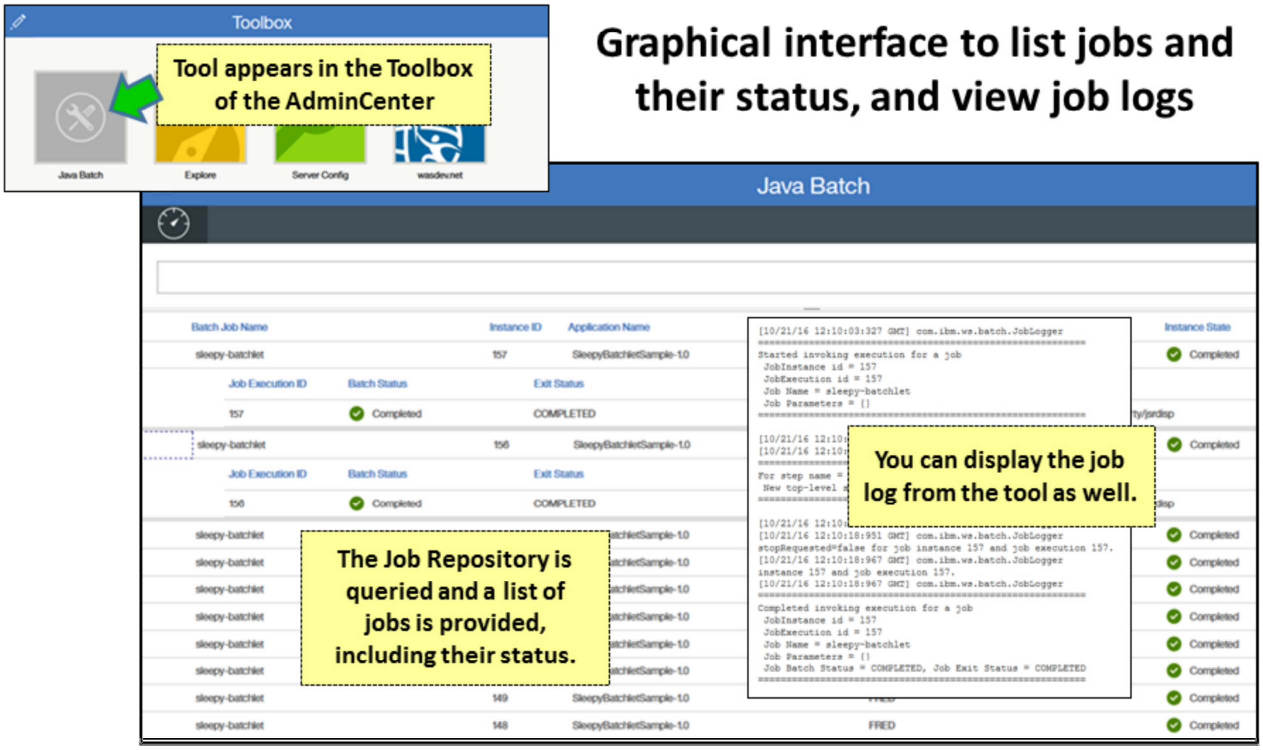
What's the advantage? As the chart indicates, this client (unlike the regular batchManager client), is not a Java client, so no JVM needs to be initiated for this. On z/OS that can be particularly important when overall CPU is being carefully monitored. A few invocations of the Java client batchManager might not be noticed, but thousands per day might. The batchManagerZos client avoids that.

This client uses WOLA, which is cross-memory, which means there's no TCP network stack involved. There's no need to coordinate certificates because there is no SSL … it's cross-memory. The limitation there is, of course, that batchManagerZos and the Liberty Profile server with JSR 352 must be in the same z/OS LPAR.

The same authorization roles as batchManager exists – admin, submitter and monitor.

AdminCenter Java Batch Tool 16.0.0.4

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



Tool appears in the Toolbox of the AdminCenter

Batch Job Name	Instance ID	Application Name		Instance State
sleepy-batchlet	157	SleepyBatchletSample-1.0		Completed
Job Execution ID	Batch Status	Exit Status		
157	Completed	COMPLETED	ty/jsrdisp	
sleepy-batchlet	156	SleepyBatchletSample-1.0		Completed
Job Execution ID	Batch Status	Exit Status		
156	Completed	COMPLETED	disp	
sleepy-batchlet		atchletSample-1.0		Completed
sleepy-batchlet		atchletSample-1.0		Completed
sleepy-batchlet		atchletSample-1.0		Completed
sleepy-batchlet		atchletSample-1.0		Completed
sleepy-batchlet		atchletSample-1.0		Completed
sleepy-batchlet		atchletSample-1.0		Completed
sleepy-batchlet	149	SleepyBatchletSample-1.0		Completed
sleepy-batchlet	148	SleepyBatchletSample-1.0	FRED	Completed

```
[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:
[10/21/16 12:10:
================
For step name =
 New top-level s
================

[10/21/16 12:10:
[10/21/16 12:10:18:951 GMT] com.ibm.ws.batch.JobLogger
stopRequested=false for job instance 157 and job execution 157.
[10/21/16 12:10:18:967 GMT] com.ibm.ws.batch.JobLogger
instance 157 and job execution 157.
[10/21/16 12:10:18:967 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.

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

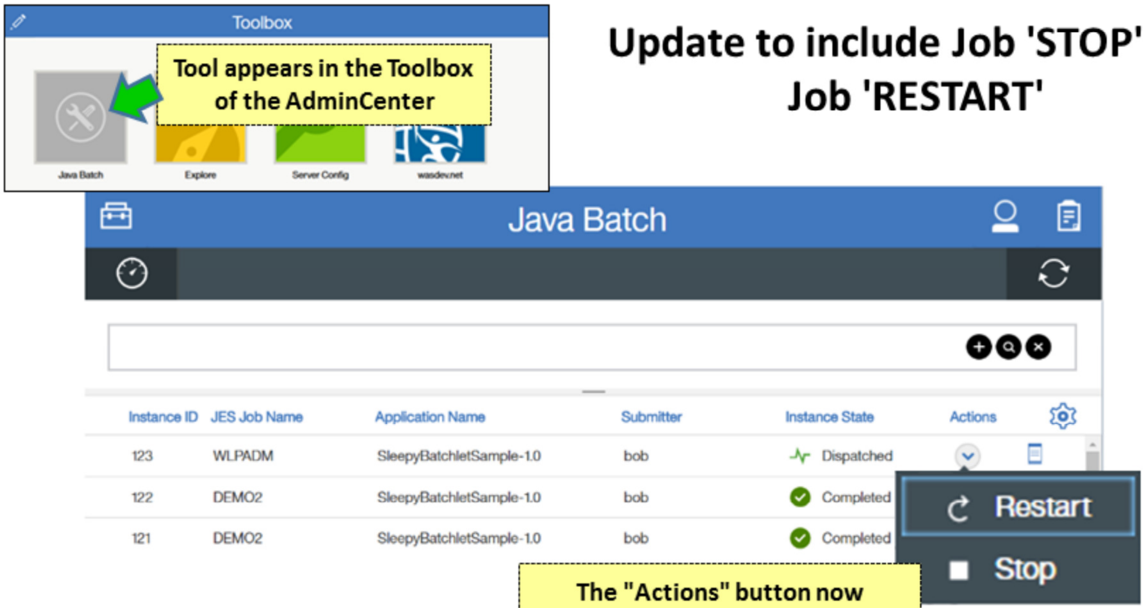
© 2018, IBM Corporation

In the 16.0.0.4 release of Liberty, the AdminCenter was updated to include a "Java Batch" tool that appears in the list of tools in the toolkit. This Java Batch tool is really a graphical interface to the information in the Job Repository: it retrieves from the database the information about the jobs and formats the result on the browser screen. There you can see the status of each job, and retrieve the job log for viewing as well.

It should be noted this is a "read only" function. You can't submit jobs through this interface or perform update actions such as delete jobs. Those roles are performed through the batchManager or batchManagerZos command line utilities. This tool is good for people responsible for monitoring jobs but do not have the authority to submit or change jobs.

AdminCenter Java Batch Tool 17.0.0.1

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



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	
121	DEMO2	SleepyBatchletSample-1.0	bob	✓ Completed	

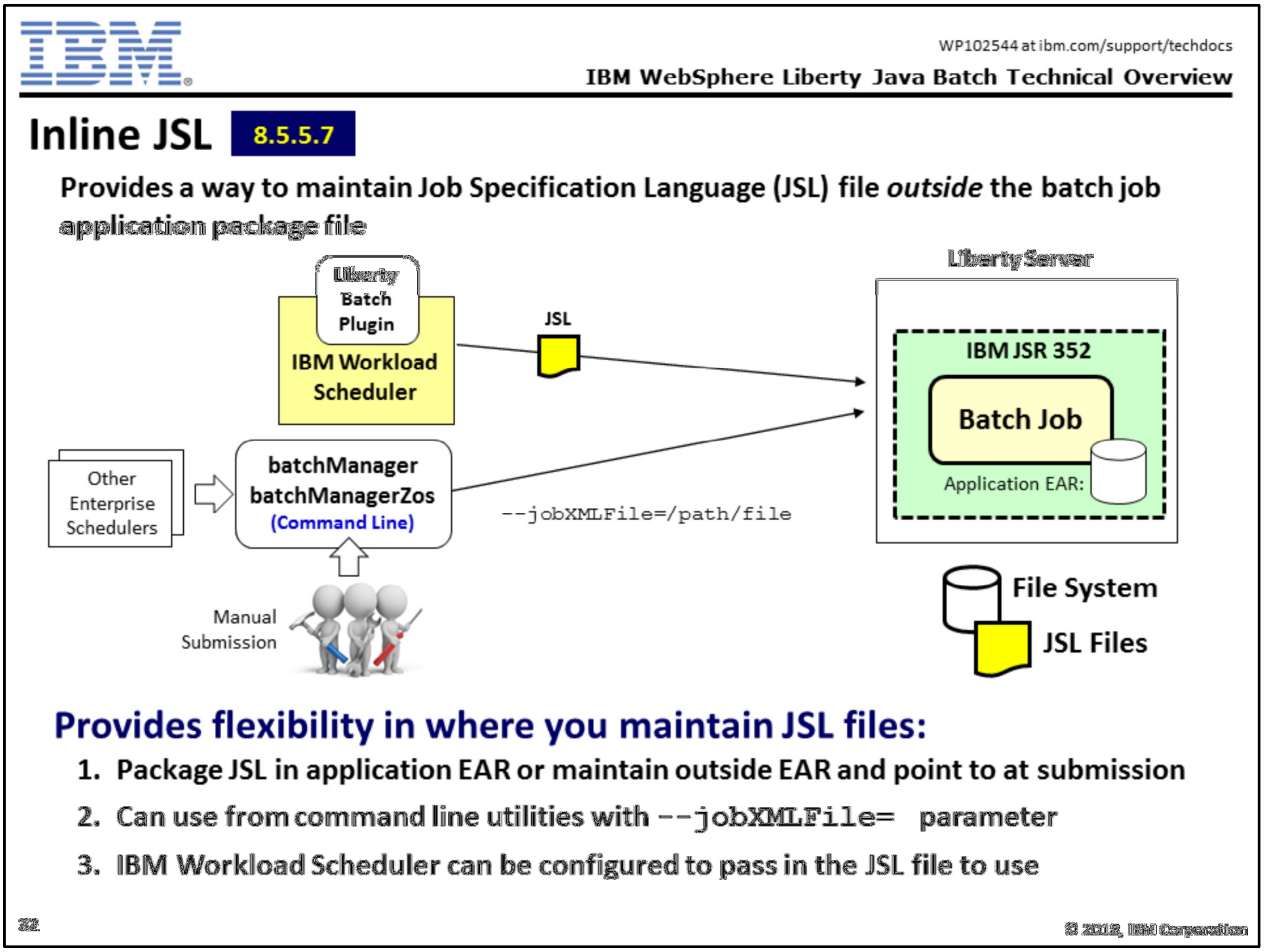
Tool appears in the Toolbox of the AdminCenter

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

31

© 2018, IBM Corporation

In the 17.0.0.1 release of Liberty, the AdminCenter Java Batch tool has been enhanced to provide additional operational controls for a batch job. For a job that's currently running, you can invoke a 'Stop' action against the job. For a batchlet step, that will stop in accordance with the stop() method that's implemented. For a chunk step, the current checkpoint interval processing will be stopped, the transaction rolled back to the last persisted checkpoint value, and the job stopped. Upon restart, a batchlet step simply runs again. A chunk step will pick up at the last checkpoint value.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

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:

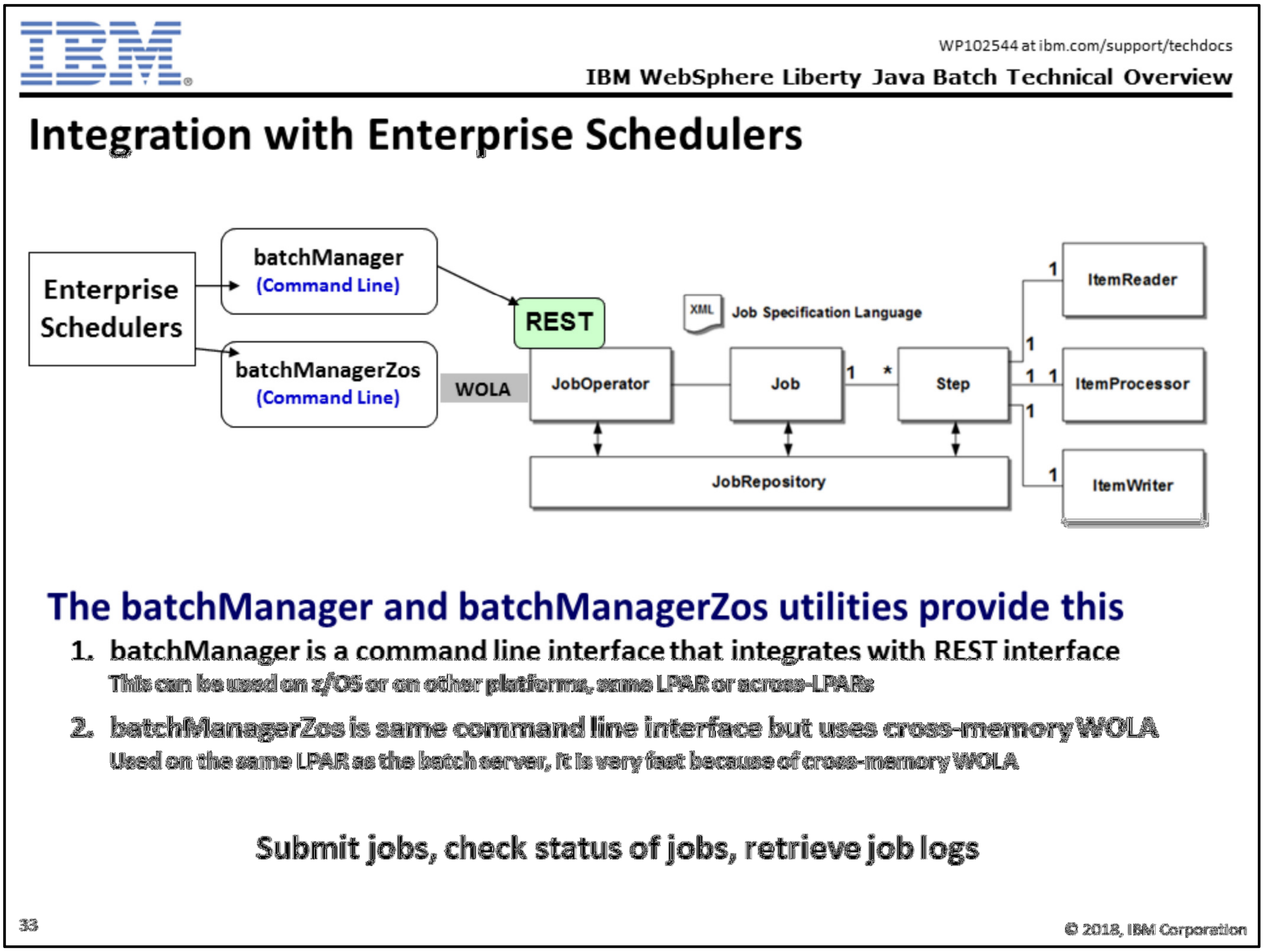
1. Package JSL in application EAR or maintain outside EAR and point to at submission
2. Can use from command line utilities with `--jobXMLFile=` parameter
3. IBM Workload Scheduler can be configured to pass in the JSL file to use

32

© 2018, IBM Corporation

Earlier we showed the Job Specification Language (JSL) file packaged with the application. When submitting a job using the Command Line Interface, the JSL to use is named using the `--jobXMLName=` parameter. That name gets passed to the server over the REST interface. The container then looks in the application packaging directory and uses the named JSL.

Another way to accomplish this is to use IBM's "Inline JSL" extension, which allows you to maintain the JSL outside the application packaging and provide the JSL at time of job submission. From the Command Line Interface this is done with the `--jobXMLFile=` parameter providing the path and name of the JSL file. The IBM Workload Scheduler product can be used to store JSL for use when submitting jobs.



Integration with Enterprise Schedulers

The batchManager and batchManagerZos utilities provide this

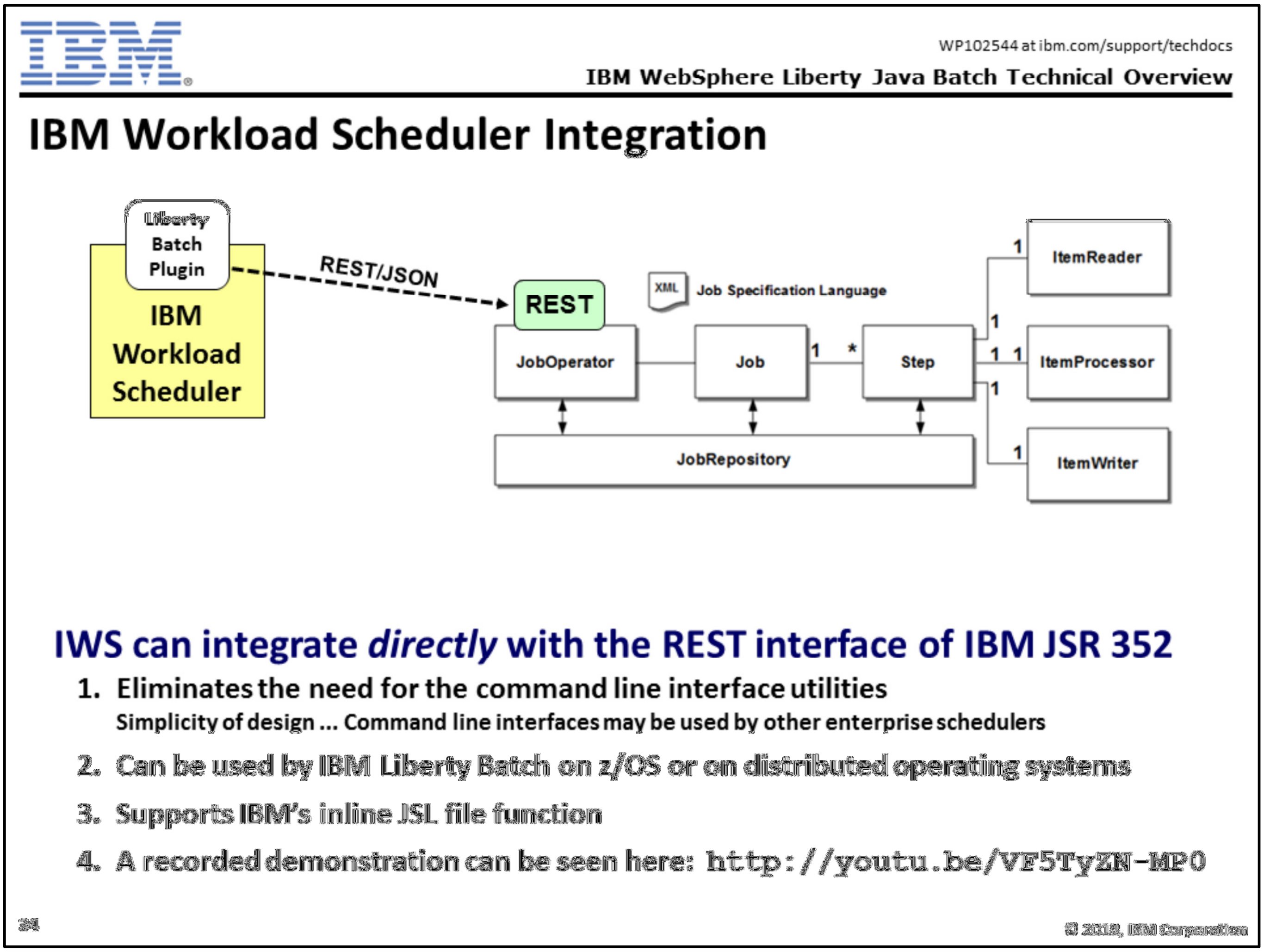
1. batchManager is a command line interface that integrates with REST interface
   This can be used on z/OS or on other platforms, same LPAR or across-LPARs
2. batchManagerZos is same command line interface but uses cross-memory WOLA
   Used on the same LPAR as the batch server, it is very fast because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

Submit jobs, check status of jobs, retrieve job logs

© 2018, IBM Corporation

Integration with enterprise scheduler functions (either from IBM or other vendors) is always a topic of interest when Java Batch is discussed. IBM's WebSphere Liberty Batch provides a command line interface function to provide this integration. The enterprise scheduler can interface with the command line to submit jobs, check the status of jobs and retrieve job logs.

Two command line interface utilities are provided: batchManager and batchManagerZos. Both have the same command line syntax. The difference is batchManager uses RESTful calls to access the REST interface of the IBM WebSphere Liberty Batch server; batchManagerZos uses the cross-memory WOLA support of Liberty z/OS to access the server. If your enterprise scheduler is on the same LPAR as the IBM WebSphere Liberty Java Batch server, the use of WOLA provides a very fast connectivity mechanism. The advantage of batchManager (REST integration) is it can be used from any platform.



IBM Workload Scheduler Integration

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

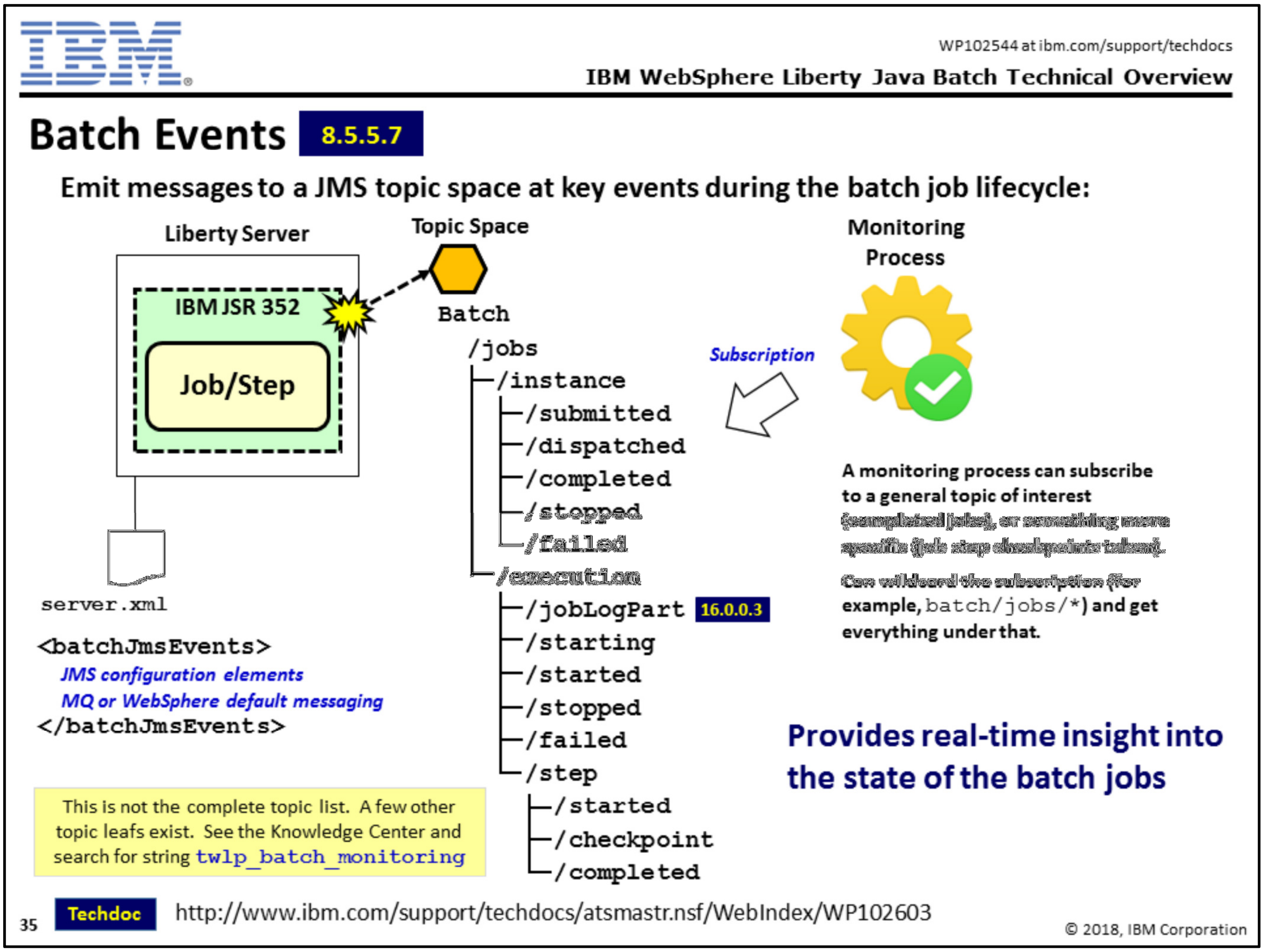
1. Eliminates the need for the command line interface utilities
   Simplicity of design ... Command line interfaces may be used by other enterprise schedulers
2. Can be used by IBM Liberty Batch on z/OS or on distributed operating systems
3. Supports IBM's inline JSL file function
4. A recorded demonstration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

© 2018, IBM Corporation

IBM Workload Scheduler is an enterprise scheduler product used to manage and coordinate batch job workloads. It has the ability to interact directly with the IBM JSR 352 REST interface for submitting batch jobs and monitoring their progress.

This function is enabled in IBM Workload Scheduler (IWS) by way of a configurable “plugin” to IWS. This is noteworthy because it means you do not need to be at the latest level of IWS to use this. And because it’s using REST, which is a network-based protocol, this can be used to integrate with IBM JSR 352 on any platform.

If you’re interested in seeing more on this function, see the video at the link shown on the chart.



Batch Events 8.5.5.7

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

Liberty Server
IBM JSR 352
Job/Step
Topic Space
Monitoring Process
Subscription

```
Batch
  /jobs
    /instance
      /submitted
      /dispatched
      /completed
      /stopped
      /failed
    /execution
      /jobLogPart 16.0.0.3
      /starting
      /started
      /stopped
      /failed
      /step
        /started
        /checkpoint
        /completed
```

server.xml

```
<batchJmsEvents>
  JMS configuration elements
  MQ or WebSphere default messaging
</batchJmsEvents>
```

This is not the complete topic list. A few other topic leafs exist. See the Knowledge Center and search for string `twlp_batch_monitoring`

A monitoring process can subscribe to a general topic of interest (completed jobs), or something more specific (job step checkpoints taken).

Can wildcard the subscription (for example, `batch/jobs/*`) and get everything under that.

Provides real-time insight into the state of the batch jobs

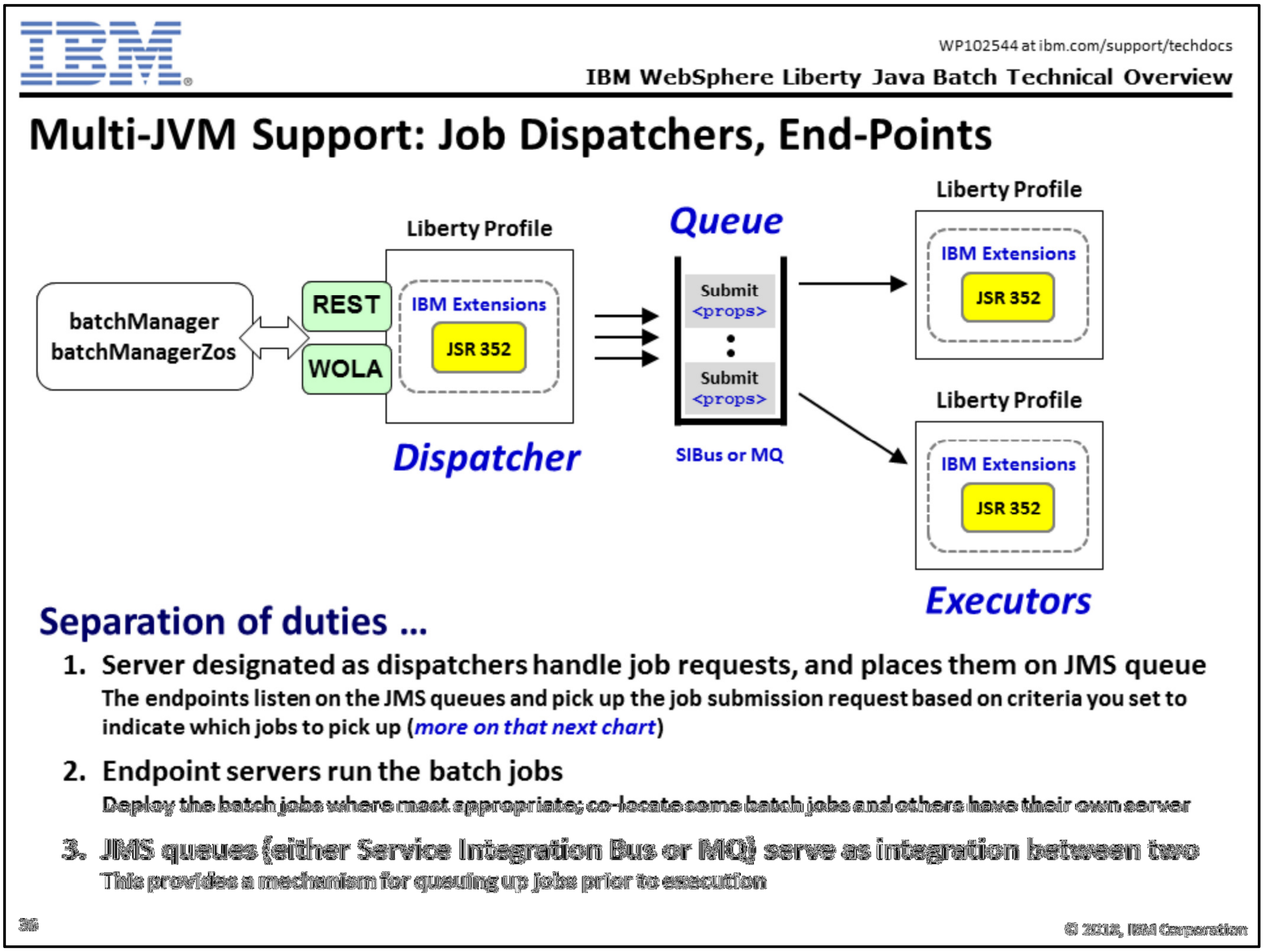
© 2018, IBM Corporation

When a JSR 352 batch job is running, the job will run through a number of states. The "Batch Events" function is designed to emit an "event" (a message) to a JMS topic space as the batch job progresses through the various job states. This is useful for processes that wish to monitor the progress of jobs.

This feature of IBM JSR 352 is optional. It is configured with an update to the `server.xml` for the IBM JSR 352 Liberty instance. That XML update enables the function and provides information about the JMS specifics -- the MQ queue manager and topic, or the Default Messaging location and topic.

The topic space is organized into a tree structure, and the IBM JSR 352 runtime will emit events associated with the leaves of the tree based on the job state. Monitoring processes may subscribe to specific leaves, or use wildcards to get events associated with multiple leaves.

Again, this provides a way for processes to subscribe to the topic and monitor the state of processing.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

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

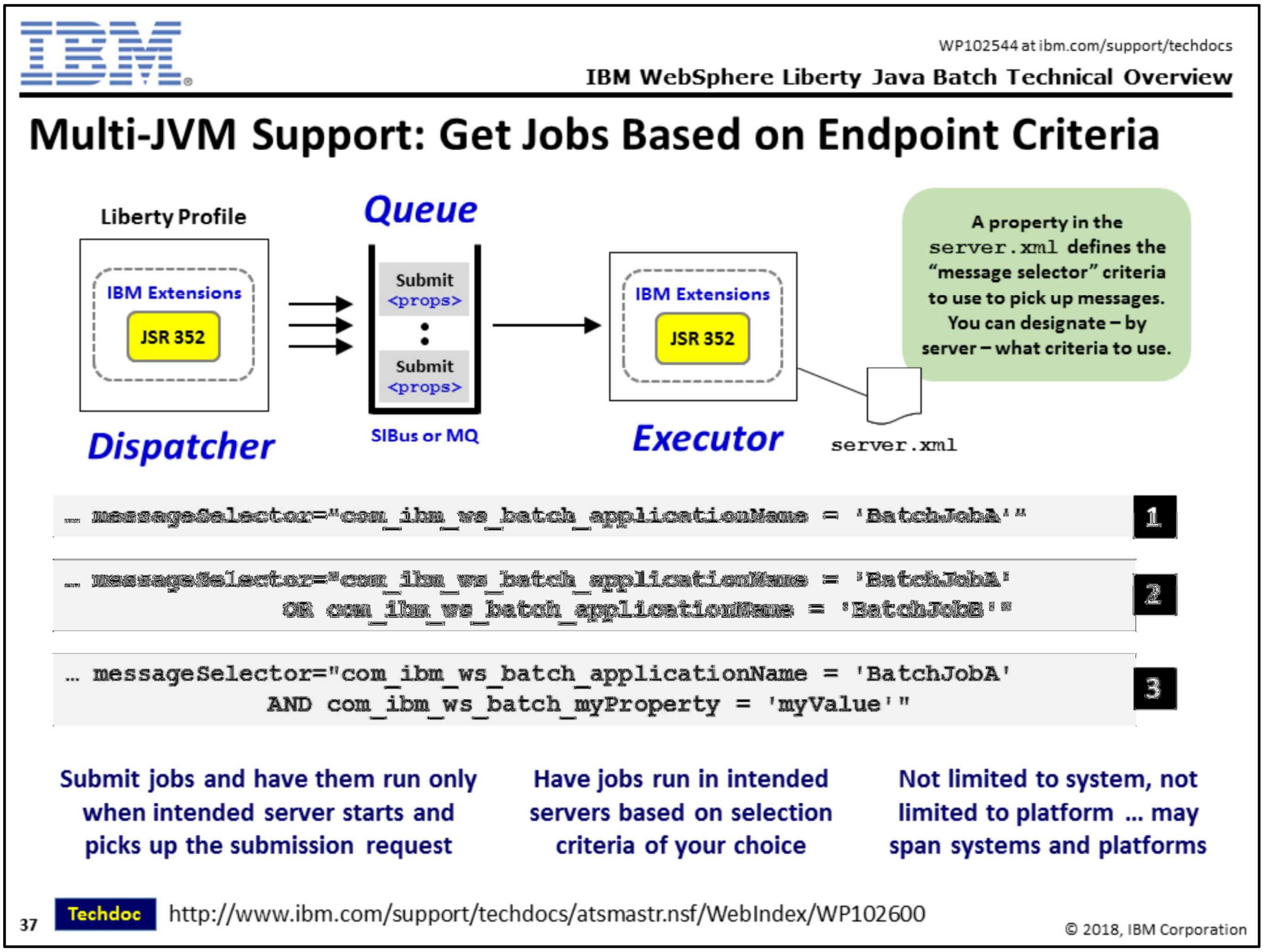
36

© 2018, IBM Corporation

The next extension we'll cover is the multiple-JVM support for JSR 352. In a nutshell, what this does is a provide a way to separate the duties, and have a Liberty Profile server act as a dispatcher of jobs, and have other Liberty Profile servers act as executors where the jobs run.

This mechanism is built around JMS queuing – either the integrated Service Integration Bus (SIBus) of Liberty Profile or MQ. The Liberty Profile server that acts as the dispatcher receives the job submission requests through the REST or WOLA interfaces as outlined earlier. The dispatcher then puts the job submission request on the queue. From there the servers designated as endpoints pick the messages up and run the jobs.

But wait … what server picks up what job requests? That's based on a selection criteria you set in the executor servers. We'll cover that on the next chart. For now the thing to understand is the executor servers will only pick up those messages they are configured to pick up.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

Multi-JVM Support: Get Jobs Based on Endpoint Criteria

A property in the `server.xml` defines the “message selector” criteria to use to pick up messages. You can designate – by server – what criteria to use.

```
… 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

37 Techdoc http://www.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102600

© 2018, IBM Corporation

The previous chart mentioned that the endpoint servers will only pick up job submission requests that the executor server is configured to pay attention to. That’s done with a property in the JMS listener configuration element. The property is `messageSelector=`, and it indicates what messages in the queue to pay attention to and pick up for execution.

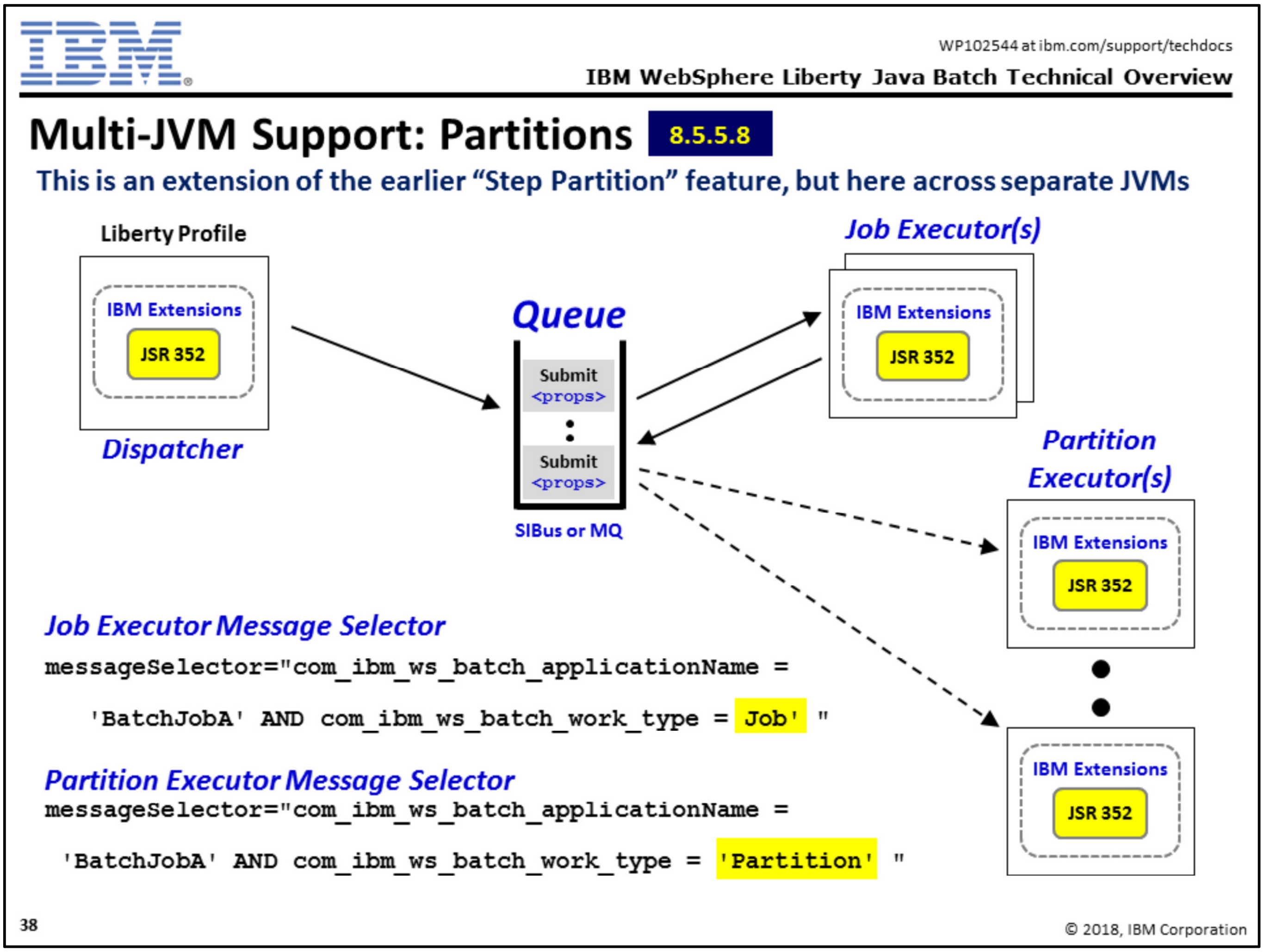
The chart offers three examples with numbered blocks. Those numbered blocks correspond to these notes:

1. The simplest example is where the endpoint is configured to look for and pick up a job request based on the application name. Imagine you have five executor servers, and one is configured with the message selector for BatchJobA. Then only that server will run that batch job. The other executor servers will see the job submission request, but since their message selector configuration is for something other than BatchJobA they will ignore it. Only the server configured for BatchJobA will run it.

2. This shows the way you would code an “or” condition for two batch jobs.

3. You are not limited to just the application name. You can make the executor select based on your own custom properties in the job submission message. This example is showing selection based on two things – the application name equaling “BatchJobA” *and* the custom property myProperty equaling a value of “myValue.”

The value of this extension is it provides a way to separate job submission from job execution. It allows jobs to be submitted and wait in the queue until a server is started that is configured to run the job. You can use this to have jobs run in only those servers you want them to run in (for example, on z/OS a server with a particular WLM service classification associated with it). This model is also not limited to just one server or just one platform … because it’s using JMS queuing between the dispatcher and executor, it’s possible to have this span servers and platforms. There’s considerable flexibility in what can be done with this.



Multi-JVM Support: Partitions 8.5.5.8

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

Liberty Profile

IBM Extensions

JSR 352

Dispatcher

Queue

Submit
<props>

Submit
<props>

SIBus or MQ

Job Executor(s)

IBM Extensions

JSR 352

Partition Executor(s)

IBM Extensions

JSR 352

IBM Extensions

JSR 352

Job Executor Message Selector

```
messageSelector="com_ibm_ws_batch_applicationName =
   'BatchJobA' AND com_ibm_ws_batch_work_type = Job' "
```

Partition Executor Message Selector

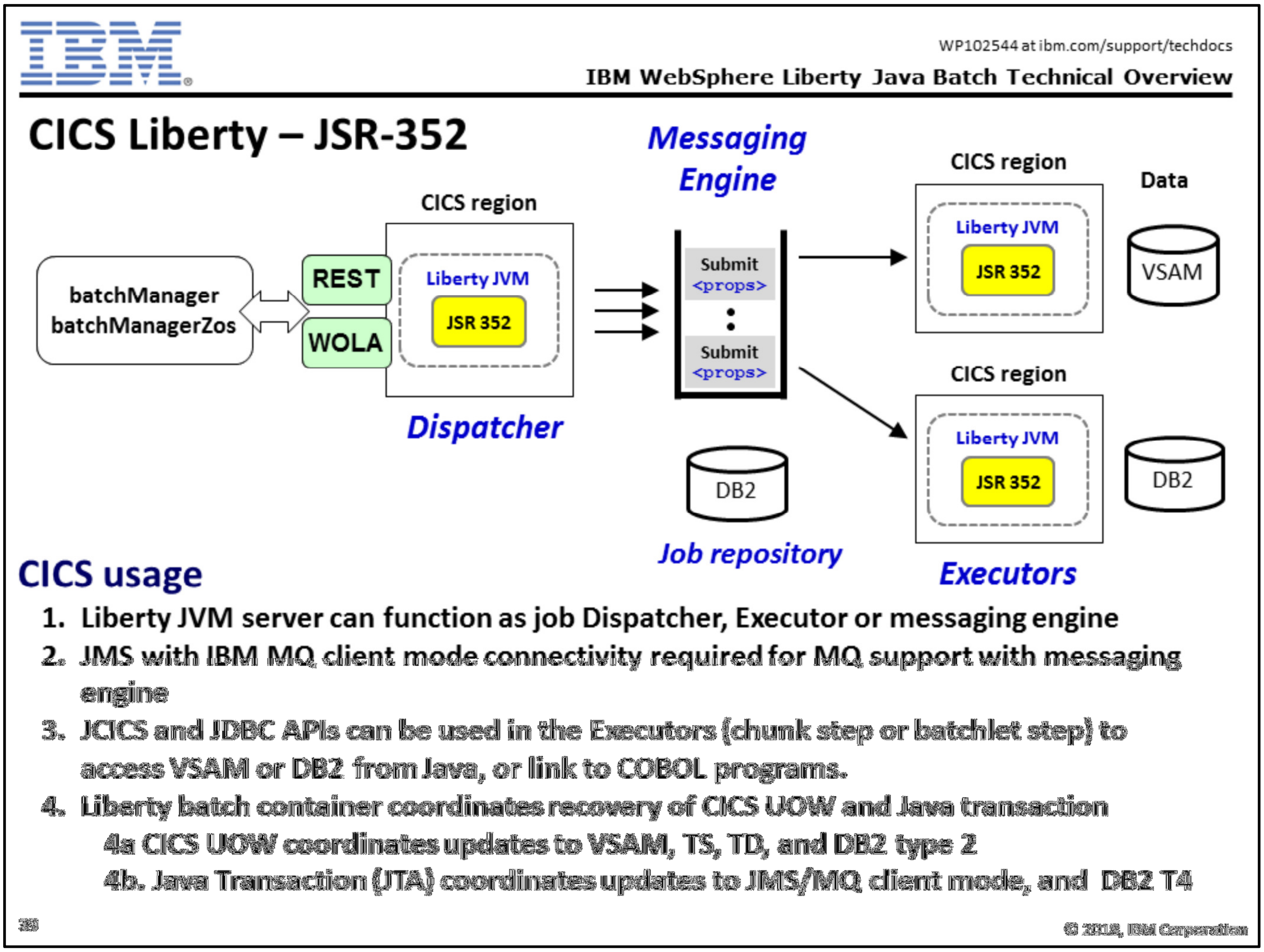
```
messageSelector="com_ibm_ws_batch_applicationName =
 'BatchJobA' AND com_ibm_ws_batch_work_type = 'Partition' "
```

38

© 2018, IBM Corporation

Another functional extension came in with the 8.5.5.8 fixpack. This is an extension of the earlier "Step Partition" feature, but whereas the JSR 352 specification provided for partitioning workload across threads in a single JVM, this splits across multiple JVMs.

This function is built on the multi-JVM support with a dispatcher and executor servers and a messaging queue between the two. A distinction is now made between a "job executor" and a "partition executor." The job executor is what listens for and picks up dispatch messages with "work_type=Job", and it then partitions the job and places the partition requests back on the queue with "work_type=Partition". The partition executors listen for and pick up these partitioned job requests and execute them. The top job (running in the Job executor, waits to hear back from the partition processes. When they all complete the top job moves on to the next step.



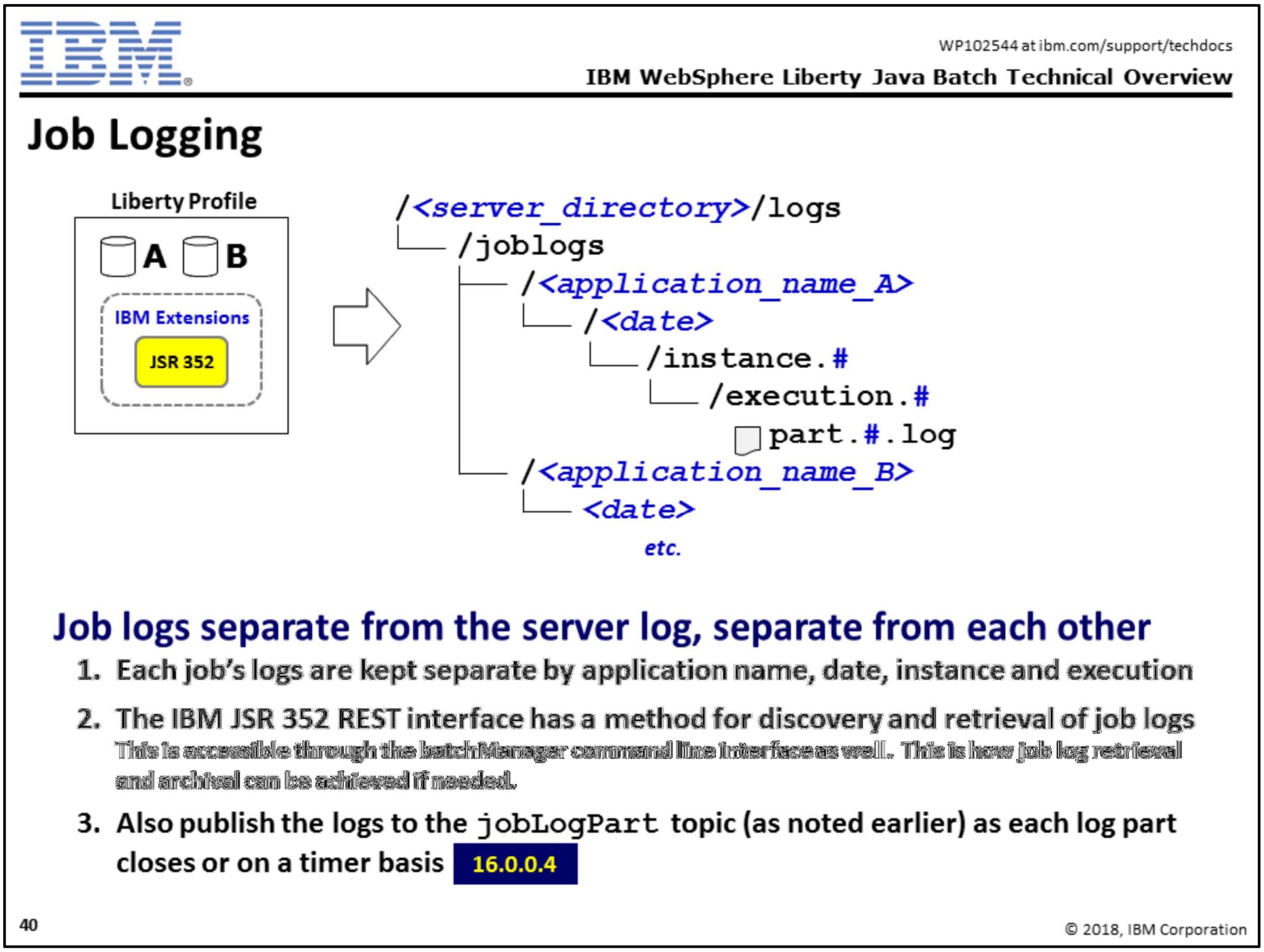
CICS Liberty – JSR-352

CICS usage

1. Liberty JVM server can function as job Dispatcher, Executor or messaging engine
2. JMS with IBM MQ client mode connectivity required for MQ support with messaging engine
3. JCICS and JDBC APIs can be used in the Executors (chunk step or batchlet step) to access VSAM or DB2 from Java, or link to COBOL programs.
4. Liberty batch container coordinates recovery of CICS UOW and Java transaction
   4a CICS UOW coordinates updates to VSAM, TS, TD, and DB2 type 2
   4b. Java Transaction (JTA) coordinates updates to JMS/MQ client mode, and DB2 T4

© 2018, IBM Corporation

Finally, here's a chart showing how you would accomplish the dispatcher/executor model using the support for Liberty inside of CICS.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

Job Logging

```
/<server_directory>/logs
└── /joblogs
       ├── /<application_name_A>
       │   └── /<date>
       │        └── /instance.#
       │             └── /execution.#
       │                   part.#.log
       └── /<application_name_B>
           └── <date>
                  etc.
```

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

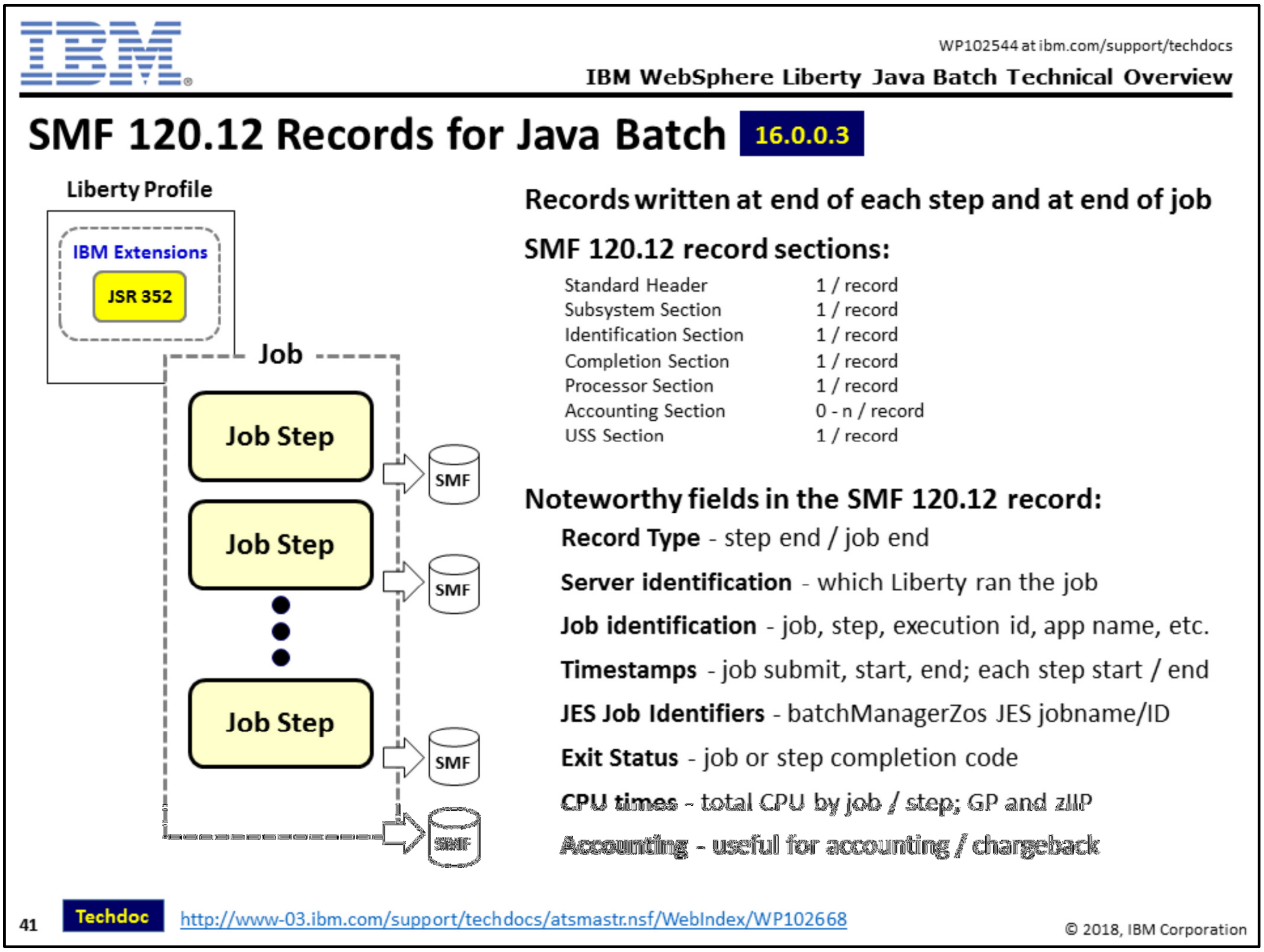
40

© 2018, IBM Corporation

Job logging does not seem a very exciting extension, but when hundreds or thousands of jobs are being run then the ability to logically organize job logs becomes very helpful.

The way job logging works with IBM's JSR 352 is to place logs into a directory tree based on the application name, the date, the job instance number and the job execution number. The chart above illustrates this. The logs by default go into the `/logs` directory for the server, but can be configured to go to any location you specify.

Once the job log has been written, the REST interface (or the batchManager command line client) will return the location of the job log for a given job execution. If you wish to retrieve the job log, the REST interface will allow you to do that.



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

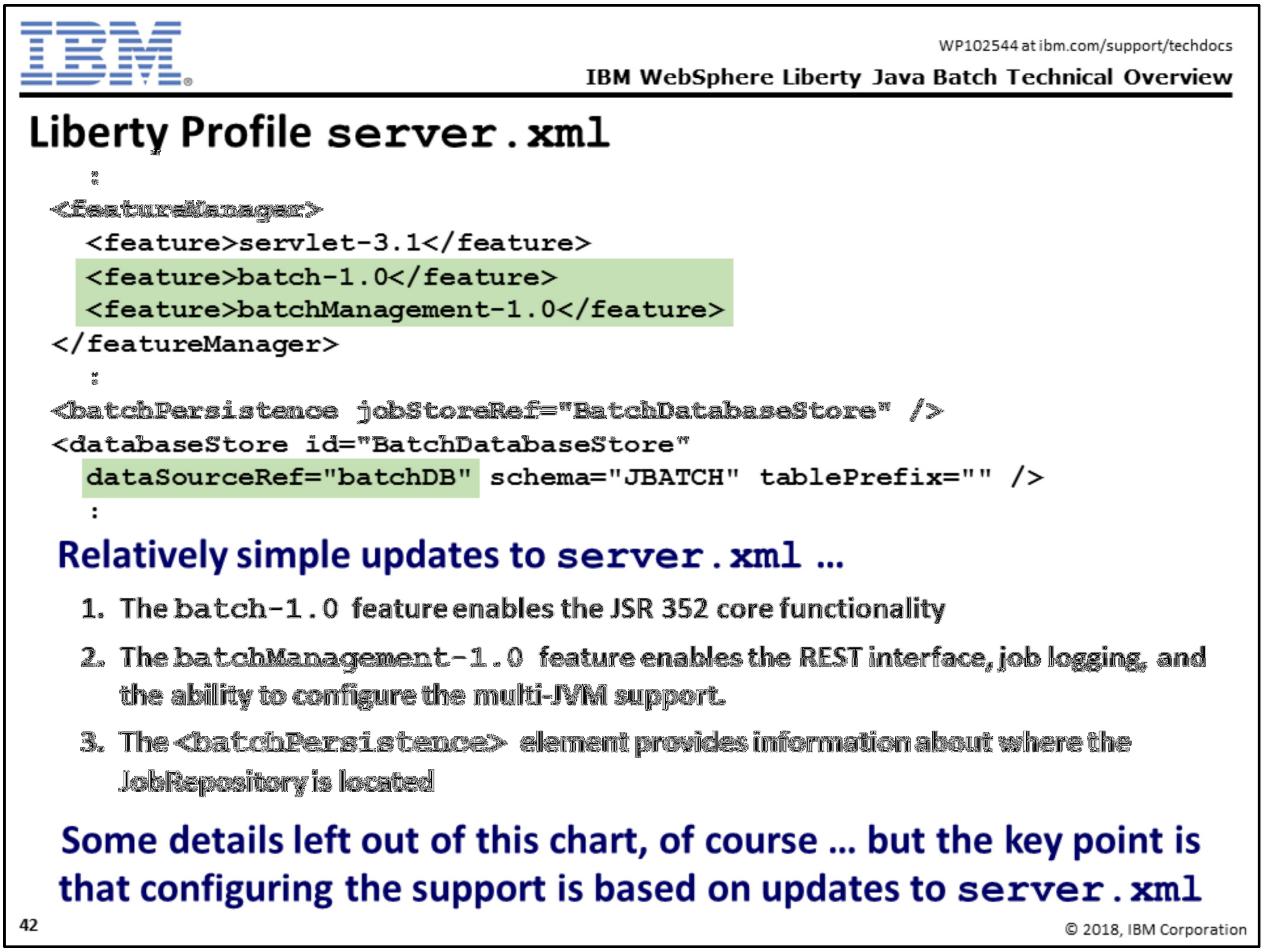
Techdoc http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102668

© 2018, IBM Corporation

SMF (Systems Management Facilities) is a mechanism on z/OS that allows components to write activity records in a highly efficient way. The content of the SMF record written by a component is based on what the component architects wish to capture. For Java Batch, the SMF record is 120, subtype 12.

A summary review of what's included with the 120.12 record is shown on the chart. The information include the job and step start and stop times; information about the server where the job ran; the exit status; as well as CPU time used information.

For a more complete review of the new SMF 120.12 record, see the WP102668 Techdoc. The URL is on the chart.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

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

42

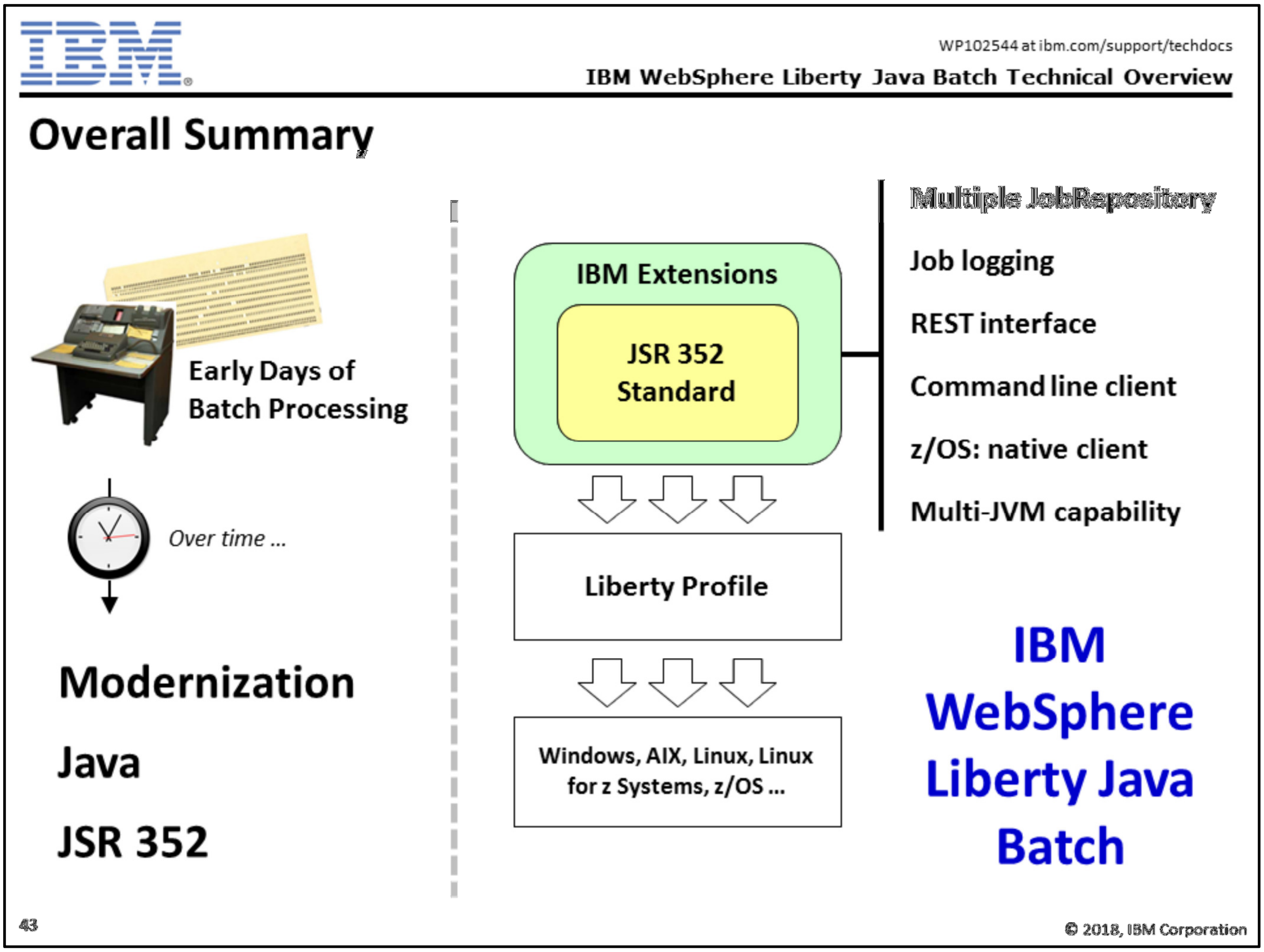
© 2018, IBM Corporation

Finally, enablement of the JSR 352 functionality within a Liberty Profile server is relatively simple. Two `<feature>` elements enable the function:

- **`batch-1.0`** … this is what enables the core JSR 352 support. This does not enable the REST interface, the job logging, or the multi-JVM support. You can run batch jobs with just batch-1.0 but you would need some function to invoke the JobOperator interface to submit the job (and as noted, the SleepyBatchlet sample illustrates how this is done).
- **`batchManagement-1.0`** … this enables the REST interface, multi-JVM support and the job logging support. If you code this then batch-1.0 is assumed and is loaded if not specified. Coding both as shown works as well.

The JSR 352 environment needs to understand where the JobRepository is going to be, and the `<batchPersistence>` element provides this. This element is used to contain the definitions about which persistence model is being used and where the database is located.

Note: there are some details that are left off this chart. We are not showing the configuration XML for the JDBC support, for example. We are not showing the JMS definitions for the multi-JVM support. Like any solution the details become very important when you are trying to configure and implement the solution. This presentation is focused on showing an overview, so for this chart we're leaving details out to keep things focused on the key points.



Overall Summary

Early Days of Batch Processing

Over time ...

Modernization

Java

JSR 352

IBM Extensions

JSR 352 Standard

Liberty Profile

Windows, AIX, Linux, Linux for z Systems, z/OS ...

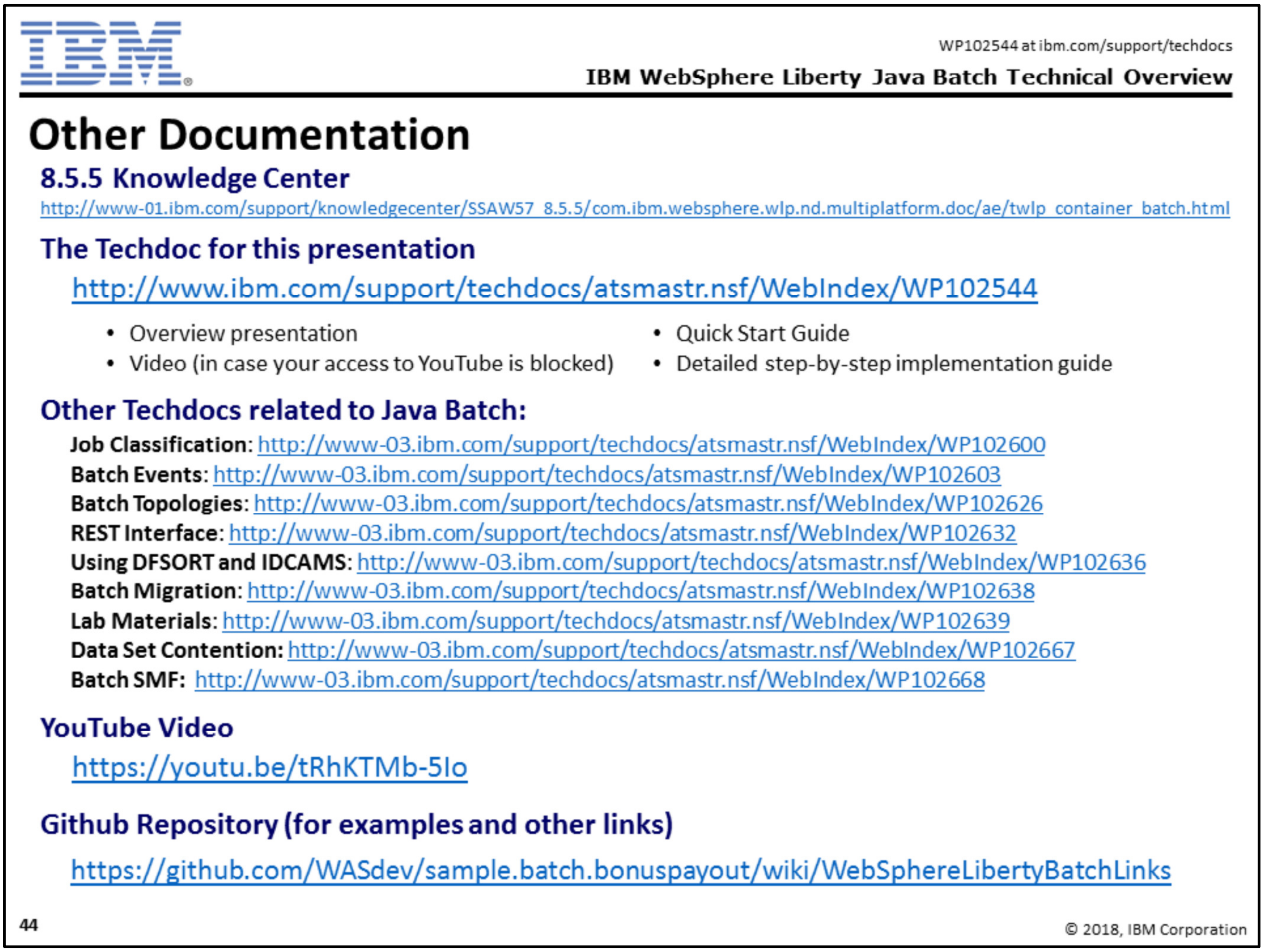
- Multiple JobRepository
- Job logging
- REST interface
- Command line client
- z/OS: native client
- Multi-JVM capability

IBM WebSphere Liberty Java Batch

43

© 2018, IBM Corporation

And this is the overall summary of the message delivered in the charts of this presentation.



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

The Techdoc for this presentation

http://www.ibm.com/support/techdocs/atsmastr.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/atsmastr.nsf/WebIndex/WP102600
Batch Events: http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102603
Batch Topologies: http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102626
REST Interface: http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102632
Using DFSORT and IDCAMS: http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102636
Batch Migration: http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102638
Lab Materials: http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102639
Data Set Contention: http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102667
Batch SMF: http://www-03.ibm.com/support/techdocs/atsmastr.nsf/WebIndex/WP102668

YouTube Video

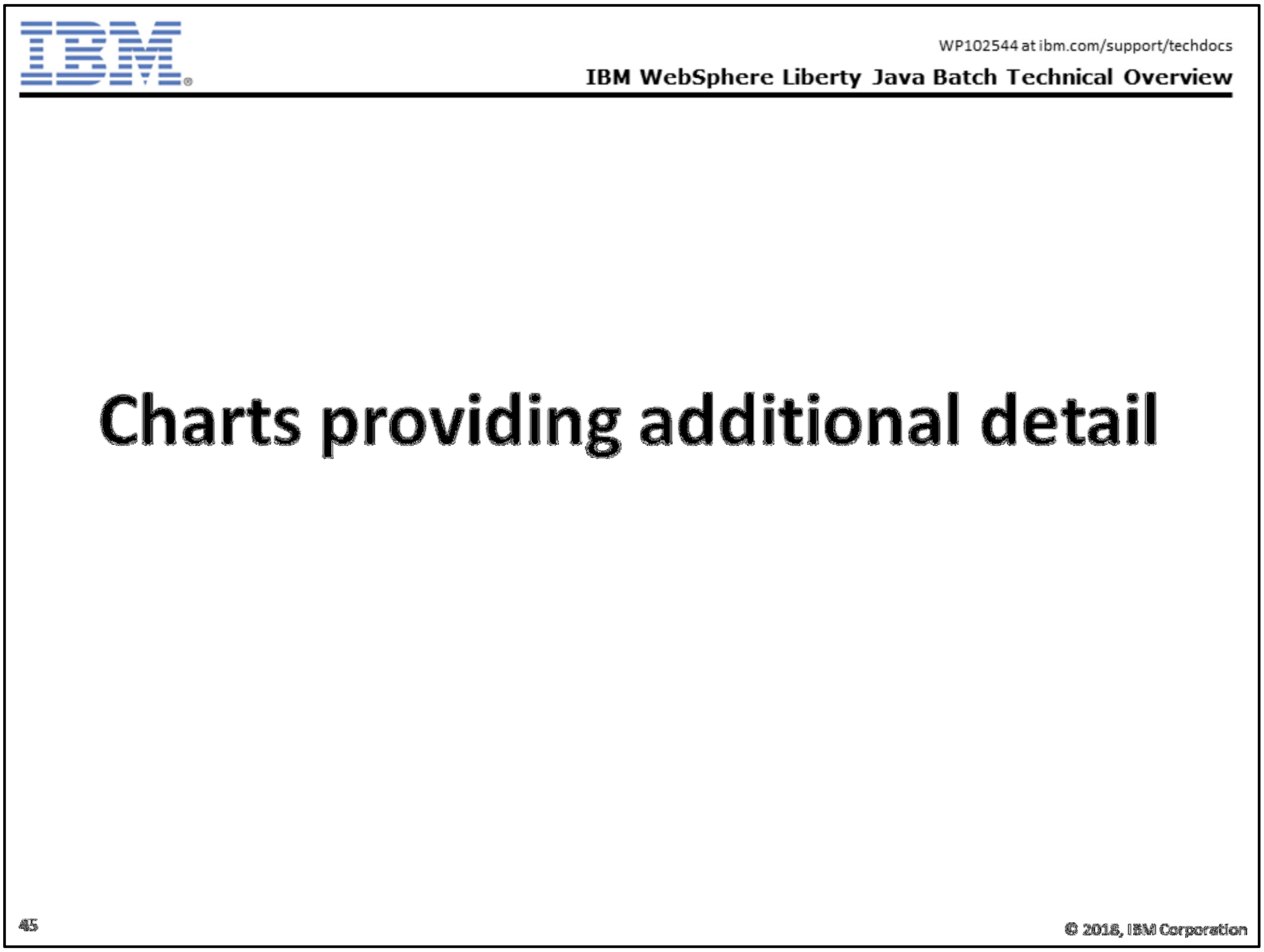
https://youtu.be/tRhKTMb-5Io

Github Repository (for examples and other links)

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

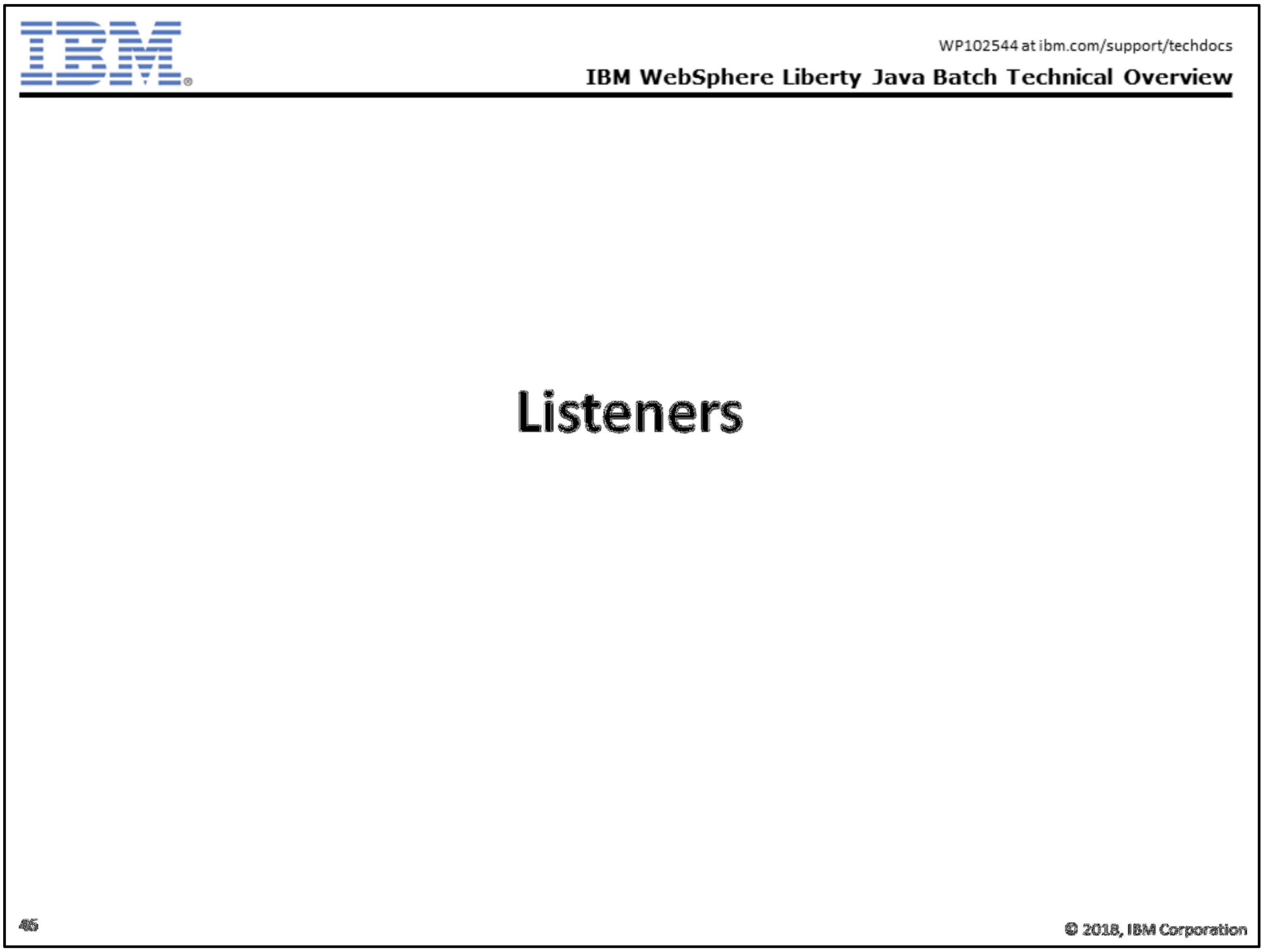
© 2018, IBM Corporation

A reference for other information.



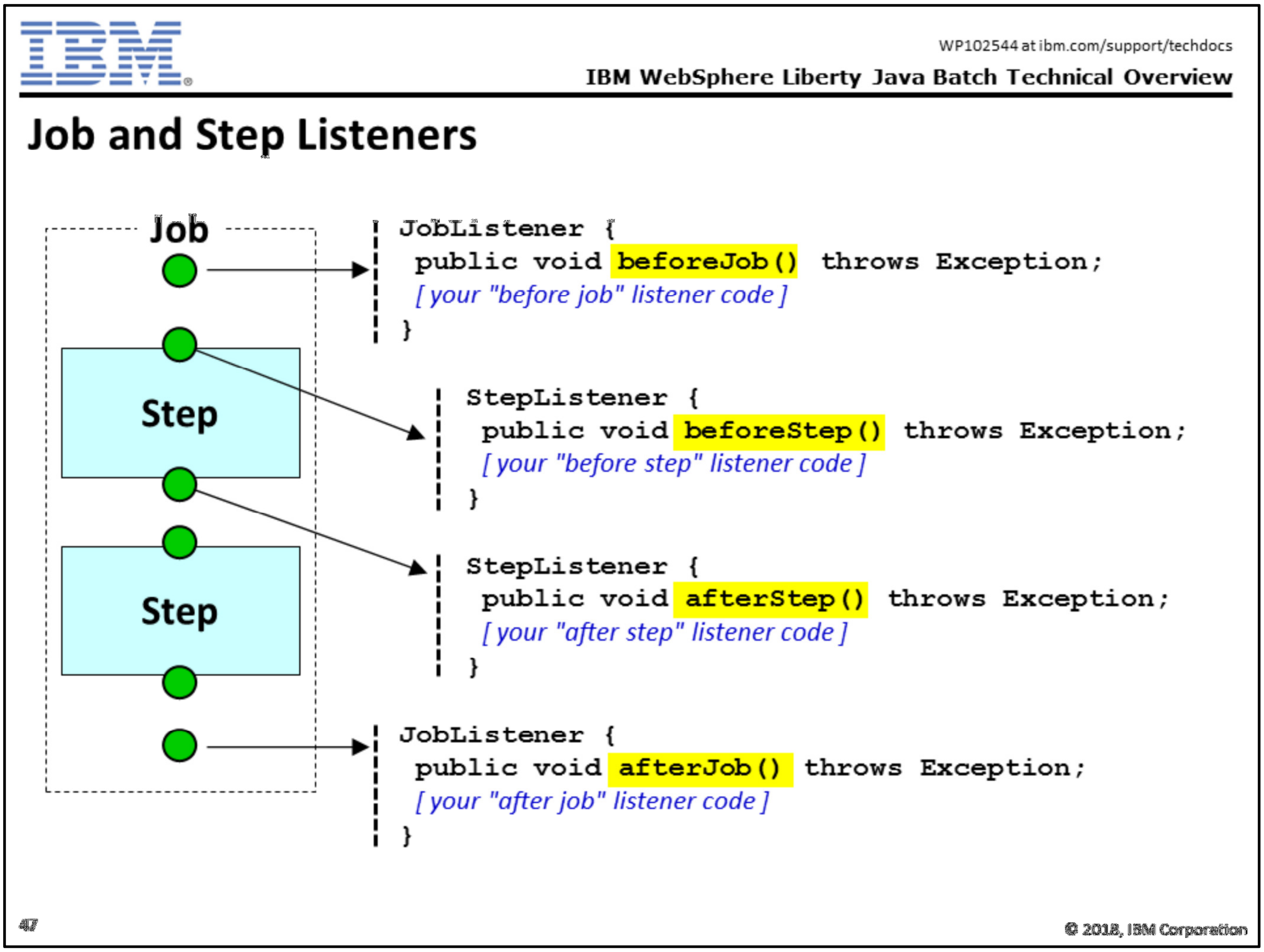
Charts providing additional detail

© 2018, IBM Corporation



Listeners

© 2018, IBM Corporation



Job and Step Listeners

```
JobListener {
 public void beforeJob() throws Exception;
 [ your "before job" listener code ]
}
```

```
StepListener {
 public void beforeStep() throws Exception;
 [ your "before step" listener code ]
}
```

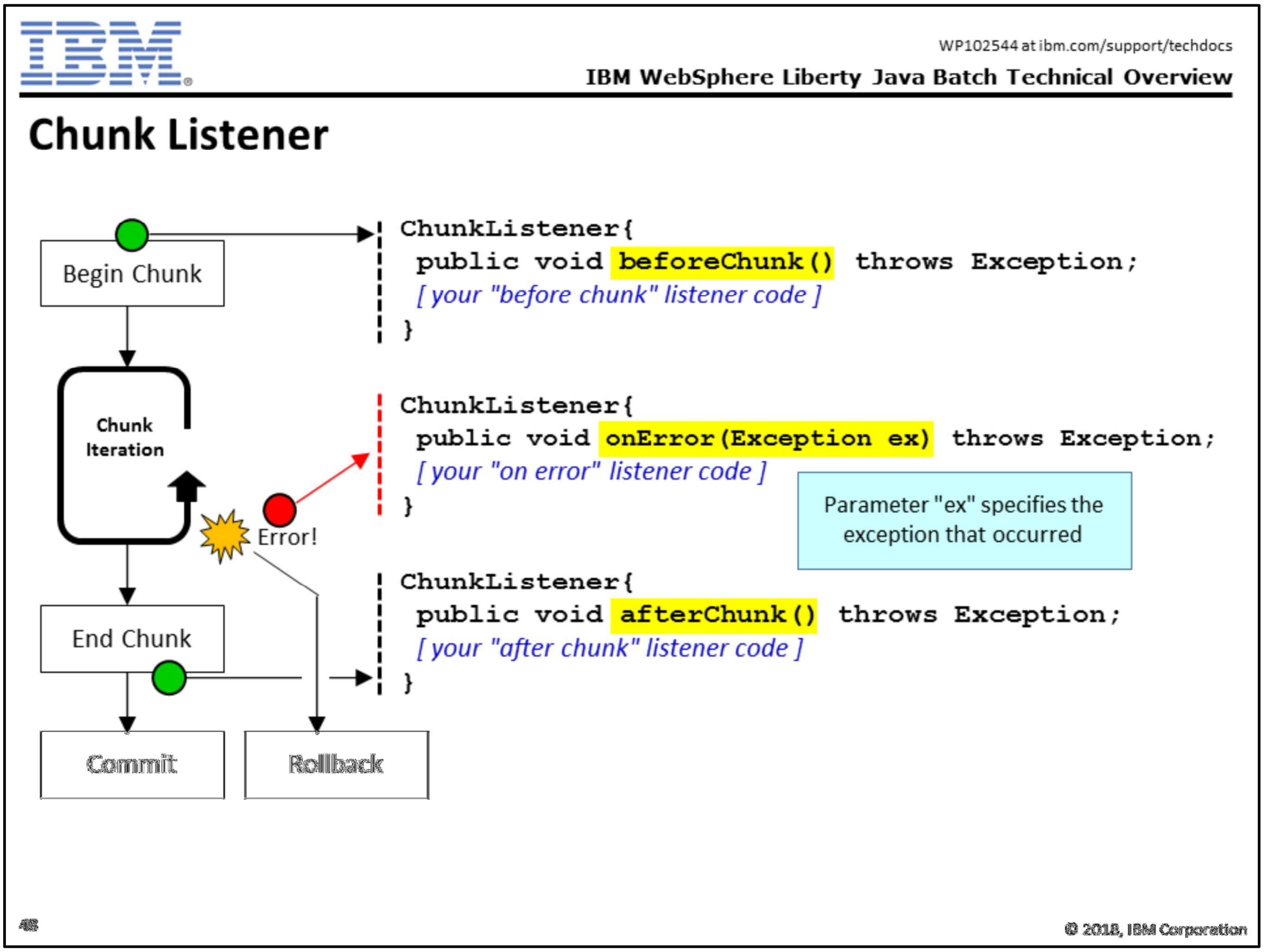
```
StepListener {
 public void afterStep() throws Exception;
 [ your "after step" listener code ]
}
```

```
JobListener {
 public void afterJob() throws Exception;
 [ your "after job" listener code ]
}
```

© 2018, IBM Corporation

Job and step listeners get control at various points during the execution of the job: at the start of the job, and the end of the job, and at the start and end of each step. You implement the listener function you desire by coding to the listener methods (as specified in the JSR-352 specification) and identifying in JSL the class that contains your implementation. Then at the start of the job the container will call beforeJob() and your code gets control; at the end of the job the container calls afterJob(). For steps the same applies, except the methods called are beforeStep() and afterStep().

These listeners are relatively simple in that they take no parameters as input, and they do not return anything.



Chunk Listener

```
ChunkListener{
 public void beforeChunk() throws Exception;
 [ your "before chunk" listener code ]
}
```

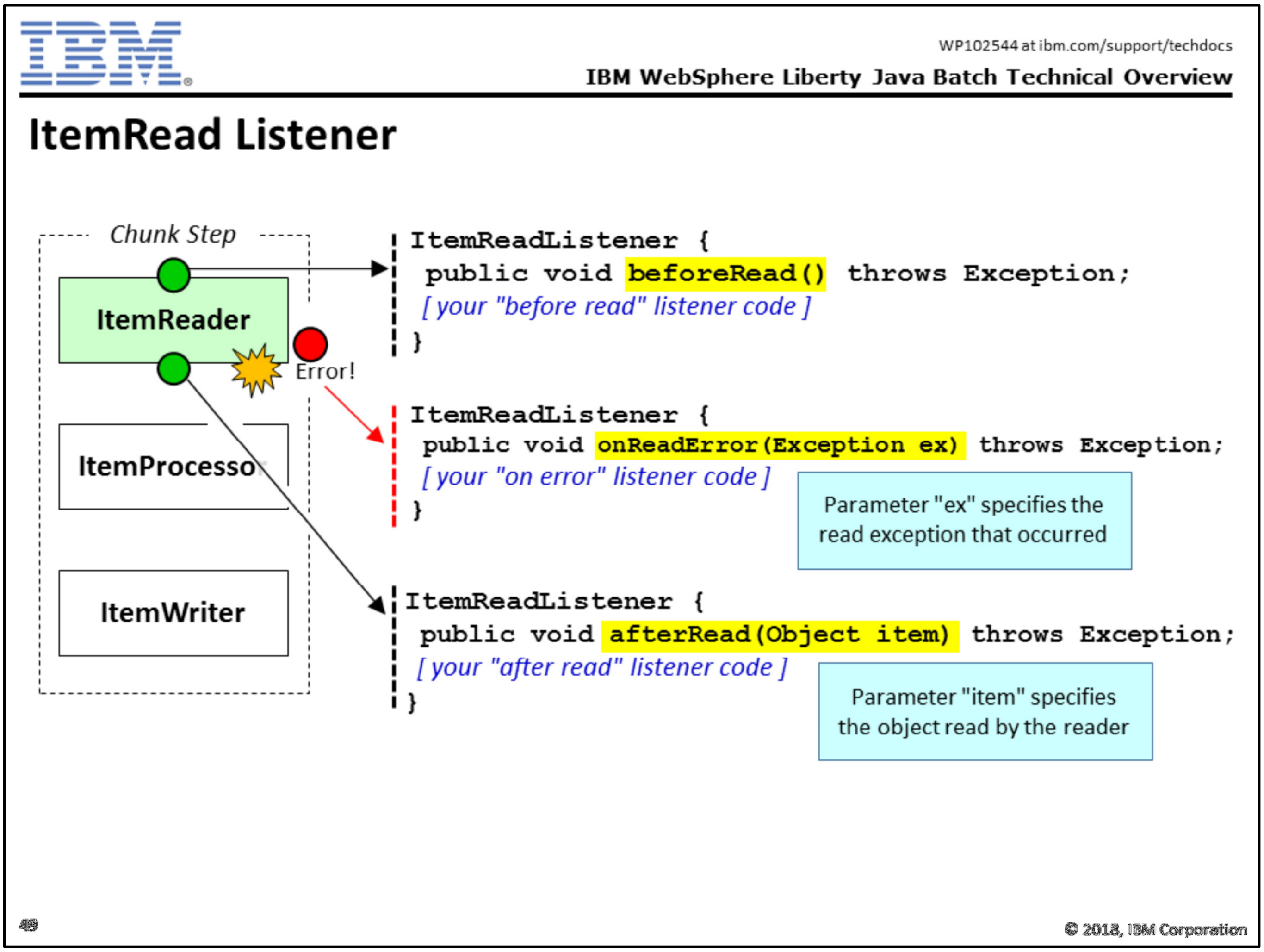
```
ChunkListener{
 public void onError(Exception ex) throws Exception;
 [ your "on error" listener code ]
}
```

Parameter "ex" specifies the exception that occurred

```
ChunkListener{
 public void afterChunk() throws Exception;
 [ your "after chunk" listener code ]
}
```

© 2018, IBM Corporation

The chunk listener has both a beforeChunk() method and an afterChunk() method. But it also has an onError() method for when an unhandled error is surfaced in the process of iterating through the chunk. When an unhandled error surfaces, the container calls the onError() method and passes the exception that was surfaced. Your onError() implementation code may then do what you wish it to do -- for example, log the exception that occurred.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

ItemRead Listener

```
ItemReadListener {
 public void beforeRead() throws Exception;
 [ your "before read" listener code ]
}
```

```
ItemReadListener {
 public void onReadError(Exception ex) throws Exception;
 [ your "on error" listener code ]
}
```

Parameter "ex" specifies the read exception that occurred

```
ItemReadListener {
 public void afterRead(Object item) throws Exception;
 [ your "after read" listener code ]
}
```

Parameter "item" specifies the object read by the reader

49

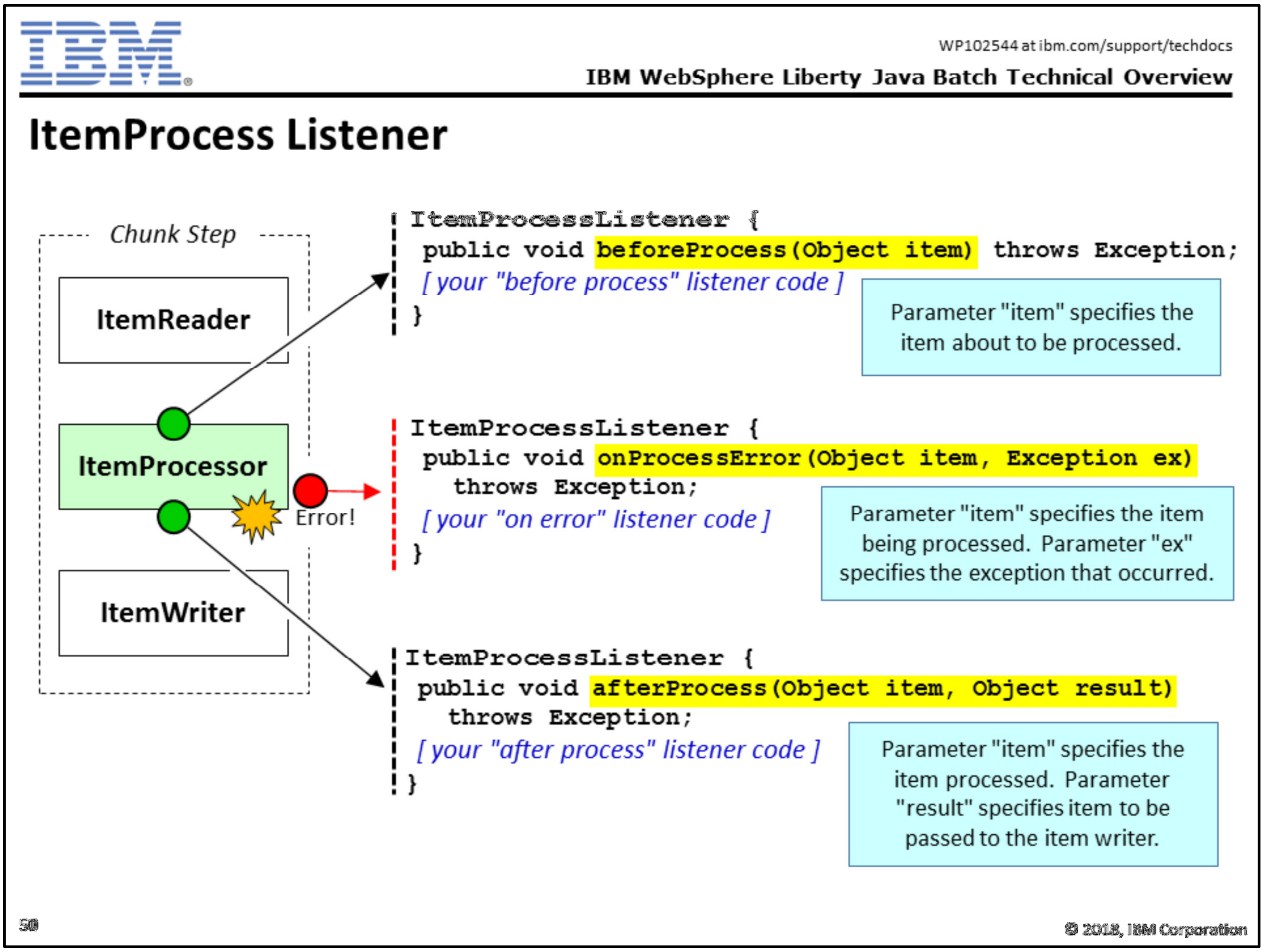
© 2018, IBM Corporation

Within chunk processing we have the ItemReader, the ItemProcessor, and the ItemWriter. Each artifact has its own set of listeners. On this chart we look at the ItemRead listeners. There are three: beforeRead(), afterRead(), and onReadError().

The beforeRead() method is relatively simple: it is called at the start of each ItemReader operation. The listener has no input parameters, and passes nothing back.

The afterRead() method takes as a parameter the item that was read by the ItemReader. That is passed in as an object, and the listener may then do what it is designed to do with that information.

The onReadError() method is called by the container when an unhandled exception surfaces from the ItemReader. The onReadError() method is called, and the exception that is surfaced is passed to the method.



ItemProcess Listener

```
ItemProcessListener {
 public void beforeProcess(Object item) throws Exception;
 [ your "before process" listener code ]
}
```

Parameter "item" specifies the item about to be processed.

```
ItemProcessListener {
 public void onProcessError(Object item, Exception ex)
   throws Exception;
 [ your "on error" listener code ]
}
```

Parameter "item" specifies the item being processed. Parameter "ex" specifies the exception that occurred.

```
ItemProcessListener {
 public void afterProcess(Object item, Object result)
   throws Exception;
 [ your "after process" listener code ]
}
```

Parameter "item" specifies the item processed. Parameter "result" specifies item to be passed to the item writer.

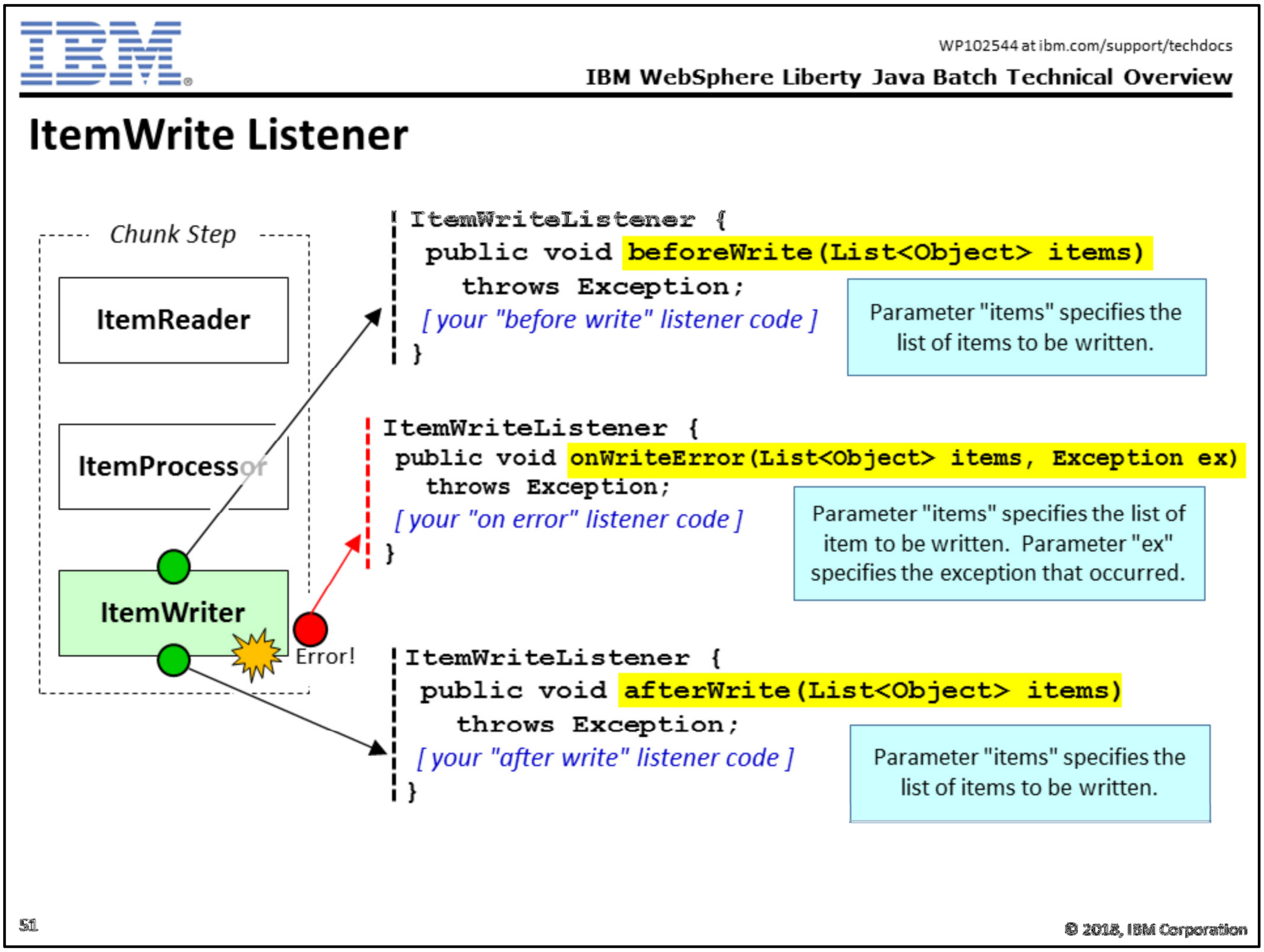
© 2018, IBM Corporation

The ItemProcessor has three listeners -- beforeProcess(), afterProcess(), and onProcessError().

The beforeProcess() method is called at the start of the ItemProcessor operation. It takes as input the item object that was read in by the ItemReader.

The afterProcess() method receives two parameters as input: the item object that was read by the ItemReader (and processed by the ItemProcessor), and the result object from the ItemProcessor.

The onProcessError() method is called by the container when an unhandled exception surfaces from the ItemProcessor. Two parameters are passed to this method: the object item that was passed to the processor from the ItemReader, and the exception that was surfaced.



ItemWrite Listener

```
ItemWriteListener {
  public void beforeWrite(List<Object> items)
     throws Exception;
  [ your "before write" listener code ]
}
```

Parameter "items" specifies the list of items to be written.

```
ItemWriteListener {
 public void onWriteError(List<Object> items, Exception ex)
   throws Exception;
 [ your "on error" listener code ]
}
```

Parameter "items" specifies the list of item to be written. Parameter "ex" specifies the exception that occurred.

```
ItemWriteListener {
 public void afterWrite(List<Object> items)
    throws Exception;
 [ your "after write" listener code ]
}
```

Parameter "items" specifies the list of items to be written.

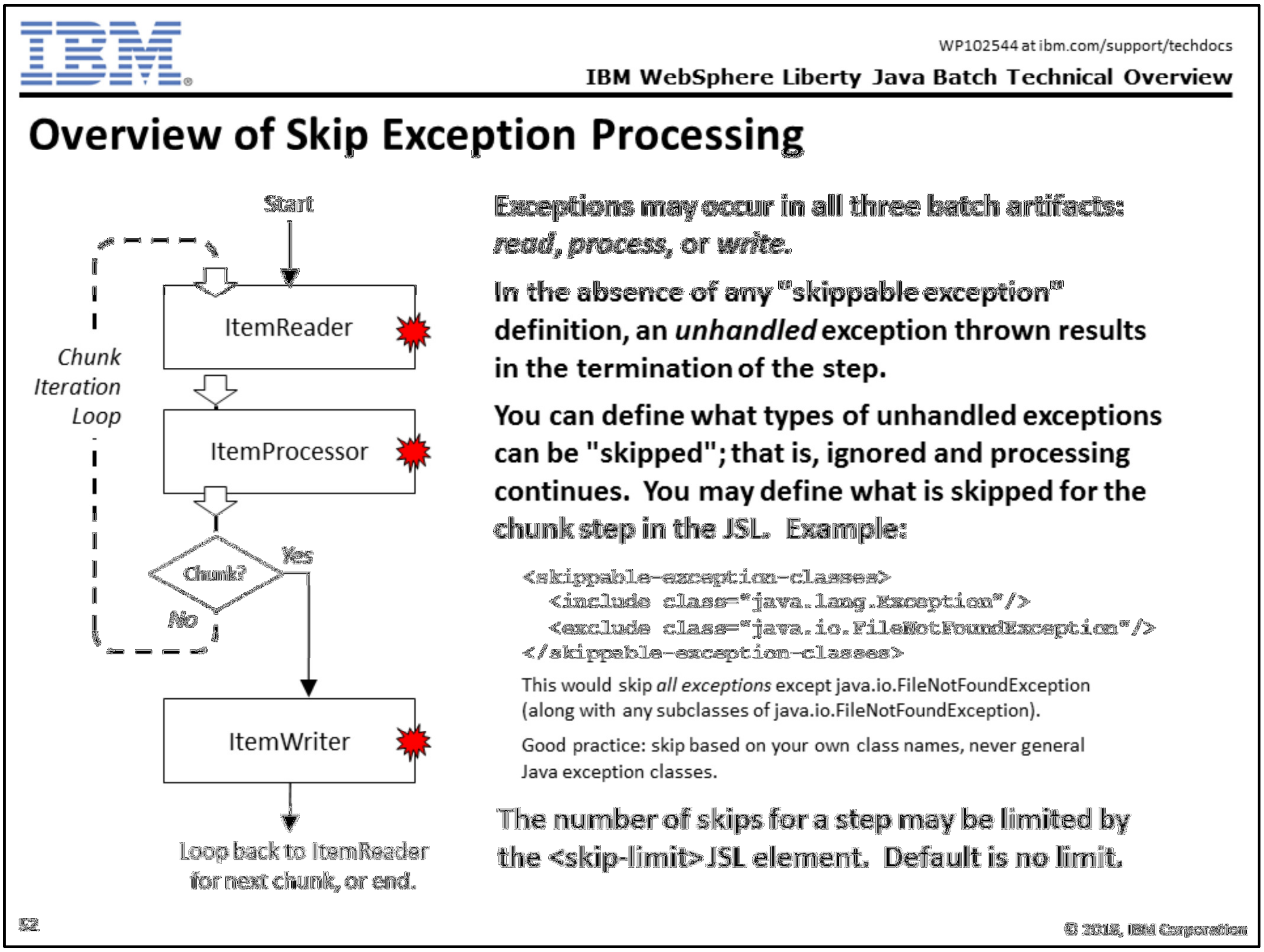
© 2018, IBM Corporation

The ItemWriter has three listeners -- beforeWrite(), afterWrite(), and onWriteError().

When the container calls beforeWrite() it passes in the list of objects that was created by the reader/processor chunk iteration. This is the list of items to be written by the ItemWriter. The listener gets this list of items so your listener code may then do what you wish this listener to do; for example, log the items.

When the container calls afterWrite() it also passes in the list of items that ItemWriter received.

The onWriteError() method is called when an unhandled exception surfaces during the ItemWriter processing. The listener takes as input two things: the list of items the ItemWriter was passed, and the exception that was thrown.



Overview of Skip Exception Processing

Start
ItemReader
ItemProcessor
Chunk?
Yes
No
Chunk
Iteration
Loop
ItemWriter
Loop back to ItemReader
for next chunk, or end.

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.

52

© 2015, IBM Corporation

We're going to next cover skip and retry listeners, but before doing that let's do a brief review of how skip processing is done. Then we'll cover the listeners for skip processing. Then we'll review retry processing, and we'll finish up with the retry listeners.

In a chunk loop, the three artifacts present (reader, processors, and writer) may experience exceptions that you wish to acknowledge but keep processing. The JSR-352 specification allows for defining what exceptions are "skippable," and also how many skipped exceptions may be tolerated before the steps is failed.

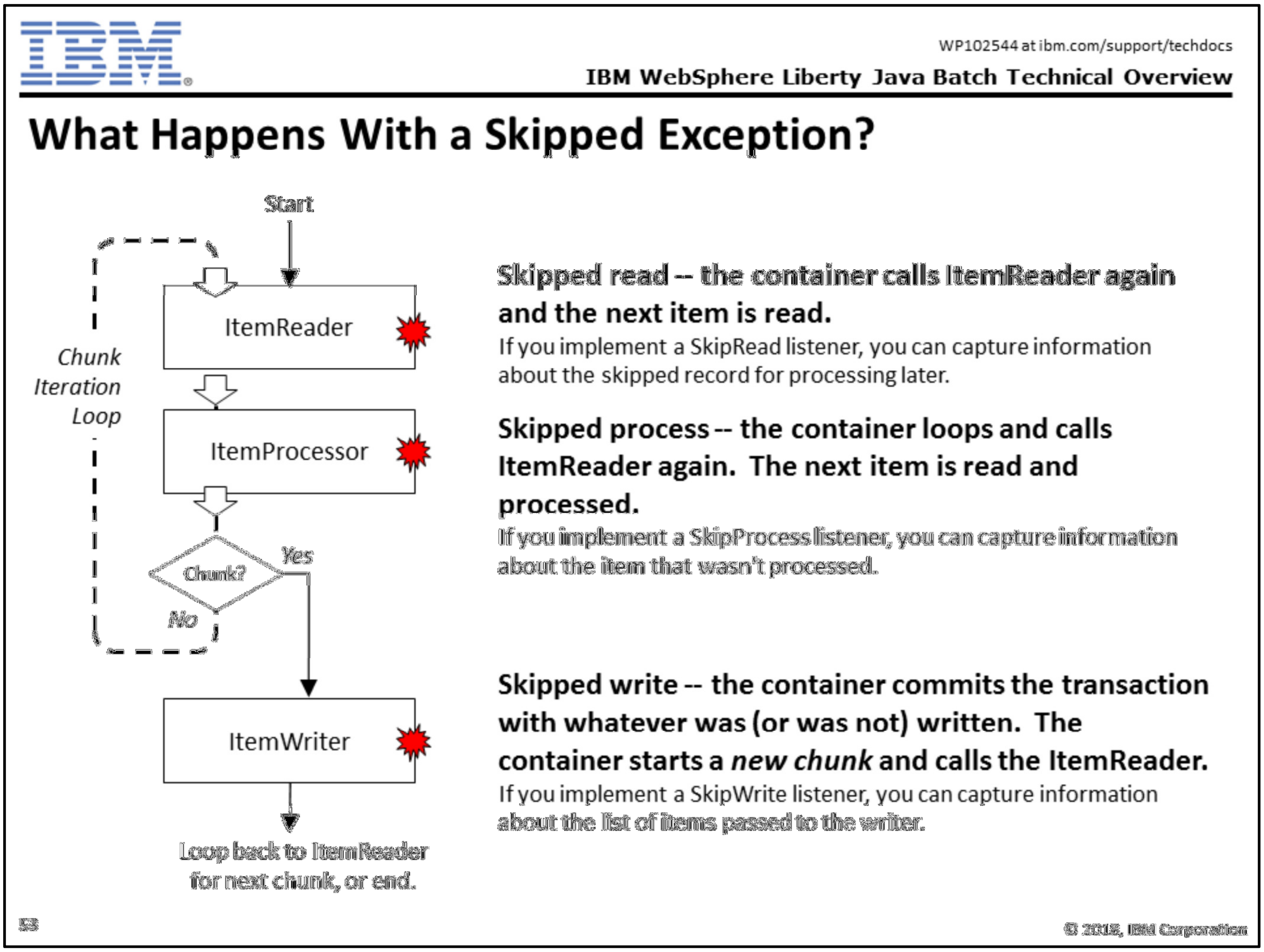
Note: the skip processing only applies to exceptions that are defined as skippable. Unhandled exceptions without a skippable definition will result in the termination of the step.

The definition of what is considered "skippable" is done in the JSL. This is defined at the <step> level. An exception is skipped (that is, ignored) if it is defined on an <include> element. An exception is not skipped if it is defined on an <exclude> element. So in the example on the chart, all exceptions that match java.lang.Exception (which is all exceptions) would be skipped, except for java.io.FileNotFoundException (and subclasses of that) because it's defined on an <exclude>.

That example is taken straight from the JSR-352 specification, and is (perhaps) not a very good example because in general, skipping on general Java exceptions is not a good practice. It is better for your code to handle exceptions, and then skip based on your own class names. Skipping on general Java exceptions may result in your batch job skipping things you didn't intend to skip. In other words, be deliberate about what you're skipping.

The <skip-limit> definition in JSL can be used to protect against excessive skips. The default is no limit, so if you're going to use exception skipping, you may want to consider limiting it to some reasonable number of skips before you determine some bigger problem is occurring and failing the step.

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.

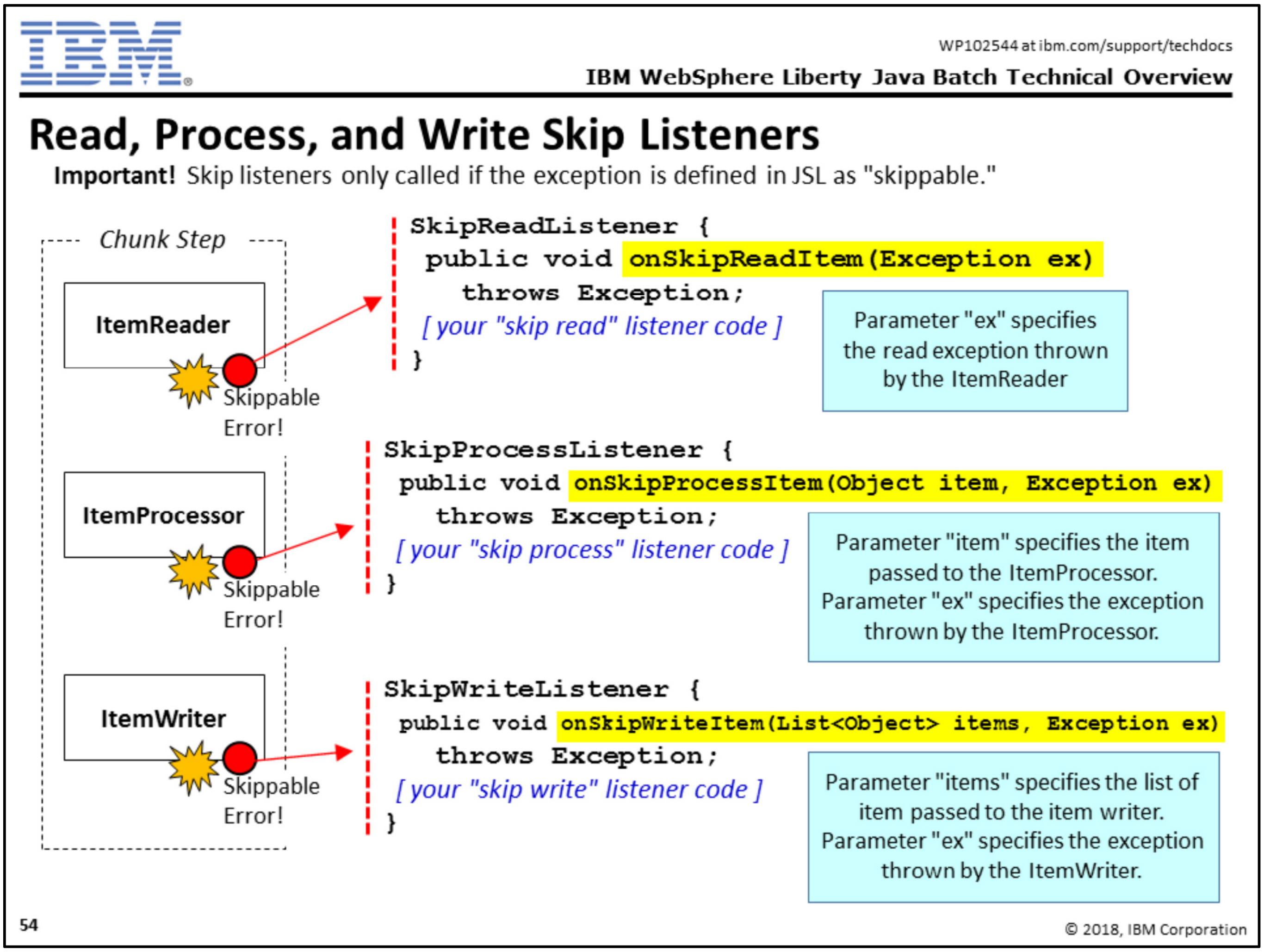
© 2015, IBM Corporation

The next question that comes up is this -- what happens when an exception is skipped? That depends on where the exception that was skipped occurred.

When an exception is thrown by the ItemReader, and a skippable definition applies, then the container simply calls ItemReader again and the next item is read. In this case you would very likely want to implement a SkipRead listener so you can capture details about the item that was skipped. We'll cover the details of the skip listeners in the upcoming charts.

When an exception is thrown by the ItemProcessor, and a skippable definition applies, then the container loops and calls the ItemReader again, which results in the *next* item being read. That means the previous item read by the ItemReader is never processed or written. So again, implementing a SkipProcess listener would be a good practice to capture information about the item whose processing was skipped.

Finally, for ItemWriter a skipped exception is a bit more involved. The container will commit the current transaction with the data written to that point. Then it starts a new chunk and calls the ItemReader. That may mean that some data was *not* written out. The SkipWrite listener can be used to capture information about the list of items the writer was working on. You can use that to analyze what was committed and what was not.



Read, Process, and Write Skip Listeners

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

```
SkipReadListener {
 public void onSkipReadItem(Exception ex)
    throws Exception;
 [ your "skip read" listener code ]
}
```

Parameter "ex" specifies the read exception thrown by the ItemReader

```
SkipProcessListener {
 public void onSkipProcessItem(Object item, Exception ex)
    throws Exception;
 [ your "skip process" listener code ]
}
```

Parameter "item" specifies the item passed to the ItemProcessor. Parameter "ex" specifies the exception thrown by the ItemProcessor.

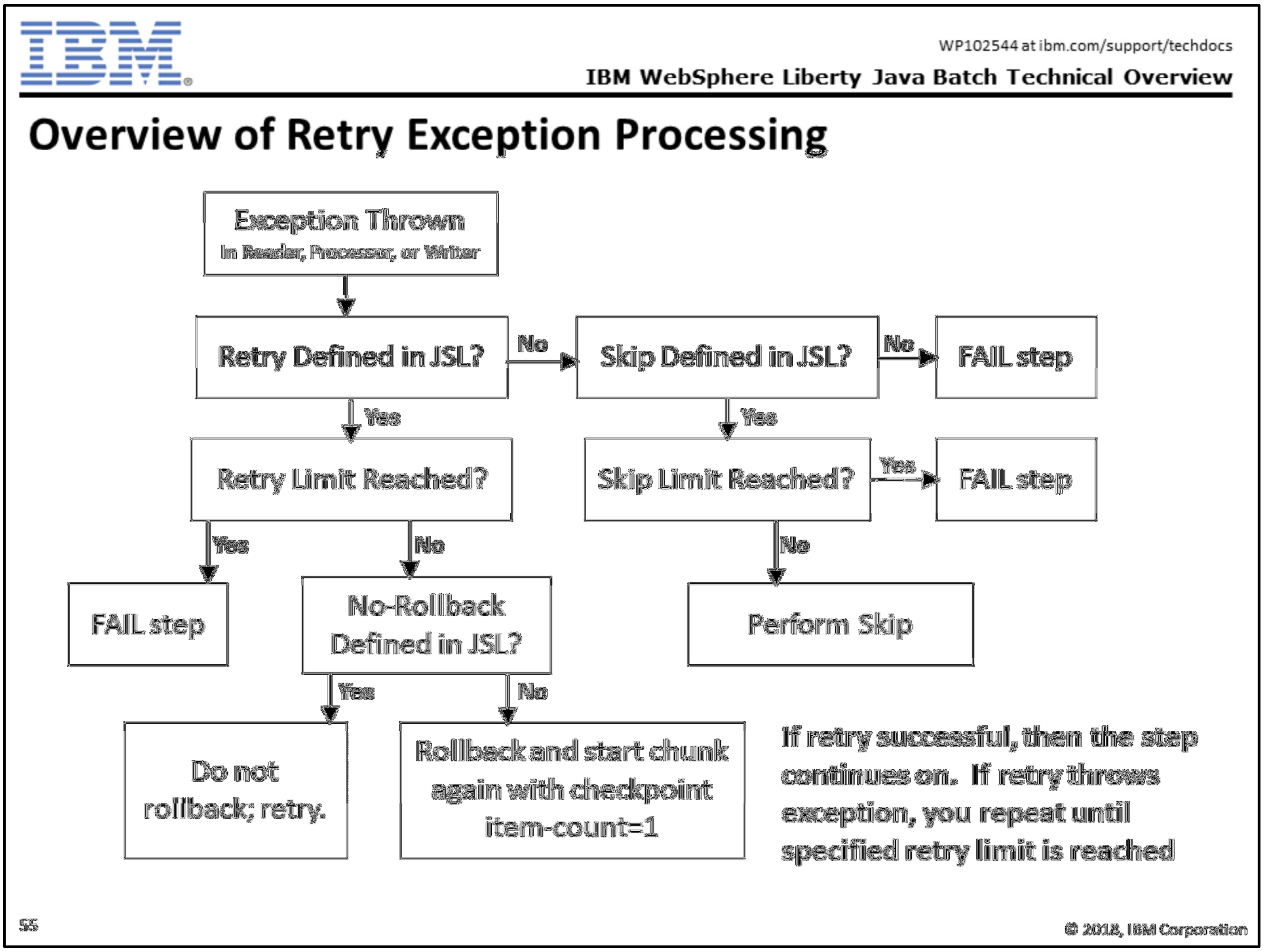
```
SkipWriteListener {
 public void onSkipWriteItem(List<Object> items, Exception ex)
    throws Exception;
 [ your "skip write" listener code ]
}
```

Parameter "items" specifies the list of item passed to the item writer. Parameter "ex" specifies the exception thrown by the ItemWriter.

© 2018, IBM Corporation

This chart provides the details on the read, process, and write skip listeners.

The onSkipReadItem() method receives as input the exception that was thrown when the skip definition took effect. The onSkipProcessItem() method receives as input the item object that was passed over from the ItemReader, as well as the exception that was thrown when the skip definition took effect. Finally, the onSkipWriteItem() method receives the list of items passed to it from the read/process loop, as well as the exception that was thrown.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

Overview of Retry Exception Processing

Exception Thrown
In Reader, Processor, or Writer

Retry Defined in JSL?

No

Skip Defined in JSL?

No

FAIL step

Yes

Yes

Retry Limit Reached?

Skip Limit Reached?

Yes

FAIL step

Yes

No

No

FAIL step

No-Rollback Defined in JSL?

Perform Skip

Yes

No

Do not rollback; retry.

Rollback and start chunk again with checkpoint item-count=1

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

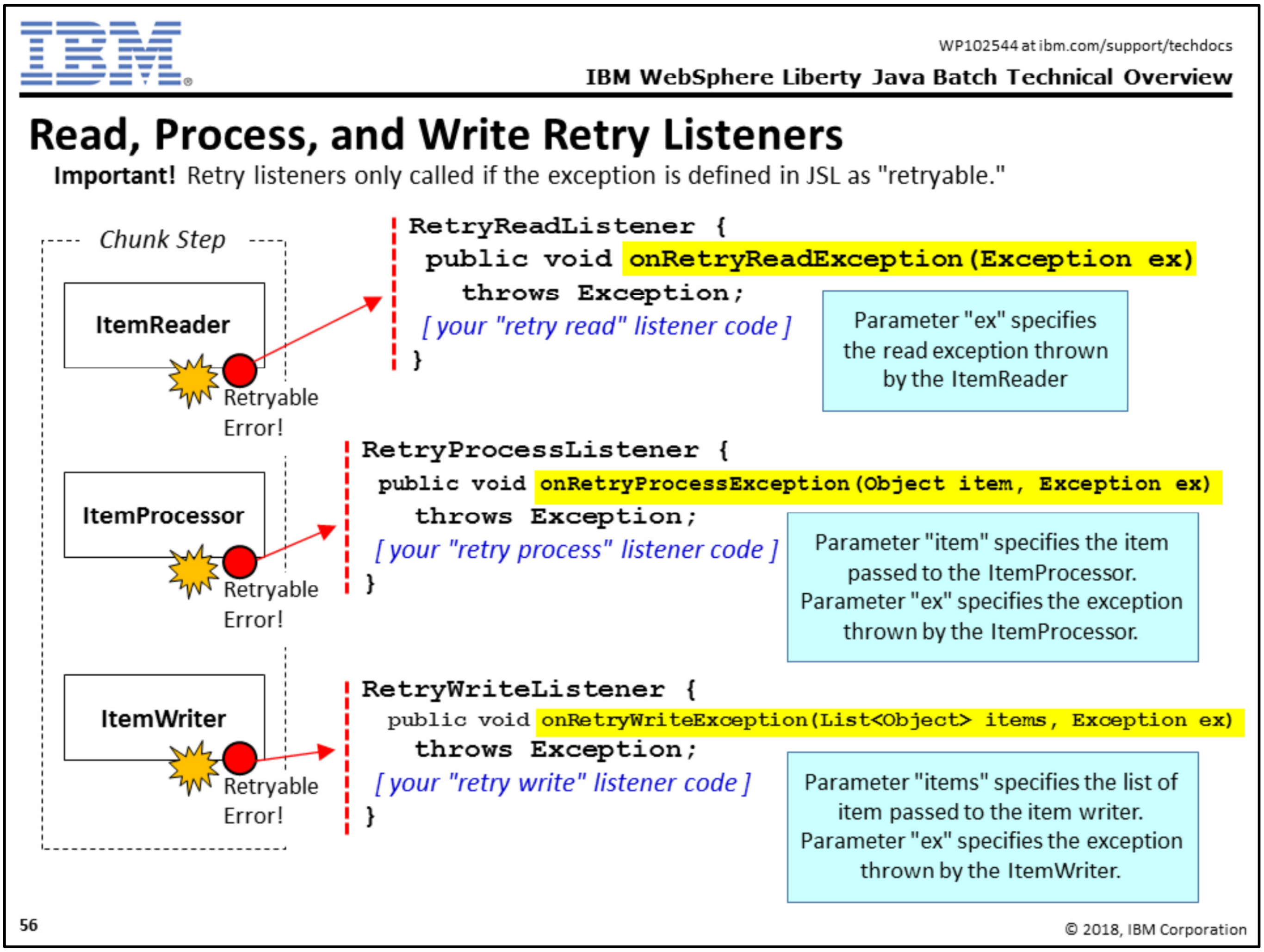
55

© 2018, IBM Corporation

The JSR-352 specification also defines the ability to retry processing when an unhandled exception surfaces from the reader, processor, or writer. What takes place depends on a number of things, so the flowchart shown above is an attempt to take you through the processing:

- Starting at the top ... assume an unhandled exception is thrown in either the reader, processor, or writer.
- The next questions is whether there is a retry definition in the JSL for the step. The retry definition is very similar to the skip definition, but the syntax is for retry, not skip. If yes, then we flow down to the retry limit question; if no, then there's no retry attempted and we flow right to the issue of whether skip is defined.
- If retry is defined and the retry limit has been reached, then we fail the step. But if the limit has not been reached, then it checks to see if a "no-rollback" definition is in the JSL. If yes (meaning: no rollback) then the container simply retries the operation that threw the exception. If no (meaning no rollback is not defined, therefore rollback), then the container rolls back the transaction and starts the chunk again, but this time with a checkpoint of item-count=1. The purpose of this is to incrementally approach the point where the exception occurred, and commit as much as possible in the event the same exception is thrown again.
- Moving back up ... no retry defined, so is skip defined? If not, then the exception results in the step being failed. But if a skip is defined, then it drops down and asks whether the skip limit has been reached. If yes, then the step is failed. But if not, then the skip operation is performed.

The retry processing is designed to overcome transient problems where the possibility exists that a retry will result in success. The retry-limit is there to protect you against a case where the problem is not transient. Repeated retries without success and no retry limit could potentially go on forever. So if you use retry processing (and skip, for that matter), then limiting the retry attempts is a good idea.



Read, Process, and Write Retry Listeners

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

```
RetryReadListener {
 public void onRetryReadException(Exception ex)
    throws Exception;
 [ your "retry read" listener code ]
}
```

Parameter "ex" specifies the read exception thrown by the ItemReader

```
RetryProcessListener {
 public void onRetryProcessException(Object item, Exception ex)
    throws Exception;
 [ your "retry process" listener code ]
}
```

Parameter "item" specifies the item passed to the ItemProcessor. Parameter "ex" specifies the exception thrown by the ItemProcessor.

```
RetryWriteListener {
 public void onRetryWriteException(List<Object> items, Exception ex)
    throws Exception;
 [ your "retry write" listener code ]
}
```

Parameter "items" specifies the list of item passed to the item writer. Parameter "ex" specifies the exception thrown by the ItemWriter.

© 2018, IBM Corporation

Here we illustrate the three retry listeners, one for ItemReader, one for ItemProcessor, and one for ItemWriter. The pattern is similar to what we've seen earlier: when an unhandled exception surfaces and the exception is defined as one that is retryable, then the container will call the listener.

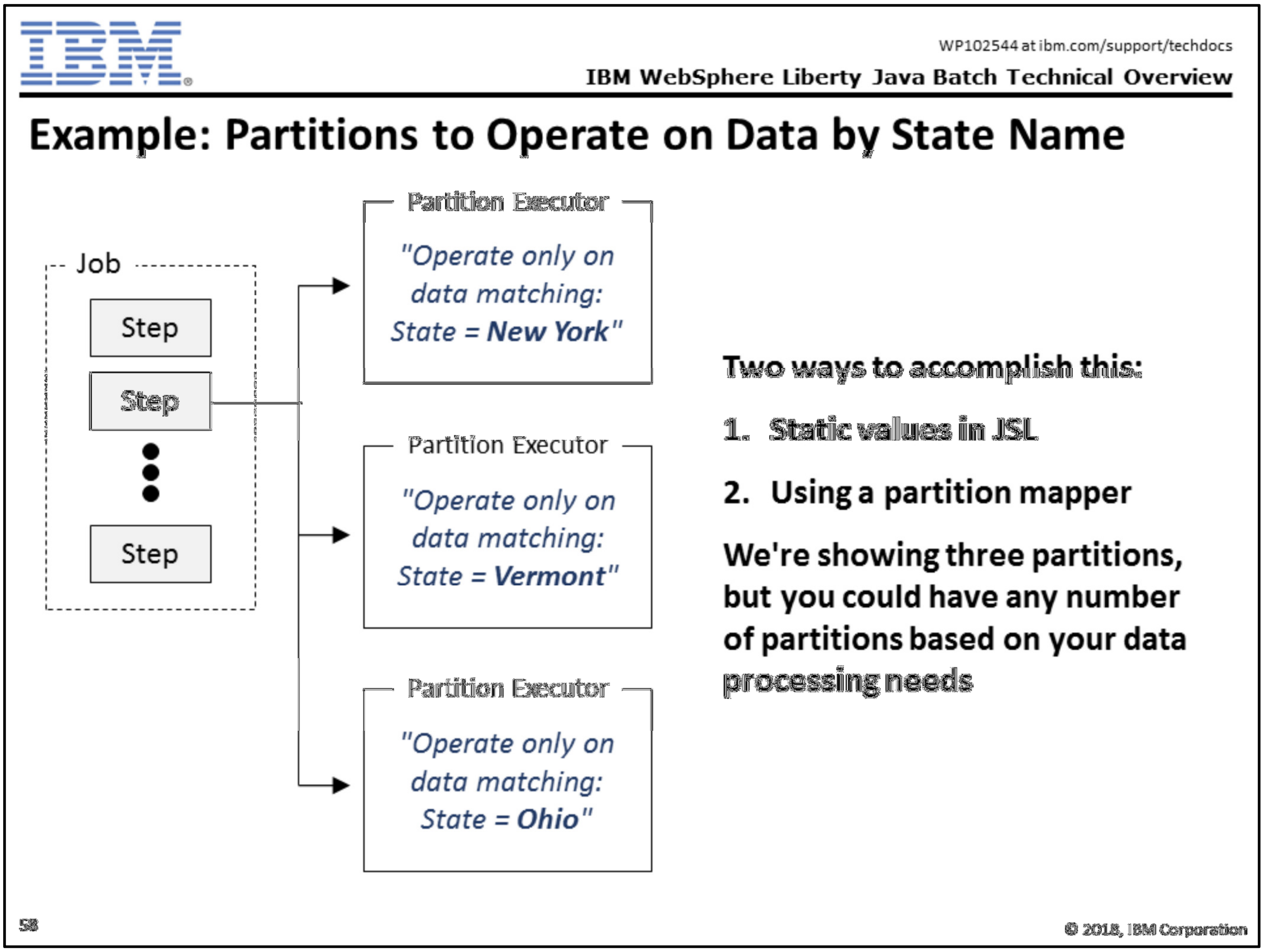
onRetryReadException() receives as input the retryable exception that was thrown.

onRetryPRocessException() receives as input the item that was passed to the processor and was being processed at the time of the exception, as well as the exception that was thrown.

onRetryWriteException() receives as input the list of items that was passed to the writer and was being processed at the time of the exception, as well as the exception that was thrown.

Partitioning

© 2018, IBM Corporation



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

Example: Partitions to Operate on Data by State Name

Job

Step

Step

Step

Partition Executor

*"Operate only on data matching: State = **New York**"*

Partition Executor

*"Operate only on data matching: State = **Vermont**"*

Partition Executor

*"Operate only on data matching: State = **Ohio**"*

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

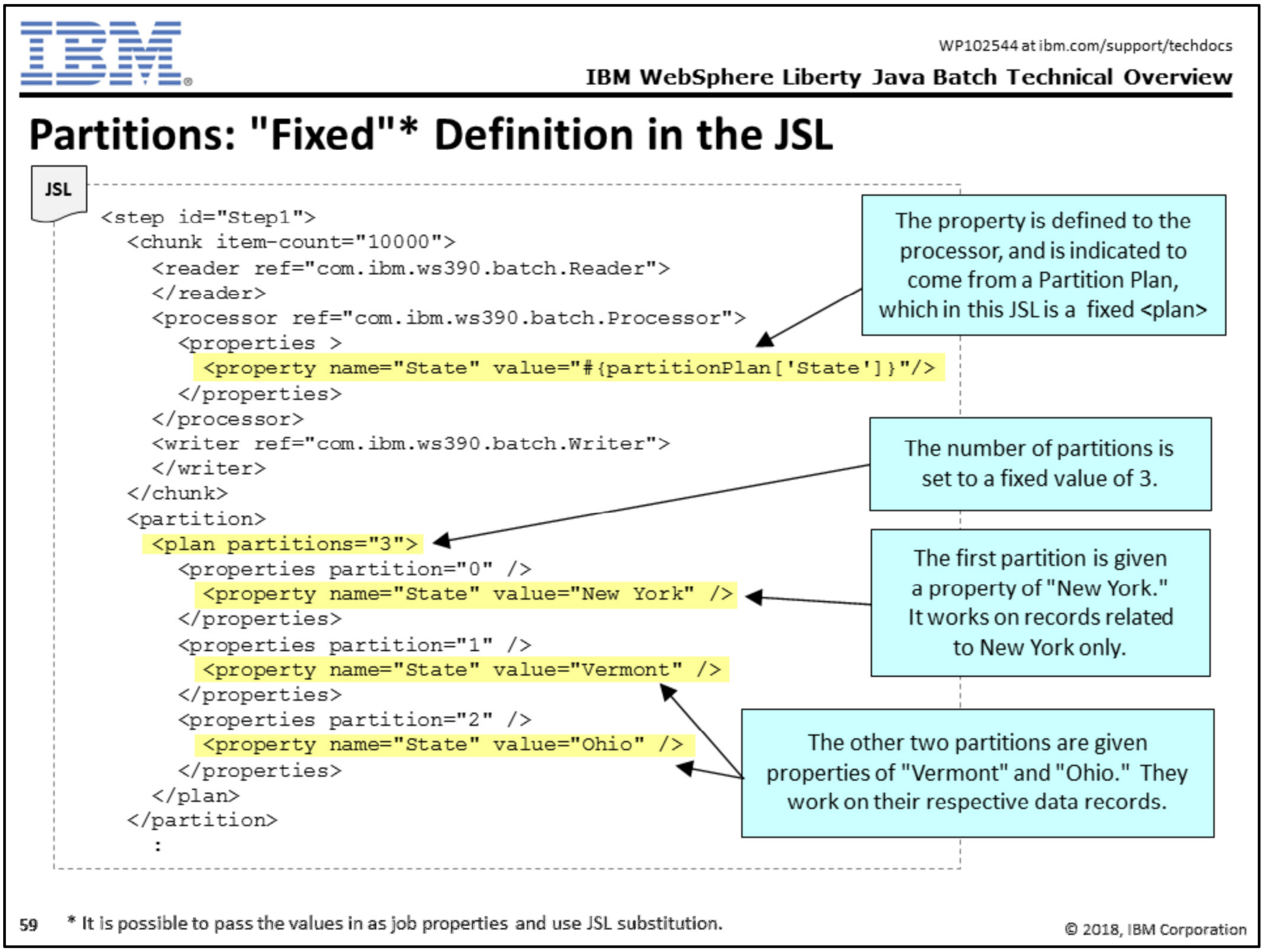
58

© 2018, IBM Corporation

To set the stage for the discussion on partitioning, we set up an example where we want three partitions; the first partition only acts up on data records related to the state of New York, the second acts only on data records from Vermont, and the third partition only acts upon data records from Ohio.

The question is how you can pass to the partitions the information about which state's data records they are to act upon. This can be done using either static values in the JSL, or by using the partition mapper and then programmatically pass the values to each partition.

For simplicity we're showing three partitions, but it could just as easily be five, or twenty, or whatever number makes sense for your application.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

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.

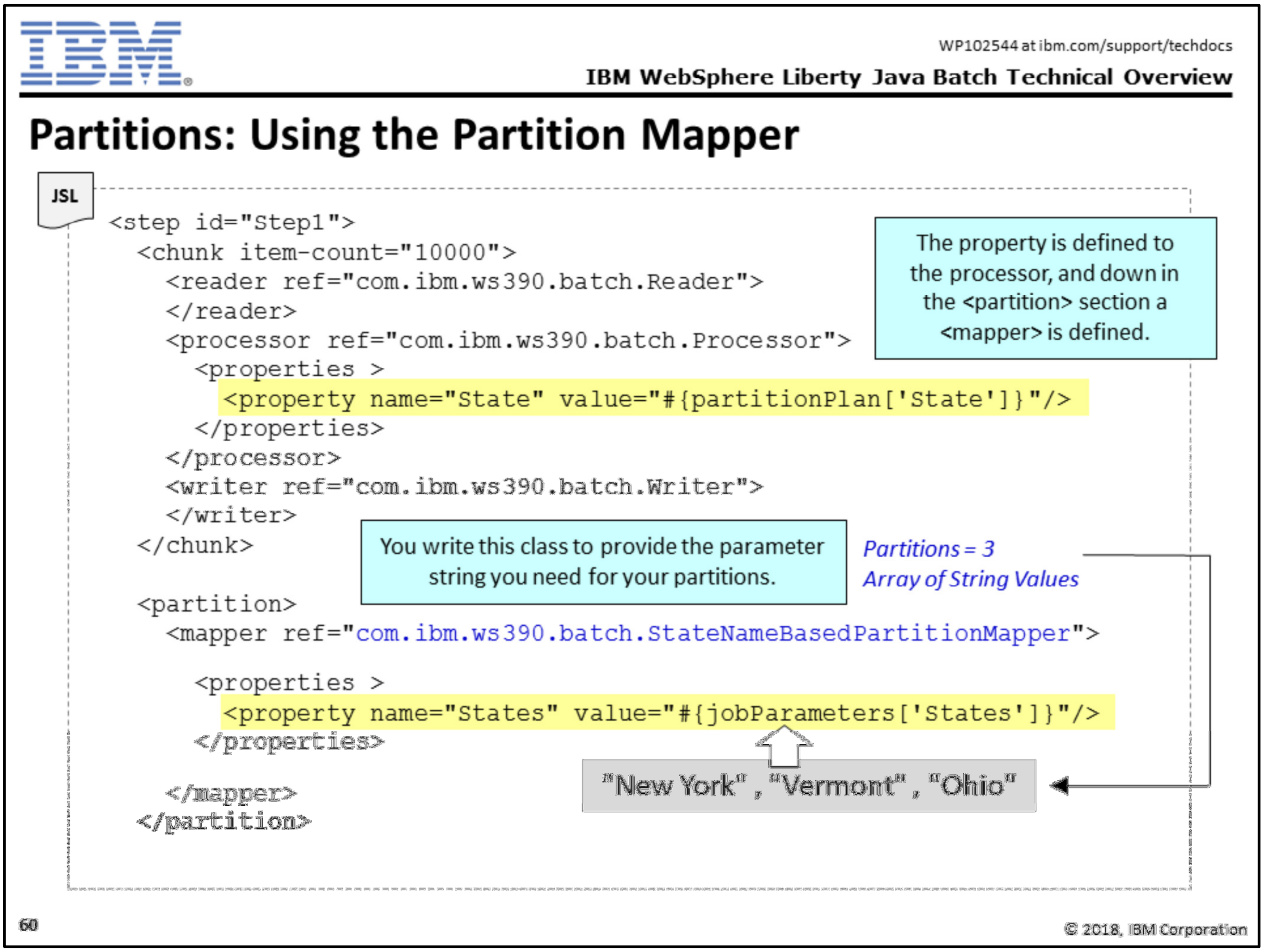
The other two partitions are given properties of "Vermont" and "Ohio." They work on their respective data records.

59 * It is possible to pass the values in as job properties and use JSL substitution.

© 2018, IBM Corporation

This chart illustrates how the state values (in our example) can be passed to the partition using "fixed" definitions in the JSL. Fixed is in quotes because, as the footnote says, the values can be passed into the JSL as job properties. That would create JSL where "New York," "Vermont," and "Ohio" don't appear in the JSL itself, but using substitution you can pass them in at time of job submission and achieve the same effect as having them hard-coded as shown here.

Down in the <partition> section you can specify a <plan> and name the number of partitions, and properties for each. In this example, the first partition (partition 0) is assigned the state name "New York." The second partition is assigned "Vermont," and the third partition is assigned "Ohio."



Partitions: Using the Partition Mapper

```
<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.

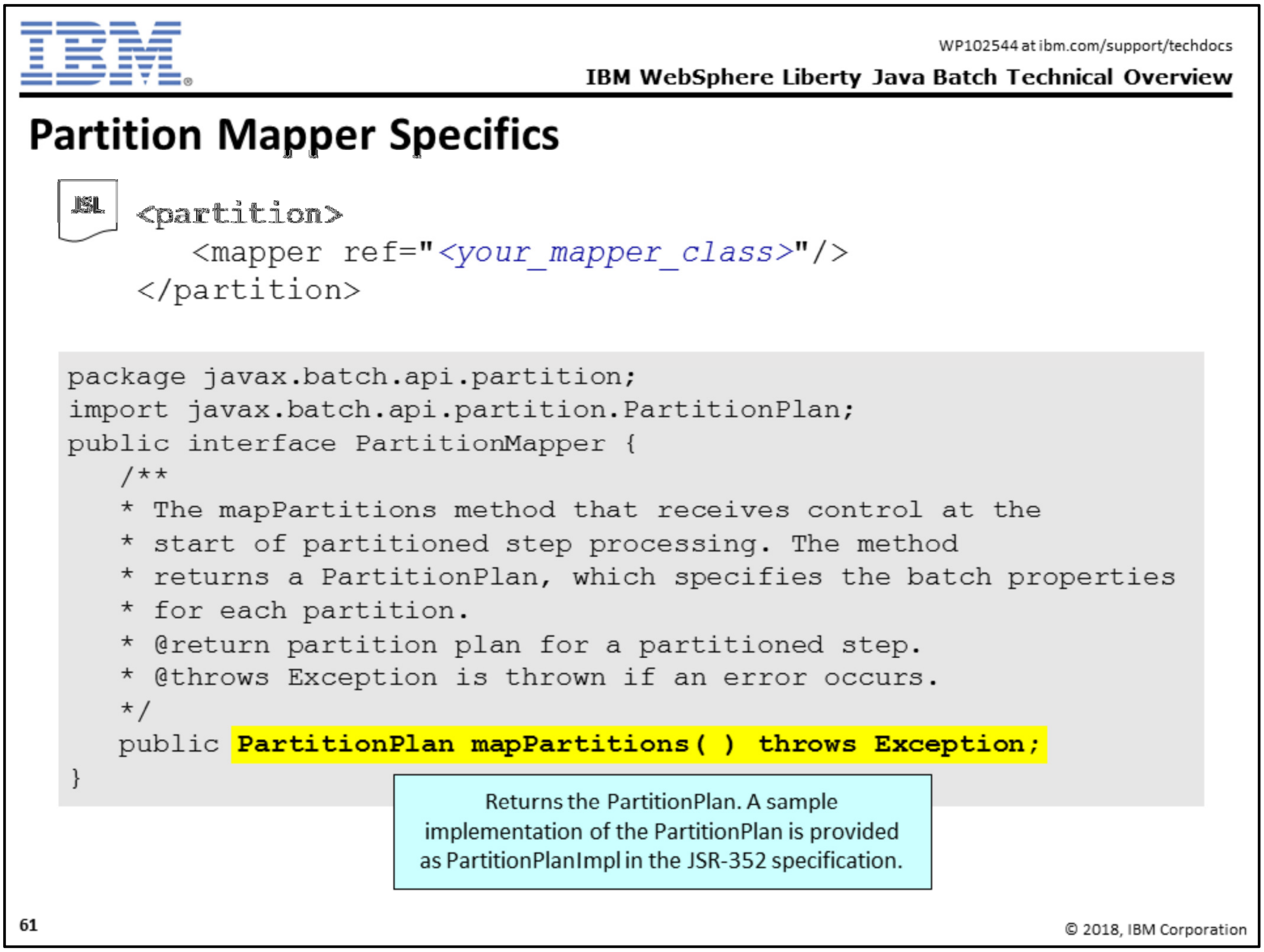
You write this class to provide the parameter string you need for your partitions.

Partitions = 3
Array of String Values

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

© 2018, IBM Corporation

The alternative is to use the partition mapper, which is code you write that produces a set of parameters that is used to determine the partition operations. In the <processor> section the property is defined just as it was with the fixed example on the previous chart, but further down in the JSL the <partition> section includes a <mapper> definition. The mapper is software you implement that is called by the container before the partitions are created. The mapper returns a set of parameters to be used for the partitions, including the number of partitions and the property value (the state name) to be used by each partition.



Partition Mapper Specifics

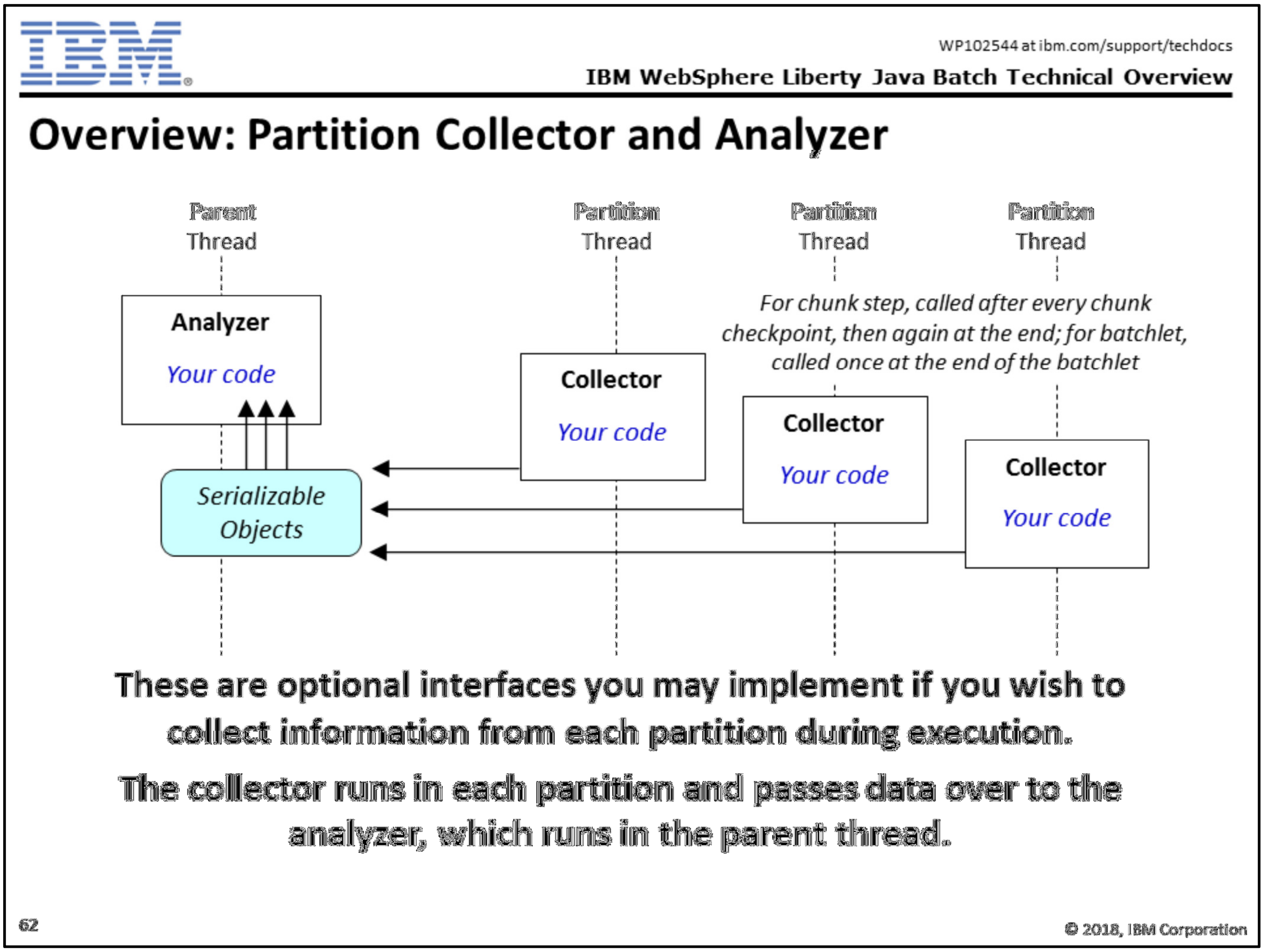
```
<partition>
   <mapper ref="<your_mapper_class>"/>
</partition>
```

```
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.

© 2018, IBM Corporation

The partition mapper is specified in the JSL using the <mapper> element. This points to the Java class that implements your mapper. A sample implementation of the PartitionPlan is provided as PartitionPlanImpl in the JSR-352 specification.



Overview: Partition Collector and Analyzer

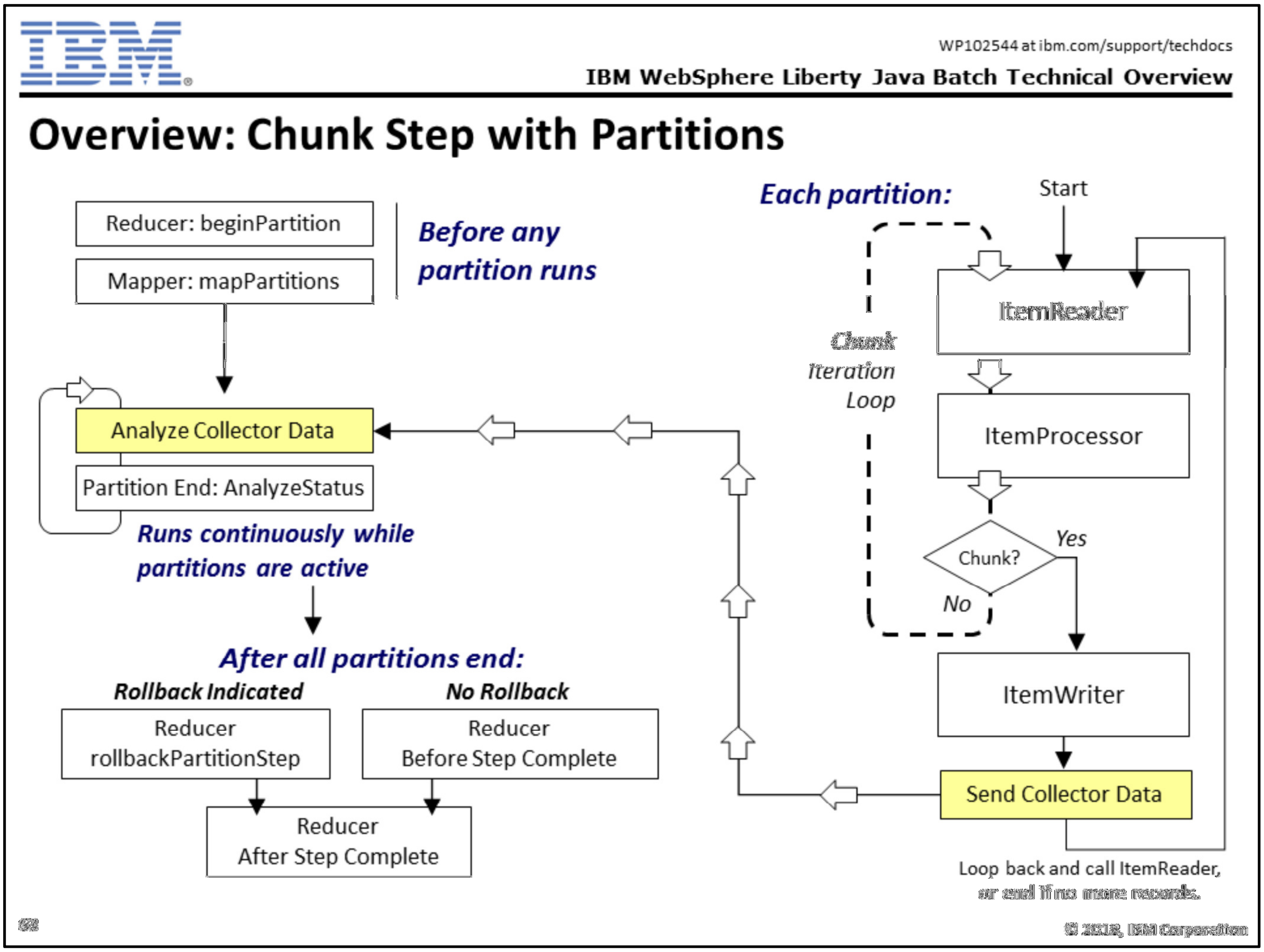
For chunk step, called after every chunk checkpoint, then again at the end; for batchlet, called once at the end of the batchlet

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.

© 2018, IBM Corporation

Two more interfaces are introduced here: the partition collector, and the partition analyzer. Collectors run in each partition. For chunk steps they are called after every chunk checkpoint, and for batchlets they are called at the end of the batchlet. They send the information collected over to the analyzer, which runs on the parent thread.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

Overview: Chunk Step with Partitions

Reducer: beginPartition
Mapper: mapPartitions
Before any partition runs
Analyze Collector Data
Partition End: AnalyzeStatus
Runs continuously while partitions are active
After all partitions end:
Rollback Indicated
No Rollback
Reducer rollbackPartitionStep
Reducer Before Step Complete
Reducer After Step Complete
Each partition:
Start
Chunk Iteration Loop
ItemReader
ItemProcessor
Chunk?
Yes
No
ItemWriter
Send Collector Data

Loop back and call ItemReader, or end if no more records.

62

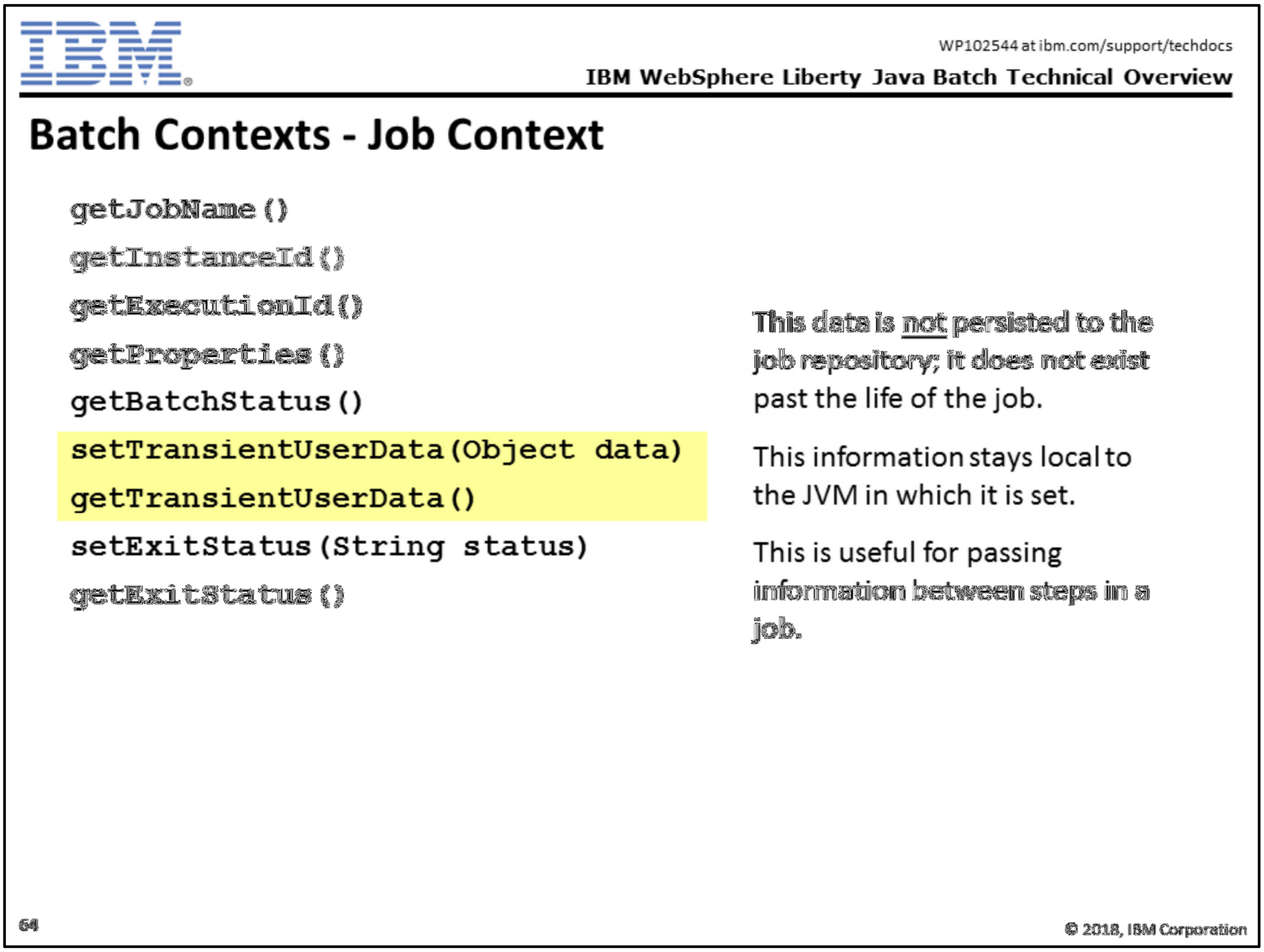
© 2018, IBM Corporation

On this chart we're bringing together the collector and analyzer, as well as another function called the partition reducer. The JSR spec says this about all three:

> 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 reducer gets control before any partition runs, then again after all partitions end. While the partitions are running, the collectors in each partition are getting control after every chunk checkpoint (for batchlets at the end of the batchlet) and they send data back to the parent thread where the analyzer receives and analyzes the data. The analyzer runs continuously while the partitions are active.

When all the partitions end, then the reducer gets control again. What it does depends on whether rollback is indicated or not. Finally, the reducer gets control one final time after the step completes.



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.

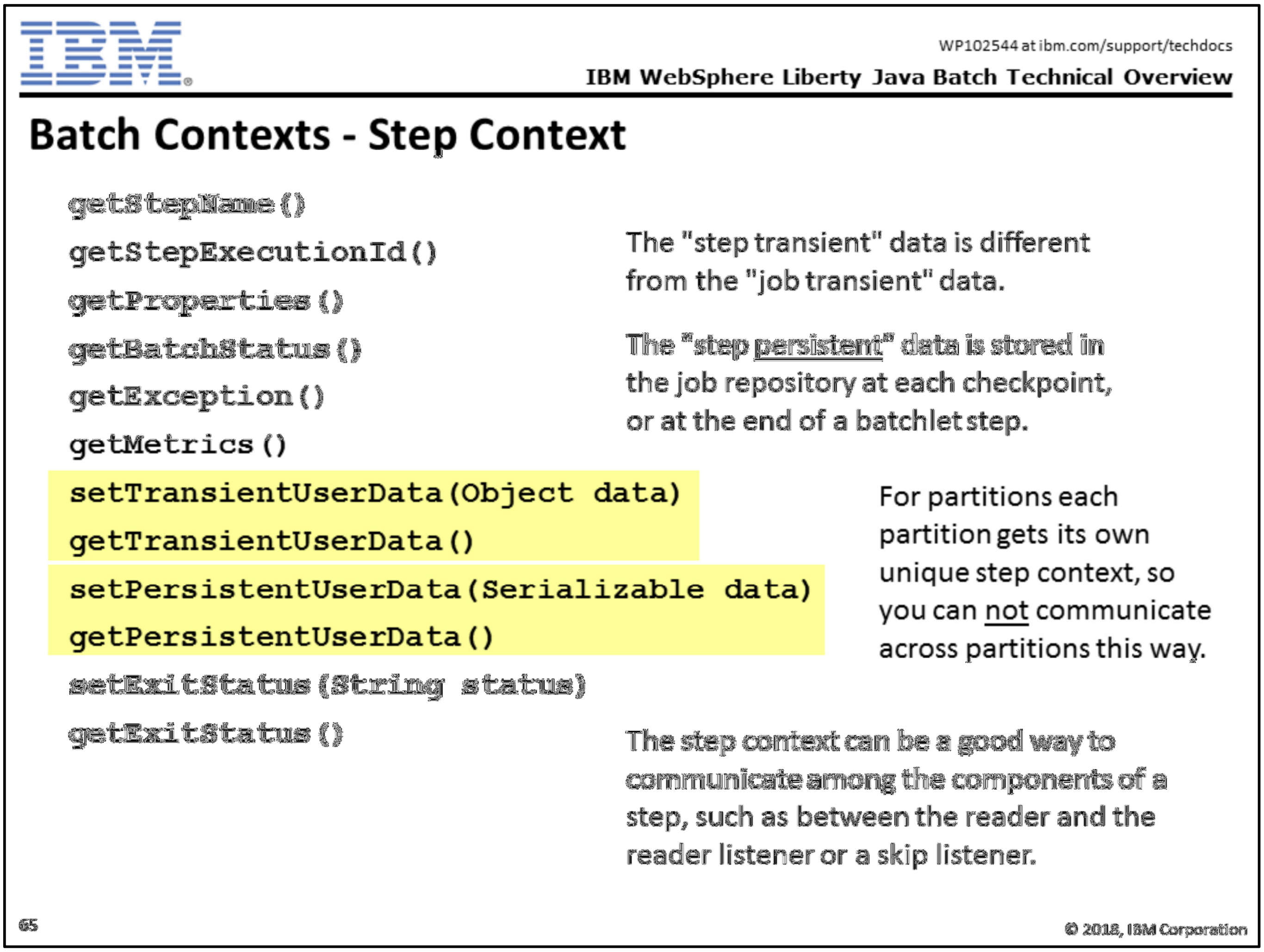
This is useful for passing information between steps in a job.

© 2018, IBM Corporation

Finally, we'll explore two more things -- Job contexts and Step contexts.

Job contexts provide a way for your batch step code to set and get information about the job during the execution of the job. The data it can get includes the job name, job properties, and status information. It can also set transient user data, and set the exit status for the job.

The transient user data is just that -- transient -- and is not persisted to the job repository. It exists during the life of the job, but not after. The information stays local to the JVM in which it is set. It's useful for passing information between steps in a job that executes within a single JVM.



WP102544 at ibm.com/support/techdocs

IBM WebSphere Liberty Java Batch Technical Overview

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.

65

© 2018, IBM Corporation

Step context is similar to job context in concept, but it is not the same thing as job context. With the step context function you can get and set information as shown on the chart. You can set either transient user data or persistent user data. Transient user data is not persisted, and exists only as long as the step exists. Persistent user data is stored in the job repository, and exists beyond the life of the step.

Some quick notes about step context -- you can't communicate across partitions using step context, as each partition gets its own unique step context. But step context is a good way to pass information between elements within a step, such as between the ItemReader and the reader listener or a skip listener.

End of Document