

# Java Batch Job Framework

---

## Intermediate Tutorial

Author: Adym Lincoln, Java Batch Job Framework  
Copyright © 2006-2010, Java Batch Job Framework Software, All Rights Reserved

<b>PREAMBLE.....</b>	<b>3</b>
<b>GLOSSARY.....</b>	<b>3</b>
<b>REFERENCES.....</b>	<b>3</b>
<b>INTERMEDIATE TUTORIAL.....</b>	<b>4</b>
<b>PREREQUISITES.....</b>	<b>4</b>
<b>BASIC RESOURCES.....</b>	<b>4</b>
<b>RESOURCES AND THE JOB STACK.....</b>	<b>6</b>
<b>REQUIRED RESOURCES.....</b>	<b>9</b>
<b>SYNOPSIS.....</b>	<b>11</b>
<b>OTHER RESOURCES.....</b>	<b>12</b>

## Preamble

This tutorial assumes that you've read through the User Guide and gone through the Basics Tutorial on JBJF. The Intermediate Tutorial will build on the code done in the Basics Tutorial. In this tutorial we will focus on the following concepts:

- ✓ Resources
- ✓ Job Stack
- ✓ Required Resources

## Glossary

Name	Description/Comments
JBJF	A document acronym for Java Batch Job Framework.
XML	Industry standard for Extensible Markup Language. A simple language for adding structure to data and documents.
XML Definition	A coding paradigm that combines Java's programming language with XML configuration files.
JBJF Batch Definition File	A specialized XML file that contains data and elements specific to a JBJF batch job.

## References

- Title: [jbjf-user-guide.pdf](#)  
Location: <http://jbjf.sourceforge.net/pdfs/jbjf-user-guide.pdf>  
Author: Adym S. Lincoln  
Referenced Revision: 1.3.0
- Title: [jbjf-basics-tutorial.pdf](#)  
Location: <http://jbjf.sourceforge.net/pdfs/jbjf-basics-tutorial.pdf>  
Author: Adym S. Lincoln  
Referenced Revision: 1.3.0

## Intermediate Tutorial

In this tutorial we'll move beyond the basics of JBJF and begin to plant resources in our task elements. We'll also illustrate how information and data get passed from one task to another via the job stack.

### Prerequisites

The tutorial focuses on the some essential concepts that JBJF relies on such as named resources and inter-task communication.

For this tutorial, I'll be touching on the following concepts:

- ✓ Named Resources – String only
- ✓ Job Stack – Passing data task-to-task
- ✓ Required Resources – Optional Contracts between JBJF and Tasks.

While the coding, builds and runtime will be done on a Windows desktop, any subtle differences to Unix, Linux or Gnome can be easily adjusted.

### Basic Resources

We will pick up where the Basics Tutorial left off. You are free to download that source project and start with that.

If you download the Basics Tutorial source and rename it for the Intermediate Tutorial there are some things you'll need to know:

- The Intermediate Tutorial uses the JBJF 1.3.0 version, so in your <jbjf-directories> element add a new sub element called "plugins":

```
<jbjf-directories>
  <directory name="base" addressing="relative">.</directory>
  <directory name="log4j" addressing="relative">etc</directory>
  <directory name="plugins" addressing="relative">.</directory>
</jbjf-directories>
```

- Adjust the <jbjf-logs> to look in the current directory ./etc/ and not the parent directory, ../etc/.

If you download the Intermediate Tutorial source, the project has two folders, ./lib and ./logs. The ./lib holds all the jarfiles and the ./logs holds all the logfiles from log4j. I'll be running the tutorial directly thru Eclipse, so these sub-directories were needed in the project.

Start by opening up the Batch Definition file in the ./etc folder. When we initially built this file we created some <task> elements and pointed them to our Task classes. In the <task t001> element add a couple of new <resource> elements with type attributes

“file-name” and “file-log4j”. Values should be ./logs/jbjf-tutorial-intermediate.log and ./etc/log4j.properties.

```
<jbjf-tasks>
  <task name="t001" order="1" active="true">
    <class>org.jbjf.tasks.Task001Basics</class>
    <resource type="file-name">./logs/jbjf-tutorial-intermedi-
ate.log</resource>
    <resource type="file-log4j">./etc/log4j.properties</re-
source>
  </task>
```

Leave the <task two> element as-is for now.

Save your Definition file.

Now open up the Task001Basics class and locate the runTask() method. In here we'll add code that is going to pickup, resolve and output the two <resource> elements we just added.

Remember in the Basics Tutorial that we “extended” the AbstractTask instead of “implementing” the ITask interface. The extension allows us to inherit the partial implementation of AbstractTask, specifically the initTask(). The default functionality of the AbstractTask.initTask() will provide the following services for your sub-class:

- ✓ Iterates through any <resource> elements, resolving the objects as needed. The type attribute of the resource element serves two purposes, a key to the task resources stack or a key to a JBJF Named Resource within the Batch Definition file. In general, if type equals an existing JBJF element name, then it gets processed. For the AbstractTask, the following JBJF elements are recognized and processed by initTask():
  - ✓ log-definition
  - ✓ ftp-definition
  - ✓ plugin (prefix)
  - ✓ Any other type value is considered a custom <resource> and stored in the Resources stack for the Task as a String.
  - ✓ Other special resources such as sql-definition and export-definition objects need to be processed in your own task by over-riding the initTask() method. Your initTask() is free to call super.initTask() to pre-process some of the <resource> elements, but your initTask() should then process the various special objects needed by the task sub-class.
- ✓ Check for archivist status and pull the Zipper and Archivist objects off the job-stack when the archivist is enabled.

The focus of this tutorial is on the first point that iterates through all the <resource> elements. JBJF will, by default, execute the initTask() automatically prior to runTask(), so by the time runTask() gets executed, all the <resource> element parsing is complete. Our <resource> elements have been parsed and placed on the Resources stack/cache

for the Task. These are reachable using the `getResources()` method. We can use this getter method to grab resources and use the type attribute as the lookup key. The only caveat is we need to “cast” the Object from the Resource stack into the proper Object type. The following code snippet illustrates this concept:

```
String llogFile = (String)getResources().get( "file-name" );
String llog4jFile = (String)getResources().get( "file-log4j" );
```

The `getResources()` method returns a `HashMap` object, sort of a poor man’s indexed table. As such, we need to “cast” our resources to the proper type, (`String`). We use the type attribute value from the Batch Definition file to “lookup” the resource, “file-name” and “file-log4j”. Once in variables, we are free to use them and/or output them like a normal variable:

```
getLog().debug( "file-name.....[ " + llogFile + " ]" );
getLog().debug( "file-log4j.....[ " + llog4jFile + " ]" );
```

When we save, compile and run these changes we get the expected result of our XML element values appearing in our output:

```
2010-01-15 10:15:52,590 [main] DEBUG org.jbjf.tutorial - Gathering
file-log4j [ ./etc/log4j.properties ]
2010-01-15 10:15:52,592 [main] DEBUG org.jbjf.tutorial - Initialize
Work Unit...Complete...
2010-01-15 10:15:52,592 [main] DEBUG org.jbjf.tutorial - Task
[ Task001Basics() ]...Starting...
2010-01-15 10:15:52,592 [main] DEBUG org.jbjf.tutorial - file-
name.....[ ./logs/jbjf-tutorial-intermediate.log ]
2010-01-15 10:15:52,593 [main] DEBUG org.jbjf.tutorial - file-
log4j.....[ ./etc/log4j.properties ]
2010-01-15 10:15:52,593 [main] DEBUG org.jbjf.tutorial - Task
[ Task001Basics() ]...Complete...
2010-01-15 10:15:52,593 [main] INFO org.jbjf.tutorial - Work Unit ...
Complete ...
2010-01-15 10:15:52,593 [main] INFO org.jbjf.tutorial - Finishing
jbjf-generic-batch...
```

The Resources cache is designed to be flexible and open. Thus, you can really list as many `<resource>` elements as you wish for any one task. The only requirement is that each type attribute needs to be unique.

## Resources and the Job Stack

So resources are listed within the `<task>` element and can get picked up using `getResources()`.

We can use the `<resource>` cache in conjunction with the Job Stack to also pass information between tasks. The Job Stack is a `HashMap` just like the Resource stack, except it’s for the Batch Process...i.e. accessible from all tasks. Using the same lookup concept, you can build on the Task Resource concept and begin to communicate

outside the Task. The only caveat is you need to pay close attention to which stack you are pushing to, remember the stack (HashMap) needs a “unique” key value, otherwise the it will “replace” your object.

Now lets open up the Batch Definition file again and expand our <resource> elements a little. The goal is to pass a value from <task t001> to <task two>. We’ll pass the file name of the logfile for the tutorial. Then we can have <task two> pickup the file name and do something with that information. In this case we’ll have <task two> simply output the logfile name...I know, stupid, but it will illustrate our concept.

Open up Task001Basics, we need to “Push” the logfile value from <task t001> onto the Job Stack. The runTask() method takes a HashMap object as the sole parameter, this is in fact the Job Stack from JBJF. However, there is a recommended getter method, getParameters(), that is the preferred way that you should use to return the Job Stack. That said, add a “Push” to the Job stack near the end of the method that places “file-name” with the value ./logs/jbjf-tutorial-intermediate.log...you’re free to use the variable we created from the Resource cache. A “Push” is done using the put() method and you pass it a key and a value:

```
// PUSH : Push the value onto the job stack...
getParameters().put( "file-name", llogfile );

getLog().debug( "Task [ " + this.SHORT_NAME + " ]...Complete..." );
```

Next, we need to configure <task two>, setting the active indicator to true. We then add a new <resource> to <task two> called “job-stack-key” with a value of “file-name”:

```
<task name="two" order="2" active="true">
  <class>org.jbjf.tasks.Task002Basics</class>
  <resource type="job-stack-key">file-name</resource>
</task>
```

Now open up the Task002Basics class. In here we need code to pickup the resource and then output it to the logfile. Unlike before though, our <resource> element really contains a “key” or lookup for the job stack, not the actual value. So we’ll need to perform two steps to actually get to our true value, which is the name of the logfile:

```
public void runTask( HashMap pjobParameters ) throws Exception {
    getLog().debug( "Task [ " + this.SHORT_NAME + " ]...Starting..." );

    // KEY : Get the key/lookup value from the resource cache...
    String lkeyFile = (String)getResources().get( "job-stack-key" );
    // VALUE : Use the key to get the Job Stack value...
    String llogfile = (String)getParameters().get( lkeyFile );

    getLog().debug( "file key.....[ " + lkeyFile + " ]" );
    getLog().debug( "file-name.....[ " + llogfile + " ]" );

    getLog().debug( "Task [ " + this.SHORT_NAME + " ]...Complete..." );
}
```

OK. I don't want to wear you out, but the above lines of code are illustrating a critical concept in JBJF. We are using a simple "abstraction" on the Job Stack using the built-in "key=value" concept of a HashMap. We do this to enhance reusability in tasks. We could have just passed the logfile name directly in a resource element by coding the following in <task two>:

```
<task name="two" order="2" active="true">
  <class>org.jbjf.tasks.Task002Basics</class>
  <resource type="file-name">./logs/jbjf-tutorial-intermedi-
ate.log</resource>
</task>
```

Yes, this will work and it would only require a single line of code in runTask() to fetch the <resource> value. The problem with this method is it limits the reuse of the Task002Basics. By using a Job Stack key (file-name in this case) in the <resource> element to lookup the value, we can now reuse this class for any number of efforts. The only requirement is that the task sending the data needs to push it onto the Job Stack using a key called "job-stack-key". This becomes more apparent when you start working on larger scale efforts involved SQL Result sets and you have a need to read many times in the same manner, passing those Result sets to some sort of export routine. I should also mention the use of a type attribute called job-stack-key is coincidental only. We could have easily used something like "input-data" or "file-key".

Anyway, back to the code. Please note in the first line how we grab the key from the Task's resource cache (getResources()) and in the second line we get the value from the "Job Stack" (getParameters()). It's very easy to use getResources() in both calls...Don't ask me how I know this...just trust me! This may seem confusing at first, but a few efforts with JBJF and the concept becomes clearer and you learn new ways to harness the power of this concept.

When we save and run this, we can see the Task002 output and how it picks up the Job Stack value:

```
2010-01-15 10:54:56,050 [main] DEBUG org.jbjf.tutorial - Task
[ Task002Basics() ]...Starting...
2010-01-15 10:54:56,051 [main] DEBUG org.jbjf.tutorial - file
key.....[ file-name ]
2010-01-15 10:54:56,051 [main] DEBUG org.jbjf.tutorial - file-
name.....[ ./logs/jbjf-tutorial-intermediate.log ]
2010-01-15 10:54:56,051 [main] DEBUG org.jbjf.tutorial - Task
[ Task002Basics() ]...Complete...
```

The key concept here is the use of a key/lookup values and a little bit of abstraction to perform inter-task communication. This enhances the reuse of tasks and makes maintenance a little easier.

## Required Resources

Now that we can use resources in our tasks and pass them between tasks, it's time to introduce the "Required Resources". As part of JBJF version 1.2.0, we wanted to provide a "contract" between JBJF and a reusable Task. Some fixed requirements so that when the Task begins to get reused, it gets used in the way it should.

Required Resources is a simple list of named resources that a Task needs to complete its work. If your task uses "Optional" resources, it should not be listed as required, but simple documented in your Task as Optional.

When you extend the AbstractTask you automatically gain the Required Resources list. By default, the list is NOT initialized as the Required Resources are an OPTIONAL tool. All you need to do is initialize and populate the list prior to runTask(). Then test the list within runTask() prior to running your code. If you choose to implement the ITask interface, you'll need to setup and control your own list, see AbstractTask for guidance.

Generally, all subclasses of AbstractTask generate a default constructor that allows the Task to be controlled by JBJF. I recommend using the default constructor to populate the Required Resources list, since we know it "always" gets executed before anything else. For this section, I'll continue building on Task001Basics and Task002Basics.

Open up the Task001Basics class and locate the Default constructor. In here we'll initialize and set our Required Resources to "file-name" and "file-log4j". We use the getRequiredResources() method to return the list.

```
/**
 * Default constructor.
 */
public Task001Basics() {
    mtaskRequired = new ArrayList();
    getRequiredResources().add( "file-name" );
    getRequiredResources().add( "file-log4j" );
}
```

By coding these lines in here we're basically "ensuring" that anyone who wishes to use Task001Basics must now have two <resource> elements coded in the <task>, one of type attribute "file-name", one of type attribute "file-log4j".

To "enforce" the requirement we code in the required resources test as part of runTask() to check for the Required Resources. We use the function hasRequiredResources() to test the list:

```
if ( hasRequiredResources() ) {
    String llogFile = (String)getResources().get( "file-name" );
    String llog4jFile = (String)getResources().get( "file-log4j" );

    getLog().debug( "file-name.....[ " + llogFile + " ]" );
    getLog().debug( "file-log4j.....[ " + llog4jFile + " ]" );
}
```

```
        getLog().debug( "Task [ " + this.SHORT_NAME +  
" ]...Complete..." );  
        getParameters().put( "file-name", llogFile );  
    }
```

The `hasRequiredResources()` has two outcomes:

- true, meaning all the required resources are present.
- exception, something is missing.

Of course, there are options to `hasRequiredResources()` default behavior:

- Override `hasRequiredResources()` in your subclass and don't throw the exception, simply return false. This could be handy if you don't want your Task to throw an exception and terminate the batch, but instead set "default" values for missing resources.
- Catch the exception and do whatever controls and recovery you wish.

Please repeat the Required Resources for Task002Basics, listing the job-stack-key as the only Required Resource.

Ok course when we run this everything works. Feel free to remove a required resource element from one of the tasks in the Batch Definition file and repeat the run. You'll probably see the following exception in your logfile/console:

```
java.lang.Exception: Required resource(s) [file-name,file-log4j] not  
found. Please check the Batch Definition file! The required resources  
for the task AbstractTask() are [file-name,file-log4j]  
    at org.jbjf.core.AbstractTask.hasRequiredResources(Unknown Source)  
    at org.jbjf.tasks.Task001Basics.runTask(Task001Basics.java:82)  
    at org.jbjf.core.AbstractBatch.runBatch(Unknown Source)  
    at org.jbjf.core.AbstractBatch._runBatch(Unknown Source)  
    at org.jbjf.core.AbstractBatch._main(Unknown Source)  
    at org.jbjf.core.DefaultBatch.main(Unknown Source)
```

## Synopsis

Resources are at the heart of your Tasks. They provide a flexible and easy way to pass adhoc and structured parameters into your Tasks. They can be mixed with the Job Stack to provide inter-task communication, thus allowing one task to pass information to other tasks. Using a little bit of abstraction with keys and values in the Resource and Job stacks allows you an added power to reuse tasks and begin to setup simple libraries of tasks for specific work.

## Other Resources

JBJF Website – <http://jbjf.sourceforge.net/>

JBJF Tutorials/Documentation - <http://jbjf.sourceforge.net/documentation.html>