z/OS

z/OS Batch Runtime: Planning and User's Guide

Version 2 Release 1

SA23-1376-00

Note

Before using this information and the product it supports, read the information in “Notices” on page 105.

This edition applies to Version 2 Release 1 of z/OS (5650-ZOS) and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Figures

Tables

About this information

This publication describes the IBM® z/OS Batch Runtime component of z/OS®. z/OS Batch Runtime provides the ability to update the DB2® database from PL/I, COBOL, and Java™ in a single transaction.

This publication is organized as follows:
- Chapter 1, "Overview and planning of z/OS Batch Runtime," on page 1. This chapter describes overview information for z/OS Batch Runtime and how to invoke the program.
- Chapter 2, "Invoking z/OS Batch Runtime," on page 5. This chapter describes how to invoke the z/OS Batch Runtime program through the job control language (JCL).
- Chapter 3, "Defining connectivity for the database," on page 17. This chapter describes planning connectivity for z/OS Batch Runtime.
- Chapter 4, "Application interfaces for z/OS Batch Runtime," on page 19. This chapter describes application programming interfaces for z/OS Batch Runtime including: options, support elements for the Java Database Connectivity (JDBC) and DB2 programs, environment variables, completion codes, and any applicable API.
- Chapter 5, "Application structure and build considerations for COBOL and Java," on page 25. This chapter describes application structuring and building procedures for z/OS Batch Runtime.
- Chapter 6, "Application structure and build considerations for PL/I and Java," on page 49. This chapter describes using PL/I for z/OS Batch Runtime.
- Chapter 7, "Troubleshooting for z/OS Batch Runtime," on page 59. This chapter describes diagnostics and troubleshooting procedures for z/OS Batch Runtime.

Who should read Batch Runtime Planning and User's Guide

This publication is intended for experienced PL/I, COBOL, and Java programmers who are familiar with DB2 and plan, develop, and test applications that run on z/OS. It describes how to improve interoperability between PL/I, COBOL, and Java applications by allowing you to share a local DB2 attachment in a single hybrid Java COBOL application. Advanced knowledge of the Java Native Interface (JNI), PL/I, COBOL, Java programming, and DB2 is required.

Note: All examples in this publication are for illustration purposes only. You must replace any example or code parameters with the correct specifications for your installation.

Where to find more information

When possible, this information uses cross-document links that go directly to the topic in reference using shortened versions of the document title. For complete titles and order numbers of the documents for all products that are part of z/OS, see *z/OS Information Roadmap*.

To find the complete z/OS library, including the z/OS Information Center, see z/OS Internet Library (http://www.ibm.com/systems/z/os/zos/bkserv/).

Internet sources

The following resources are available through the Internet to provide additional information about z/OS:

- **Online library**

  To view and print online versions of the z/OS publications, use this address:

  `http://www.ibm.com/systems/z/os/zos/bkserv/`

- **Redbooks®**

  The documents known as IBM Redbooks that are produced by the International Technical Support Organization (ITSO) are available at the following address:

  `http://www.redbooks.ibm.com`

How to send your comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or provide any other feedback that you have.

Use one of the following methods to send your comments:

1. Send an email to mhvrcfs@us.ibm.com.
2. Send an email from the Contact z/OS.
3. Mail the comments to the following address:
   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Building 707
   2455 South Road
   Poughkeepsie, NY 12601-5400
   US
4. Fax the comments to us, as follows:
   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address.
- Your email address.
- Your telephone or fax number.
- The publication title and order number:
  z/OS V2R1.0 Batch Runtime Planning and User's Guide
  SA23-1376-00
- The topic and page number that is related to your comment.
- The text of your comment.

When you send comments to IBM, you grant IBM a nonexclusive right to use or distribute the comments in any way appropriate without incurring any obligation to you.

IBM or any other organizations use the personal information that you supply to contact you only about the issues that you submit.

If you have a technical problem

Do not use the feedback methods that are listed for sending comments. Instead, take one of the following actions:

- Contact your IBM service representative.
- Call IBM technical support.
- Visit the IBM Support Portal at IBM support portal.

z/OS Version 2 Release 1 summary of changes

See the following publications for all enhancements to z/OS Version 2 Release 1 (V2R1):

- *z/OS Migration*
- *z/OS Planning for Installation*
- *z/OS Summary of Message and Interface Changes*
- *z/OS Introduction and Release Guide*

Chapter 1. Overview and planning of z/OS Batch Runtime

In today's z/OS environment, many installations want to re-engineer their existing native z/OS PL/I, COBOL, and Java applications to incorporate the Java language. By doing so, they can keep their heritage of existing z/OS PL/I, COBOL, and Java batch applications, while taking advantage of the larger developer skill base and many language features of Java. As such, there is a requisite need to share a local DB2 for z/OS attachment across the PL/I, COBOL, and Java language boundary. This enables mixed language programs to process DB2 for z/OS requests in the same unit of work (UOW). When these batch application suites are re-engineered or updated, they should also allow transparent local DB2 for z/OS access from PL/I, COBOL, and Java to the following programs:

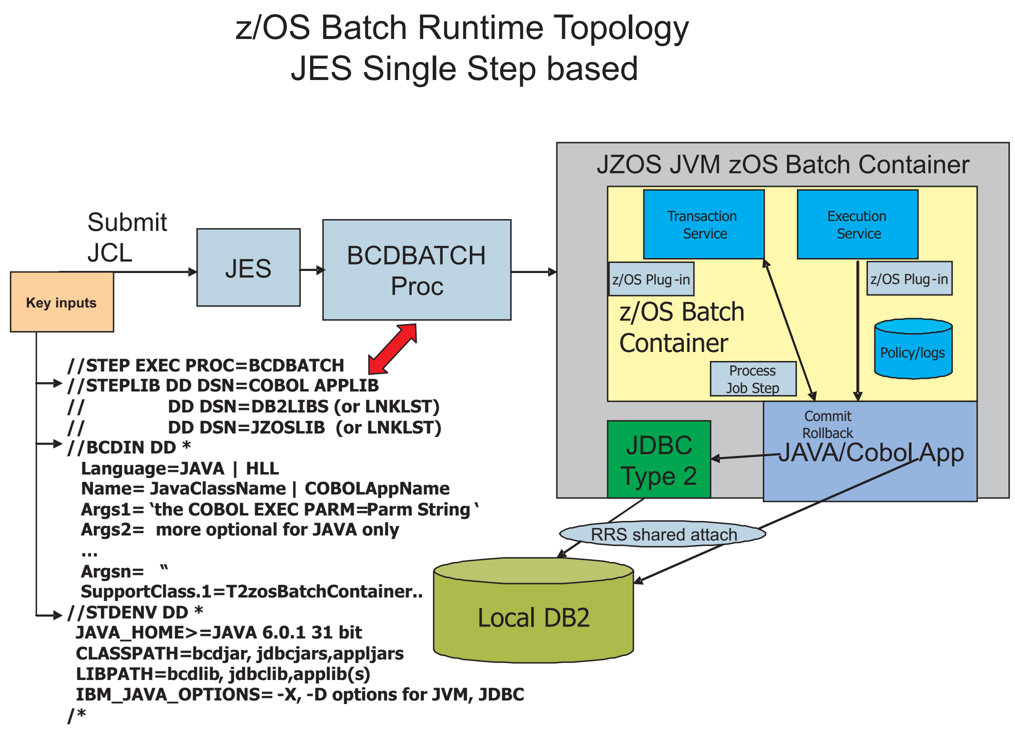
- Embedded Structured Query Language (SQL) DB2 access, which is used in Enterprise PL/I, COBOL, and Java
- Java Database Connectivity (JDBC) for Dynamic SQL
- Embedded Structured Query Language for Java (SQLJ)

z/OS Batch Runtime allows for this interoperability between PL/I, COBOL, and Java applications that run on z/OS. It is a program designed to provide a managed environment that enables shared access to a DB2 connection by PL/I, COBOL, and Java programs. Updates to DB2 are committed in a single transaction. (Note that updates to multiple databases are not supported.)

A new feature of z/OS Batch Runtime is support for the IBM Java Batch Programming Model. It enables pure Java batch applications to be both data source and sink neutral with an XML like definition and is fully described in Appendix A. A new JCL statement, //BCDXJCL is added as the source for this descriptor along with a new and special language type, XJCL.

Figure 1 on page 2 shows a high-level overview of the z/OS Batch Runtime environment. The batch container performs the initialization that sets up the environment for PL/I, COBOL, Java, and DB2 interoperability. This includes the following tasks:

- Setting up the proper Language Environment® for the PL/I or COBOL programs to run
- Setting up the job step under the umbrella of a Resource Recovery Services (RRS)-managed global transaction
- Initiating the DB2 JDBC driver in this special “BatchContainer” mode
- Invoking the DB2 JDBC driver to create a DB2 connection and attachment thread
- Invoking the primary PL/I, COBOL or Java application after the environment is properly initialized.



```
//STEP EXEC PROC=BCDBATCH
//STEPLIB DD DSN=COBOL APPLIB
//           DD DSN=DB2LIBS (or LNKLST)
//           DD DSN=JZOSLIB  (or LNKLST)
//BCDIN DD *
  Language=JAVA | HLL
  Name= JavaClassName | COBOLAppName
  Args1= 'the COBOL EXEC PARM=Parm String'
  Args2=  more optional for JAVA only
  ...
  Argsn=   "
  SupportClass.1=T2zosBatchContainer..
//STDENV DD *
 JAVA_HOME>=JAVA 6.0.1 31 bit
 CLASSPATH=bcdjar, jdbcjars,appljars
 LIBPATH=bcdlib, jdbclib,applib(s)
 IBM_JAVA_OPTIONS= -X, -D options for JVM, JDBC
/*
```

Figure 1. Overview of the z/OS Batch Runtime environment

Requirements for z/OS Batch Runtime

z/OS Batch Runtime requires the following programs:

- IBM SDK for z/OS, Java Technology Edition, V6.0.1 or higher (5655-R31). (For details, see "Configuring Java" on page 5.)
- Pure Java applications using IBM Java Batch Common Programming Model Support: may use either 31-bit or 64-bit JVM
- IBM COBOL and PL/I interoperability functions: 31-bit JVM only
- IBM Enterprise PL/I and COBOL Version 4.2
- One of the following:
  - DB2 V9 with PTF UK62190 for JDBC 3.0 specification level, or PTF UK62191 for JDBC 4.0 specification level
  - DB2 V10 with PTF UK62141 for JDBC 3.0 specification level, or PTF UK62145 for JDBC 4.0 specification level

For more information about these required programs, see the appropriate reference listed in Table 1.

Table 1. Summary of reference information for required programs

For information about	Refer to
Java	`http://www.ibm.com/systems/z/os/zos/tools/java/`
IBM Enterprise COBOL Version 4 Release 2	`http://www.ibm.com/software/awdtools/cobol/zos/library/`
DB2	`http://www.ibm.com/software/data/db2/zos/family/`

Table 1. Summary of reference information for required programs (continued)

For information about	Refer to
PL/I	`http://www-01.ibm.com/software/awdtools/pli/plizos/library/`

Planning for z/OS Batch Runtime

When planning use of z/OS Batch Runtime, a good application to consider using is a native procedural z/OS COBOL or PL/I application that you want to functionally enhance with Java method calls. The entire application code must be single threaded. Also, see Chapter 5, "Application structure and build considerations for COBOL and Java," on page 25 for more information.

Chapter 2. Invoking z/OS Batch Runtime

The z/OS Batch Runtime is established by launching the Java program `com.ibm.zos.batch.container.BCDBatchContainer` with the proper configuration and environment settings that allows your PL/I, COBOL, and Java programs to be invoked with the correct arguments. The JZOS launcher, a component of the IBM JDK for z/OS, is used to establish the environment and pass control to z/OS Batch Runtime which, in turn, will launch your PL/I, COBOL, or Java program and provide necessary services. To facilitate the use of z/OS Batch Runtime, z/OS includes:

- Environment tailoring shell scripts: `bcdconfig.sh` and `bcdconfigend.sh in /usr/lpp/bcp`
- A JCL procedure to be invoked by batch jobs: BCDPROC in SYS1.PROCLIB
- A sample batch job to use BCDPROC: BCDBATCH in SYS1.SAMPLIB

Configuring Java

You must configure the CLASSPATH and LIBPATH variables with the list of Java archive (JAR) files and dynamic link library (DLL) files that are required to run both the z/OS Batch Runtime and the application. z/OS Batch Runtime is itself a Java application and uses the JZOS toolkit to launch the JVM. You should tailor the z/OS Batch Runtime sample BCDBATCH JCL and the environment variables it provides.

Additionally, JZOS defines several environment variables that allow you to control the Java options that JZOS uses when it creates the JVM and main method program arguments. Find these options and complete information in *JZOS Batch Launcher and Toolkit function in IBM SDK for z/OS*, at `www.ibm.com/systems/z/os/zos/tools/java/products/jzos/overview.html`.

Note: Although JZOS also defines environment variables that allow you to control the encoding of output, z/OS Batch Runtime only supports EBCDIC file encoding.

Improving Java start up time

For short-running jobs, improving Java start up time is important. This is especially true when running numerous small Java batch jobs, as the Java start up elapsed time and CPU time may affect performance. Using the following Java options can make it possible to reduce the Java startup times for applications that frequently start a new JVM :

- -Xquickstart Java option

  Note: Quickstart may improve startup time for short running jobs, but it may degrade performance of long running applications.
- Shared classes and AOT Java options

For more details about this topic as well as the latest considerations for using Java, performance information, hints and tips, and information about developing and running applications see:

`http://www.ibm.com/systems/z/os/zos/tools/java`

Java environment variables for z/OS Batch Runtime

Java applications use the following environment variables for z/OS Batch Runtime that are specified in the JCL:

- JAVA_HOME
- CLASSPATH
- LIBPATH
- IBM_JAVA_OPTIONS

See "Procedure for modifying the BCDBATCH job" on page 10 for examples of how to specify these environment variables.

JAVA_HOME

The application must set the JAVA_HOME environment variable to a minimum level of JAVA 6.0.1.

CLASSPATH

The application must set the CLASSPATH to include the .JAR files for z/OS Batch Runtime, the DB2 driver for JDBC (DB2 JCC), and the application. To do so, use the CLASSPATH environment variable specified in the BCDBATCH JCL procedure.

The configuration script automatically updates the CLASSPATH for z/OS Batch Runtime .jar files, based on the exported BCD_HOME variable in the BCDBATCH JCL procedure.

LIBPATH

In the BCDBATCH JCL procedure, the application must set LIBPATH to the location of the DLLs for z/OS Batch Runtime, DB2 JCC, and any that are associated with application. The configuration script performs the function.

IBM_JAVA_OPTIONS

This environment variable is a concatenation of the IBM JVM runtime options, which are typically prefixed with -X, and any Java system properties, which are prefixed with -D. This can include, for example, the JVM heap size runtime option and the DB2 package list system property.

31-bit support

z/OS Batch Runtime for COBOL and PL/I interoperability supports only 31-bit applications; you must use the 31-bit JVM.

64-bit support

IBM Java Batch Common Programming Model Support supports both 31-bit and 64-bit applications.

Main JCL statements needed for BCDBATCH

This section of the documentation uses reference keys, such as **1** , **2** , to match the instructions with the sample JCL.

z/OS Batch Runtime supplies a sample BCDBATCH job which you modify to suit your application. Table 2 on page 7 summarizes the main JCL statements for the BCDBATCH job. "Procedure for modifying the BCDBATCH job" on page 10 contains complete steps to modifying the sample BCDBATCH job.

Table 2. JCL summary for BCDBATCH job

JCL statement	Explanation

1
 `//BCDBATCH  JOB  (1),'name'`
 `//BATCH EXEC BCDPROC,REGION=0M,LOGLVL='+I'`
	The JCL that invokes z/OS Batch Runtime. Throughout this publication, the JCL used to invoke the z/OS Batch Runtime is referred to as the BCDBATCH job. Use any job name that is acceptable to your installation.

2
 `//*STEPLIB DD  DSN=hlq.yourapp.loadlib,DISP=SHR`
 `//*        DD  DSN=hlq.jzos.loadlib,DISP=SHR`
	Add any load libraries your application requires to the STEPLIB; for example, this could be the data set containing your PL/I or COBOL application load modules. If the JZOS Java launcher is not installed in the LNKLST, add a STEPLIB for it. For more information about installing JZOS, see the JZOS Java Launcher and Toolkit Overview at `www.ibm.com/systems/z/os/zos/tools/java/`.
	
	 Any COBOL application modules must be in either the //STEPLIB concatenation or added to a STEPLIB environment variable in //STDENV DD *. Do not use LIBPATH for starting a COBOL application.

3 `//STDENV DD *`	Specifies the environment variables used for this run, including JAVA_HOME, CLASSPATH, and LIBPATH.
4 `//BCDIN DD *`	Specifies a file containing the batch configuration options. Note that some support elements obtain their options from Java system properties. See "JCL for BCDIN configurations options" on page 12 for more information.
5 `//BCDXJCL`	Names a file containing the xJCL XML definition describing the application to run. The syntax of the xJCL is described in the WebSphere® Compute Grid Info Center. Compute Grid provides some tooling to assist in the development of XJCL, which is part of the Modern Batch project support in Rational® Application Developer V8. This statement is required when an xJCL application is processed.

Table 2. JCL summary for BCDBATCH job (continued)

JCL statement	Explanation
6 `//BCDXPROP`	Names a file containing substitution properties to be applied the xJCL when the job is submitted. The file is in keyword=value format using the same syntax rules as the //BCDIN file. This statement is optional.

JCL for the BCDBATCH job

A current sample of BCDBATCH job for z/OS Batch Runtime is in SYS1.SAMPLIB. For convenience and planning purposes, this documentation contains the following "Sample BCDBATCH JCL," "Procedure for modifying the BCDBATCH job" on page 10, and "Sample BCDPROC to invoke z/OS Batch Runtime" on page 15.

Note: All examples in this publication are for illustration purposes only. You must replace any example or code parameters with the correct specifications for your installation.

Sample BCDBATCH JCL

Figure 2 on page 9 is an example of JCL procedure for running the sample BCDBATCH job.

```
1
//BCDBATCH  JOB  (1),'name'
//BATCH  EXEC  BCDPROC,REGION=0M,LOGLVL='+I'
//*
//*******************************************************************
//* Update: Add the load libraries your application requires,        *
//*         such as the data set containing your COBOL               *
//*         application load modules to the STEPLIB.                 *
//*                                                                  *
//*         If the JZOS Java launcher has not been installed in      *
//*         the lnklst, add a steplib for it.                        *
//*******************************************************************
2
//*STEPLIB DD  DSN=hlq.yourapp.loadlib,DISP=SHR
//*        DD  DSN=hlq.jzos.loadlib,DISP=SHR
//*
//*
3
//STDENV DD *
#
#-------------------------------------------------------------------
#  UPDATE: Installation path for Batch Runtime.
#          Note: because the Batch Runtime is a component of z/OS,
#          the installation defaults to /usr/lpp/bcp
#-------------------------------------------------------------------
export BCD_HOME=/usr/lpp/bcp
# 4
#-------------------------------------------------------------------
#  UPDATE: Installation path for Java.
#-------------------------------------------------------------------
export JAVA_HOME=/usr/lpp/java/J6.0.1
#
# 5
#-------------------------------------------------------------------
#  The following runs the z/OS
Batch Runtime configuration script.
#  This script processes the exported environment variables that
#  were defined above.
#-------------------------------------------------------------------
. $BCD_HOME/bcdconfig.sh
#
# 6
#-------------------------------------------------------------------
#  UPDATE: JDBC home directory, jar files, and DLLs.
#-------------------------------------------------------------------
#JDBC_HOME=/usr/lpp/db2910_jdbc
#CLASSPATH="$CLASSPATH":$JDBC_HOME/classes/db2jcc.jar
#CLASSPATH="$CLASSPATH":$JDBC_HOME/classes/db2jcc_javax.jar
#export CLASSPATH="$CLASSPATH"
#
#LIBPATH="$LIBPATH":$JDBC_HOME/lib
#export LIBPATH="$LIBPATH"
```

Figure 2. Example: BCDBATCH JCL procedure (Part 1 of 2)

```
#
# 7
#-------------------------------------------------------------------
#   UPDATE: Add your application jar files to the CLASSPATH here.
#-------------------------------------------------------------------
#CLASSPATH="$CLASSPATH":/your/extra/app.jar
#CLASSPATH="$CLASSPATH":/your/extra/app2.jar
#export CLASSPATH="$CLASSPATH"
#
# 8
#-------------------------------------------------------------------
#   UPDATE: Add your application libraries to the LIBPATH here.
#   The LIBPATH defines points to any application-defined DLLs,
#   which may include Java Native Interface (JNI) routines.
#-------------------------------------------------------------------
#LIBPATH="$LIBPATH":/your/extra/lib
#LIBPATH="$LIBPATH":/your/extra/lib2
#export LIBPATH="$LIBPATH"
#
# 9
#-------------------------------------------------------------------
#   UPDATE: Uncomment to enable z/OS
Batch Runtime tracing for diagnosis.
#-------------------------------------------------------------------
#IJO="$IJO -Dcom.ibm.zos.batch.container.BCDTraceConfig.trace=all"
#
# 10
#-------------------------------------------------------------------
#   UPDATE: Uncomment and add any additional JVM options here.
#-------------------------------------------------------------------
#IJO="-Xms256m -Xmx512m"
#
# 11
#-------------------------------------------------------------------
#   UPDATE: Uncomment and add JDBC options here.
#-------------------------------------------------------------------
#IJ0=$IJ0 -Ddb2.jcc.ssid=XXXX -Ddb2.jcc.pkList=NULLID.*,COBOLPKG.*"
#
# 12
#-------------------------------------------------------------------
#   Exports JVM options set above.
#-------------------------------------------------------------------
export IBM_JAVA_OPTIONS="$IJO "
#
# 13
#-------------------------------------------------------------------
#  The following runs the z/OS
Batch Runtime configuration completion
#  script.  This command must be last in the STDENV file.
#-------------------------------------------------------------------
. $BCD_HOME/bcdconfigend.sh
#
//
```

Figure 3. Example: BCDBATCH JCL procedure (Part 2 of 2)

Procedure for modifying the BCDBATCH job

The following JCL procedure summarizes the key statements to modify in the BCDBATCH job (see Figure 2 on page 9) that invokes z/OS Batch Runtime.

- **1** Modify the JOB and EXEC statements to add any parameters required by your installation.

  For example in the following statement, BCDPROC is the batch container JCL procedure.

  ```
  //BATCH EXEC BCDPROC,REGION=0M
  ```

For details and options, including the symbolic to override defaults, see "Sample BCDPROC to invoke z/OS Batch Runtime" on page 15.

- **2** For the STEPLIB statement, specify any load libraries that the application requires (for example, the data set that contains your COBOL application load modules) for DSN=, where *hlq.yourapp.loadlib* is the name:

  ```
  //*STEPLIB DD DSN=hlq.yourapp.loadlib,DISP=SHR
  //  DD DSN=hlq.jzos.loadlib,DISP=SHR
  ```

  This may include requisite DB2 and COBOL libraries that are not in LNKLST but are loaded during program execution. Note that any COBOL modules that are bound as DLLs should usually be found through the LIBPATH environment variable.

  The batch container uses the Java SDK JZOS launcher utility. If you installed the Java SDK using SMP/E, JZOS is installed in the LNKLST. However, if you did not use SMP/E, you must install the JZOS launcher into a data set, and add that to your STEPLIB concatenation.

  For more information about installing JZOS, see the JZOS Java Launcher and Toolkit Overview at `www.ibm.com/systems/z/os/zos/tools/java/`

- **3** Update the installation paths for z/OS Batch Runtime. To tailor the runtime environment, use the `//STDENV DD` statement in the BCDBATCH JCL to define a shell script. The batch container processes the exported `BCD_HOME` environment variable referenced by the script as the installation path for z/OS Batch Runtime (default is `/usr/lpp/bcp`).

- **4** Update the installation path for Java to the correct level of Java:

  ```
  export JAVA_HOME=/usr/lpp/java/J6.0.1
  ```

- **5** Run the z/OS Batch Runtime configuration shell script, bcdconfig, to process the exported environment variables you just defined:

  ```
  . $BCD_HOME/bcdconfig.sh
  ```

  To set up the batch container, you must use the . (dot) command to invoke the bcdconfig.sh.

- **6** Update the JDBC home directory, jar files, and DLLs:

  ```
  JDBC_HOME=/usr/lpp/db2910_jdbc
  ```

- **7** Add additional application jar files to the CLASSPATH:

  ```
  CLASSPATH="$CLASSPATH":/your/extra/app.jar
  #CLASSPATH="$CLASSPATH":/your/extra/app2.jar
  #export CLASSPATH="$CLASSPATH"
  ```

- **8** Add your application DLLs to the LIBPATH directories:

  ```
  LIBPATH="$LIBPATH":/your/extra/lib
  LIBPATH="$LIBPATH":/your/extra/lib2
  export LIBPATH="$LIBPATH"
  ```

- **9** Enable tracing for z/OS Batch Runtime:

  ```
  IJO="$IJO -Dcom.ibm.zos.batch.container.BCDTraceConfig.trace=all
  ```

- **10** Add any additional JVM options:

  ```
  IJO="-Xms256m -Xmx512m"
  ```

  You may add, for example, the -Xquickstart option or any other -D or -X JVM runtime option you want to use.

- **11** Add any additional JDBC options for the DB2 subsystem:

  ```
  IJO="$IJO -Ddb2.jcc.ssid=XXXX -Ddb2.jcc.pkList=NULLID.*,COBOLPKG.*"
  ```

  For more information about the Java Database Connectivity (JDBC) options, see DB2 Application Programming Guide and Reference for Java.

  Do not specify -Dfile.encoding in the IBM_JAVA_OPTIONS string. z/OS Batch Runtime only supports the default z/OS file.encoding of IBM-1047.

- **12** Export the JVM options:

  ```
  export IBM_JAVA_OPTIONS="$IJO "
  ```

  The IBM_JAVA_OPTIONS string must be set and exported before invoking the bcdconifgend.sh script.
- **13** Run the following z/OS Batch Runtime completion script:

  ```
  . $BCD_HOME/bcdconfigend.sh
  ```

  Note: This script must always be run last in `STDENV`.

JCL for BCDIN configurations options

Use the `//BCDIN` JCL statement to control the z/OS Batch Runtime configuration options. Some support elements obtain their options from Java system properties.

When creating a configuration options file, the following rules apply:

- Options must appear in the `keyword=value` format
- Options must be coded in columns 1 through 71 of the record. Long options can be continued by coding a non-blank character in column 72 and continuing on the next line.
- Comment lines contain a # in column one.
- Blank lines are ignored.
- Options are case sensitive.
- When you specify an option more than once, the last occurrence is used.

Sample BCDIN File

Figure 4 on page 13 shows an example file that contains additional details and explanations. You can modify the sample as needed for individual jobs at your installation.

```
//*
//*********************************************************************
//*                                                                   *
//*           Batch Runtime Options                                   *
//*                                                                   *
//* Syntax rules for specifying options:                              *
//*                                                                   *
//*   1. Options are specified in keyword=value format.               *
//*                                                                   *
//*   2. Options are coded using columns 1-71.                        *
//*                                                                   *
//*      Long options may be continued by coding a non-blank          *
//*      character in column 72 and continuing on the next line.      *
//*                                                                   *
//*   3. Comment lines contain a # in column 1.                       *
//*                                                                   *
//*   4. Blank lines are ignored.                                     *
//*                                                                   *
//*   5. Option names are case sensitive.                             *
//*                                                                   *
//*   6. When the same option is specified more than once,            *
//*      the last occurrence of the option is used.                   *
//*                                                                   *
//*********************************************************************
//*
//BCDIN DD *
# 1
#--------------------------------------------------------------------*
# UPDATE: Uncomment the option corresponding to the language of
#         the application being launched
#--------------------------------------------------------------------*
#bcd.applicationLanguage=COBOL
#bcd.applicationLanguage=JAVA
#
# 2
#--------------------------------------------------------------------*
# UPDATE: The program name or fully qualified Java class name
#         of the application to be launched
#--------------------------------------------------------------------*
bcd.applicationName=your.application.name
```

Figure 4. Example: JCL BCDIN configuration options (Part 1 of 2)

```
#
# 3
#--------------------------------------------------------------------*
# UPDATE: Arguments to be passed to the launched application.
#
#         For Java applications, any number of arguments can be used.
#         Each argument is passed as an element of the initial
#         args array passed to the main method.
#
#         For COBOL applications, a single argument with a maximum
#         length of 100 characters can be used.
#--------------------------------------------------------------------*
#bcd.applicationArgs.1=Java argument element 1
#bcd.applicationArgs.2=Java argument element 2
#bcd.applicationArgs.3=Java argument element 3
#
#bcd.applicationArgs.1=COBOL single argument up to 100 characters
#
# 4
#--------------------------------------------------------------------*
# REQUIRED UPDATE: Support class names used to manage transactions.
#
#         For the DB2 JDBC driver, replace with the correct statement
#         for your installation.
#         If your application uses DB2 for z/OS, you MUST uncomment
#         this statement.
#
#         NOTE: A bcd.supportClass.1=class_name must be specified.
#         If you use the one provided by DB2, the DB2-related .jar
#         files and executables must be on the CLASSPATH and LIBPATH,
#         respectively.
#--------------------------------------------------------------------*
#bcd.supportClass.1=com.ibm.db2.jcc.t2zos.T2zosBatchContainerSupport
#
# 5
#--------------------------------------------------------------------*
# UPDATE: Verbose mode for additional diagnostics (default is false).
#--------------------------------------------------------------------*
#bcd.verbose=true
#
//
```

Figure 5. Example: JCL BCDIN configuration options (Part 2 of 2)

Procedure for modifying the BCDIN JCL

The following list summarizes the `BCDIN` JCL statements to use for configuring the Batch Runtime options:

- **1** Specify the option that corresponds to the language of the application.

  For example, in PL/I or COBOL:

  `bcd.applicationLanguage=LE`

  For example, in Java:

  `bcd.applicationLanguage=JAVA`

- **2** Specify the program name or fully qualified Java class name of the application, where *MYPGMNAM* or *yourpackagename* is the name of the application.

  For example, in COBOL:

  `bcd.applicationName=`*`MYPGMNAM`*

  For example, in Java:

  `bcd.applicationName=com.xyz.`*`yourpackagename`*`.classname`

- **3** Specify the program arguments that you want to pass to the program. Java and COBOL each have there own format.

  For Java applications, you can use any number of arguments. Each argument is passed as an element of the initial arguments array passed to the main method. For example:

  ```
  bcd.applicationArgs.1=java arg1
  bcd.applicationArgs.2=java arg2
  bcd.applicationArgs.3=java arg3
  ```

  For COBOL applications, you can use a single argument with a maximum length of 100 characters. For example:

  ```
  bcd.applicationArgs.1=COBOL single argument up to 100 characters
  ```

  The *COBOL single argument...* value corresponds to the *PARM='string <=100 chars'* value of an `//EXECPGM EXEC PGM=Cobol_Main,PARM=` JCL statement.
- **4** Specify the name of the support class used to manage the transaction. For example, the following statement for the DB2 JDBC driver should be uncommented from Figure 4 on page 13.

  ```
  bcd.supportClass.1=com.ibm.db2.jcc.t2zos.T2zosBatchRuntimeSupport
  ```
- **5** Specify the verbose mode, using *true* or *false*.

  If you want diagnostic information, use the following statement:

  ```
  bcd.verbose=true
  ```

  If you do not want verbose mode, use the following:

  ```
  bcd.verbose=false
  ```

Sample BCDPROC to invoke z/OS Batch Runtime

Figure 4 on page 13 shows an example of a BCDPROC statement. You can use a symbolic to override defaults on BCDPROC.

VERSION
Specifies the Java SDK version (default 61).

version 6 (Java 6.0.1)

version 70 (Java 7 31-bit) - Interoperable

version 76 (Java 7 64-bit) - IBM Java Batch Common Programming Model Support

LOGLVL
Specifies the following JZOS trace level:

+I	informational (default)
+T	detail trace (used for additional diagnostics and debugging `//STDENV` script)

LEPARM
Allows for additional Language Environment options to be specified by providing by a `//CEEDOPTS DD` statement. For more information, see *z/OS Language Environment Programming Reference* .

Note: z/OS Batch Runtime only supports EBCDIC file encoding.

```
//BCDPROC PROC VERSION='61',      JVMLDM version: 61 (Java 6.0.1 31bit)
//             LOGLVL='+I',       Debug level: +I(info) +T(trc)
//             LEPARM=''          Language Environment parms
//*
//********************************************************************
//*                                                                  *
//*  Proprietary Statement:                                          *
//*                                                                  *
//*    Licensed Materials - Property of IBM                          *
//*    5694-A01                                                      *
//*    Copyright IBM Corp. 2011.                                     *
//*                                                                  *
//*    Status = HBB7780                                              *
//*                                                                  *
//*  Component = z/OS Batch Runtime (SC1BC)                          *
//*                                                                  *
//*  EXTERNAL CLASSIFICATION = OTHER                                 *
//*  END OF EXTERNAL CLASSIFICATION:                                 *
//*                                                                  *
//*  Sample procedure JCL to invoke z/OS Batch Runtime               *
//*                                                                  *
//*  Notes:                                                          *
//*                                                                  *
//*  1. Override the VERSION symbolic parameter in your JCL          *
//*     to match the level of the Java SDK you are running.          *
//*                                                                  *
//*       VERSION=61    Java SDK 6.0.1 (31 bit)                      *
//*                                                                  *
//*  2. Override the LOGLVL symbolic parameter to control            *
//*     the messages issued by the jZOS Java launcher.               *
//*                                                                  *
//*     Use the +T option when reporting problems to IBM or          *
//*     to diagnose problems in the STDENV script.                   *
//*                                                                  *
//*  3. Override the LEPARM symbolic parameter to add any            *
//*     application specific language environment options           *
//*     needed.                                                      *
//*                                                                  *
//*  Change History =                                                *
//*                                                                  *
//*    $L0=BATCH,HBB7780,100324,KDKJ:                                *
//*                                                                  *
//*                                                                  *
//********************************************************************
//JAVA EXEC PGM=JVMLDM&VERSION,REGION=0M,
//             PARM='&LEPARM/&LOGLVL'
//*
//SYSPRINT  DD  SYSOUT=*          System stdout
//SYSOUT    DD  SYSOUT=*          System stderr
//STDOUT    DD  SYSOUT=*          Java System.out
//STDERR    DD  SYSOUT=*          Java System.err
//BCDOUT    DD  SYSOUT=*          Batch container messages
//BCDTRACE  DD  SYSOUT=*          Batch container trace
//*
//CEEDUMP   DD  SYSOUT=*
//*
```

Figure 6. Example: BCDPROC statement

Chapter 3. Defining connectivity for the database

This chapter describes basic information about setting up the z/OS Batch Runtime environment with the DB2 database and how the processing of transactions works for requests from PL/I, COBOL or Java applications.

Considerations for setting up z/OS Batch Runtime services for a database resource

For the DB2 or database resource that z/OS Batch Runtime uses to make connections for interoperability functions, the database must do the following:

- Initialize the z/OS Batch Runtime environment processing
- End the z/OS Batch Runtime environment processing
- Obtain notification of the start of a global transaction
- Obtain notification of the completion of a global transaction.

DB2 Java Database Connectivity (JDBC) and z/OS Batch Runtime

At startup, the z/OS Batch Runtime calls the Java Database Connectivity (JDBC) driver for DB2 to establish a connection that can then be shared by the PL/I, COBOL or Java applications. The DB2 JDBC detects the mode of z/OS Batch Runtime and creates the single physical attachment for processing applications. JDBC maintains this application attachment for any connection requests that an application makes. The PL/I, COBOL, and Java applications use the same "BatchRuntime" attachment to access the DB2 resources.

Establishing a connection to DB2 from a PL/I or COBOL application usually requires three calls to the RRS Attach Facility (RRSAF):

- IDENTIFY
- SIGNON
- CREATE THREAD

Because the JDBC has created the DB2 resource attachment for the thread during z/OS Batch Runtime initialization, the PL/I or COBOL application must not code these RRSAF calls to initialize or end a DB2 connection; otherwise, RRSAF fails the request. z/OS Batch Runtime performs resource clean up after processing ends for the request.

Transaction management and global transactions

z/OS Batch Runtime performs basic transaction management functions for the application through the Java Transaction API (JTA). It can manage the PL/I, COBOL or Java application clients and can coordinate transaction management between itself and the z/OS RRS transaction management services.

All transactions that run on z/OS Batch Runtime are considered global transactions. z/OS Batch Runtime calls z/OS RRS to start a transaction to associate the transaction with the calling thread before it invokes the PL/I, COBOL or Java application. The JDBC provides the following methods to perform transaction synchronization:

beforeCompletion
Invoked before the transaction process starts

afterCompletion
Invoked after the transaction is performed

The JDBC informs all of the active connections about the DB2 commit or rollback events for consistency in processing database requests. You cannot initiate DB2 commit or rollback requests from the PL/I, COBOL or Java applications themselves. For this release, support for multiple resource managers is not available in z/OS Batch Runtime.

Commit and rollback services of z/OS Batch Runtime

PL/I and COBOL invokes Batch Runtime methods for commit and rollback. For PL/I and COBOL applications, z/OS Batch Runtime offers callable procedures for commit and rollback of a transaction. Before committing the unit of work, z/OS Batch Runtime invokes the beforeCompletion method on the JDBC to indicate the start of the commit. (This in turn invokes the z/OS RRS Commit_UR service to commit the transaction.) After the commit transaction is committed, z/OS Batch Runtime invokes the afterCompletion method on the JDBC to indicate the completion of the commit.

Before processing the rollback transaction, z/OS Batch Runtime invokes the beforeCompletion method on the JDBC to indicate the start of the rollback. (This in turn invokes the z/OS RRS Backout_UR service to back out the transaction.) After the rollback transaction is completed, z/OS Batch Runtime invokes the afterCompletion method on the JDBC to indicate completion of the rollback

End-of-job clean up processing

If the applications complete with no issues, z/OS Batch Runtime commits any outstanding transaction. z/OS Batch Runtime invokes the z/OS RRS end_transaction service to clean up a global transaction. It rolls back any outstanding global transaction and invokes the z/OS Resource Recovery Services (RRS) end_transaction service to pass a rollback action. It also communicates the start and completion of the transaction rollback process.

For additional information about RRS, see *z/OS MVS Programming: Resource Recovery* for topics about:

- Using Resource Recovery Services
- Callable Resource Recovery Services.

Chapter 4. Application interfaces for z/OS Batch Runtime

This section describes the following interfaces, considerations, and samples for z/OS Batch Runtime:

- Configuration options. See "Configuration options reference."
- Helper functions including commit and rollback in Java. See "Helper functions for z/OS Batch Runtime" on page 22.
- Support elements for JDBC and DB2 communications. See"Support elements for JDBC and DB2" on page 23.
- Java environment variables. See "Java environment variables for z/OS Batch Runtime" on page 6.
- Language Environment considerations and restrictions for COBOL and Java applications. See "Language Environment restrictions for z/OS Batch Runtime" on page 24.
- Completion codes. See "Completion codes for z/OS Batch Runtime" on page 24.
- Code examples. See "Example: Java code calling COBOL" on page 33.

Configuration options reference

You can control z/OS Batch Runtime by using configuration options that you specify on the `//BCDIN JCL` statement. This section provides reference information about the supported input parameters. These *keyword=value* pairs are prefixed with '`bcd`'. For a description of the JCL conventions to specify options, see "JCL for BCDIN configurations options" on page 12.

Configuration option types

As Table 3 shows, the syntax of a configuration option varies according to the following types.

Table 3. Configuration option types

Type	Description and Example	Default
Keyword	An option in *keyword=value* format. Values can contain embedded blanks. Trailing blanks are removed. `bcd.applicationLanguage=COBOL`	None
Stem	An option you use to specify multiple values for the option. A numeric suffix (the stem) is appended to the option name and indicates the value number. A stem suffix must be numeric. Values can be skipped and can appear in any order; however, z/OS Batch Runtime processes the stem values in their numeric order. `bcd.supportClass.1=any.class.name`	None

Configuration option names

The following options are read from the `//BCDIN` JCL file. The name, description, and example of the option are provided.

bcd.applicationLanguage=*language*
Names the language of the application to be launched, where *language* is one of the following values:

COBOL
indicates the program is written in COBOL; although this parameter is supported, the LE parameter is recommended

JAVA indicates the program is written in Java

LE indicates the program is written in either COBOL or PL/I.

XJCL indicates the program is XJCL that is provided by the user; the xJCL is read using the //BCDXJCL DD statement.

Default
None; the statement is required.

Example

```
bcd.applicationLanguage=JAVA
```

bcd.applicationName=*application-name*
Names the fully qualified PL/I, COBOL or Java class program name of the application, where *application-name* is the name of the application.

For PL/I or COBOL applications, *application-name* is a 1-8 character module name. The z/OS Batch Runtime uses the typical z/OS LNKLST/STEPLIB search order for locating the COBOL application.

For Java applications, *application-name* is the fully qualified class name. The z/OS Batch Runtime uses the CLASSPATH environment variable to locate the main() method of the specified classname.

Default
None

Example

```
bcd.applicationName=XMPCOBJX
```

bcd.applicationArgs.*n=argument*
Names an argument to be passed to the application, where *n=argument* specifies the suffix number of the argument position.

For Java applications, each argument is passed as an element of the argument array that is passed to the main method.

For PL/I or COBOL applications, you can specify only one argument. The argument can contain a maximum of 100 characters and is passed using the same convention as the `PARM= keyword` of the `// EXEC` JCL statement.

Default
None

Example

```
bcd.applicationArgs.1=java arg1
```

bcd.supportClass.*n=support-class-name*
Names a support class to be used with z/OS Batch Runtime, where *n=support-class-name* specifies a suffix number that indicates the order in which the support class is invoked.

Default
None, but at least one support class is required.

Example

```
bcd.supportClass.1=com.ibm.db2.jcc.t2zos.T2zosBatchRuntimeSupport
```

Note: For DB2, the following support class is provided by the JDBC driver. To use it, you must uncomment the following statement provided in the sample BCDBATCH job.

```
com.ibm.db2.jcc.t2zos.T2zosBatchRuntimeSupport
```

`bcd.verbose=`*`value`*
Specifies the verbose mode for the batch runtime, where *value* is either TRUE or FALSE. z/OS Batch Runtime generates additional diagnostics when you specify TRUE for verbose mode and can affect performance.

`Default`
FALSE

`Example`
`bcd.verbose=true`

`bcd.supportPropertiesDDName.`*`n=ddname`*
References a ddname defining a properties file containing initialization statements for the support class. The properties are in keyword=value format suitable for a Java Properties object.

This statement maps one-to-one with the bcd.supportClass.n statements that define the support class.

In the case of the DB2 JDBC driver, the properties read from this file are the same as those referenced by the info parameter of the DriverManager.getConnection method. Reading the properties from the file is an alternative to specifying them as -D options when invoking Java.

This statement is optional. If not provided for a support class, an initialization file is not read.

`bcd.xJCLEncoding``=`*`encoding-name`*
where encoding-name names an encoding to be used when reading the xJCL defined by the /BCDXJCL DD statement. This statement is optional and defaults to the JVM file.encoding value.

`bcd.xJCLRestartEnabled=`*`boolean`*
Specifies whether the xJCL defined job is restartable. When restart is enabled, the Batch Runtime will update its persistence files upon each checkpoint. If the job subsequently abnormally terminates and is in the restartable state, the job can be restarted and will use the state saved since the last checkpoint. The state includes positioning information for each batch data stream.

true When restart is enabled, the Batch Runtime creates checkpoint files in the file system. By default, the files are created under the user home directory. The directory can be changed using the bcd.xJCLRestartHome option.

false When restart is not enabled, the job cannot be restarted using the Batch Runtime restart facility if it abnormally terminates.

This statement is optional and defaults to false.

`bcd.xJCLRestartHome``=`*`path`*
Names the home directory where restart control files will be placed in the XJCL environment when restart is enabled.

The directory must exist. The files will be deleted when the job terminated successfully.

The default is the working directory for the job as returned by the Java System.getProperty("user.dir") method.

`bcd.xJCLRestartJobId`=*`jobid`*
Names a previously failed xJCL defined job to be restarted. The job must be in the restartable state.

When an xJCL job is submitted, the job Id assigned to the job is recorded in message BCD0310I. Use this jobid as the value of the bcd.xJCLRestartJobId statement when restarting the job.

Program arguments

You can pass program arguments to the PL/I, COBOL or Java main application from z/OS Batch Runtime.

For PL/I or COBOL programs, you can pass a single argument in standard format as it is specified on the PARM= keyword of the `//EXEC` JCL statement. The following statement shows an example:

```
bcd.applicationArgs.1=This is PARM= main arg to Cobol
```

For Java programs, you can pass program arguments as a string array to the Java main method, as shown in the following example; Java main methods accept this as a variable length string array per the usual specified behavior:

```
bcd.applicationArgs.1=500
bcd.applicationArgs.2=string input 1
bcd.applicationArgs.3=My userid
```

You do not have to include a single quote (') in the string value you are passing. Also, note that trailing blanks are not supported in the string.

Helper functions for z/OS Batch Runtime

As part of the interoperability commit and rollback database functions for PL/I, COBOL, and Java applications, z/OS Batch Runtime provides helper functions to simplify the processing.

For Java, methods for commit and rollback functions are available with the following package:

```
com.ibm.batch.spi.UserControlledTransactionHelper
```

Java function for commit and rollback

The following class contains commit and rollback functions for Java applications:

```
com.ibm.batch.spi.UserControlledTransactionHelper
```

The class contains the following static methods that initiate the commit or rollback process:

Commit
`UserControlledTransactionHelper.commit()`

Rollback
`UserControlledTransactionHelper.rollback()`

z/OS Batch Runtime uses z/OS Resource Recovery Services (RRS) to manage the unit of work that is active across the PL/I, COBOL, and Java language boundary. All commits and rollbacks must be managed by z/OS Batch Runtime; your applications should not call commit and rollback directly. Rather, they should use helper functions to call the functions. When your Java application needs to perform a commit or rollback, you must call these helper functions to perform the function. For COBOL applications, you can use the COBOL INVOKE statement to invoke these helper methods.

Direct use of JTA (Java transaction API) by Java programs is not allowed. Also, any use of SQL COMMIT or ROLLBACK APIs by Java or COBOL will be rejected with `SQLSTATE = '2D521 SQL COMMIT or ROLLBACK are invalid in the current operating environment'`. As such, Java programs should avoid setting the JDBC autocommit connection option. See "Code examples" on page 32 for examples.

Support elements for JDBC and DB2

You can use a support element (also referred to as a support class) to allow z/OS Batch Runtime to interoperate with a database or other resource manager.

For this release, the only support element is one that manages the JDBC driver that communicates with DB2. The support element must implement the following interface:

```
com.ibm.zos.zbatch.runtime.support.BCDBatchRuntimeSupport
```

This interface defines the following Java methods:

initializeBatchRuntimeEnv(Properties props)
Initializes z/OS Batch Runtime environment. Startup options are passed in the properties object

terminateBatchRuntimeEnv()
Ends the z/OS Batch Runtime environment.

notifyNewGlobalTransaction(BCDTransaction transaction)
Informs the support element of a new global transaction. The support element of this method calls the following, which z/OS Batch Runtime implements:

```
transaction.registerSynchronization(BCDSynchronization sync)
```

getVersion()
Retrieves a string representation of the version of the support element for diagnostic purposes. The content of the string is determined by the support element.

Transaction and synchronization processing that are normally part of the J2EE javax.transaction package are part of the following z/OS Batch Runtime package:

```
com.ibm.zos.batch.runtime.support.transaction
```

The classes for this package are called:
- BCDTransaction
- BCDSynchronization

In addition, a support element is required to implement a static getInstance() method that returns an instance of the support element class. You must add any .JAR files or DLLs to the CLASSPATH and LIBPATH in the BCDBATCH JCL. For more details, see Chapter 2, "Invoking z/OS Batch Runtime," on page 5.

Language Environment restrictions for z/OS Batch Runtime

Certain restrictions apply to PL/I, COBOL, and Java applications that use the Language Environment in the z/OS Batch Runtime environment.

- COBOL applications must not use the STOP RUN statement. Using the option in COBOL programs prevents the z/OS Batch Runtime environment from receiving control. Instead, use the GOBACK statement to end the COBOL application.
- COBOL will no longer be the first program entered. COBOL-specific runtime options might be affected.
- Java applications must be single threaded and must not use the *system.exit* method. Using the *system.exit* method ends the JVM and prevents the z/OS Batch Runtime environment from receiving control. Instead, end the main Java procedure with a simple return statement.

Completion codes for z/OS Batch Runtime

Upon completion, z/OS Batch Runtime processing returns the condition codes shown in Table 4.

Table 4. Completion codes for z/OS Batch Runtime

Code	Description
00	The processing has successfully completed.
08	The processing has launched the application, but the application has returned a non-zero condition code. See the z/OS Batch Runtime messages in the job log for errors.
12	An error occurred during z/OS Batch Runtime processing. See the z/OS Batch Runtime messages in the job log for errors and *z/OS MVS System Messages, Vol 3 (ASB-BPX)* for more information.

Chapter 5. Application structure and build considerations for COBOL and Java

The following sections describe considerations for structuring and building COBOL and Java applications when using z/OS Batch Runtime.

DLL considerations for COBOL and Java

In effort to simplify, some information from *Enterprise COBOL for z/OS, V4R2, Programming Guide, SC23-8529* is repeated in this section of the documentation. For complete details, see *Enterprise COBOL for z/OS, V4R2, Programming Guide* at `http://publibfp.boulder.ibm.com/epubs/pdf/igy3pg50.pdf`

It is important to recognize the structural implications to COBOL source files when they are calling out to Java. In particular, you need DLL and RECURSIVE on COBOL classes and methods or on COBOL programs that invoke Java methods.

To compile COBOL source code that contains OO syntax, such as INVOKE statements or class definitions, or that use Java services, you must use these compiler options:
- RENT
- DLL
- THREAD

Any programs that you compile with the THREAD compiler option must be recursive. You must specify the recursive clause in the PROGRAM-ID paragraph of each OO COBOL client program. This can affect the overall COBOL program structure because a program compiled with a DLL cannot make a traditional COBOL dynamic call. It can, however, be statically linked with and call into another COBOL program compiled dynamic. This separate but statically linked program can then use a traditional dynamic call to other external COBOL modules with built dynamic programs.

In general, DLL linkage built COBOL programs can only call out to other external DLL linkage built programs. Similarly, dynamic call built COBOL programs can only call out to other external dynamic call built programs. However, static linking of objects with either two of these external program call mechanisms is allowed. This provides the bridging between the DLL linkage that Java requires and the traditional COBOL dynamic call.

For additional details, see the topic about "Using DLL linkage and dynamic calls together " in *Enterprise COBOL for z/OS, V4R2, Programming Guide*.

Example of a COBOL COMMIT wrapper

Figure 7 on page 26 is a simple example of a COBOL COMMIT wrapper that, while compiled with the DLL option required for Java, can be statically linked with a main non-DLL application module. In this example, a procedural COBOL program invokes a Java method. Only the non-DLL module objects that need to call the new COMMIT function need to be recompiled. You can also perform a similar function for ROLLBACK.

```
*----------------------------------------------------------------
*
* Program Name   : COBCOMIT
* Objective      : Call RSS global commit via batch container
*
*----------------------------------------------------------------
 IDENTIFICATION DIVISION.
 PROGRAM-ID. "COBCOMIT" IS RECURSIVE.

/
 ENVIRONMENT DIVISION.
*--------------------
 CONFIGURATION SECTION.
 REPOSITORY.
     Class JavaException is "java.lang.Exception"
     Class UserControlledTransaction is
           "com.ibm.batch.spi.UserControlledTransactionHelper".
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.

 DATA DIVISION.
 FILE SECTION.

 WORKING-STORAGE SECTION.
 01 ex object reference JavaException.

 LINKAGE SECTION.
 01 RETCODE PIC S9(9) USAGE IS BINARY.
 COPY JNI.

 PROCEDURE DIVISION RETURNING RETCODE.
*----------------------------------------------------------------
*
*    Test batch cobol commit.
*
*----------------------------------------------------------------
 PROGRAM-BEGIN.
     SET ADDRESS OF JNIENV TO JNIENVPTR
     SET ADDRESS OF JNINATIVEINTERFACE TO JNIENV
     Display "Calling into Java commit"
     Invoke UserControlledTransaction "commit"
     Display "Returned from Java commit"
     Perform ErrorCheck
     Goback
     .
 PROGRAM-END.
     GOBACK.
* need to perform exception check / stack trace at this point ?

ErrorCheck.
    Compute RETCODE = 0
    Call ExceptionOccurred
         using by value JNIEnvPtr
         returning ex
    If ex not = null then
         Call ExceptionClear using by value JNIEnvPtr
         Display "Caught an unexpected exception"
         Invoke ex "printStackTrace"
         Compute RETCODE = 99
    End-if
     .
  End program "COBCOMIT".
```

Figure 7. Example: COBOL COMMIT wrapper

Figure 8 on page 27 shows an example of the JCL that would be needed to compile the COMMIT wrapper shown in Figure 7.

```
//COMPCMIT JOB ,'STEVE PROGRAM ',
//            NOTIFY=&SYSUID,
//            MSGCLASS=X,
//            CLASS=A,
//            REGION=0M,
//            TIME=120
//COMPILE  EXEC IGYWC,LNGPRFX='SYSPROG.MNT.COBOL42',
//        COND=(4,LT),
//        PARM.COBOL=(NOSEQUENCE,RENT,LIB,THREAD,
//        NODYNAM,DLL)
//COBOL.SYSLIB DD DSN=SUIMGJB.PRIVATE.JNI.COPY,
//            DISP=SHR
//COBOL.SYSIN  DD DSN=SUIMGJB.PRIVATE.DOCXMP.COBOL(COBCOMIT),
//            DISP=SHR
//COBOL.SYSLIN DD DSN=SUIMGJB.PRIVATE.COBOL.OBJ(COBCOMIT),
//            DISP=SHR
//
```

Figure 8. Example: JCL used to compile COMMIT wrapper

Using the bcdcommit() and bcdrollback() helpers from COBOL

The bcdcommit() and bcdrollback() helpers can be used from a COBOL application, although there is not as much advantage as when called from PL/I. Since COBOL supports the “invoke” statement, the batch container commit and rollback routines can be called directly with this verb. However, this does a serve again as a simple model for building any COBOL calling into the Java environment -- including source, compile, and bind options.

Figure 9 on page 28 shows a COBOL application calling the bcdcommit() and bcdrollback() helpers.

```
PROGRAM-ID. 'COBHELP' RECURSIVE.

ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.

DATA DIVISION.
FILE SECTION.

WORKING-STORAGE SECTION.
77  RC    PIC 9(4) USAGE IS BINARY.

LINKAGE SECTION.
PROCEDURE DIVISION.

PROGRAM-BEGIN.

    DISPLAY 'Starting COBHELP ...'.

    Display 'Calling bcdcommit ...';
    Call 'bcdcommit' returning rc.
    Display 'bcdcommit rc=' rc.

    Display 'Calling bcdrollback ...';
    Call 'bcdrollback' returning rc.
    Display 'bcdrollback rc=' rc.

PROGRAM-END.
      GOBACK.
```

Figure 9. Sample COBOL Calling bcdcommit() and bcdrollback() Helpers

Figure 10 on page 29 is sample JCL to compile the COBOL program in Figure 9.

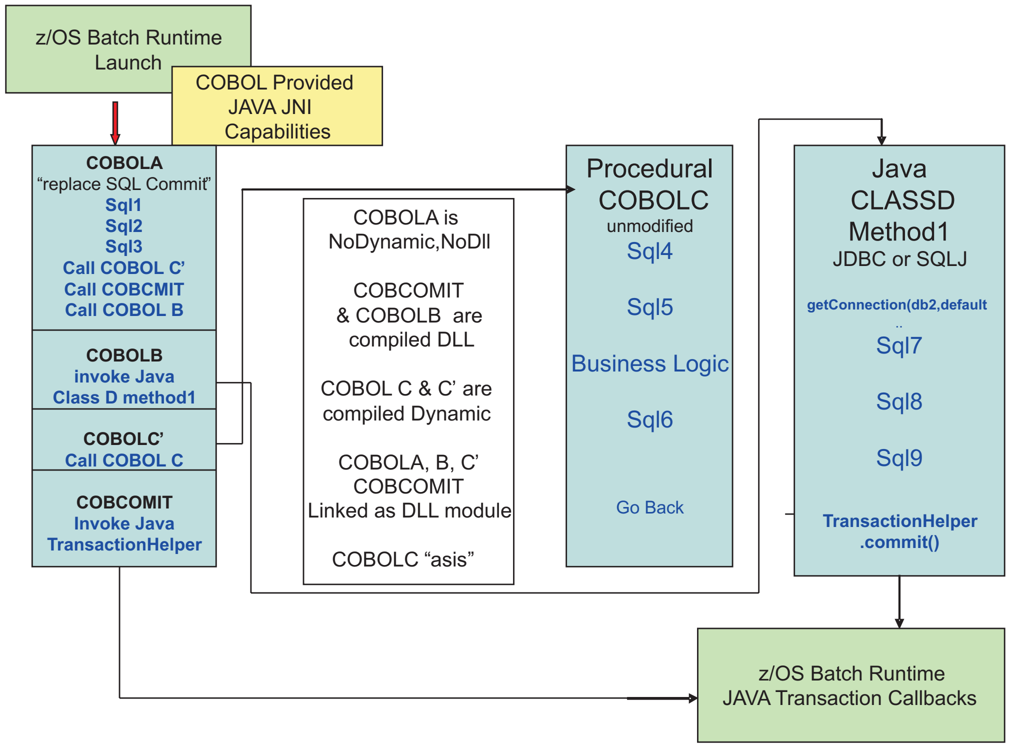
```
//jobname JOB  (1)
//*
//JCLLIB   JCLLIB ORDER=COB.COBOL42.SIGYPROC
//*
//STEP1    EXEC  IGYWCL,REGION=0M,
//       LNGPRFX='COB.COBOL42',
//       PARM.COBOL=('OPTFILE'),
//       PARM.LKED=('OPTIONS=OPTS')
//COBOL.SYSOPTF  DD  *
 MAP,
 NOOPT,
 SZ(MAX),
 DLL,
 NODYNAM,
 THREAD,
 PGMNAME(LONGMIXED),
 NOTERM,
 DATA(31),
 LIB,
 LIST,
 XREF,
 SOURCE
//COBOL.SYSIN  DD  DSN=IBMUSER.BATCH.SOURCE(COBHELP),DISP=SHR
//*
//LKED.SYSLMOD DD  DSN=IBMUSER.BATCH.LOAD,DISP=SHR
//LKED.OPTS    DD  *
MAP
RENT
DYNAM=DLL
CASE=MIXED
LIST=ALL
XREF
//LKED.SYSIN   DD  *
 INCLUDE '/usr/lpp/bcp/lib/libbcduser.x'
 NAME COBHELP(R)
```

Figure 10. Sample COBOL Compile and Bind JCL for bcdcommit() and bcdrollback() Helpers

Examples of program structures

This section demonstrates several types of program structures and interaction between COBOL Java, and z/OS Batch Runtime.

Figure 11 on page 30 shows an overview of a COBOL program that interacts with a Java program. In this example, the program flow starts in COBOL and then flows to a Java program and to another COBOL program. OOCOBOL methods are not used; however, the programs use both COBOL JNI and user JNI.



z/OS Batch Runtime Launch
COBOL Provided JAVA JNI Capabilities
COBOLA
"replace SQL Commit"
Sql1
Sql2
Sql3
Call COBOL C'
Call COBCMIT
Call COBOL B
COBOLB
invoke Java Class D method1
COBOLC'
Call COBOL C
COBCOMIT
Invoke Java TransactionHelper
COBOLA is NoDynamic,NoDll
COBCOMIT & COBOLB are compiled DLL
COBOL C & C' are compiled Dynamic
COBOLA, B, C' COBCOMIT Linked as DLL module
COBOLC "asis"
Procedural COBOLC
unmodified
Sql4
Sql5
Business Logic
Sql6
Go Back
Java CLASSD Method1
JDBC or SQLJ
getConnection(db2,default
Sql7
Sql8
Sql9
TransactionHelper .commit()
z/OS Batch Runtime JAVA Transaction Callbacks

Figure 11. Example: COBOL program calling Java and unmodified COBOL

In Figure 12, a Java program flows to a COBOL program. In this example, the Java program uses an OOCOBOL factory wrapper to call COBOL.

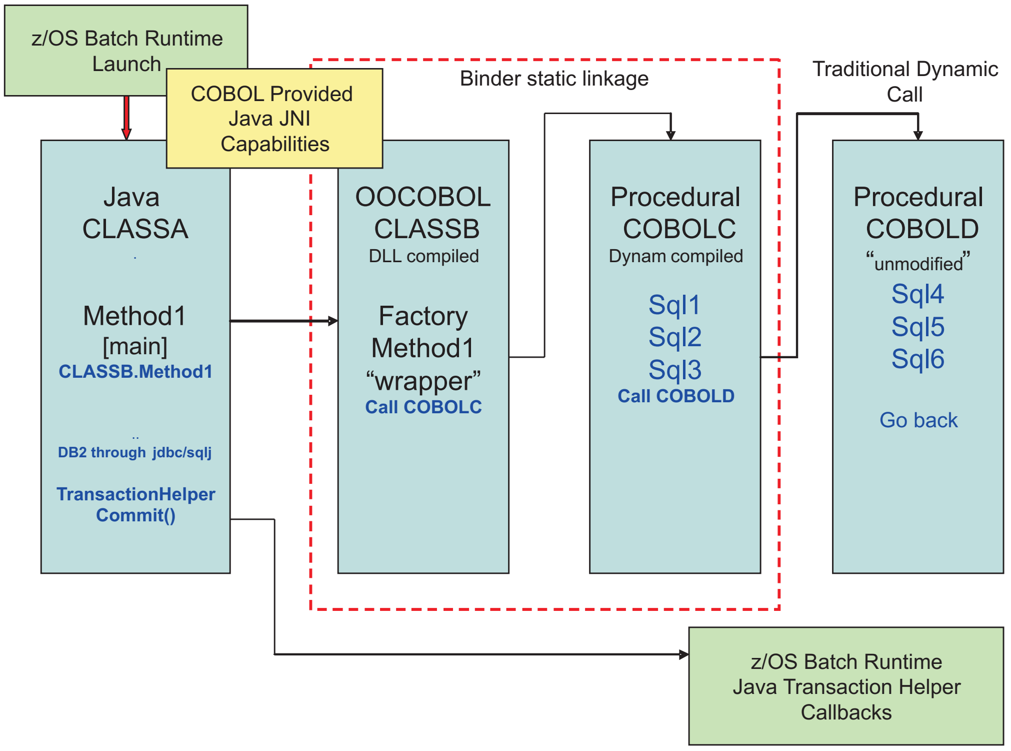


Figure 12. Example: Java program using OOCOBOL to call COBOL

Building programs: compile and link JCL examples

For complete documentation about building COBOL applications, including Object Oriented (OO) COBOL, see *Enterprise COBOL for z/OS, V4R2, Programming Guide*.

For compiling with JCL, IBM provides a set of cataloged procedures to reduce the amount of JCL coding that you need to write. If the cataloged procedures do not meet your needs, you can write your own JCL. Using JCL, you can compile a single program or compile several programs as part of a batch job. See Chapter 2, "Invoking z/OS Batch Runtime," on page 5 for more information.

The compiler translates your COBOL program into language that the computer can process (object code). The compiler also lists errors in your source statements and provides supplementary information to help you debug and tune your program. Use compiler-directing statements and compiler options to control your compilation. After compiling your program, you need to review the results of the compilation and correct any compiler-detected errors.

To build Java programs, use the `javac` command to create the classes and the `jar` command for packaging. This documentation focuses on building a typical use case that updates a traditional COBOL program to call out to Java methods in which either or both can use DB2.

The JCL example shown in Figure 13 on page 32 is a modification of a sample COBOL DB2 phone program that ships as part of IBM DB2 for z/OS. This program is typically found in `*hlq*.sdsnsamp(DSN8BC3)` and is often used in the DB2 installation verification program (IVP). The COBOL source is modified to invoke a simple Java "Hello World" method that also selects rows from the DB2 catalog using the SYSIBM schema. The following are implications on the DB2 provided COBOL build procedure to run in the z/OS Batch Runtime container:

- The Language Environment Runtime library CEE.SCEERUN must be in the JOBLIB for the Java JNI support.
- The ATTACH(RRSAF) must be in the preprocessor portion of the catalogued procedure. Although optional, this forces the generation of RRS attach entry point at compile time. Omit this option for attach-neutral code generation. The z/OS Batch Runtime requires the use of RRS attach to be bound at compile (as in this example), link (include `DSNRLI`), or runtime (include `DSNULI`).
- The use of Java from COBOL source requires the compile options `RENT,DLL,THREAD`.
- The long names required for the Java JNI imply use of PDSE libraries by the binder (rather than traditional PDS load libraries).
- The input to the Binder requires both the Enterprise COBOL Java linkage and JNI export (*.x) files.

```
//COBBUILD   JOB (MOP,1458),'STEVE',CLASS=A,REGION=0M,
//     MSGLEVEL=(1,1),MSGCLASS=X,TIME=1440,NOTIFY=&SYSUID
//*
//*********************************************************************
//*  NAME = DSNTEJ2C -- MODIFIED FOR RRS AND Java BATCH CONTAINER RUN  *
//*                     BUILD ONLY WITH APP CALL TO JAVA               *
//*  DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION W CALL TO JAVA          *
//*                     PHASE 2                                        *
//*                     COBOL                                          *
//*                                                                    *
//*    LICENSED MATERIALS - PROPERTY OF IBM                            *
//*    5635-DB2                                                        *
//*    (C) COPYRIGHT 1982, 2006 IBM CORP.       *
//*                                                                    *
//*    STATUS = VERSION 9                                              *
//*                                                                    *
//*  FUNCTION = THIS JCL PERFORMS THE PHASE 2 COBOL SETUP FOR THE      *
//*             SAMPLE APPLICATIONS.  IT PREPARES AND EXECUTES         *
//*             COBOL BATCH PROGRAMS.                                  *
//*                                                                    *
//*             THIS JOB IS RUN AFTER PHASE 1.                         *
//*                                                                    *
//*                                                                    *
//* CHANGE ACTIVITY =                                                  *
//*                                                                    *
//*********************************************************************
//JOBLIB  DD  DISP=SHR,DSN=DSN910.NEWFUNC.SDSNEXIT
//        DD  DISP=SHR,DSN=DSN910.SDSNLOAD
//        DD  DISP=SHR,DSN=CEE.SCEERUN
//*
//*        PREPARE COBOL PHONE PROGRAM
//PH02CS03 EXEC DSNHNCOB,MEM=XMPCOBJV,
//         COND=(4,LT),
//         PARM.PC=('HOST(IBMCOB)',APOST,APOSTSQL,SOURCE,
//         NOXREF,'SQL(DB2)','DEC(31)','ATTACH(RRSAF)'),
//         PARM.COB=(NOSEQUENCE,LIB,QUOTE,RENT,'PGMNAME(LONGUPPER)',
//         DLL,THREAD)
//PC.DBRMLIB   DD DSN=DSN910.DBRMLIB.DATA(XMPCOBJV),
//             DISP=SHR
//PC.SYSLIB    DD DSN=SUIMGJB.PRIVATE.DSN910.SRCLIB.DATA,
//             DISP=SHR
//PC.SYSIN     DD DSN=SUIMGJB.PRIVATE.JCL.CNTL(XMPCOBJV),
//             DISP=SHR
//COB.SYSLIB   DD DSN=SUIMGJB.PRIVATE.JNI.COPY,
//             DISP=SHR
//LKED.SYSLMOD DD DSN=SUIMGJB.PRIVATE.LIBRARY(XMPCOBJV),
//             DISP=SHR
//LKED.RUNLIB  DD DSN=DSN910.RUNLIB.LOAD,
//             DISP=SHR
//LKED.SYSIN   DD *
     INCLUDE SYSLIB(DSNRLI)
     INCLUDE RUNLIB(DSN8MCG)
     INCLUDE '/home/cob42/cobol/lib/igzcjava.x'
     INCLUDE '/usr/lpp/java/J6.0/lib/s390/j9vm/libjvm.x'
//
```

Figure 13. Example: JCL for COBOL DB2 phone program

Code examples

This section contains the following code examples:

- "Example: Java code calling COBOL" on page 33
- "Example: C DLL calling COBOL from Java" on page 34
- "Example: COBOL code invoking Java" on page 35

Example: Java code calling COBOL

Figure 14 shows an example of Java code calling COBOL.

```
package com.ibm.zos.batch.container.test;

import java.sql.*;
import com.ibm.batch.spi.UserControlledTransactionHelper;
import com.ibm.ws.gridcontainer.exceptions.TransactionManagementException;

public class Sample
{
  //Native method declaration
  private native int CallCOBOL();
  //Load the library
  static {
  System.loadLibrary("c_to_cobol");
}

public static void main(String[] args)
{
Connection conn = DriverManager.getConnection(url);
  String url = "jdbc:default:connection";

  Statement stmt;
  int maxRows = 25;
  String pnumber = "";
  int pnum = 0;
  int rc = 0;
  String formatted;

  try
    {
      System.out.println ( "Establishing Connection to URL: " + url );

      conn = DriverManager.getConnection(url);
      System.out.println ( " successful connect" );
      stmt = conn.createStatement();
      System.out.println ( " Successful creation of Statement" );
      // Limit the number of rows to return
      stmt.setMaxRows ( maxRows );
```

Figure 14. Example: Java code calling COBOL (Part 1 of 2)

```
    // SELECT from an DB2 sample table
    String sqlText =
    "SELECT PHONENUMBER " +
    "FROM DSN8910.VEMPLP " +
    "WHERE EMPLOYEENUMBER = '000260'";
    ResultSet results = stmt.executeQuery ( sqlText );
    pnumber = results.getString ( "PHONENUMBER" );
    pnum = Integer.parseInt(pnumber.trim());
    pnum++;
    pnum = pnum % 10000;
    formatted = String.format("%04d", pnum);

    sqlText =
    "UPDATE DSN8910.VEMPLP " +
    " SET PHONENUMBER = " + "'"+formatted+"'" +
    " WHERE EMPLOYEENUMBER = '000260' ";
    int updateCount = stmt.executeUpdate(sqlText);
    System.out.println ( "Successful execution of UPDATE. Rows updated= " + updateCount );

    // close ResultSet and Statement
    results.close();
    // Call COBOL via a C DLL
    Sample call_cobol = new Sample();
    //Call native method
    rc = call_cobol.CallCOBOL();
    System.out.println ( "Returned from COBOL with a rc: " + rc );

    if (rc == 0)
      {
        try
        {
          UserControlledTransactionHelper.commit();
        }
        catch (TransactionManagementException e)
        {
          e.printStackTrace();
        }
      }
    else
      {
        try
        {

        UserControlledTransactionHelper.rollback();
        }
        catch (TransactionManagementException e)
        {
          e.printStackTrace();
        }
      }
    }
    catch (SQLException ex)
      {
        System.out.println("SQLException information");
        while(ex!=null) {
        System.err.println ("Error msg: " + ex.getMessage());
        System.err.println ("SQLSTATE: " + ex.getSQLState());
        System.err.println ("Error code: " + ex.getErrorCode());
        ex.printStackTrace();
        ex = ex.getNextException();
      }
    }
  }
}
```

Figure 15. Example: Java code calling COBOL (Part 2 of 2)

Example: C DLL calling COBOL from Java

The example in Figure 16 on page 35 shows the C interface DLL to use when calling COBOL.

```
/ c99  -o libc_to_cobol.so -Wc,exportall -Wl,
dll -I/usr/lpp/java/J6.0.1/include
    -I/usr/lpp/java/J6.0.1/include/zos c_to_cobol.c

#include <jni.h>
#include <sys/types.h>
#include <stdio.h>
#include <stdlib.h>
#include "com_ibm_zos_batch_container_test_Java_Calls_Cobol.h"

void (*fetch(const char *name))();
typedef void cfunc();

JNIEXPORT jint JNICALL
Java_com_ibm_zos_batch_container_test_Java_1Calls_1Cobol_CallCOBOL(JNIEnv * jenv, jobject jobj)
{
 cfunc *cobfetch_ptr;
 cobfetch_ptr = (cfunc *) fetch("XMPCOBJ3");    // loads fetched module
 if (cobfetch_ptr == NULL){
     printf("\tfetch failed\n");
 }
 else
 {
    printf("\tShould be going off to COBOL\n\n");
    (*cobfetch_ptr)();         // sets up the proper linkage for the call
 }

 return(0);
}
```

Figure 16. Example: C interface DLL for calling COBOL from Java

Example: COBOL code invoking Java

Figure 17 on page 36 is an example of a modified DB2 sample phone application that uses COBOL code to invoke the "sayHello" Java method. Descriptions for each of the code blocks precede the example.

Figure 17 on page 36 includes changes that were made in the sample program to provide an interface to Java. These changes are **highlighted** and are located in the following areas of the example:

A Identification Division

B Environment Division

C Linkage Section

D Main Program Routine

E Updates Phone Numbers For Employees

F Perform Rollback

G Java Exception Check

Note: This sample was provided by DB2 , typically in *hlq*.sdsnsamp(DSN8BC3). For more details, see http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.db29.doc/db2prodhome.htm

```
    IDENTIFICATION DIVISION.
    *-----------------------
A   PROGRAM-ID. DSN8BC3 RECURSIVE.

    ****** DSN8BC3 - DB2 SAMPLE PHONE APPLICATION - COBOL - BATCH ***
    * *
    * MODULE NAME = DSN8BC3 *
    * *
    * DESCRIPTIVE NAME = DB2 SAMPLE APPLICATION *
    * PHONE APPLICATION *
    * BATCH *
    * COBOL *
    * *
    *LICENSED MATERIALS - PROPERTY OF IBM *
    *5695-DB2 *
    *(C) COPYRIGHT 1982, 1995 IBM CORP. *
    * * *-----------------------------------------------------------------*
    /

    ENVIRONMENT DIVISION.
    *--------------------
    CONFIGURATION SECTION.
    SPECIAL-NAMES. C01 IS TO-TOP-OF-PAGE.
    REPOSITORY.
B   Class HelloJ is
    "com.ibm.zos.batch.container.test.HelloJ"
    Class JavaException is "java.lang.Exception"
    Class BCDTranHelper is
    "com.ibm.batch.spi.UserControlledTransactionHelper".
    INPUT-OUTPUT SECTION.
    FILE-CONTROL.
    SELECT CARDIN
    ASSIGN TO DA-S-CARDIN.
    SELECT REPOUT
    ASSIGN TO UT-S-REPORT.

    DATA DIVISION.
    *-------------
    FILE SECTION.
    FD CARDIN
    RECORD CONTAINS 80 CHARACTERS
    BLOCK CONTAINS 0 RECORDS
    LABEL RECORDS ARE OMITTED.
    01 CARDREC PIC X(80).

    FD REPOUT
    RECORD CONTAINS 120 CHARACTERS
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS REPREC.
    01 REPREC PIC X(120).
```

Figure 17. Example: COBOL DB2 phone application that invokes Java under z/OS Batch Runtime (Part 1 of 10)

```
/
WORKING-STORAGE SECTION.

******************************************************
* STRUCTURE FOR INPUT *
******************************************************
01 IOAREA.
02 ACTION PIC X(01).
02 LNAME PIC X(15).
02 FNAME PIC X(12).
02 ENO PIC X(06).
02 NEWNO PIC X(04).
02 FILLER PIC X(42).

01 ex object reference JavaException.
******************************************************
* REPORT HEADER STRUCTURE *
******************************************************
01 REPHDR1.
02 FILLER PIC X(29)
VALUE '-----------------------------'.
02 FILLER PIC X(21)
VALUE ' TELEPHONE DIRECTORY '.
02 FILLER PIC X(29)
VALUE '-----------------------------'.
01 REPHDR2.
02 FILLER PIC X(09) VALUE 'LAST NAME'.
02 FILLER PIC X(07) VALUE SPACES.
02 FILLER PIC X(10) VALUE 'FIRST NAME'.
02 FILLER PIC X(03) VALUE SPACES.
02 FILLER PIC X(08) VALUE 'INITIAL'.
02 FILLER PIC X(07) VALUE 'PHONE'.
02 FILLER PIC X(09) VALUE 'EMPLOYEE'.
02 FILLER PIC X(05) VALUE 'WORK'.
02 FILLER PIC X(04) VALUE 'WORK'.
01 REPHDR3.
02 FILLER PIC X(37) VALUE SPACES.
02 FILLER PIC X(07) VALUE 'NUMBER'.
02 FILLER PIC X(09) VALUE 'NUMBER'.
02 FILLER PIC X(05) VALUE 'DEPT'.
02 FILLER PIC X(05) VALUE 'DEPT'.
02 FILLER PIC X(04) VALUE 'NAME'.

******************************************************
* REPORT STRUCTURE *
******************************************************
01 REPDATA.
02 RLNAME PIC X(15).
02 FILLER PIC X(01) VALUE SPACES.
02 RFNAME PIC X(12).
02 FILLER PIC X(04) VALUE SPACES.
02 RMIDINIT PIC X(01).
02 FILLER PIC X(04) VALUE SPACES.
02 RPHONE PIC X(04).
02 FILLER PIC X(03) VALUE SPACES.
02 REMPNO PIC X(06).
02 FILLER PIC X(03) VALUE SPACES.
02 RDEPTNO PIC X(03).
02 FILLER PIC X(02) VALUE SPACES.
02 RDEPTNAME PIC X(36).
```

Figure 18. Example: COBOL DB2 phone application that invokes Java under z/OS Batch Runtime (Part 2 of 10)

```
*****************************************************
* WORKAREAS *
*****************************************************
01 LNAME-WORK.
49 LNAME-WORKL PIC S9(4) COMP.
49 LNAME-WORKC PIC X(15).
01 FNAME-WORK.
49 FNAME-WORKL PIC S9(4) COMP.
49 FNAME-WORKC PIC X(12).
77 INPUT-SWITCH PIC X VALUE 'Y'.
88 NOMORE-INPUT VALUE 'N'.
77 NOT-FOUND PIC S9(9) COMP VALUE +100.
*****************************************************
* VARIABLES FOR ERROR-HANDLING *
*****************************************************
01 ERROR-MESSAGE.
02 ERROR-LEN PIC S9(4) COMP VALUE +960.
02 ERROR-TEXT PIC X(120) OCCURS 10 TIMES
INDEXED BY ERROR-INDEX.
77 ERROR-TEXT-LEN PIC S9(9) COMP VALUE +120.

77 W09-WAIT-TIME PIC S9(8) COMP VALUE 0005.
77 W09-RESPONSE PIC S9(8) COMP VALUE 0000.

*****************************************************
* SQL INCLUDE FOR SQLCA *
*****************************************************
EXEC SQL INCLUDE SQLCA END-EXEC.

*****************************************************
* SQL DECLARATION FOR VIEW VPHONE *
*****************************************************
EXEC SQL DECLARE DSN8910.VPHONE TABLE
(LASTNAME VARCHAR(15) NOT NULL,
FIRSTNAME VARCHAR(12) NOT NULL,
MIDDLEINITIAL CHAR(01) NOT NULL,
PHONENUMBER CHAR(04) ,
EMPLOYEENUMBER CHAR(06) NOT NULL,
DEPTNUMBER CHAR(03) NOT NULL,
DEPTNAME VARCHAR(36) NOT NULL)
END-EXEC.

*****************************************************
* STRUCTURE FOR PHONE RECORD *
*****************************************************
01 PPHONE.
02 LASTNAME.
49 LASTNAMEL PIC S9(4) COMP.
49 LASTNAMEC PIC X(15) VALUE SPACES.
02 FIRSTNAME.
49 FIRSTNAMEL PIC S9(4) COMP.
49 FIRSTNAMEC PIC X(12) VALUE SPACES.
02 MIDDLEINITIAL PIC X(01).
02 PHONENUMBER PIC X(04).
02 EMPLOYEENUMBER PIC X(06).
02 DEPTNUMBER PIC X(03).
02 DEPTNAME.
49 DEPTNAMEL PIC S9(4) COMP.
49 DEPTNAMEC PIC X(36) VALUE SPACES.
*
77 PERCENT-COUNTER PIC S9(4) COMP.
```

Figure 19. Example: COBOL DB2 phone application that invokes Java under z/OS Batch Runtime (Part 3 of 10)

```
*****************************************************
* SQL DECLARATION FOR VIEW VEMPLP *
*****************************************************
EXEC SQL DECLARE DSN8910.VEMPLP TABLE
(EMPLOYEENUMBER CHAR(06) NOT NULL,
PHONENUMBER CHAR(04) )
END-EXEC.
*****************************************************
* SQL CURSORS *
*****************************************************
*** CURSOR LISTS ALL EMPLOYEE NAMES

EXEC SQL DECLARE TELE1 CURSOR FOR
SELECT *
FROM DSN8910.VPHONE
END-EXEC.

*** CURSOR LISTS ALL EMPLOYEE NAMES WITH A PATTERN (%) OR (_)
*** FOR LAST NAME

EXEC SQL DECLARE TELE2 CURSOR FOR
SELECT *
FROM DSN8910.VPHONE
WHERE LASTNAME LIKE :LNAME-WORK
AND FIRSTNAME LIKE :FNAME-WORK
END-EXEC.

*** CURSOR LISTS ALL EMPLOYEES WITH A SPECIFIC
*** LAST NAME

EXEC SQL DECLARE TELE3 CURSOR FOR
SELECT *
FROM DSN8910.VPHONE
WHERE LASTNAME = :LNAME
AND FIRSTNAME LIKE :FNAME-WORK
END-EXEC.
/
/****************************************************
* FIELDS SENT TO MESSAGE ROUTINE *
*****************************************************
01 MAJOR PIC X(07) VALUE 'DSN8BC3'.

01 MSGCODE PIC X(4).

01 OUTMSG PIC X(69).

01 MSG-REC1.
02 OUTMSG1 PIC X(69).
02 RETCODE PIC S9(9).

01 MSG-REC2.
02 OUTMSG2 PIC X(69).

C LINKAGE SECTION.
COPY JNI.

PROCEDURE DIVISION.
*------------------
```

Figure 20. Example: COBOL DB2 phone application that invokes Java under z/OS Batch Runtime (Part 4 of 10)

```
*****************************************************
* SQL RETURN CODE HANDLING *
*****************************************************
EXEC SQL WHENEVER SQLERROR GOTO DBERROR END-EXEC.
EXEC SQL WHENEVER SQLWARNING GOTO DBERROR END-EXEC.
EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.

*****************************************************
* MAIN PROGRAM ROUTINE *
*****************************************************
PROG-START.
MOVE 0 to RETURN-CODE.
SET ADDRESS OF JNIENV TO JNIENVPTR
SET ADDRESS OF JNINATIVEINTERFACE TO JNIENV
```
D
```
Invoke HelloJ "sayHello"
Display "Returned from Java sayHello to MAIN"
Perform ErrorCheck
* **OPEN FILES
OPEN INPUT CARDIN
OUTPUT REPOUT.

* **GET FIRST INPUT
READ CARDIN RECORD INTO IOAREA
AT END MOVE 'N' TO INPUT-SWITCH.

* **MAIN ROUTINE
PERFORM PROCESS-INPUT
UNTIL NOMORE-INPUT.
PROG-END.
* **CLOSE FILES
CLOSE CARDIN
REPOUT.

GOBACK.

*****************************************************
* CREATE REPORT HEADING *
* SELECT ACTION *
*****************************************************
PROCESS-INPUT.
* **PRINT HEADING
WRITE REPREC FROM REPHDR1
AFTER ADVANCING TO-TOP-OF-PAGE.
WRITE REPREC FROM REPHDR2
AFTER ADVANCING 2 LINES.
WRITE REPREC FROM REPHDR3.

* **SELECT ACTION
IF ACTION = 'L'
PERFORM LIST-FUNCTION
ELSE
IF ACTION = 'U'
PERFORM TELEPHONE-UPDATE
```

Figure 21. Example: COBOL DB2 phone application that invokes Java under z/OS Batch Runtime (Part 5 of 10)

```
ELSE
* **INVALID REQUEST
* **PRINT ERROR MESSAGE
MOVE '068E' TO MSGCODE
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG
MOVE OUTMSG TO OUTMSG2
WRITE REPREC FROM MSG-REC2
AFTER ADVANCING 2 LINES.
READ CARDIN RECORD INTO IOAREA
AT END MOVE 'N' TO INPUT-SWITCH.
/
*****************************************************
* DETERMINE FORM OF NAME USED TO LIST EMPLOYEES *
*****************************************************
LIST-FUNCTION.
* **NO LAST NAME GIVEN
IF LNAME = SPACES
MOVE '%' TO LNAME.
* **NO FIRST NAME GIVEN
IF FNAME = SPACES
MOVE '%' TO FNAME.
* **LIST ALL EMPLOYEES
IF LNAME = '*'
PERFORM LIST-ALL
ELSE
* **UNSTRING LAST NAME
UNSTRING LNAME
DELIMITED BY SPACE
INTO LNAME-WORKC
COUNT IN LNAME-WORKL
* **UNSTRING FIRST NAME
UNSTRING FNAME
DELIMITED BY SPACE
INTO FNAME-WORKC
COUNT IN FNAME-WORKL
* **COUNT %'S
MOVE ZERO TO PERCENT-COUNTER
INSPECT LNAME
TALLYING PERCENT-COUNTER FOR ALL '%'
IF PERCENT-COUNTER > ZERO
* **IF NO %'S THEN
* **LIST SPECIFIC NAME(S)
* **ELSE
* **LIST GENERIC NAME(S)
PERFORM LIST-GENERIC
ELSE
PERFORM LIST-SPECIFIC.
/
*****************************************************
* LIST ALL EMPLOYEES *
*****************************************************
LIST-ALL.
* **OPEN CURSOR
EXEC SQL OPEN TELE1 END-EXEC
* **GET EMPLOYEES
EXEC SQL FETCH TELE1 INTO :PPHONE END-EXEC.
```

Figure 22. Example: COBOL DB2 phone application that invokes Java under z/OS Batch Runtime (Part 6 of 10)

```
IF SQLCODE = NOT-FOUND
* **NO EMPLOYEE FOUND
* **PRINT ERROR MESSAGE
MOVE '008I' TO MSGCODE
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG
MOVE OUTMSG TO OUTMSG2
WRITE REPREC FROM MSG-REC2
AFTER ADVANCING 2 LINES
ELSE
* **LIST ALL EMPLOYEES
PERFORM PRINT-AND-GET1
UNTIL SQLCODE IS NOT EQUAL TO ZERO.

* **CLOSE CURSOR
EXEC SQL CLOSE TELE1 END-EXEC.

PRINT-AND-GET1.
PERFORM PRINT-A-LINE.
EXEC SQL FETCH TELE1 INTO :PPHONE END-EXEC.
/
*****************************************************
* LIST GENERIC EMPLOYEES *
*****************************************************
LIST-GENERIC.
* **OPEN CURSOR
EXEC SQL OPEN TELE2 END-EXEC.

* **GET EMPLOYEES
EXEC SQL FETCH TELE2 INTO :PPHONE END-EXEC.

IF SQLCODE = NOT-FOUND
* **NO EMPLOYEE FOUND
* **PRINT ERROR MESSAGE
MOVE '008I' TO MSGCODE
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG
MOVE OUTMSG TO OUTMSG2
WRITE REPREC FROM MSG-REC2
AFTER ADVANCING 2 LINES
ELSE
* **LIST GENERIC EMPLOYEE(S)
PERFORM PRINT-AND-GET2
UNTIL SQLCODE IS NOT EQUAL TO ZERO.

* **CLOSE CURSOR
EXEC SQL CLOSE TELE2 END-EXEC.

PRINT-AND-GET2.
PERFORM PRINT-A-LINE.
EXEC SQL FETCH TELE2 INTO :PPHONE END-EXEC.
/
*****************************************************
* LIST SPECIFIC EMPLOYEES *
*****************************************************
LIST-SPECIFIC.
* **OPEN CURSOR
EXEC SQL OPEN TELE3 END-EXEC.
```

Figure 23. Example: COBOL DB2 phone application that invokes Java under z/OS Batch Runtime (Part 7 of 10)

```
* **GET EMPLOYEES
EXEC SQL FETCH TELE3 INTO :PPHONE END-EXEC.

IF SQLCODE = NOT-FOUND
* **NO EMPLOYEE FOUND
* **PRINT ERROR MESSAGE
MOVE '008I' TO MSGCODE
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG
MOVE OUTMSG TO OUTMSG2
WRITE REPREC FROM MSG-REC2
AFTER ADVANCING 2 LINES
ELSE
* **LIST SPECIFIC EMPLOYEE(S)
PERFORM PRINT-AND-GET3
UNTIL SQLCODE IS NOT EQUAL TO ZERO.

* **CLOSE CURSOR
EXEC SQL CLOSE TELE3 END-EXEC.

PRINT-AND-GET3.
PERFORM PRINT-A-LINE.
EXEC SQL FETCH TELE3 INTO :PPHONE END-EXEC.
/
*****************************************************
* PRINT A LINE OF INFORMATION FROM DIRECTORY *
*****************************************************
PRINT-A-LINE.
* **GET INFORMATION
MOVE LASTNAMEC TO RLNAME.
MOVE FIRSTNAMEC TO RFNAME.
MOVE MIDDLEINITIAL TO RMIDINIT.
MOVE PHONENUMBER OF PPHONE TO RPHONE.
MOVE EMPLOYEENUMBER OF PPHONE TO REMPNO.
MOVE DEPTNUMBER TO RDEPTNO.
MOVE DEPTNAMEC TO RDEPTNAME.
* **PRINT INFORMATION
WRITE REPREC FROM REPDATA
AFTER ADVANCING 2 LINES.

MOVE SPACES TO LASTNAMEC
FIRSTNAMEC
DEPTNAMEC.
/
*****************************************************
* UPDATES PHONE NUMBERS FOR EMPLOYEES *
*****************************************************
TELEPHONE-UPDATE.
EXEC SQL UPDATE DSN8910.VEMPLP
SET PHONENUMBER = :NEWNO
WHERE EMPLOYEENUMBER = :ENO END-EXEC.
IF SQLCODE = ZERO
* **EMPLOYEE FOUND
* **UPDATE SUCCESSFUL
* **PRINT CONFIRMATION
* **MESSAGE
E INVOKE BCDTranHelper "commit"
DISPLAY "After the BCcommit"
Perform ErrorCheck
MOVE '004I' TO MSGCODE
ELSE
* **NO EMPLOYEE FOUND
* **UPDATE FAILED
```

Figure 24. Example: COBOL DB2 phone application that invokes Java under z/OS Batch Runtime (Part 8 of 10)

```
* **PRINT ERROR MESSAGE
MOVE '007E' TO MSGCODE.
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
MOVE OUTMSG TO OUTMSG2.
WRITE REPREC FROM MSG-REC2
AFTER ADVANCING 2 LINES.
/
*****************************************************
* SQL ERROR OCCURRED - GET ERROR MESSAGE *
*****************************************************
DBERROR.
* **SQL ERROR
* **PRINT ERROR MESSAGE
MOVE '060E' TO MSGCODE
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
MOVE OUTMSG TO OUTMSG1 OF MSG-REC1.
MOVE SQLCODE TO RETCODE OF MSG-REC1.
WRITE REPREC FROM MSG-REC1
AFTER ADVANCING 2 LINES.
CALL 'DSNTIAR' USING SQLCA ERROR-MESSAGE ERROR-TEXT-LEN.
IF RETURN-CODE = ZERO
PERFORM ERROR-PRINT VARYING ERROR-INDEX
FROM 1 BY 1 UNTIL ERROR-INDEX GREATER THAN 10
ELSE

* **MESSAGE FORMAT
* **ROUTINE ERROR
* **PRINT ERROR MESSAGE
MOVE '075E' TO MSGCODE
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG
MOVE OUTMSG TO OUTMSG1 OF MSG-REC1
MOVE RETURN-CODE TO RETCODE OF MSG-REC1
WRITE REPREC FROM MSG-REC1
AFTER ADVANCING 2 LINES.

***********************************************************
* SQL RETURN CODE HANDLING WHEN PROCESSING CANNOT PROCEED *
***********************************************************
EXEC SQL WHENEVER SQLERROR CONTINUE END-EXEC.
EXEC SQL WHENEVER SQLWARNING CONTINUE END-EXEC.
EXEC SQL WHENEVER NOT FOUND CONTINUE END-EXEC.

F * **PERFORM ROLLBACK
  INVOKE BCDTranHelper "rollback"
  DISPLAY "After the BCrollback"
  Perform ErrorCheck

IF SQLCODE = ZERO

* **ROLLBACK SUCCESSFUL
* **PRINT CONFIRMATION
* **MESSAGE
MOVE '053I' TO MSGCODE
ELSE

* **ROLLBACK FAILED
* **PRINT ERROR MESSAGE
MOVE '061E' TO MSGCODE.
CALL 'DSN8MCG' USING MAJOR MSGCODE OUTMSG.
MOVE OUTMSG TO OUTMSG1 OF MSG-REC1.
MOVE SQLCODE TO RETCODE OF MSG-REC1.
WRITE REPREC FROM MSG-REC1
AFTER ADVANCING 2 LINES.
GO TO PROG-END.
```

Figure 25. Example: COBOL DB2 phone application that invokes Java under z/OS Batch Runtime (Part 9 of 10)

```
    *****************************************************
    * PRINT MESSAGE TEXT *
    *****************************************************
    ERROR-PRINT.
    WRITE REPREC FROM ERROR-TEXT (ERROR-INDEX)
    AFTER ADVANCING 1 LINE.

    *****************************************************
G   * Java Exception Check *
    *****************************************************
    ErrorCheck.
    Compute RETCODE = 0
    Call ExceptionOccurred
    using by value JNIEnvPtr
    returning ex
    If ex not = null then
    Call ExceptionClear using by value JNIEnvPtr
    Display "Caught an unexpected exception"
    Invoke ex "printStackTrace"
    MOVE 99 to RETURN-CODE
    End-if.
```

Figure 26. Example: COBOL DB2 phone application that invokes Java under z/OS Batch Runtime (Part 10 of 10)

Binding DB2 with Java JDBC and COBOL embedded SQL

Note: Before you begin, it is important to be familiar with the DB2 for z/OS package creation for SQLJ programs. For additional details, see the following information about writing and preparing Java programs that access DB2 for z/OS databases:

- The topic about "Programming for Java" in `http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=/com.ibm.db29.doc.java/db2z_java.htm`
- The topic about "Preparing and running JDBC and SQLJ programs" in DB2 Application Programming Guide and Reference for Java.
- The topic "Binding an application" in DB2 Application Programming and SQL Guide and in `http://publib.boulder.ibm.com/infocenter/dzichelp/v2r2/index.jsp?topic=/com.ibm.db29.doc.apsg/db2z_bindplanpanel.htm`

As input, the JDBC driver of z/OS supports application package collections or a plan name. Embedded SQL in IBM Enterprise COBOL routines typically use a bound DB2 plan as input. Packages provide more flexibility by minimizing full application rebuilds when only one SQL source file is updated. Therefore, a best practice for the hybrid mixture of COBOL and Java JDBC sharing a local RRSAF attachment is to use a package list passed to the JDBC driver through the JDBC property `db2.jcc.pkList`. These JDBC properties can be passed on the Java command line using `-Dprop_name=`*`value`*. When many properties are involved, you can use the special JDBC property `-Ddb2.jcc.PropertiesFile=`*`pathname of the file`*, where the `PropertiesFile` contains the list of desired `jcc.db2.*` system properties. You can also use JDBC APIs to set properties; for more information, refer to DB2 Application Programming Guide and Reference for Java.

For the commands necessary to build SQLJ packages for Java programs containing SQLJ, see “Commands for SQLJ program preparation” on page 46.

There is considerable flexibility when binding with existing packages and DBRM members, or both. To bind a COBOL program containing embedded SQL, which has been preprocessed or co-processed to produce DBRM member XMPCOBJX (for instance, COBOL extended with Java JDBC), you can use **--**.

```
//BINDCOBX JOB (1),'name'
//            NOTIFY=&SYSUID,
//            MSGCLASS=X,
//            CLASS=A,
//            REGION=0M,
//            TIME=120
//BINDEXE  EXEC PGM=IKJEFT01,DYNAMNBR=20,COND=(4,LT)
//DBRMLIB  DD DSN=SUIMGJB.DBRMLIB.DATA,DISP=SHR
//SYSTSPRT DD SYSOUT=*
//SYSPRINT DD SYSOUT=*
//SYSUDUMP DD SYSOUT=*
//SYSOUT   DD SYSOUT=*
//REPORT   DD SYSOUT=*
//SYSIN    DD *
//SYSTSIN DD *
 DSN SYSTEM(DSN9)
  BIND PACKAGE(XMPCOBJX) MEMBER(XMPCOBJX) -
  ACT(REP) ISO(CS) CURRENTDATA(YES) ENCODING(EBCDIC)
/*
```

Figure 27. Example: JDBC-only case

Using the example in Figure 27, you now have a new COBOL collection named XMPCOBJX.*. This can be passed to z/OS Batch Runtime as the system property `db2.jcc.pkList`, which can be appended to the default JDBC-provided `NULLID` collection. On the Java command line, for example, this would be seen as follows:

```
-Ddb2.jcc.pkList=NULLID.*,XMPCOBJX.*
```

You should also grant package privileges, according to the specific security standards that are in place at your installation. Using the example in Figure 27, you would specify the following statement, where *authid* can be either a user ID or secondary ID, such as a RACF® (SAF) group name.

`GRANT EXECUTE ON PACKAGE XMPCOBJX.* TO` *`authid`*

Commands for SQLJ program preparation

To build SQLJ packages for Java programs that contain SQLJ embedded SQL, knowledge and use of the following commands is a must:

`sqlj - SQLJ translator`
: The `sqlj` command translates an SQLJ source file into a Java source file and zero or more SQLJ serialized profiles. By default, the `sqlj` command also compiles the Java source file.

`db2sqljcustomize - SQLJ profile customizer`
: The `db2sqljcustomize` command augments the profile with DB2-specific information for use at run time. It processes an SQLJ profile, which contains embedded SQL statements. By default, `db2sqljcustomize` produces four DB2 packages: one for each isolation level.

Remember, also, to include `SQLJ.JAR` in the classpath set up of your BCDBATCH JCL.

For the complete details, syntax, authorization, and parameters, see the topic on "JDBC and SQLJ reference information" in DB2 Application Programming Guide and Reference for Java.

Chapter 6. Application structure and build considerations for PL/I and Java

The z/OS Batch Runtime component, introduced in z/OS V1R13, is part of an ongoing z/OS effort to integrate Java and modernize JES batch with traditional application language assets prevalent in many z/OS installations. In that initial release, the z/OS Batch Runtime provided the ability for applications written in COBOL and Java to inter operate with DB2 in a Resource Recovery Services (RRS) global transaction environment.

In z/OS V2R1, the batch runtime is enhanced to include PL/I as a supported language. When mixing COBOL, PL/I, and Java programs in a batch environment, the z/OS Batch Runtime provides a managed environment to govern the different programming models being used. Just as with the IBM COBOL support, the environment is bound and centered around traditional JES submitted job steps.

In particular, this managed environment provides a framework and APIs to enable shared access to a local DB2 for z/OS database connection by COBOL, PL/I, and Java programs. Updates to the database across language boundaries are committed within a single RRS managed transaction scope. Also new in z/OS V2R1 is transparent support for Transactional VSAM (TVS) across these same language environments.

Support is added to allow the z/OS Batch Runtime to launch an Enterprise PL/I main routine similar to the launch of COBOL that was done in V1R13.

Further enhancements are made in z/OS V2R1 Batch Runtime to support the IBM Batch Programming Model. This support, fully documented in Appendix AAppendix A, is for JES submitted new JAVA applications described in an XML like policy (termed xJCL) and following the rules intrinsic to the IBM Batch Programming Model. This is the same descriptive language used in IBM Websphere Batch support, albeit with a traditional JES submission and limited to a single non-persistent JVM per job step.

In the descriptions that follow, we use the term “batch container” to distinguish the surrounding transaction management subset of the full z/OS Batch Runtime.

PL/I External Control JCL Statement

Identically to the IBM COBOL support, z/OS Batch Runtime control statements are read from the //BCDIN file defined in the batch JCL.

The bcd.applicationLanguage statement is changed to accept the new LE value to indicate the program being launched is written in either COBOL or PL/I. Although the COBOL language value will still be accepted, the new LE value is the preferred syntax. The batch runtime processes both COBOL and PL/I applications in the same way.

The revised syntax is as follows:

```
bcd.applicationLanguage=JAVA | COBOL | LE
```

Figure 28. bcd.applicationLanguage Syntax

The bcd.applicationArgs statement is used to pass an application argument to the launched program. For PL/I, a single string up to 100 characters in length can be used.

PL/I Compile and Bind Considerations for PL/I Main Routines

The z/OS Batch Runtime launches an application by fetching and calling it. As a result, the PL/I external procedure being launched must specify the “fetchable” and “assembler” options. The “main” option cannot be used. So at a minimum, and launched PL/I “Main” routine must be at least slightly modified as shown below. Additionally, the assembler option is also needed so that any launched PL/I “Main” application can set a return code using the PLIRETC function upon return to the z/OS Batch Runtime.

For example, the external procedure could be written as follows:

```
PLITEST: Procedure Options( Fetchable
                            Assembler );
```

When binding the PL/I application, the ENTRY CEESTART must not be used. Instead, the entry point should specify the name of the external procedure. For example, the procedure shown above would be bound as:

```
//BIND.SYSIN DD *
ENTRY PLITEST
NAME PLITEST(R)
```

Complete JCL examples will be shown in the sections that follow.

Commit and Rollback Callbacks

The z/OS Batch Runtime coordinates updates to data through commit and rollback operations. When an application needs to commit or rollback a transaction, the batch runtime is called to initiate the activity. This is done through Java callbacks to the batch container.

The batch container provides two Java helper methods that must be invoked by the application:

- Commit: com.ibm.batch.spi.UserControlledTransactionHelper.commit()
- Rollback: com.ibm.batch.spi.UserControlledTransactionHelper.rollback()

Because these helper methods are written in Java, the PL/I application must use the Java Native Interface (JNI) to call them. The batch runtime will document a sample that shows how to do this but will not deliver any PL/I sample code.

PL/I provides a mapping of the JNI functions which are needed to invoke Java methods in the %ibmzjni.inc include member. The PL/I application then invokes a series of JNI calls to find the helper method and invoke it.

This process is described generally below and is followed with an example:

1. Obtain the JNIEnv pointer (which is needed to access JNI functions) using the JNI_GetCreatedJavaVMs function

2. Invoke the FindClass JNI function to locate the com.ibm.batch.spi.UserControlledTransactionHelper class
3. Invoke the GetStaticMethodID JNI function to locate the commit or rollback methods
4. Invoke the CallStaticVoidMethod JNI function to invoke the commit or rollback method located above

It is likely the PL/I application will already have similar code due to interoperability since the process is the same when invoking any Java method.

Sample PL/I Source to Invoke Commit Callback

The following source file contains an example of a PL/I module calling Java through the JNI to invoke the batch container commit method using the steps as described above. One may als0 think of this as simple proxy for any PL/I “main” routine launched from the z/OS Batch Runtime and invoking simple Java methods.

```
*Process Limits( Extname( 100 ) );
*Process Margins( 1, 100 );
*Process Display(STD) Rent;
*Process Default( ASCII ) Or('|');
*Process Options InSource Source Nest Macro Storage;
*Process Aggregate Offset;
*Process List Flag(I) MarginI('|');
*Process Opt(2) Attributes(Full) Xref(Short);
/* PL/I calling batch container commit using JNI */
PLICOMIT: Procedure Options( Fetchable
                             Assembler );

/*********************************************************************/
/* */
/* This sample invokes the z/OS batch runtime to commit a */
/* transaction. */
/* */
/* The batch runtime transaction helper class is loaded and the */
/* commit method is located and invoked. */
/* */
/*********************************************************************/

     /************/
     /* Includes */
     /************/
%INCLUDE ibmzjni;


     /*************/
     /* Constants */
     /*************/
Dcl NULLPTR Pointer Static Init(PtrValue(0));

Dcl TRANSACTION_HELPER_CLASS_NAME Char(60) VaryingZ Static
      Init("com/ibm/batch/spi/UserControlledTransactionHelper");

     /*******************/
     /* Local variables */
     /*******************/
Dcl myRC Fixed Bin(31);

Dcl myClass Type jclass;
Dcl myMethodID Type jmethodID;
Dcl myjstring Type jstring;

Dcl javaNumVMs Type jsize;

Dcl myNull Pointer;


Put skip list("Starting PLICOMIT ...");
     /*****************/
     /* Query the JVM */
     /*****************/
myRC = JNI_GetCreatedJavaVMs( addr(JavaVM),
                              1,
                              addr(javaNumVMs) );
```

Figure 29. PL/I Sample to Commit Transaction Using JNI

```
If myRC = JNI_OK then
  Do;
    If javaNumVMs = 1 then
      Do;
       /*******************************/
       /* Get env pointer for our JVM */
       /*******************************/
       myRC = JGetEnv( JavaVM,
                       addr(JNIEnv),
                       JNI_VERSION_1_4 );

       If myRC ^= JNI_OK then
         Do;
           Put skip list("JGetEnv failed rc=",myRC);
           Stop;
         End;
       Else;
     End;
  Else
    Do;
      Put skip list("javaNumVMs not 1",javanumVMs);
      Stop;
    End;
  End;
Else
  Do;
    Put skip list("getCreatedJavaVMs failed",myRC);
    Stop;
  End;

     /******************************/
     /* Get the Java class to call */
     /******************************/
myClass = FindClass( JNIEnv,
                     TRANSACTION_HELPER_CLASS_NAME );
If myClass = NULLPTR then
  Do;
    Put skip list("myClass is null");
    Stop;
  End;
Else;

      /*************************************/
      /* Get method id for commit method */
      /*************************************/
myMethodID = GetStaticMethodID( JNIEnv,
                                myClass,
                                "commit",
                                "()V" );

If myMethodID = NULLPTR then
  Do;
    Put skip list("myMethodID is null");
    Stop;
  End;
Else;

Put skip list("Calling commit method ...");
myNull = CallStaticVoidMethod( JNIEnv,
                               myClass,
                               myMethodID );

Call PLIRETC(0); /* Set rc=0 */

Put skip list("PLICOMIT ended.");
End PLICOMIT;
```

Sample Compile and Bind JCL

The following is an example of the JCL needed to compile and bind the sample source as shown in Figure 29 on page 52

```
//jobname JOB (1)
//*
//*   Complies and links PLICOMIT
//*
//JCLLIB   JCLLIB   ORDER=PLI.PLI410.SIBMZPRC
//*
//STEP1    EXEC   IBMZCB,
//       REGION=0M,
//       LNGPRFX='PLI.PLI410',
//       PARM.BIND=('OPTIONS=OPTS')
//*
//PLI.SYSLIB DD DSN=PLI.PLI410.SIBMZSAM,DISP=SHR
//*
//PLI.SYSIN  DD DSN=IBMUSER.BATCH.SOURCE(PLICOMIT),DISP=SHR
//*
//BIND.SYSLMOD DD DSN=IBMUSER.BATCH.LOAD,DISP=SHR
//BIND.OPTS    DD *
MAP
RENT
DYNAM=DLL
CASE=MIXED
LIST=ALL
XREF
//*
//BIND.SYSIN   DD *
INCLUDE '/usr/lpp/java/J6.0/bin/j9vm/libjvm.x'
ENTRY PLICOMIT
NAME PLICOMIT(R)
//
```

Figure 30. Sample Compile and Bind JCL to Commit Transaction Using JNI

Commit and Rollback Helpers

Although the batch runtime commit and rollback methods can be called using the JNI as shown above, the process is cumbersome. To simplify the process, the batch runtime is providing the convenience methods bcdcommit() and bcdrollback() that can be called directly from a PL/I or COBOL application. Use of the helpers replaces the JNI calls to callback to the batch container.

The methods reside in a new DLL bcdlibuser.so that will be shipped with this line item. For PL/I callers, an include file is provided in SYS1.SAMPLIB(BCDPLIH) that defines the entry points. COBOL applications do not need the include file and can just call the methods directly.

Both COBOL and PL/I applications must include the bcdlibuser.x side deck when binding their applications to make the methods accessible to the program.

Syntax

The syntax of calling the methods varies based on the calling language as shown in the figures below.

```
rc=bcdcommit();
```

Figure 31. PL/I bcdcommit() Syntax

```
rc=bcdrollback();
```

Figure 32. PL/I bcdrollback() Syntax

```
Call 'bcdcommit' returning rc.
```

Figure 33. COBOL bcdcommit() Syntax

```
Call 'bcdrollback' returning rc.
```

Figure 34. COBOL bcdrollback() Syntax

Return Codes

The bcdcommit() and bcdrollback() methods set a return code as follows:

Table 5. bcdcommit() and bcdrollback() Return Codes

Return Code	Description
00	Success
16	Function not performed, unable to obtain JNI pointer

The methods can also throw a Java exception in the event the callback to the batch runtime fails.

The following is sample JCL to compile and bind the PL/I sample. Note that the helper method side deck must be included in the bind step to make the methods accessible to the application.

```
//jobname JOB (1)
//*
//*   Complies and links PLIHELP
//*
//JCLLIB   JCLLIB   ORDER=PLI.PLI410.SIBMZPRC
//*
//STEP1    EXEC   IBMZCB,
//     REGION=0M,
//     LNGPRFX='PLI.PLI410',
//     PARM.BIND=('OPTIONS=OPTS')
//*
//PLI.SYSLIB DD DSN=SYS1.SAMPLIB,DISP=SHR
//*
//PLI.SYSIN  DD DSN=IBMUSER.BATCH.SOURCE(PLIHELP),DISP=SHR
//*
//BIND.SYSLMOD DD DSN=IBMUSER.BATCH.LOAD,DISP=SHR
//BIND.OPTS    DD *
MAP
RENT
DYNAM=DLL
CASE=MIXED
LIST=ALL
XREF
//*
//BIND.SYSIN    DD *
INCLUDE '/usr/lpp/bcp/lib/libbcduser.x'
ENTRY PLIHELP
NAME PLIHELP(R)
//
```

Figure 35. Sample PL/I Compile and Bind JCL for bcdcommit() and bcdrollback() Helpers

Using the bcdcommit() and bcdrollback() helpers from PL/I

The sample below shows a PL/I caller invoking the bcdcommit() and bcdrollback() methods to commit a transaction instead of calling the JNI directly.

```
*Process Limits( Extname( 100 ) );
*Process Margins( 1, 100 );
*Process Display(STD) Rent;
*Process Default( ASCII ) Or('|');
*Process Options InSource Source Nest Macro Storage;
*Process Aggregate Offset;
*Process List Flag(I) MarginI('|');
*Process Opt(2) Attributes(Full) Xref(Short);
/* PL/I Module calling batch container helpers */
PLIHELP: Procedure Options( Fetchable
                            Assembler );

%INCLUDE BCDPLIH;

Dcl rc fixed bin(31);

Display("Calling bcdcommit helper ...");
rc = bcdcommit();
Display("bcdcommit rc=" || rc);

Display("Calling bcdrollback helper ...");
rc = bcdrollback();
Display("bcdrollback rc=" || rc);

End PLIHELP;
```

Figure 36. Sample PL/I Commit Transaction Using bcdcommit() and bcdrollback()

PL/I Embedded SQL and Transactional VSAM Considerations

Just as with IBM COBOL and DB2, there are no particular changes required to PL/I source for use of embedded SQL in the z/OS Batch Runtime environment. Since the RRSAF attachment is acquired by the runtime prior to the application receiving control, it is not necessary for any application code to initialize a DB2 RRSAF local attachment (identify, create thread, etc RRS protocal). Any such code should either be removed or recognize an existing attachment. Normal RRSAF pre-compilation or DB2 coprocessor is sufficient to establish the correct runtime linkage to DB2. See the the sample DB2 “phone” application DSN8BP3 provided and built with DSNTEJ2P in the hlqprefix.SDSNSAMP library provided with your DB2 installation. Simply include DSNRLI in the bind step of your program to get the correct RRSAF linkage. This sample application is described in the DB2 for z/OS Application Programming & SQL Guide SC18-9841 in the “DB2 sample applications and data” chapter.

For Transactional VSAM, once the TVS environment is established there are no required changes for the VSAM RLS (Record Level Sharing) APIs used in the application program. Transactional VSAM is a systems programmer only setup with correct TVS server, RLS server, VSAM dataset, and Logger definitions. The z/OS Batch Runtime provided RRS commit and rollback operations initiate any 2-phase commit processing required of the TVS server. Good references are z/OS DFSMStvs Planning and Operating Guide SC26-7348-00 and z/OS DFSMStvs Administration Guide GC26-7483-00.

Calling PL/I from Java

Java method naming must conform to java conventions for native PL/I calls. The full documentation for PL/I and Java interoperability and linkage is contained in Chapter 16, Interfacing with Java of the Enterprise PL/I for z/OS V4.2

Programming Guide (GI11-9145-01). There are no special considerations of this standard JNI protocol for the z/OS Batch Runtime. We include here a small extracted sample for completeness. Note that the PL/I in this example is built into a JNI required DLL named “hiFromPLI”

```
*Process Limits( Extname( 100 ) ) Margins( 1, 100 ) ;
     *Process Display(Std) Dllinit Extrn(Short);
     *Process Rent Default( ASCII IEEE );
     PliJava_Demo: Package Exports(*);
      Java_callingPLI_callToPLI:
      Proc( JNIEnv , MyJObject )
        External( "Java_callingPLI_callToPLI" )
        Options( FromAlien NoDescriptor ByValue );
       %include ibmzjni;
       Dcl myJObject        Type jObject;
       Display('Hello from Enterprise PL/I!');
     End;
```

Figure 37. PL/I example to be called from Java

```
public native void callToPLI();
  public class callingPLI {
       public native void callToPLI();
       static {
          System.loadLibrary("hiFromPLI");
       }
       public static void main(String[] argv) {
          callingPLI callPLI = new callingPLI();
          callPLI.callToPLI();
          System.out.println("And Hello from Java, too!");
       }
     }
```

Figure 38. Corresponding Java which calls the PL/I native method

Chapter 7. Troubleshooting for z/OS Batch Runtime

In addition to the standard z/OS messages (in the format BCD*nnnnx*, where *nnnn* is the message number and *x* is the message severity), z/OS Batch Runtime provides logging and tracing facilities for troubleshooting problems. The following topics explore trace and logging in more detail. For more information about messages, see *z/OS MVS System Messages, Vol 3 (ASB-BPX)*.

Trace facilities for z/OS Batch Runtime

All z/OS Batch Runtime classes are designed to use the standard Java trace facilities available in the `java.util.logging` package. At a minimum, z/OS Batch Runtime traces entry and exit to all significant methods, all exceptions, and all significant events. Tracing is controlled through a system property or by the logging configuration file, which by default is specified in the `jre/lib/logging.properties` file. You can override the location of the file using the following Java system property when you invoke z/OS Batch Runtime:

```
java.util.logging.config.file
```

Use a trace to diagnose problems in z/OS Batch Runtime. Obtain the trace using the following system property:

```
com.ibm.zos.batch.container.BCDTraceConfig=trace-level
```

The property values for *trace-level* are ALL, which indicates that all events will be traced, or NONE, which indicates no tracing. When diagnosing problems, use a trace level of ALL.

Log facilities for z/OS Batch Runtime

z/OS Batch Runtime provides a verbose mode to provide additional messages that can assist in diagnosing batch runtime problems. When running in verbose mode, all messages are created for all commit and rollback requests. Messages are written to `//BCDOUT`.

Signalling and exception handling by z/OS Batch Runtime

PL/I and COBOL applications have a specific signal or error condition handling process. Java has a defined signal handling process as well as a set of JNI processes for signal and error condition handling. Language Environment also has application programming interfaces (APIs) for application code that allows you to customize condition handling to override the default settings.

z/OS Batch Runtime uses the JVM startup option -XCEEHDLR. This option informs the JVM to register a stack-based Language Environment condition handler before COBOL JNI calls. It is then able to translate potentially-recoverable Language Environment exceptions into a Java exception and pass it back to the calling Java code. z/OS Batch Runtime catches and reports percolated runtime exceptions out of the Java application.

Appendix A. Common Batch Container

Introduction

This appendix describes the IBM WebSphere Common Batch Container. z/OS embeds this component, and supports a subset of its overall functionality. The IBM WebSphere Common Batch Container enables JES submission of the Transactional Batch programming model. This batch model is an XML-like policy driven control flow of looping business logic with defined input and output data streams. z/OS Resource Recovery Services (RRS) provides transaction support for z/OS data streams, such as DB2, that support transactional behavior.

To use the z/OS Batch Runtime, you must build a set of Java class files corresponding to the described transactional batch job step interface and data streams, and submits these files as a traditional job step to the z/OS JES scheduler. As explained further in the following sections, there can be multiple and sequential transactional batch steps defined in a single JES job step. z/OS supports an inherently serial flow of these transactional batch steps.

Transactional batch programming model

The common batch container supports the transactional batch programming model in which batch applications conform to a few well-defined interfaces that allow the batch runtime environment to manage the start of batch jobs destined for the application. In the z/OS batch container environment, this model is implemented as a Java object.

The transactional batch programming model provides details on how the common batch container manages the life cycle of the application and jobs that are submitted to the common batch container. Central to all common batch containers is the concept that a job represents an individual unit of work that needs to be processed.

Batch job steps

A batch job can be comprised of one or more batch steps. All steps in a job are processed sequentially. Dividing a batch application into steps allows for separation of distinct tasks in a batch application. Batch steps are created by implementing the interface com.ibm.websphere.batch.BatchJobStepInterface. The implementation of this interface provides the business logic of the batch step that the batch runtime invokes to run the batch application. See “Creating batch job steps” on page 83 for more information.

Batch data streams

Batch data streams (BDS) are Java objects that provide an abstraction for the data stream processed by a batch step. A batch step can have zero or more BDS objects associated with it. The batch endpoints make the BDS associated with the batch step available at run time. The batch endpoints also manage the lifecycle of a BDS by invoking batch-specific callbacks.

A BDS object implements the com.ibm.websphere.batch.BatchDataStream interface. The implementing object can retrieve data from any type of data source.

See “Implementing batch data stream framework and patterns” on page 63 for more information.

Batch data stream framework

Common batch container provides a batch data stream (BDS) framework that includes pre-built code to work with popular streams like text, byte, database, data sets, and so on. You can implement an interface where the business logic for processing the stream is added. The pre-built code manages such operations as the opening, closing, and the externalizing/internalizing of checkpoints for the batch data stream.

Checkpoint algorithms

The batch runtime environment uses checkpoint algorithms to record progress during a step. he XML Job Control Language (xJCL) definition of a batch job defines the checkpoint algorithms to be used.

Properties specified for checkpoint algorithms in xJCL allow for checkpoint behavior, such as checkpoint intervals, to be customized for batch steps. The common batch container provides time-based and record-based checkpoint algorithms. A checkpoint algorithm SPI is also provided for building additional custom checkpoint algorithms. See “Implementing Checkpoint algorithms” on page 98 for more information about implementing checkpoint algorithms.

Results algorithm

Results algorithms are an optional feature of the batch programming model. Results algorithms are applied to batch steps through XML Job Control Language (xJCL). The algorithms are used to manipulate the return codes of batch jobs. Additionally, these algorithms are place holders for triggers based on step return codes. See “Implementing a Results algorithm” on page 99 for more information about implementing results algorithms.

Batch job return codes

Batch job return codes fall into two groups named *system* and *user*. System return codes are defined as negative integers. User application return codes are defined as positive integers. Both system and user ranges include the return code of zero (0). If a user application return code is specified in the system return code range, a warning message is posted in the job and system logs. See “Batch job return codes explanations” on page 91 for explanations of the return codes that might be issued.

Developing a simple batch application

The following procedure for developing a batch application is kept simple from a processing standpoint to highlight the steps involved in developing a new batch application.

Procedure

1. Create batch job steps.

   Create a new Java class that implements the interface com.ibm.websphere.BatchJobStepInterface. Implement business logic. If your step has exactly one input and one output stream you could alternatively use the Generic batch step (GenericXDBatchStep).
2. Create batch data streams.

   Batch data streams are accessed from the business logic (i.e. from the batch job steps) by calling BatchDataStreamMgr with jobID and stepID. JobID and stepID are retrieved from the step bean properties list using keys BatchConstants.JOB_ID and BatchConstants.STEP_ID. Map BatchConstants.JOB_ID to com.ibm.websphere.batch.JobID and map

BatchConstants.STEP_ID to com.ibm.websphere.batch.StepID. You should already have access to the BatchConstants class.

The batch datastream framework provides several ready-to-use patterns to work with different types of datastreams. For example, file, database, and so on. To use the batch datastream framework:

a. Identify the data stream type you want to operate with (TextFile, ByteFile, JDBC, z/OS stream)
b. Identify whether you would be reading or writing data from/to this stream.
c. Refer to the table in "Implementing batch data stream framework and patterns," and select the class from the supporting classes column that matches your data stream type and operation.

   For example, if you want to read data from a text file then you would choose TextFileReader.
d. Implement the interface listed in the pattern name column that corresponds to the supporting class you selected in the previous step.

   The supporting class handles all the book keeping activities related to the stream and the batch programming model, allowing the implementation class to focus on the stream processing logic.

   Example:

```
<batch-data-streams>
   <bds>
 <logical-name>inputStream</logical-name>
  <props>
      <prop name="PATTERN_IMPL_CLASS" value="MyBDSStreamImplementationClass"/>
            <prop name="file.encoding" value="8859_1"/>
            <prop name="FILENAME" value="${inputDataStream}" />
            <prop name="PROCESS_HEADER" value="true"/>
            <prop name="AppendJobIdToFileName" value="true"/>
        </props>
 <impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.FileByteReader
</impl-class>
   </bds>
```

   Note: The PATTERN_IMPL_CLASS denotes your implementation of the BDS framework pattern and the impl-class property denotes the supporting class.
e. Declare the supporting class and your implementation class in the xJCL.
f. Repeat this procedure for each datastream required in your step.

Implementing batch data stream framework and patterns

The batch data stream (BDS) framework pattern interface is a simple Java interface for a particular type of data stream into which you an insert business logic. The BDS framework has several supporting classes for each pattern that do most of the mundane tasks related to stream management. The following table shows the patterns that the common batch container provides.

The following main methods exist for the BatchDataStream interface. See the API for the BatchDataStream interface for additional information.

- void open(): Called by batch jobs to open the BDS
- void close(): Called by batch jobs to close the BDS
- void initialize(String ilogicalname, String ijobstepid): Called by batch jobs to initialize the BDS and let it knows its logical name and batch step ID.

- String externalizeCheckpointInformation(): Called by batch jobs right before a checkpoint to record the current cursor of the BDS
- void internalizeCheckpointInformation(String chkpointInfo(): Called by batch jobs to inform the BDS of the previously recorded cursor, chkpointInfo. Typically, the positionAtCurrentCheckpoint is called after this call to position the BDS to this cursor.
- void positionAtCurrentCheckpoint(): Called by batch jobs after calling internalizeCheckpointInformation to position the BDS to the cursor indicated by the chkpointInfo passed in through the internalizeCheckpointInformation call.

The BatchDataStream interface does not have methods for retrieving or writing data. There are no getNextRecord and putNextRecord methods defined on the interface that a batch step calls to read or write to the BDS object. Methods for passing data between the batch step and the BDS object are left up to the implementation of the BDS object. Review the batch samples that are provided to see how to implement batch data streams.

Table 6. Batch data stream patterns

Pattern name	**Description**	**Supporting classes**
“JDBCReaderPattern” on page 66	Used to retrieve data from a database using a JDBC connection.	• LocalJDBCReader • JDBCReader • CursorHoldableJDBCReader
“JDBCWriterPattern” on page 69	Used to write data to a database using a JDBC connection.	• LocalJDBCWriter • JDBCWriter
“ByteReaderPattern” on page 71	Used to read byte data from a file.	FileByteReader
“ByteWriterPattern” on page 72	Used to write byte data from a file.	FileByteWriter
“FileReaderPattern” on page 73	Used to read a text file.	TextFileReader
“FileWriterPattern” on page 75	Used to write to a text file.	TextFileWriter
“RecordOrientedDatasetReaderPattern” on page 76	Used to read a z/OS dataset.	• ZFileStreamOrientedTextReader • ZFileStreamOrientedByteReader • ZFileRecordOrientedDataReader
“RecordOrientedDatasetWriterPattern” on page 78	Used to write to a z/OS dataset.	• ZFileStreamOrientedTextWriter • ZFileStreamOrientedByteWriter • ZFileRecordOrientedDataReader
“JPAReaderPattern” on page 80	Used to retrieve data from a database using OpenJPA	JPAReader
“JPAWriterPattern” on page 82	Used to write data to a database using a Java Persistence API (JPA) connection.	JPAWriter

Using the batch data stream framework

Before you start this task, you must identify the correct pattern to use. Select a pattern based on the type of data stream you need to use. For example, if you want to read text from a file, then select the FileReaderPattern. See "Implementing batch data stream framework and patterns" on page 63 for a description of the available patterns

1. Implement the pattern interface:

```
<codeblock>package com.ibm.websphere.samples;

import java.io.BufferedReader;
import java.io.IOException;
import java.util.Properties;

import com.ibm.websphere.batch.devframework.configuration.BDSFWLogger;
import com.ibm.websphere.batch.devframework.datastreams.patternadapter.FileReaderPattern;

// Implement the FileReaderPattern
public class TransactionListStream implements FileReaderPattern {
 private Properties properties;
 private BDSFWLogger logger;

/**
   Save properties specified in the xJCL
*/

 public void initialize(Properties props) {
// create logger
  logger = new BDSFWLogger(props);

  if (logger.isDebugEnabled())
   logger.debug("entering TransactionListInputStream.initialize()");
  properties = props;

 }


// This method is where you should add the business logic of processing the read //string
 public Object fetchRecord(BufferedReader reader) throws IOException {
  String str = null;
  Posting posting = null;
  if (logger.isDebugEnabled())
logger.debug("Entering TransactionListInputStream.fetchRecord");
  if(reader.ready()) {
   str = reader.readLine();
  }
  if(str != null) {

   posting = _generateRecord(str);

  }

  if (logger.isDebugEnabled())
   logger.debug("Exiting TransactionListInputStream.fetchRecord with " + posting);
  return posting;

 }
 // Helper method that parses the read string and
creates an internal object for use ///by other parts of the code
 private Posting _generateRecord(String str) {
  Posting post = null;
  String [] tokens = str.split(",", 3);

  if(tokens.length == 3) {

   String txTypeStr = tokens[0];
   String actNoStr = tokens[1];
   String amtStr = tokens[2];

   int txType = Integer.parseInt(txTypeStr);
   double amt = Double.parseDouble(amtStr);
   post = new Posting(txType,actNoStr,amt);


  } else {
   logger.error("Invalid csv string" + str);
  }
  if(logger.isDebugEnabled())
   logger.debug("Loaded posting record " + post);
  return post;
 }
 public void processHeader(BufferedReader reader) throws IOException {
  // NO OP for this sample

 }



}
</codeblock>
```

2. Add a reference to the class that you just created in the previous step, along with the supporting class in the xJCL.

```
<codeblock><batch-data-streams>
  <bds>
          <logical-name>txlististream</logical-name>
          <props>
<prop name="IMPLCLASS" value= "com.ibm.websphere.samples.TransactionListStream"/>
             <prop name="FILENAME" value="/opt/inputfile.txt"/>
             <prop name="debug" value="true"/>
          </props>
<impl-class> com.ibm.websphere.batch.devframework.datastreams.patterns.TextFileReader </impl-class>
 </bds>
</batch-data-streams>

</codeblock>
```

JDBCReaderPattern

This pattern is used to retrieve data from a database using a Java Database Connectivity (JDBC) connection.

Supporting classes

- CursorHoldableJDBCReader

  This class is referenced when the usage pattern of your JDBC input stream retrieves a set of results at the beginning of the step, and then iterates over them throughout the step-processing logic. The CursorHoldableJDBCReader uses a stateful session bean with a cursor-holdable, non-XA data source. A cursor-holdable JDBCReader is a pattern that is implemented in such a way that the cursor is not lost when the transaction is committed. As a result, ResultSets do not need to be repopulated after every checkpoint, which improves performance. To use CursorHoldableJDBCReader, package the CursorHoldableSessionBean in your application. To create the package, add the **nonxadsjndiname=***jndi_name_of_a_non-XA_data_source_to_database* property to the properties file that is used by the BatchPackager. For example:

  ```
  nonxadsjndiname=jdbc/nonxads
  ```

  If you want to add multiple non-XA datasources enter the following:

  ```
  name1>;<jndi name2>...
  ```

  Restriction: Currently, the resource reference name of the JDBC data source is the same as the Java Naming and Directory Interface (JNDI) name.

- JDBCReader

This class is referenced when the usage pattern of your JDBC input stream retrieves a single result from a query, which is used and discarded after every iteration of the step.

- LocalJDBCReader

  This class is referenced when data is read from a local database.

Required properties

Table 7. Required properties

Property	Value	LocalJDBCReader	CursorHoldableJDBCReader	JDBCReader
PATTERN_IMPL _CLASS	Class implementing JDBCReaderPattern interface	Applicable	Applicable	Applicable
ds_jndi_name	Datasource JNDI name.	Applicable	Not applicable	Applicable
jdbc_url	The JDBC URL. For example, jdbc:derby:C:\\mysample\\ CREDITREPORT.	Applicable	Not applicable	Not applicable
jdbc_driver	The JDBC driver. For example, org.apache.derby.jdbc. EmbeddedDriver	Applicable	Not applicable	Not applicable
userid	The user ID for the database. For example, Myid	Applicable	Not applicable	Not applicable
pswd	User password. For example, mypwd. LocalJDBCReader only.	Applicable	Not applicable	Not applicable

Optional properties

Table 8. Optional properties

Property name	Value	Description	LocalJDBCReader	CursorHoldableJDCReader	JDBCReader
debug	true or false (default is false)	Enables detailed tracing on this batch datastream.	Applicable	Applicable	Applicable
EnablePerformance Measurement	true or false (default is false)	Calculates the total time spent in the batch data-streams and the processRecord method, if you are using the GenericXDBatchStep.	Applicable	Applicable	Applicable
EnableDetailedPerformance Measurement	true or false (default is false)	Provides a more detailed breakdown of time spent in each method of the batch data-streams.	Applicable	Applicable	Applicable

Interface definition

```
public interface JDBCReaderPattern {

 /**
  * This method is invoked during the job setup phase.
  *
  * @param props properties provided in the xJCL
  */

 public void initialize(Properties props);

 /**
  * This method should retrieve values for the various columns for the current row
  * from the given resultset object. Typically this data would be used to populate
  * an intermediate object which would be returned
```

```
   * @param resultSet
  * @return
  */
 public Object fetchRecord(ResultSet resultSet);

 /**
  * This method should return a SQL query that will be used during setup of the
   * stream to retrieve all relevant data that would be processed part of the job
  * steps @return object to be used during process step.
  */
 public String getInitialLookupQuery();

 /**
  * This method gets called during Job Restart. The restart token should be used to
  * create an SQL query that will retrieve previously unprocessed records.
  * Typically the restart token would be the primary key in the table and the query
  * would get all rows with primary key value > restarttoken
  * @param restartToken
  * @return The restart query
  */
 public String getRestartQuery(String restartToken);

 /**
  * This method gets called just before a checkpoint is taken.
  * @return The method should return a string value identifying the last record
   * read by the stream.
  */
 public String getRestartTokens();

}
```

xJCL examples

Example 1:

```
<batch-data-streams>
<bds>
<logical-name>inputStream</logical-name>
<props>
<prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.tests.bds.EchoReader"/>
<prop name="ds_jndi_name" value="jdbc/fvtdb"/>
<prop name="debug" value="true"/>
<prop name="DEFAULT_APPLICATION_NAME" value="XDCGIVT"/>
</props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.CursorHoldableJDBCReader
</impl-class>
</bds>
</batch-data-streams>
```

Example 2:

```
<batch-data-streams>
<bds>
<logical-name>inputStream</logical-name>
<props>
<prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.tests.bds.EchoReader"/>
<prop name="jdbc_url" value="jdbc:derby:C:\\mysample\\CREDITREPORT"/>
<prop name="jdbc_driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
<prop name="user_id" value="myuserid"/>
<prop name="pswd" value="mypswd"/>
<prop name="debug" value="true"/>
</props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.LocalJDBCReader
</impl-class>
</bds>
</batch-data-streams>
```

JDBCWriterPattern

This pattern is used to write data to a database using a JDBC connection.

Supporting classes

- JDBCWriter
- LocalJDBCWriter

Required properties

Table 9. Required properties

Property	Value	LocalJDBCWriter	JDBCWriter
PATTERN_IMPL_CLASS	Class implementing JDBCWriterPattern interface	Applicable	Applicable
ds_jndi_name	Datasource JNDI name.	Applicable	Not applicable
jdbc_url	The JDBC URL. For example, jdbc:derby:C:\\mysample\\ CREDITREPORT.	Applicable	Not applicable
jdbc_driver	The JDBC driver. For example, org.apache.derby.jdbc.EmbeddedDriver	Applicable	Not applicable
user_id	The user ID for the database. For example, Myid	Applicable	Not applicable
pswd	User password. For example, mypwd. LocalJDBCReader only.	Applicable	Not applicable

Optional properties

Table 10. Optional properties

Property name	Value	Description	LocalJDBCReader	JDBCWriter
debug	true or false (default is false)	Enables detailed tracing on this batch datastream.	Applicable	Applicable
EnablePerformance Measurement	true or false (default is false)	Calculates the total time spent in the batch data-streams and the processRecord method, if you are using the GenericXDBatchStep.	Applicable	Applicable
EnableDetailedPerformance Measurement	true or false (default is false)	Provides a more detailed breakdown of time spent in each method of the batch data-streams.	Applicable	Applicable

Table 10. Optional properties (continued)

Property name	Value	Description	LocalJDBCReader	JDBCWriter
batch_interval	Default value is 20. This value should be less than the checkpoint interval for record-based checkpointing.	Denotes the number of SQL updates to batch before committing.	Applicable	Applicable

Interface definition

```
public interface JDBCWriterPattern {

 public void initialize(Properties props);

 /**
  * This is typically an Update query used to write data into the DB
  * @return
  */
 public String getSQLQuery();

 /**
  * The parent class BDSJDBCWriter creates a new preparedstatement and
  * passes it to this method. This method populates the preparedstatement
  * with appropriate values and returns it to the parent class for execution
  * @param pstmt
  * @param record
  * @return
  */
 public PreparedStatement writeRecord(PreparedStatement pstmt, Object record);
}
```

xJCL examples

Example 1:

```
<batch-data-streams>
<bds>
<logical-name>outputStream</logical-name>
<props>
<prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.tests.bds.EchoWriter"/>
<prop name="ds_jndi_name" value="jdbc/fvtdb"/>
<prop name="debug" value="true"/>
</props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.JDBCWriter</impl-class>
</bds>
</batch-data-streams>
```

Example 2:

```
<batch-data-streams>
<bds>
<logical-name>outputStream</logical-name>
<props>
<prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.tests.bds.EchoWriter"/>
<prop name="jdbc_url" value="jdbc:derby:C:\\mysample\\CREDITREPORT"/>
<prop name="jdbc_driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
<prop name="user_id" value="myuserid"/>
<prop name="pswd" value="mypswd"/>
<prop name="debug" value="true"/>
```

```
</props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.LocalJDBCWriter</impl-class>
</bds>
</batch-data-streams>
```

ByteReaderPattern

This pattern is used to read byte data from a file.

Supporting classes

- FileByteReader

Required properties

Table 11. Required properties

Property name	Value
PATTERN_IMPL_CLASS	Class implementing ByteReaderPattern interface
FILENAME	Complete path to the input file

Optional properties

Table 12. Optional properties

Property name	Value	Description
debug	true or false (default is false)	Enables detailed tracing on this batch datastream.
EnablePerformanceMeasurement	true or false (default is false)	Calculates the total time spent in the batch data-streams and the processRecord method, if you are using the GenericXDBatchStep.
EnableDetailedPerformanceMeasurement	true or false (default is false)	Provides a more detailed breakdown of time spent in each method of the batch data-streams.
file.encoding	Encoding of the file.	For example, 8859_1
AppendJobIdToFileName	true or false (default is false)	Appends the JobID to the file name before loading the file.

Interface definition

```
public interface ByteReaderPattern {

 /**
  * Is called by the framework during Step setup stage
  * @param props
  */
 public void initialize(Properties props);

 /**
  *
  * @param reader
  * @throws IOException
  */
 public void processHeader(BufferedInputStream reader) throws IOException;

 /**
  * Get the next record from the input stream
  * @param reader
  * @return
  * @throws IOException
  */
 public Object fetchRecord(BufferedInputStream reader) throws IOException;
}
```

xJCL example

```
<batch-data-streams>
<bds>
<logical-name>inputStream</logical-name>
<props>
<prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.tests.bds.EchoReader"/>
<prop name="file.encoding" value="8859_1"/>
<prop name="FILENAME" value="/opt/txlist.txt" />
<prop name="debug" value="true"/>
</props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.FileByteReader</impl-class>
</bds>
</batch-data-streams>
```

ByteWriterPattern

This pattern is used to write byte data to a file.

Supporting classes

- FileByteWriter

Required properties

Table 13. Required properties

Property name	Value
PATTERN_IMPL_CLASS	Class implementing ByteWriterPattern interface
FILENAME	Complete path to the input file

Optional properties

Table 14. Optional properties

Property name	Value	Description
debug	true or false (default is false)	Enables detailed tracing on this batch datastream.
EnablePerformanceMeasurement	true or false (default is false)	Calculates the total time spent in the batch data-streams and the processRecord method, if you are using the GenericXDBatchStep.
EnableDetailedPerformanceMeasurement	true or false (default is false)	Provides a more detailed breakdown of time spent in each method of the batch data-streams.
file.encoding	Encoding of the file	For example, 8859_1

Table 14. Optional properties (continued)

Property name	Value	Description
AppendJobIdToFileName	true or false (default is false)	Appends the JobID to the file name before loading the file.

Interface definition

```
public interface ByteWriterPattern {

 /**
  * Invoked during the step setup phase
  * @param props
  */
 public void initialize(Properties props);

 /**
  * Writes the given object onto the given outputstream. Any processing
  * that needs to be done before writing can be added here
  * @param out
  * @param record
  * @throws IOException
  */
 public void writeRecord(BufferedOutputStream out, Object record) throws IOException;

 /**
  * Write header information if any
  * @param out
  * @throws IOException
  */
 public void writeHeader(BufferedOutputStream out) throws IOException;

         /**
  * This method can be optionally called during process step to explicity
  * initialize and write the header.
  * @param header
  */
 public void writeHeader(BufferedOutputStream out, Object header) throws IOException;
}
```

xJCL example

```
<batch-data-streams>
<bds>
<logical-name>outputStream</logical-name>
<props>
     <prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.tests.bds.EchoWriter"/>
     <prop name="file.encoding" value="8859_1"/>
     <prop name="FILENAME" value="/opt/txlist.txt" />
     <prop name="debug" value="true"/>
</props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.FileByteWriter</impl-class>
</bds>
</batch-data-streams>
```

FileReaderPattern

This pattern is used to read text data from a file.

Supporting classes

- TextFileReader

Required properties

Table 15. Required properties

Property name	Value
PATTERN_IMPL_CLASS	Class implementing FileReaderPattern interface
FILENAME	Complete path to the input file

Optional properties

Table 16. Optional properties

Property name	Value	Description
debug	true or false (default is false)	Enables detailed tracing on this batch datastream.
EnablePerformanceMeasurement	true or false (default is false)	Calculates the total time spent in the batch data-streams and the processRecord method, if you are using the GenericXDBatchStep.
EnableDetailedPerformanceMeasurement	true or false (default is false)	Provides a more detailed breakdown of time spent in each method of the batch data-streams.
file.encoding	Encoding of the file.	For example, 8859_1
AppendJobIdToFileName	true or false (default is false)	Appends the JobID to the file name before loading the file.

Interface definition

```
public interface FileReaderPattern {

 /**
  * Invoked during the step setup phase
  * @param props
  */
 public void initialize(Properties props);
 /**
  * This method is invoked only once. It should be used
  * to read any header data if necessary.
  * @param reader
  * @throws IOException
  */
 public void processHeader(BufferedReader reader) throws IOException;

 /**
  * This method should read the next line from the reader
  * and return the data in suitable form to be processed
  * by the step.
  * @param reader
  * @return
  * @throws IOException
  */
 public Object fetchRecord(BufferedReader reader) throws IOException;

   /**
   * This method can be optionally invoked from the process step
```

```
    * to obtain the header data that was previously obtained during the
    * processHeader call
    * @return
    */

     public Object fetchHeader();
}
```

xJCL example

```
<batch-data-streams>
<bds>
<logical-name>inputStream</logical-name>
<props>
     <prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.tests.bds.EchoReader"/>
     <prop name="file.encoding" value="8859_1"/>
     <prop name="FILENAME" value="/opt/txlist.txt" />
     <prop name="debug" value="true"/>
</props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.TextFileReader</impl-class>
</bds>
</batch-data-streams>
```

FileWriterPattern

The FileWriterPattern pattern is used to write text data to a file.

Supporting classes

- TextFileWriter

Required properties

Table 17. Required properties

Property name	Value
PATTERN_IMPL_CLASS	Class that implements the FileWriterPattern interface
FILENAME	Complete path to the input file

Optional properties

Table 18. Optional properties

Property name	Value	Description
debug	true or false (default is false)	Enables detailed tracing on this batch datastream.
EnablePerformanceMeasurement	true or false (default is false)	Calculates the total time spent in the batch data-streams and the processRecord method, if you are using the GenericXDBatchStep.
EnableDetailedPerformanceMeasurement	true or false (default is false)	Provides a more detailed breakdown of time spent in each method of the batch data-streams.
file.encoding	Encoding of the file	For example, 8859_1
AppendJobIdToFileName	true or false (default is false)	Appends the JobID to the file name before loading the file.

Interface definition

```
public interface FileWriterPattern {

 /**
  * Invoked during step setup phase
  * @param props
  */
 public void initialize(Properties props);

 /**
  * This method should write the given record
  * object to the bufferedwriter.
  * @param out
  * @param record
  * @throws IOException
  */
 public void writeRecord(BufferedWriter out, Object record) throws IOException;

 /**
  * This method is invoked only once just after the bufferedwriter
  * is opened. It should be used to write any header information
  * @param out
  * @throws IOException
  */
 public void writeHeader(BufferedWriter out) throws IOException;

        /**
  * This method can be optionally called during process step to explicity
  * initialize and write the header.
  * @param header
  * @throws IOException
  */
 public void writeHeader(BufferedWriter out, Object header) throws IOException;

}
```

xJCL example

```
<batch-data-streams>
<bds>
<logical-name>outputStream</logical-name>
<props>
     <prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.tests.bds.EchoWriter"/>
     <prop name="file.encoding" value="8859_1"/>
     <prop name="FILENAME" value="/opt/txlist.txt" />
     <prop name="debug" value="true"/>
</props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.TextFileWriter</impl-class>
</bds>
</batch-data-streams>
```

RecordOrientedDatasetReaderPattern

The RecordOrientedDatasetReaderPattern pattern is used to read data from a z/OS dataset.

Supporting classes

- ZFileStreamOrientedTextReader: Reads text data
- ZFileStreamOrientedByteReader: Reads byte data
- ZFileRecordOrientedDataReader: Reads sequential data

Required properties

Table 19. Required properties

Property name	Value	Description
PATTERN_IMPL_CLASS	Java class name	Class that implements the RecordOrientedDatasetReaderPattern interface
DSNAME	Dataset name	For example, USER216.BATCH.RECORD.OUTPUT

Optional properties

Table 20. Optional properties

Property name	Value	Description
ds_parameters	Parameters used to open the dataset.	Default for ZFileRecordOrientedDataReader is rb,recfm=fb,type=record,lrecl=80 and Default for ZFileStreamOrientedByteReader and ZFileStreamOrientedTextReader is rt
debug	true or false (default is false)	Enables detailed tracing on this batch datastream.
EnablePerformanceMeasurement	true or false (default is false)	Calculates the total time spent in the batch data-streams and the processRecord method, if you are using the GenericXDBatchStep.
EnableDetailedPerformanceMeasurement	true or false (default is false)	Provides a more detailed breakdown of time spent in each method of the batch data-streams.
file.encoding	Encoding of the file.	For example, 8859_1.

Interface definition

```
public interface RecordOrientedDatasetReaderPattern {

 /**
  * This method is invoked during the job setup phase.
  * The properties are the ones specified in the xJCL.
  * @param props
  */
 public void initialize(Properties props);

 /**
  * This method is invoked only once immediately after
  * the Zfile is opened. It should be used to process
  * header information if any.
  * @param reader
  * @throws IOException
  */
 public void processHeader(ZFile reader) throws IOException;

 /**
  * This method should read the next record from the Zfile
  * and return it in an appropriate form (as an intermediate object)
  * @param reader
  * @return
  * @throws IOException
  */
 public Object fetchRecord(ZFile reader) throws IOException;
}
```

xJCL example

```
<batch-data-streams>
<bds>
<logical-name>inputStream</logical-name>
<props>
  <prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.tests.bds.EchoReader"/>
  <prop name="DSNAME" value="USER216.BATCH.RECORD.INPUT"/>
  <prop name="ds_parameters" value="rt"/>
  <prop name="file.encoding" value="CP1047"/>
  <prop name="debug" value="true"/>
</props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.ZFileStreamOrientedByteReader
</impl-class>
</bds>
   <bds>
   <logical-name>outputStream</logical-name>
   <props>
    <prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.tests.bds.EchoWriter"/>
    <prop name="DSNAME" value="USER216.BATCH.RECORD.OUTPUT"/>
    <prop name="ds_parameters" value="wt"/>
    <prop name="file.encoding" value="CP1047"/>
    <prop name="debug" value="${debug}"/>
    </props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.ZFileStreamOrientedByteWriter
</impl-class>
</bds>
</batch-data-streams>
```

RecordOrientedDatasetWriterPattern

The RecordOrientedDataSetWriterPattern pattern is used to write data to a z/OS dataset.

Supporting classes

- ZFileStreamOrientedTextWriter: Writes text data
- ZFileStreamOrientedByteWriter: Writes byte data
- ZFileRecordOrientedDataWriter: Writes sequential data

Required properties

Table 21. Required properties

Property name	Value	Description
PATTERN_IMPL_CLASS	Java class name	Class implementing RecordOrientedDatasetWriterPattern interface
DSNAME	Dataset name	For example, USER216.BATCH.RECORD.OUTPUT

Optional properties

Table 22. Optional properties

Property name	Value	Description
ds_parameters	Parameters used to open the dataset.	Default for ZFileRecordOrientedDataWriter is wb,recfm=fb,type=record,lrecl=80and Default forZFileStreamOrientedByteWriter and ZFileStreamOrientedTextWriter is wt
debug	true or false (default is false)	Enables detailed tracing on this batch datastream.
EnablePerformanceMeasurement	true or false (default is false)	Calculates the total time spent in the batch data-streams and the processRecord method, if you are using the GenericXDBatchStep.
EnableDetailedPerformanceMeasurement	true or false (default is false)	Provides a more detailed breakdown of time spent in each method of the batch data-streams.
file.encoding	Encoding of the file.	For example, CP1047

Interface definition

```
/**
 *
 * This pattern is used to write data to z/OS dataset using
 * jzos apis
 */
public interface RecordOrientedDatasetWriterPattern {

 /**
  * This method is called during the job setup phase allowing
  * the user to do initialization.
  * The properties are the ones passed in the xJCL
  * @param props
  */
 public void initialize(Properties props);

 /**
  * This method should be used to write the given
  * object into the dataset
  * @param out
  * @param record
  * @throws IOException
  */
 public void writeRecord(ZFile out, Object record) throws IOException;

 /**
  * This method should be used to write header information
  * if any
  * @param out
  * @throws IOException
  */
 public void writeHeader(ZFile out) throws IOException;

 /**
```

```
  * This method can be optionally called during process step to explicity
  * initialize and write the header.
  * @param header
  */
 public void writeHeader(ZFile out, Object header);

}
```

xJCL example

```
<batch-data-streams>
<bds>
<logical-name>outputStream</logical-name>
<props>
<prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.tests.bds.EchoWriter"/>
<prop name="DSNAME" value="USER216.BATCH.RECORD.OUTPUT"/>
<prop name="ds_parameters" value="wt"/>
<prop name="file.encoding" value="CP1047"/>
<prop name="debug" value="${debug}"/>
</props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.ZFileStreamOrientedByteWriter
</impl-class>
</bds>
</batch-data-streams>
```

JPAReaderPattern

This pattern is used to retrieve data from a database using OpenJPA.

Supporting classes

- JPAReader: This class implements the basic JPA operations of obtaining an EntityManager and joining or begin/commit of transactions.

  By default the JPAWriter joins an existing global transaction. The user should package a persistence.xml that sets the transaction-type attribute to JTA and declare a jta-data-source

  Optionally the JPAWriter can be configured to begin and commit transactions in synch with the global transactions for use with non-jta-data-sources and connection URLs. In this case the persistence.xml should set the transaction-type to RESOURCE_LOCAL and declare a non-jta-data-source or connection URLs

Required properties

Table 23. Required properties

Property	**Value**
PATTERN_IMPL_CLASS	Class implementing JPAWriterPattern interface
PERSISTENT_UNIT	The OpenJPA persistent unit name.
Any other JPA properties that a user wishes to set on the EntityManager	The values of these properties.

Optional properties

Table 24. Optional properties

Property name	**Value**	**Description**
debug	true or false (default is false)	Enables detailed tracing on this batch data stream

Table 24. Optional properties (continued)

Property name	Value	Description
use_JTA_transactions	true or false (default is true)	If a user wishes to use non-jta-data-source or connection URLs, this value should be false.
EnablePerformanceMeasurement	true or false (default is false)	Calculates the total time spent in the batch data-streams and the processRecord method, if you are using the GenericXDBatchStep.

Interface definition

```
public interface JPAReaderPattern {

 /**
  * This method is invoked during the job setup phase.
  *
  * @param props properties provided in the xJCL
  */

 public void initialize(Properties props);

 /**
  * This method should retrieve values for the various columns for the current
    * row from the given Iterator object. Typically this data would be used
  * to populate an intermediate object which would be returned
  * @param listIt
  * @return
  */
 public Object fetchRecord(Iterator listIt);

 /**
  * This method should return a JPQL query that will be used during setup of the
  * stream to retrieve all relevant data that would be processed part of the job
  *  steps @return object to be used during process step.
  */
 public String getInitialLookupQuery();

 /**
  * This method gets called during Job Restart. The restart token should be used
  * to create an JPQL query that will retrieve previously unprocessed records.
  * Typically the restart token would be the primary key in the table and the
  * query would get all rows with primary key value > restarttoken
  * @param restartToken
  * @return The restart query
  */
 public String getRestartQuery(String restartToken);

 /**
  * This method gets called just before a checkpoint is taken.
  * @return The method should return a string value identifying the last record
    * read by the stream.
  */
 public String getRestartTokens();

}
```

xJCL example

```
<batch-data-streams>
<bds>
<logical-name>inputStream</logical-name>
<props>
```

```
<prop name="openjpa.ConnectionDriverName" value="org.apache.derby.jdbc.EmbeddedDriver"/>
<prop name="openjpa.ConnectionURL" value="jdbc:derby:/opt/tmp/hellojpadb;create=true"/>
<prop name="openjpa.ConnectionUserName" value="" />
<prop name="openjpa.ConnectionPassword" value="" />
<prop name="openjpa.jdbc.SynchronizeMappings" value="buildSchema"/>
<prop name="openjpa.Log" value="DefaultLevel=WARN,SQL=TRACE"/>
<prop name="PERSISTENT_UNIT" value="hellojpa"/>
<prop name="debug" value="true"/>
<prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.samples.JPAOutputStream"/>
</props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.LocalJDBCReader</impl-class>
</bds>
</batch-data-streams>
```

JPAWriterPattern

This pattern is used to write data to a database using a Java Persistence API (JPA) connection.

Supporting classes

- JPAWriter: This class implements the basic JPA operations of obtaining an EntityManager and joining or begin/commit of transactions.

  By default the JPAWriter joins an existing global transaction. The user should package a persistence.xml that sets the transaction-type attribute to JTA and declare a jta-data-source

  Optionally the JPAWriter can be configured to begin and commit transactions in synch with the global transactions for use with non-jta-data-sources and connection URLs. In this case the persistence.xml should set the transaction-type to RESOURCE_LOCAL and declare a non-jta-data-source or connection URLs

Required properties

Table 25. Required properties

Property	**Value**
PATTERN_IMPL_CLASS	Class implementing JPAWriterPattern interface
PERSISTENT_UNIT	The OpenJPA persistent unit name.
Any other JPA properties that a user wishes to set on the EntityManager	The values of these properties.

Optional properties

Table 26. Optional properties

Property name	**Value**	**Description**
debug	true or false (default is false)	Enables detailed tracing on this batch data stream
use_JTA_transactions	true or false (default is true)	If a user wishes to use non-jta-data-source or connection URLs, this value should be false.
EnablePerformanceMeasurement	true or false (default is false)	Calculates the total time spent in the batch data-streams and the processRecord method, if you are using the GenericXDBatchStep.

Interface definition

```
public interface JPAWriterPattern {

 /**
  * This method is invoked during create job step to allow the JPAWriter stream to
  * initialize.
  * @param props Properties passed via xJCL
  */
public void initialize(Properties props);

 /**
  * This method is invoked to actually persist the passed object to the database
  * using JPA EntityManager
  * @param manager
  * @param record
  */
 public void writeRecord(EntityManager manager, Object record);
}
```

xJCL example

```
<batch-data-streams>
<bds>
<logical-name>outputStream</logical-name>
<props>
<prop name="PATTERN_IMPL_CLASS" value="com.ibm.websphere.batch.samples.tests.bds.EchoWriter"/>
<prop name="jdbc_url" value="jdbc:derby:C:\\mysample\\CREDITREPORT"/>
<prop name="jdbc_driver" value="org.apache.derby.jdbc.EmbeddedDriver"/>
<prop name="user_id" value="myuserid"/>
<prop name="pswd" value="mypswd"/>
<prop name="debug" value="true"/>
</props>
<impl-class>com.ibm.websphere.batch.devframework.datastreams.patterns.LocalJDBCWriter</impl-class>
</bds>
</batch-data-streams>
```

Common batch container jobs

All jobs contain the following information:

- The identity of the common batch container job that performs the work
- One or more job steps that need to be performed to complete the work
- The identity of an artifact within the application that provides the logic for each job step
- Key and value pairs for each job step to provide additional context to the application artifacts
- Definitions of sources and destinations for data
- Definitions of checkpoint algorithms

Creating batch job steps

Call back methods in the BatchJobStepLocalInterface allow the common batch container to run batch steps when it runs a batch job. Typically, a batch step contains code to read a record from a batch data stream, perform business logic with that record and then continue to read the next batch container in a batch loop. This method contains all the logic that can be batched to perform on data.

The following common batch container callback methods exist on the BatchJobStepLocalInterface that are invoked by the common batch container in the following ordered list:

1. setProperties(java.util.Properties properties): Makes properties defined in XML Job Control Language (xJCL) available to batch step in a java.util.Properties object.
2. void createJobStep(): Indicates to the step that it has been initialized. Initialization logic, such as retrieving a handle to a batch data stream, can be placed here.
3. int processJobStep(): Repeatedly invoked by common batch container in a batch loop until the return code integer of this method indicates that the step has finished processing. Review BatchConstants in the batch API to see which return codes can be returned. A return code of BatchConstants.STEP_CONTINUE signals to the common batch container to continue calling this method in the batch loop. A return code of BatchConstants.STEP_COMPLETE indicates to the common batch container that the step has finished, and destroyJobStep should be called.
4. int destroyJobStep() - indicates to the step that completion has occurred. The integer return code of this method is arbitrary and can be chosen by the batch application developer. This return code is saved in the common batch container database and represents the return code of the batch step. If the results algorithm is associated with the batch job, then this return code is passed to it. If there is a return code-based conditional logic in the xJCL of the batch job, then the common batch container uses this return code to evaluate that logic.

The getProperties() method on the BatchJobStepLocalInterface is not currently called by the common batch container. The method is included in the interface for symmetry and possible later use.

Generic batch step (GenericXDBatchStep)

A generic batch step works with exactly one input and one output stream. This step during each iteration of the batch loop reads a single entry from the BDS Input Stream passes it to the BatchRecordProcessor for processing. The BatchRecordProcessor returns the processed data which is then passed to the BDS output stream.

Table 27. Required properties

Property name	Value	Description
BATCHRECORDPROCESSOR	java class name	Class implementing the BatchRecordProcessor interface

Table 28. Optional properties

Property	Value	Description
debug	true or false (default is false)	Enable tracing and debugging on the step
EnablePerformanceMeasurement	true or false (default is false)	Measure time spent within the step

Procedure:

1. Implement the following interface to provide the business logic for the step.
   `com.ibm.websphere.batch.devframework.steps.technologyadapters.BatchRecordProcessor`

The xJCL for the step should declare a property BATCHRECORDPROCESSOR with the value set to the implementation of the interface. For example:

```
...
<props>
 <prop name="BATCHRECORDPROCESSOR"  value=
  "com.ibm.websphere.batch.samples.tests.steps.InfrastructureVerificationTest"/>
</props>
...
```

2. Set the BDS input stream logical name to inputStream and a BDS output stream logical name to outputStream.

   The logical names are declared in the xJCL. For example:

```
<batch-data-streams>
   <bds>
 <logical-name>inputStream</logical-name>
 <props>
        ....
   </bds>
    <bds>
 <logical-name>outputStream</logical-name>
 <props>
 ...
   </bds>
</batch-data-streams>
```

3. While using the BatchPackager for packaging, jobstepclass for the application must be set to the following value:

```
com.ibm.websphere.batch.devframework.steps.technologyadapters.GenericXDBatchStep
```

   For example:

```
ejbname.1=IVTStep1
jndiname.1=ejb/GenericXDBatchStep
jobstepclass.1=
   com.ibm.websphere.batch.devframework.steps.technologyadapters.GenericXDBatchStep
```

Error Tolerant step

An error tolerant generic batch step works with exactly one input, one output stream and one error stream. This step during each iteration of the batch loop reads a single entry from the batch data stream (BDS) input stream passes it to the BatchRecordProcessor for processing

The BatchRecordProcessor may either return a valid data object or a null value in case of a tolerable error. If the returned value is null, the record read from the inputStream is logged onto the errorstream and the invalidRecordEncountered method is invoked on the ThresholdPolicy. The threshold policy determines whether the error tolerance threshold has been reached. If so, it returns STEP_CONTINUE_FORCE_CHECKPOINT_BEFORE_PROCESSING_CANCEL, which forces a checkpoint and puts the job in the restartable state. Otherwise, the job continues as normal. If the data returned by BatchRecordProcessor.processRecord is valid, then the data is passed to the BDS Outputstream.

Table 29. Required properties

Property name	Value	Description
threshold_policy	java class name	Class implementing the com.ibm.websphere.batch.devframework.thresholdpolicies.ThresholdPolicy interface

Table 29. Required properties (continued)

Property name	Value	Description
BATCHRECORDPROCESSOR	java class name	Class implementing the BatchRecordProcessor interface

Table 30. Optional properties

Property	Value	Description
debug	true or false (default is false)	Enable tracing and debugging on the step
EnablePerformanceMeasurement	true or false (default is false)	Measure time spent within the step

Procedure:

1. Implement the following interface to provide the business logic for the step:

   `com.ibm.websphere.batch.devframework.steps.technologyadapters.BatchRecordProcessor`

   The xJCL for the step should declare a property BATCHRECORDPROCESSOR with the value set to the implementation of the interface. For example:

```
...
<props>
   <prop name="BATCHRECORDPROCESSOR"
      value="com.ibm.websphere.batch.samples.tests.steps.InfrastructureVerificationTest"/>
</props>
...
```

2. Implement the following interface to provide the threshold policy for the step:

   `com.ibm.websphere.batch.devframework.thresholdpolicies.ThresholdPolicy`

   Declare the ThresholdPolicy to use in the xJCL as shown in the following code snippet:

```
...
<props>
   <prop name="threshold_policy"
      value="com.ibm.websphere.batch.devframework.thresholdpolicies.PercentageBasedThresholdPolicy"/>
</props>
...
```

   You can also use the product implementations such as the following:

```
com.ibm.websphere.batch.devframework.thresholdpolicies.PercentageBasedThresholdPolicy
```

   or

```
com.ibm.websphere.batch.devframework.thresholdpolicies.RecordBasedThresholdPolicy
```

3. Set the BDS input stream logical name to inputStream and a BDS output stream logical name to outputStream and the BDS output stream for errors to errorStream.

   The logical names are declared in the xJCL. For example:

```
<batch-data-streams>
   <bds>
 <logical-name>inputStream</logical-name>
 <props>
       ....
   </bds>
    <bds>
 <logical-name>outputStream</logical-name>
```

```
<props>
...
   </bds>
   <bds>
<logical-name>errorStream</logical-name>
<props>
...
    </bds>
</batch-data-streams>
```

4. While using the BatchPackager for packaging, the jobstepclass for the application must be set to the following value:

   `com.ibm.websphere.batch.devframework.steps.technologyadapters.ThresholdBatchStep`

   For example:

```
ejbname.1=IVTStep1
jndiname.1=ejb/MyThresholdBatchStep
jobstepclass.1=com.ibm.websphere.batch.devframework.steps.technologyadapters.ThresholdBatchStep
```

PercentageBasedThresholdPolicy

This policy provides a common batch container implementation of the ThresholdPolicy interface.

The percentageBasedThresholdPolicy is applicable only if the ThresholdBatchStep is used. It calculates the percentage of the number of error records processed to the total number processed. If the result is greater than the threshold, it forces the job to go into restartable state.

Table 31. Optional properties

Property	Value	Description
debug	true or false (default is false)	Enable tracing and debugging on the step
minimum_threshold_sample_size	Integer value (default is 20)	The minimum number of records to process before checking for threshold breach.
threshold_threshold	Double value (default is 0.1)	The acceptable percentage of errors.

Procedure:

1. Declare the ThresholdPolicy to use in the xJCL as a property of the step.

```
...
<props>
  <prop name="threshold_policy"
      value="com.ibm.websphere.batch.devframework.thresholdpolicies.PercentageBasedThresholdPolicy"/>
</props>
...
```

RecordBasedThresholdPolicy

This policy provides a grid implementation of the ThresholdPolicy interface.

The RecordBasedThresholdPolicy is applicable only if the ThresholdBatchStep is used. It counts the number of error records processed if the result is greater than the threshold forces the job to go into restartable state.

Table 32. Optional properties

Property	Value	Description
debug	true or false (default is false)	Enable tracing and debugging on the step
minimum_threshold_size	Integer value (default is 20)	The minimum number of records to process before checking for threshold breach.
error_threshold	Double value (default is 100)	The number of error records.

Procedure: Declare the ThresholdPolicy to use in the xJCL as a property of the step as follows:

```
...
<props>
  <prop name="threshold_policy"
     value="com.ibm.websphere.batch.devframework.thresholdpolicies.RecordBasedThresholdPolicy"/>
</props>
...
```

xJCL elements

Jobs are described using a job control language. Batch jobs use an XML-based job control language. The job description identifies which application to run, its inputs, and outputs.

Jobs are expressed using an Extensible Markup Language XML dialect called XML xJCL Job Control Language. See the xJCL provided with the Sample applications, the xJCL table and xJCL XSD schema document for more information about xJCL. The xJCL definition of a job is not part of the batch application, but is constructed separately.

The following table summarizes the xJCL elements:

Table 33. xJCL elements

Element	Sub-element	Attributes	Description
job	not-applicable	not-applicable	Scopes the description of a batch job.
not-applicable	not-applicable	name	Name of the job. This name must match the name of the batch application.
not-applicable	step-scheduling-criteria	See step-scheduling-criteria element	not-applicable
not-applicable	checkpoint-algorithm	See checkpoint-algorithm element	not-applicable
not-applicable	job-step	See job-step element	not-applicable
job-step	not-applicable	not-applicable	not-applicable
not-applicable	not-applicable	name	Name of the step. This information is returned on operational commands.
not-applicable	step-scheduling	See step-scheduling element	Allows for conditional logic based on return codes of steps that determine if the batch step should be invoked or not

Table 33. xJCL elements (continued)

Element	Sub-element	Attributes	Description
not-applicable	checkpoint-algorithm-ref	See checkpoint-algorithm-ref element	Specifies the checkpoint algorithm to use for the batch job step.
not-applicable	classname	not-applicable	Fully-qualified name of class that implements the compute intensive job.
not-applicable	props	See props element	Name-value properties to pass to the step
not-applicable	batch-data-streams	See batch-data-streams element	A sequence of `bds` elements. Each `bds` is the configuration information necessary to create a batch data stream.
prop	not-applicable	not-applicable	Single instance of a name value pair, that serves as a property.
not-applicable	not-applicable	name	Name of the property.
not-applicable	not-applicable	value	Value of the property.
props	not-applicable	not-applicable	Series of `prop` elements that are used to pass name value pair properties to steps, `bds`, checkpoint algorithms and results algorithms.
not-applicable	prop	See prop element	not-applicable
bds	not-applicable	not-applicable	Single instance of a batch data stream implementation that is made available to the batch job to use.
not-applicable	logical-name	not-applicable	A string that is embedded in batch step that the batch step uses to ask the batch runtime environment for a specific batch data stream instance.
not-applicable	impl-class	not-applicable	Fully qualified class name of the batch data stream implementation class.
not-applicable	props	See Props elements	List of properties that are passed to the batch data stream implementation class.
batch-data-streams	not-applicable	not-applicable	Series of `bds` elements
not-applicable	bds	see bds element	not-applicable
step-scheduling	not-applicable	not-applicable	Can be applied to job steps to create return code-based conditional flows for a batch job. Can compare values of return codes defined for this batch job to decide whether a step is invoked or not while processing a batch job. The values of return codes are compared using the return code-expression element.
not-applicable	returncode- expression	see returncode- expression	Returncode- expression for evaluation.
not-applicable	not-applicable	condition	If there is more than one returncode-expression element in the step-scheduling element, then conditional operators can be applied to them. Conditional operators supported are: AND, OR.

Table 33. xJCL elements (continued)

Element	Sub-element	Attributes	Description
returncode-expression	not-applicable	not-applicable	Used for step scheduling tags to decide if a batch job step is run based on return codes of other job steps.
not-applicable	not-applicable	step	Name of step whose return code is to be compared in this expression.
not-applicable	not-applicable	operator	Operator to use for the return code expression; the supported operators are: eq equals, lt less than, gt greater than, le less than or equal to, ge greater than or equal to.
not-applicable	not-applicable	value	The value with which to compare the return code.
step-scheduling-criteria	not-applicable	not-applicable	Describes the sequence in which the job steps are processed. Currently sequential scheduling is supported, and steps get invoked in the order in which they are displayed in xJCL.
not-applicable	scheduling-mode	not-applicable	Sequence in which to invoke steps: only possible value is sequential
checkpoint-algorithm	not-applicable	not-applicable	Declares a checkpoint algorithm that can be used for a batch job step.
not-applicable	not-applicable	name	Name of algorithm.
not-applicable	classname	not-applicable	Class that implements this algorithm.
not-applicable	props	see props element	Sequence of `props` elements for the checkpoint algorithm.
checkpoint-algorithm-ref	not-applicable	not-applicable	Reference to a checkpoint algorithm element.
not-applicable	not-applicable	name	Name of checkpoint algorithm to which you are referring.
not-applicable	props	see props element	Sequence of prop elements for the checkpoint algorithm.
++ The xJCL element substitution-props is discussed in the following section.			

xJCL substitution-props

The job xJCL can contain symbolic variables. A symbolic variable is an expression of the form ${variable-name}, which is found outside a comment in an otherwise well-formed document. For example:

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

The xJCL element, substitution-props, defines a default name and value pairs for symbolic variables. Following is an example of the substitution-props element, taken from the postingSampleXJCL.xml document:

```
<substitution-props>
<prop name="wsbatch.count" value="5" />
<prop name="checkpoint" value="timebased" />
<prop name="checkpointInterval" value="15" />
<prop name="postingsDataStream"
   value="${was.install.root}${file.separator}temp${file.separator}postings" />
</substitution-props>
```

Substitution for symbolic variables occurs at run time. At run time, the string ${variable-name} is replaced with the value of the property when the xJCL is submitted for execution. Using the properties in the previous example, the string ${checkpoint} is replaced with the string timebased before the job is submitted.

Symbolic variables can be indirect. For example: name=FILENAME value=${${filename}} used with the name/value pair filename=postingsDataStream yields the same result as specifying name=FILENAME value=${postingsDataStream}.

Symbolic variables can also be compound. For example:

```
name=postingsDataStream
  value=${was.install.root}${file.separator}temp${file.separator}postings.
```

The name/value pairs must be defined in the job document substitution-props element. The props name and value pairs are initially defined to the default values for the named variables. The following two conditions must be met before an xJCL is considered valid:.

- Every symbolic variable defined in the body of a job document must be resolved.
- Every name/value pair defined in the job document must resolve to a symbolic variable which is found in the body of the xJCL.

Batch job return codes explanations

The following table lists the system batch job return codes that the common batch container uses. Do not confuse the batch job return code with either the JobStatusConstants (see the com.ibm.websphere.longrun.JobStatusConstants API) or the JobSchedulerConstants (see the com.ibm.websphere.longrun.JobSchedulerConstants API). The JobStatusConstants represent the status of the job such as submitted, ended, restartable, canceled, or execution failed.

The JobSchedulerConstants represent operating conditions returned by the job scheduler on requests involving multiple jobs. For example, int[] cancelJob(String[] jobid)). These conditions include:

1. Job does not exist
2. Job is in an invalid state
3. Database exception has occurred.

Table 34. Return codes and explanations

Return code	Explanation
0	Job ended normally
-4	Job was suspended
-8	Job was canceled
-10	Job was forcibly canceled
-12	Job failed and is in restartable state
-14	Job failed and is in execution failed state**

Table 34. Return codes and explanations (continued)

Return code	Explanation
** This return code value does not apply in the case where the application returns BatchConstants.STEP_COMPLETE_EXECUTION_FAILED from the processJobStep method. In this case, the return code is determined by the application.	

There are two options that are used to report an error in a batch application. The first option is for the application to throw an exception when an error is encountered. This results in termination of the job with a batch job return code of -12 and a batch job status of *restartable*. The second option is for the application to return a BatchConstants.STEP_COMPLETE_EXECUTION_FAILED return code (see the com.ibm.websphere.batch.BatchConstants API) from the processJobStep method and return an application-specific error return code from the destroyJobStep method. This results in termination of the job and a batch job status of *execution failed*. The step return code set in the destroyJobStep method is passed to any results algorithm specified on the job step and is used to influence the return code of the job to indicate the specific cause of the failure.

Job logs

A job log is a file that contains a detailed record of the execution details of a job. It is comprised of both system and application messages.

A job log is a file that contains a detailed record of the execution details of a job. System messages from the batch container and output from the job executables are collected. By examining job logs, you can see the life cycle of a batch job, including output from the batch applications themselves.

A job log is composed of the following three types of information:

1. xJCL - A job log contains a copy of the xJCL used to run the job, including xJCL substitution values.
2. System messages - A set of system messages that communicate the major life cycle events corresponding to the job. The following system events are recorded in a job log:
   - Begin and end of a job
   - Begin and end of a step
   - Begin and end of a checkpoint
   - Open, close, and checkpoint of a batch data stream
   - Checkpoint algorithm invocation / results
   - Results algorithm invocation / results
3. Application messages - A set of messages written to standard out and standard error by a job step program.

Output of a job log

Job log output is collected in a directory which has the format:

```
specified_root/joblogs/job-directory/timeStamp-directory
```

In this directory path:

- *specified_root* is the root directory for your logs.
- *job-directory* is generated at run time from the job name. For example, if the job ID is PostingsSampleEar:99, then the generated directory name is PostingsSampleEar_99.

- *timeStamp-directory* Is generated at run time from the current date. It is in the format ddmmyyy_hhmmss, where dd is the day of the month, mm is a month (00 - 11), and yyyy is the year. hh is the hour of the day (00 - 23), mm is the minute of the hour (00 - 59) and ss is the seconds of the minutes (00 - 59). For example, a timestamp directory with the name 14022010_164535 means that the job began processing on 14 Mar 2010, at 16:45:35.

The job log output contains both application output and batch endpoint runtime messages. This output includes any application generated output directed to the System.out and System.err output streams. Job log output from the common batch container is collected in the job log directory in files with names such as part.1.log and part.2.log. Each of the log parts contains approximately 1000 records. The following example shows the contents of part.1.log:

```
CWLRB5588I: [03/13/07 08:25:32:104 EDT] Setting up j2ee job SimpleCIEar:44 for execution in batch endpoint
   dmgrCell/lreeNode/lreeServer: [jobClass Default] [jobName SimpleCIEar] [module null]
   [user UNAUTHENTICATED] [applicationName SimpleCIEar] [applicationType j2ee]
CWLRB5784I: [03/13/07 08:25:32:696 EDT] Setting step SLSB property: calculationTimeInSecs=30
CWLRB5784I: [03/13/07 08:25:32:696 EDT]
   Setting step SLSB property: outputFileName=/dtc/bin/XD/temp/simpleCI.txt
System.out: [03/13/07 08:25:32:708 EDT] Tue Mar 13 08:25:32 EDT 2007: SimpleCI application starting...
System.out: [03/13/07 08:25:32:708 EDT] -->Will loop processing a variety of math functions for
   approximately 30.0 seconds!
System.out: [03/13/07 08:26:02:752 EDT] Tue Mar 13 08:26:02 EDT 2007: SimpleCI application complete!
System.out: [03/13/07 08:26:02:753 EDT] -->Actual Processing time = 30.043 seconds!
CWLRB5764I: [03/13/07 08:26:03:069 EDT] Job SimpleCIEar:44 ended
```

XML schema for a batch job

The following example shows the XML schema for a batch job:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:element name="classname" type="xsd:string" />
 <xsd:element name="impl-class" type="xsd:string" />
 <xsd:element name="jndi-name" type="xsd:string" />
 <xsd:element name="logical-name" type="xsd:string" />

 <xsd:element name="scheduling-mode">
     <xsd:simpleType>
         <xsd:restriction base="xsd:string">
             <xsd:pattern value="sequential"/>
         </xsd:restriction>
     </xsd:simpleType>
 </xsd:element>

 <xsd:element name="required" >
     <xsd:simpleType>
         <xsd:restriction base="xsd:string">
             <xsd:pattern value="[YNyn]"/>
         </xsd:restriction>
     </xsd:simpleType>
 </xsd:element>

 <xsd:element name="batch-data-streams">
   <xsd:complexType>
       <xsd:sequence>
          <xsd:element maxOccurs="unbounded" minOccurs="1" ref="bds" />
       </xsd:sequence>
   </xsd:complexType>
 </xsd:element>

 <xsd:element name="job-scheduling-criteria">
   <xsd:complexType>
       <xsd:sequence>
```

```
            <xsd:element maxOccurs="unbounded" minOccurs="1" ref="required-capability" />
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

<xsd:element name="bds">
   <xsd:complexType>
       <xsd:all>
           <xsd:element ref="logical-name" minOccurs="1" maxOccurs="1"/>
           <xsd:element ref="impl-class" minOccurs="1" maxOccurs="1"/>
           <xsd:element ref="props" minOccurs="0" maxOccurs="1"/>
       </xsd:all>
   </xsd:complexType>
</xsd:element>

<xsd:element name="checkpoint-algorithm">
   <xsd:complexType>
       <xsd:sequence>
           <xsd:element ref="classname" minOccurs="1" maxOccurs="1"/>
           <xsd:element ref="props" minOccurs="0" maxOccurs="unbounded"/>
       </xsd:sequence>
       <xsd:attribute name="name" type="xsd:string" use="required" />
   </xsd:complexType>
</xsd:element>

<xsd:element name="checkpoint-algorithm-ref">
   <xsd:complexType>
       <xsd:attribute name="name" type="xsd:string" use="required" />
   </xsd:complexType>
</xsd:element>

<xsd:element name="required-capability">
   <xsd:complexType>
       <xsd:attribute name="expression" type="xsd:string" use="required" />
   </xsd:complexType>
</xsd:element>

<xsd:element name="results-algorithm">
   <xsd:complexType>
       <xsd:sequence>
           <xsd:element ref="classname" minOccurs="1" maxOccurs="1"/>
           <xsd:element ref="props" minOccurs="0" maxOccurs="1"/>
           <xsd:element ref="required" minOccurs="0" maxOccurs="1"/>
       </xsd:sequence>
       <xsd:attribute name="name" type="xsd:string" use="required" />
   </xsd:complexType>
</xsd:element>

<xsd:element name="results-algorithms">
       <xsd:complexType>
           <xsd:sequence>
               <xsd:element maxOccurs="unbounded" minOccurs="1" ref="results-algorithm" />
           </xsd:sequence>
       </xsd:complexType>
</xsd:element>

<xsd:element name="results-ref">
       <xsd:complexType>
           <xsd:attribute name="name" type="xsd:string" use="required" />
       </xsd:complexType>
</xsd:element>

<xsd:element name="substitution-props">
   <xsd:complexType>
       <xsd:sequence>
           <xsd:element ref="prop" minOccurs="0" maxOccurs="unbounded"/>
       </xsd:sequence>
   </xsd:complexType>
</xsd:element>


<xsd:element name="job">
   <xsd:complexType>
     <xsd:sequence>
         <xsd:element ref="jndi-name" minOccurs="1" maxOccurs="1"/>
         <xsd:element ref="job-scheduling-criteria" minOccurs="0" maxOccurs="1"/>
         <xsd:element ref="step-scheduling-criteria" minOccurs="0" maxOccurs="1"/>
         <xsd:element ref="checkpoint-algorithm" maxOccurs="unbounded"  minOccurs="1"/>
         <xsd:element ref="results-algorithms" maxOccurs="1"  minOccurs="0"/>
         <xsd:element ref="substitution-props" minOccurs="0" maxOccurs="1"/>
         <xsd:element ref="job-step" maxOccurs="unbounded" minOccurs="1" />
     </xsd:sequence>
     <xsd:attribute name="name" type="xsd:string" use="required" />
     <xsd:attribute name="class" type="xsd:string" use="optional" />
     <xsd:attribute name="accounting" type="xsd:string" use="optional" />
     <xsd:attribute name="default-application-name" type="xsd:string" use="optional" />
   </xsd:complexType>
</xsd:element>

<xsd:element name="job-step">
   <xsd:complexType>
       <xsd:sequence>
           <xsd:element ref="step-scheduling" minOccurs="0" maxOccurs="1"/>
           <xsd:element ref="jndi-name" minOccurs="1" maxOccurs="1"/>
           <xsd:element ref="checkpoint-algorithm-ref" minOccurs="0" maxOccurs="1"/>
           <xsd:element ref="results-ref" minOccurs="0" maxOccurs="unbounded"/>
           <xsd:element ref="batch-data-streams" minOccurs="0" maxOccurs="1"/>
           <xsd:element ref="props" minOccurs="0" maxOccurs="1"/>
       </xsd:sequence>
       <xsd:attribute name="name" type="xsd:string" use="optional" />
       <xsd:attribute name="application-name" type="xsd:string" use="optional" />
   </xsd:complexType>
</xsd:element>


<xsd:element name="prop">
   <xsd:complexType>
       <xsd:attribute name="name" type="xsd:string" use="required" />
       <xsd:attribute name="value" type="xsd:string" use="required" />
   </xsd:complexType>
</xsd:element>

<xsd:element name="props">
   <xsd:complexType>
       <xsd:sequence>
           <xsd:element ref="prop" maxOccurs="unbounded" minOccurs="0"/>
       </xsd:sequence>
   </xsd:complexType>
</xsd:element>

<xsd:element name="returncode-expression">
   <xsd:complexType>
       <xsd:attribute name="step" type="xsd:string" use="required" />
       <xsd:attribute name="operator" type="xsd:string" use="required" />
       <xsd:attribute name="value" type="xsd:string" use="required" />
   </xsd:complexType>
</xsd:element>

<xsd:element name="step-scheduling">
   <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="returncode-expression" minOccurs="1" maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="condition" type="xsd:string" use="optional" />
   </xsd:complexType>
</xsd:element>

<xsd:element name="step-scheduling-criteria">
   <xsd:complexType>
       <xsd:sequence>
           <xsd:element ref="scheduling-mode" minOccurs="1" maxOccurs="1" />
       </xsd:sequence>
   </xsd:complexType>
</xsd:element>

</xsd:schema>
```

xJCL sample for a batch job

The following sample illustrates a batch job.

```
<job name="PostingsSampleEar" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">

        <jndi-name>ejb/com/ibm/websphere/samples/PostingsJob</jndi-name>


    <step-scheduling-criteria>
  <scheduling-mode>sequential</scheduling-mode>
 </step-scheduling-criteria>


    <checkpoint-algorithm name="${checkpoint}">
  <classname>com.ibm.wsspi.batch.checkpointalgorithms.${checkpoint}</classname>
  <props>
   <prop name="interval" value="${checkpointInterval}" />
  </props>
 </checkpoint-algorithm>


    <results-algorithms>
  <results-algorithm name="jobsum">
   <classname>com.ibm.wsspi.batch.resultsalgorithms.jobsum</classname>
  </results-algorithm>
 </results-algorithms>


    <substitution-props>
        <prop name="wsbatch.count" value="5" />
        <prop name="checkpoint" value="timebased" />
        <prop name="checkpointInterval" value="15" />
        <prop name="postingsDataStream"
           value="${was.install.root}${file.separator}temp${file.separator}postings" />
    </substitution-props>

    <job-step name="Step1">


        <jndi-name>ejb/DataCreationBean</jndi-name>

        <!-- apply checkpoint policy to step1 -->
        <checkpoint-algorithm-ref name="${checkpoint}" />


        <results-ref name="jobsum"/>


  <batch-data-streams>
   <bds>

                <logical-name>myoutput</logical-name>
                <impl-class>com.ibm.websphere.samples.PostingOutputStream</impl-class>

                <props>

                    <prop name="FILENAME" value="${postingsDataStream}" />

    </props>
   </bds>
  </batch-data-streams>


        <props>

            <prop name="wsbatch.count" value="${wsbatch.count}" />
  </props>
 </job-step>


 <job-step name="Step2">


        <step-scheduling condition="OR">
   <returncode-expression step="Step1" operator="eq" value="0" />
            <returncode-expression step="Step1" operator="eq" value="4" />
  </step-scheduling>

        <jndi-name>ejb/PostingAccountData</jndi-name>
  <checkpoint-algorithm-ref name="${checkpoint}" />
  <results-ref name="jobsum"/>

  <batch-data-streams>
   <bds>

                <logical-name>myinput</logical-name>
    <impl-class>com.ibm.websphere.samples.PostingStream</impl-class>


                <props>
     <prop name="FILENAME" value="${postingsDataStream}" />
    </props>

   </bds>
  </batch-data-streams>
 </job-step>

         <job-step name="Step3">
       <step-scheduling>
           <returncode-expression step="Step2" operator="eq" value="4" />
       </step-scheduling>

       <jndi-name>ejb/OverdraftAccountPosting</jndi-name>
       <checkpoint-algorithm-ref name="${checkpoint}" />
       <results-ref name="jobsum" />


      <batch-data-streams>
           <bds>

              <logical-name>dbread</logical-name>
               <impl-class>com.ibm.websphere.samples.OverdraftInputStream</impl-class>
          </bds>
      </batch-data-streams>
   </job-step>
</job>
```

Implementing Checkpoint algorithms

The common batch container use checkpoint algorithms to determine when to commit global transactions under which batch steps are invoked. These algorithms are applied to a batch job through the XML Job Control Language (xJCL) definition. Properties specified for checkpoint algorithms in xJCL allow for checkpoint behavior, such as transaction timeouts and checkpoint intervals, to be customized for batch steps. The common batch container supports both a time-based checkpoint algorithm and a record-based algorithm, and defines a service provider interface (SPI) for building additional custom checkpoint algorithms.

On each batch step iteration of the processJobStep method, the common batch container consults the checkpoint algorithm applied to that step if it commits the global transaction or not. Callback methods on the checkpoint algorithms allow the common batch container to inform the algorithm when a global transaction is committed or started. This behavior enables the algorithm to keep track of the global transaction life cycle. On each iteration of the processJobStep method, the common batch container calls the ShouldCheckpointBeExecuted callback method on the algorithm to determine if the transaction is committed. The algorithm controls the checkpoint interval through this method.

Review the batch API for the checkpoint algorithm SPI that you can use to create custom checkpoint algorithms. The class name is com.ibm.wsspi.batch.CheckpointPolicyAlgorithm.

Time-based algorithm

The time-based checkpoint algorithm commits global transactions at a specified time interval. The following example declares a time-based algorithm in xJCL:

```
<checkpoint-algorithm name="timebased">
     <classname>com.ibm.wsspi.batch.checkpointalgorithms.timebased</classname>
     <props>
           <prop name="interval" value="15" />
           <prop name="TransactionTimeOut" value="30" />
     </props>
</checkpoint-algorithm>
```

The units of interval and TransactionTimeOut properties in the previous example are expressed in seconds.

Record-based algorithm

The record-based checkpoint algorithm commits global transactions at a specified number of iterations of the processJobStep method of batch step. Each call to the processJobStep method is treated as iterating through one record. The processJobStep method can retrieve multiple records from a batch data stream on each call. However, for this checkpoint algorithm one record is the equivalent one call to the processJobStep method.

The following example declares a record-based algorithm in xJCL:

```
<checkpoint-algorithm name="recordbased">
     <classname>com.ibm.wsspi.batch.checkpointalgorithms.recordbased</classname>
     <props>
           <prop name="recordcount" value="1000" />
           <prop name="TransactionTimeOut" value="60" />
     </props>
</checkpoint-algorithm>
```

The unit of the TransactionTimeOut property in the previous example is expressed in seconds.

If not specified in xJCL, the default transaction timeout is 60 seconds and the default record count is 10000.

Applying a checkpoint algorithm to a batch step

Checkpoint algorithms are applied to a batch job through xJCL. You can declare multiple checkpoint algorithms in xJCL, and you can apply a different algorithm to each batch step. You can apply no more than one checkpoint algorithm to a batch step.

The following example applies checkpoint algorithms in xJCL:

```
<job name="PostingsSampleEar">

<checkpoint-algorithm name="timebased">
     <classname>com.ibm.wsspi.batch.checkpointalgorithms.timebased</classname>
     <props>
           <prop name="interval" value="15" />
           <prop name=" TransactionTimeOut" value="30" />
     </props>
</checkpoint-algorithm>

<checkpoint-algorithm name="recordbased">
     <classname>com.ibm.wsspi.batch.checkpointalgorithms.recordbased</classname>
     <props>
           <prop name="recordcount" value="1000" />
           <prop name="TransactionTimeOut" value="60" />
     </props>
</checkpoint-algorithm>

<job-step name="Step1">
     <checkpoint-algorithm-ref name="timebased" />
</job-step>

<job-step name="Step2">
     <checkpoint-algorithm-ref name="recordbased" />
</job-step>
</job>
```

Implementing a Results algorithm

Results algorithms are an optional feature of the batch programming model.

A results algorithm allows for two types of actions to occur at the end of a batch step:

- To influence the return code of the batch job based on the return code of the batch step that just ended. Note that there are two types of return codes: the return code of an individual batch step and the return code of the batch job to which the step belongs.
- To provide a place holder for triggers or actions to take based on various step return codes.

Results algorithms are applied to a batch job through XML Job Control Language (xJCL). These algorithms are declared in xJCL and then applied to batch steps.

At the end of a batch step, the common batch container checks the xJCL of the batch job to determine which results algorithm to invoke. For each results algorithm specified, the common batch container passes to the algorithm the return code of the batch step, which is the integer returned by the destroyJobStep method of the step, and the current return code of the batch job in the common batch container database. The results algorithm can then take any action based on the return codes passed in. The algorithm then passes a return code for the batch job back to the common batch container which is persisted to the common batch container database as the current return code of the batch job. This return code can be the same as the return code that the common batch container passed to the results algorithm initially, or the return code can be different, depending on logic coded into the results algorithm. If a results algorithm is not specified on a batch step, the job return code is that of the results algorithm from the previous step. If no results algorithms are specified, the job return code is zero (0).

A results algorithm system programming interface (SPI) is also provided, which you can use to write your own algorithms and apply them to batch jobs.

Appendix B. Accessibility

Accessible publications for this product are offered through the z/OS Information Center, which is available at www.ibm.com/systems/z/os/zos/bkserv/.

If you experience difficulty with the accessibility of any z/OS information, please send a detailed message to mhvrcfs@us.ibm.com or to the following mailing address:

IBM Corporation
Attention: MHVRCFS Reader Comments
Department H6MA, Building 707
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Accessibility features

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use software products successfully. The major accessibility features in z/OS enable users to:

- Use assistive technologies such as screen readers and screen magnifier software
- Operate specific or equivalent features using only the keyboard
- Customize display attributes such as color, contrast, and font size.

Using assistive technologies

Assistive technology products, such as screen readers, function with the user interfaces found in z/OS. Consult the assistive technology documentation for specific information when using such products to access z/OS interfaces.

Keyboard navigation of the user interface

Users can access z/OS user interfaces using TSO/E or ISPF. Refer to *z/OS TSO/E Primer*, *z/OS TSO/E User's Guide*, and *z/OS ISPF User's Guide Vol I* for information about accessing TSO/E and ISPF interfaces. These guides describe how to use TSO/E and ISPF, including the use of keyboard shortcuts or function keys (PF keys). Each guide includes the default settings for the PF keys and explains how to modify their functions.

Dotted decimal syntax diagrams

Syntax diagrams are provided in dotted decimal format for users accessing the z/OS Information Center using a screen reader. In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), they can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read out punctuation. All the syntax elements that have the same dotted decimal number (for example, all the syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, you know that your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, it is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is given the format 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol giving information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, this indicates a reference that is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

- ? means an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element, (for example 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.
- ! means a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines 2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP will be applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1!

(KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

- * means a syntax element that can be repeated 0 or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data area, you know that you can include one data area, more than one data area, or no data area. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

  Note:
  1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
  2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
  3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.
- + means a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times; that is, it must be included at least once and can be repeated. For example, if you hear the line 6.1+ data area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. Similar to the * symbol, the + symbol can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

Policy for unsupported hardware

Various z/OS elements, such as DFSMS, HCD, JES2, JES3, and MVS™, contain code that supports specific hardware servers or devices. In some cases, this device-related element support remains in the product even after the hardware devices pass their announced End of Service date. z/OS may continue to service element code; however, it will not provide service related to unsupported hardware devices. Software problems related to these devices will not be accepted for service, and current service activity will cease if a problem is determined to be associated with out-of-support devices. In such cases, fixes will not be issued.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: http://www.ibm.com/software/support/systemsz/lifecycle/
- For information about currently-supported IBM hardware, contact your IBM representative.

Minimum supported hardware

The minimum supported hardware for z/OS releases identified in z/OS announcements can subsequently change when service for particular servers or devices is withdrawn. Likewise, the levels of other software products supported on a particular release of z/OS are subject to the service support lifecycle of those products. Therefore, z/OS and its product publications (for example, panels, samples, messages, and product documentation) can include references to hardware and software that is no longer supported.

- For information about software support lifecycle, see: http://www.ibm.com/software/support/systemsz/lifecycle/
- For information about currently-supported IBM hardware, contact your IBM representative.

Index

Product Number: 5650-ZOS

Printed in USA

SA23-1376-00