Create an application to process variable length and multiple segment output messages

Contents

Create a J2C application for an IMS transaction containing variable length and multiple segments

This tutorial describes how to use the J2C Java™ Bean wizard to build a simple Web application that processes an IMS™ transaction with input and output data containing variable length and multiple segments.

Learning objectives

This tutorial enables you to:

- Use the J2C Java Bean wizard to create a J2C Java bean that runs an IMS transaction.
- Create a message buffer class and edit this class using doclet annotations.
- Create a method for the J2C Java bean to run the IMS transaction and provide input and output data types for the method.
- Create Java data bindings for the segments of the output message.
- Create a test Java class to test the application

Time required

30 minutes

Related information

View the PDF version

Multisegment Output sample

Create an application to process variable length and multiple segment output messages

This tutorial leads you through the detailed steps to generate a J2C application that processes variable length and multiple segment IMS transaction output messages.

This tutorial might require some optionally installable components. To ensure that you have installed the appropriate optional components, see the System requirements list.

This tutorial is divided into several exercises that must be completed in sequence for the tutorial to work properly. This tutorial teaches you how to use the J2C Java bean wizard to create a Java bean that runs a transaction in IMS. While completing the exercises, you will:

- Use the J2C Java Bean wizard to create a J2C Java bean that runs an IMS transaction.
- Create a message buffer class, CCIBuffer.java, and edit this class using doclet annotations.
- Create a method for the J2C Java bean to run the IMS transaction and provide input and output data types for the method.
- Create Java data bindings for the segments of the output message.
- Create a test Java class, TestMultiSeg.java, to invoke the J2C Java bean method that runs the IMS transaction, then populate the output segments from the buffer of data returned by the IMS transaction.

  Note: The test Java class, *TestMultiSeg.java*, was created for an English locale; you may have to make modifications in the code for other locales.

Time required

This tutorial should take approximately 30 minutes to finish. If you explore other concepts related to this tutorial, it could take longer to complete.

Skill level

Experienced

Audience

This tutorial is intended for users who are familiar with Enterprise Information systems (EIS) and IMS in particular.

System requirements

To complete this tutorial, you need to have the following tools and components installed:

- IBM® WebSphere® Application Server, version 6.1
- J2EE Connector (J2C) tools
- **Information about your IMS environment**: In this tutorial, your application interacts with an IMS application program in IMS. You need to obtain information such as the host name and port number of IMS Connect and the name of the IMS datastore where the transaction will run. Contact your IMS systems administrator for this information. Specifically, you need to perform some setup work in IMS if you want to run the IMS\MultiSegmentOutput IMS program.
- **A copy of the COBOL file `MSOut.cbl`**: You may locate this file in your product installation directory: \rad\eclipse\plugins\com.ibm.j2c.cheatsheet.content_7.0.0\samples\IMS\MultiSegmentOutput. If you wish to store it locally, you can copy the code from here: ../resources/msoutcbl.pdf
- **A clean workspace**.

To use this tutorial, you must have an application server installed and configured. To verify that a server runtime environment is available, click **Window → Preferences**, expand **Server**, and then click **Installed Runtimes**. You can use this pane to add, remove, or edit installed server runtime definitions. You can also download and install support for a new server.

Prerequisites

In order to complete this tutorial end to end, you should be familiar with:

- J2EE and Java programming
- Basic IMS Transaction Manager (IMS TM) concepts

Lesson 1.1: Select a resource adapter

This lesson leads you through the detailed steps to select and configure the resource adapter to connect to the IMS server.

The IMS transaction that is used in this tutorial is not one of the IMS Installation Verification Programs. This tutorial uses DFSDDLT0, an IMS application program that issues calls to IMS based on control statement information. The DFSDDLT0 control statements for this tutorial are provided below. However, to run this tutorial you must configure your environment for DFSDDLT0 and provide the necessary JCL. This tutorial uses SKS2 as the transaction code for the DFSDDLT0 application.

DFSDDLT0 control statements

```
S11 1 1 1 1    TP     1
L        GU
E      OK
E  Z0017 DATA  SKS2 M2 SI1M3 SI1
WTO SEGMENT SI1 RECEIVED
L        GN
E      QD
WTO END OF INPUT SEGMENTS
L        ISRT  IW06OUT
L  Z0012 DATA  *******M1SO1
E      OK
WTO SEGMENT SO1 INSERTTED
L        ISRT
L  Z0027 DATA  ********M1SO2*********M2SO2
E      OK
WTO SEGMENT SO2 INSERTTED
L        ISRT
L  Z0048 DATA  **********M1SO3***********M2SO3************M3SO3
E      OK
WTO SEGMENT SO3 INSERTTED
WTO CURRENT PROGRAM STLDDLT2 TERMINATED
L        GU
```

This tutorial uses COBOL data structures to describe the IMS transaction input and output messages. Note that the output message returned by IMS consists of three fixed length segments:

- OUTPUT-SEG1 (16 bytes)
- OUTPUT-SEG2 (31 bytes)
- OUTPUT-SEG3 (52 bytes)

The output message returned by this particular IMS application is a fixed size of 99 bytes and is represented by the COBOL 01 structure OUTPUT-MSG.

One way of developing this multi-segment application is to use the COBOL definition OUTPUT-MSG to define the output of the transaction. A second way is to create an output message for the output of the transaction. The code provided with this tutorial uses the second method, since it can also be used to build an application that processes a variable length output message. The COBOL definitions for the individual message segments will continue to be used to simplify access to the data of the individual segments

Connecting to the IMS server

1. If the J2EE icon,  , does not appear in the top right tab of the workspace, you need to switch to the J2EE perspective. From the menu bar, select **Window > Open Perspective > Other**. The Select Perspective window opens.
2. Select **J2EE**.
3. Click **OK**. The J2EE perspective opens.
4. In the J2EE perspective, select **File > New > Other**.
5. In the New page, select **J2C > J2C Java Bean**. Click **Next**
6. In the Resource Adapters Selection page, select the J2C 1.5 IMS resource adapter. For this tutorial select **IMS Connector for Java (IBM: 9.1.0.2.3)**. Click **Next**.
7. In the Connection Properties page, clear **Managed Connection** and select **Nonmanaged connection**. (For this tutorial, you will use a non-managed connection to directly access IMS.) Accept the default Connection class name of `com.ibm.connector2.ims.ico.IMSManagedConnectionFactory`. In the blank fields, enter all the required connection information. Required fields, indicated by an asterisk (*), include the following:
   a. **For TCP/IP connection**:

1) **Host name**: (Required) The IP address or host name of IMS Connect.
2) **Port Number**: (Required) The number of the port used by the target IMS connect.

b. **For local option connection**:
1) **IMS Connect name**: (Required) The name of the target IMS connect.
2)

c. **For both**:
1) **Data Store Name**: (Required) The name of the target IMS datastore.

8. You may obtain the connection information from your IMS system administrator. When you have provided the required connection information, click **Next**.

MSout.cbl

Here is the code from MSout.cbl:

MSout.cbl

```
      IDENTIFICATION DIVISION.
      program-id. pgm1.
      ENVIRONMENT DIVISION.
      CONFIGURATION SECTION.
      DATA DIVISION.
      *
      *    IMS TOC Connector for Java, Multi-segment Output Example
      *
      ************************************************************************/
      *                                                                      */
      * (c) Copyright IBM Corp. 1998                                         */
      * All Rights Reserved                                                  */
      * Licensed Materials - Property of IBM                                 */
      *                                                                      */
      * DISCLAIMER OF WARRANTIES.                                            */
      *                                                                      */
      * The following (enclosed) code is provided to you solely for the      */
      * purpose of assisting you in the development of your applications.    */
      * The code is provided "AS IS." IBM MAKES NO WARRANTIES, EXPRESS OR    */
      * IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF      */
      * MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, REGARDING      */
      * THE FUNCTION OR PERFORMANCE OF THIS CODE.                            */
      * IBM shall not be liable for any damages arising out of your use      */
      * of the generated code, even if they have been advised of the         */
      * possibility of such damages.                                         */
      *                                                                      */
      * DISTRIBUTION.                                                        */
      *                                                                      */
      * This generated code can be freely distributed, copied, altered,      */
      * and incorporated into other software, provided that:                 */
      *   - It bears the above Copyright notice and DISCLAIMER intact        */
      *   - The software is not for resale                                   */
      *                                                                      */
      ************************************************************************/
      *
        LINKAGE SECTION.

        01  INPUT-MSG.
            02  IN-LL          PICTURE S9(3) COMP.
            02  IN-ZZ          PICTURE S9(3) COMP.
            02  IN-TRCD        PICTURE X(5).
            02  IN-DATA1       PICTURE X(6).
            02  IN-DATA2       PICTURE X(6).

        01  OUTPUT-MSG.
            02  OUT-ALLSEGS  PICTURE X(99) VALUE SPACES.
```

```
01  OUTPUT-SEG1.
    02  OUT-LL       PICTURE S9(3) COMP VALUE +0.
    02  OUT-ZZ       PICTURE S9(3) COMP VALUE +0.
    02  OUT-DATA1    PICTURE X(12) VALUE SPACES.

01  OUTPUT-SEG2.
    02  OUT-LL       PICTURE S9(3) COMP VALUE +0.
    02  OUT-ZZ       PICTURE S9(3) COMP VALUE +0.
    02  OUT-DATA1    PICTURE X(13) VALUE SPACES.
    02  OUT-DATA2    PICTURE X(14) VALUE SPACES.

01  OUTPUT-SEG3.
    02  OUT-LL       PICTURE S9(3) COMP VALUE +0.
    02  OUT-ZZ       PICTURE S9(3) COMP VALUE +0.
    02  OUT-DATA1    PICTURE X(15) VALUE SPACES.
    02  OUT-DATA2    PICTURE X(16) VALUE SPACES.
    02  OUT-DATA3    PICTURE X(17) VALUE SPACES.

PROCEDURE DIVISION.
```

Lesson 1.2: Set up a Web project and Java Interface and Implementations

Lesson 1.2 steps you through the creation of a the Web project and Java Interface and Implementations

Before you begin, you must complete Lesson 1.1. In this lesson you will:

- Create a J2C Java bean
- Create a dynamic Web project

1. All work done in the workbench must be associated with a project. Projects provide an organized view of the work files and directories, optimized with functions based on the type of project. In the workbench, all files must reside in a project, so before you create the J2C Java bean, you need to create a project to put it in. In the New J2C Java Bean page, type the value `MultiSegOutput` in the **Project Name** field.
2. Click the **New** button beside the **Project Name** field to create the new project
3. Now you will create a dynamic Web project. In the New Source Project Creation page, select **Web project**, and click **Next**.
4. In the New Dynamic Web Project page, click **Show Advanced**.
5. Ensure that the following values are selected:
   a. **Project name**: MultiSegOutput
   b. **Project contents**: accept default
   c. **Target runtime** : WebSphere Application Server v6.1
   d. **Configurations**: accept default
   e. **Add project to an EAR**: checked
   f. **EAR Project name**: MultiSegOutputEAR
6. Click **Next**.
7. On the Project Facets page, accept the defaults.
8. Click **Next**.
9. On the Web Modules page, accept the defaults.
10. Click **Finish**.
11. A dialog box may appear asking if you would like to switch to the Dynamic Web perspective. Click **Yes**.
12. On the J2C Java Bean Output Properties page:
    a. In the **Package Name** field, click **Browse** and select the `MultiSegOutput` project. Click **OK**.

b. Type `sample.ims` in the **Package Name** field.
c. Type `MSO` in the **Interface Name** field.
d. Type `MSOImpl` in the **Implementation Name** field.
e. Leave the **Save session as ant script** unchecked.

13. Click **Finish**.

Lesson 1.3: Create a message buffer class

Lesson 1.3 leads you through the creation of a message buffer class.

Before you begin, you must complete Lesson 1.2: Setting up the Web project and Java Interface and Implementations. In this lesson you will:

- Create a message buffer class
- Edit the message buffer class using doclet annotations
- Create the input and output binding operations
- Create the output segment data mappings

1. **First you will create a message buffer class:** Expand the **MultiSegOutput** project, expand **Java Resources**, and expand **JavaSource**.
2. Right click on the **sample.ims** package, and select **New > Class** to launch the New Class wizard.
3. Type `CCIBuffer` as the name of the class. Accept all default settings.
4. Click **Finish**. The **CCIBuffer** class opens in the Java editor.
5. In the comment section of the **CCIBuffer** class, type in the tag `@type-descriptor.message-buffer`.
6. Press **CTRL-S** to save the changes. Note that new code is automatically generated in the ../resources/ccibuffer.pdf.
7. **Next you will create a method to run the IMS transaction and the input message data type:** In the Project Explorer view, right click on **MSOImpl.java**, and select **Source > Add method to J2C Java bean**.
8. In the New Java Method page, click **Add**.
9. Type `runMultiSegOutput` as the Java method name. Click **Next**.
10. Click **New** to define the input type.
11. Select **COBOL_to_Java** mapping. Click **Browse**.
12. Locate the **MSOut.cbl** COBOL file. Click **Open**.
13. Click **Next**.
14. In the COBOL Importer page, click **Show Advanced**.
    a. Select the following options:

Table 1. COBOL Importer Parameter Settings

Parameter	Value
Platform Name	Z/OS
Codepage	IBM-037
Floating point format name	IBM Hexadecimal
External decimal sign	EBCDIC
Endian name	Big
Remote integer endian name	Big
Quote name	DOUBLE
Trunc name	STD
Nsymbol name	DBCS

b. Click **Query** to load the data.
c. A list of data structures is shown. Select **INPUT-MSG** in the **Data structures** field.
d. Click **Next**.
e. Leave the **Save session as Ant script** unchecked.

15. In the Saving Properties page, the default Class Name is **INPUTMSG**. Overwrite the Class Name with **InputMsg**. Click **Finish**.
16. **Next you will create the output message data type:** Click **Browse** to define the output type.
17. Type `CC` in the Select a data type field, and **CCIBuffer** will appear in the Matching types field. Select **CCIBuffer** as the output type. Click **Finish**.
18. In the Java Method page, click **Finish**.
19. In the Java Methods page, ensure that the **interactionVerb** is set to **SYNC_SEND_RECEIVE (1)** to indicate that the interaction with IMS involves a send followed by a receive interaction.
20. Click **Finish**.
21. **Next you will create the output segment data mappings. First you will create the OutputSeg1.java class:** To accomplish this step, you need to use a standalone data mapping wizard so that you create only the data mapping files.
22. Select **File > New > Other > CICS/IMS Java Data Binding** to invoke the Data Binding wizard.
23. Click **Next**.
24. Select **COBOL_To_Java** in the **Choose mapping** list. Click **Browse** to find the **MSOut.cbl** COBOL copy book.
25. Click **Next**.
26. In the COBOL Importer page, click **Show Advanced**.
    a. Select the following options:

Table 2. COBOL Importer Parameter Settings

Parameter	Value
Platform Name	Z/OS
Codepage	IBM-037
Floating point format name	IBM Hexadecimal
External decimal sign	EBCDIC
Endian name	Big
Remote integer endian name	Big
Quote name	DOUBLE
Trunc name	STD
Nsymbol name	DBCS

    b. Click **Query** to load the data.
    c. A list of data structures is shown. Select **OUTPUT-SEG1** in the **Data structures** field.
    d. Click **Next**.
27. In the Saving properties wizard, click **Browse**, and select the **MultiSegOutput** project you created before.
28. Click **Browse** to select the package name: **sample.ims.data**.
29. Change the Java Class Name from **OUTPUTSEG1** to **OutputSeg1**.
30. Click **Finish**.
31. **Now you will create the OutputSeg2.java class:** Select **File > New > Other > CICS/IMS Java Data Binding** to invoke the Data Binding wizard.

32. Click **Next**.
33. Select **COBOL_To_Java** in the **Choose mapping** list. Click **Browse** to find the **MSOut.cbl** COBOL copy book.
34. In the COBOL Importer page, click **Show Advanced**.
    a. Select the following options:

Table 3. COBOL Importer Parameter Settings

Parameter	Value
Platform Name	Z/OS
Codepage	IBM-037
Floating point format name	IBM Hexadecimal
External decimal sign	EBCDIC
Endian name	Big
Remote integer endian name	Big
Quote name	DOUBLE
Trunc name	STD
Nsymbol name	DBCS

    b. Click **Query** to load the data.
    c. A list of data structures is shown. Select **OUTPUT-SEG2** in the **Data structures** field.
    d. Click **Next**.
35. In the Saving properties wizard, click **Browse** to select the **MultiSegOutput** project you created before.
36. Click **Browse** to select the package name: **sample.ims.data**.
37. Change the Java Class Name from **OUTPUTSEG2** to **OutputSeg2**.
38. Click **Finish**.
39. **Now you will create the OutputSeg3.java class:** Select **File > New > Other > CICS/IMS Java Data Binding** to invoke the Data Binding wizard.
40. Click **Next**.
41. Select **COBOL_To_Java** in the **Choose mapping** list. Click **Browse** to find the **MSOut.cbl** COBOL copy book.
42. In the COBOL Importer page, click **Show Advanced**.
    a. Select the following options:

Table 4. COBOL Importer Parameter Settings

Parameter	Value
Platform Name	Z/OS
Codepage	037
Floating point format name	IBM 390 Hexadecimal
External decimal sign	EBCDIC
Endian name	Big
Remote integer endian name	Big
Quote name	DOUBLE
Trunc name	STD
Nsymbol name	DBCS

b. Click **Query** to load the data.
c. A list of data structures is shown. Select **OUTPUT-SEG3 in** in the **Data structures** field.
d. Click **Next**.

43. In the Saving properties wizard, click **Browse** to select the **MultiSegOutput** project you created before.
44. Click **Browse** to select the package name: **sample.ims.data**.
45. Change the Java Class Name from **OUTPUTSEG3** to **OutputSeg3**.
46. Click **Finish**.

CCIBuffer.java

Here is the generated code in the CCIBuffer.java class:

CCIBuffer.java

```
/*
 * Created on Oct 13, 2004
 *
 * TODO To change the template for this generated file go to
 * Window - Preferences - Java - Code Style - Code Templates
 */
package sample.ims;

/**
 * @author ivyho
 *
 * TODO To change the template for this generated type comment go to
 * Window - Preferences - Java - Code Style - Code Templates
 * @type-descriptor.message-buffer
 */
public class CCIBuffer implements javax.resource.cci.Record,
  javax.resource.cci.Streamable, com.ibm.etools.marshall.RecordBytes {

 private byte[] buffer_ = null;

 /**
  * @generated
  */
 public CCIBuffer() {
  return;
 }

 /**
  * @generated
  * @see javax.resource.cci.Record#getRecordShortDescription()
  */
 public String getRecordShortDescription() {
  return (this.getClass().getName());
 }

 /**
  * @generated
  * @see javax.resource.cci.Record#hashCode()
  */
 public int hashCode() {
  return (super.hashCode());
 }

 /**
  * @generated
  * @see javax.resource.cci.Streamable#write(OutputStream)
  */
 public void write(java.io.OutputStream outputStream)
   throws java.io.IOException {
```

```
 outputStream.write(buffer_);
}

/**
 * @generated
 * @see javax.resource.cci.Record#setRecordShortDescription(String)
 */
public void setRecordShortDescription(String shortDescription) {
 return;
}

/**
 * @generated
 */
public int getSize() {
 if (buffer_ != null)
  return (buffer_.length);
 else
  return (0);
}

/**
 * @generated
 * @see java.lang.Object#toString
 */
public String toString() {
 StringBuffer sb = new StringBuffer(super.toString());
 sb.append("\n");
 com.ibm.etools.marshall.util.ConversionUtils.dumpBytes(sb, buffer_);
 return (sb.toString());
}

/**
 * @generated
 * @see javax.resource.cci.Record#getRecordName()
 */
public String getRecordName() {
 return (this.getClass().getName());
}

/**
 * @generated
 */
public byte[] getBytes() {
 return (buffer_);
}

/**
 * @generated
 * @see javax.resource.cci.Record#clone()
 */
public Object clone() throws CloneNotSupportedException {
 return (super.clone());
}

/**
 * @generated
 * @see javax.resource.cci.Record#setRecordName(String)
 */
public void setRecordName(String recordName) {
 return;
}

/**
 * @generated
 * @see javax.resource.cci.Record#equals()
 */
```

```
public boolean equals(Object object) {
 return (super.equals(object));
}

/**
 * @generated
 * @see javax.resource.cci.Streamable#read(InputStream)
 */
public void read(java.io.InputStream inputStream)
  throws java.io.IOException {
 byte[] input = new byte[inputStream.available()];
 inputStream.read(input);
 buffer_ = input;
}

/**
 * @generated
 */
public void setBytes(byte[] bytes) {
 buffer_ = bytes;
}

}
```

Lesson 1.4: Creating a Java test class to test your application

Lesson 1.4 leads you through the creation of a Java test class to test your application.

Before you begin, you must complete Lesson 1.3: Creating a message buffer class. In this lesson you will

- Create a Java test class.
- Edit the class using the code supplied below.
- Run the test class to test your application.

1. **First you will create a Java test class:** Expand the MultiSegOutput project, expand **Java Resources**, and select the **sample.ims** package.
2. Right click and select **New**. Select the  class option to create a new Java class.
3. In the Java class name field, type `TestMultiSeg`. Note that the TestMultiSeg.java class is provided as an example only; you may need to change the transaction code to your IMS machine specifications. Consult your IMS administrator for the transaction code. You can locate the statement `input.setIn__trcd("SKS6 ")` in the TestMultiSeg.java class and make the modifications.
4. Ensure that **Source Folder** contains **MultiSegOutput/JavaSource** and that the **Package Name** contains **sample.ims**.
5. Click **Finish**.
6. Open **TestMultiSeg.java** in the Java class editor.
7. Replace all the code in the editor with the following:

```
/************************************************************
 * Licensed Materials - Property of IBM
 *
 * com.ibm.j2c.cheatsheet.content
 *
 *Copyright IBM Corporation 2004. All Rights Reserved.
 *
 * Note to U.S. Government Users Restricted Rights:  Use,
 * duplication or disclosure restricted by GSA ADP  Schedule
 * Contract with IBM Corp.
 ************************************************************/
package sample.ims;

import com.ibm.etools.marshall.util.MarshallIntegerUtils;
```

```
import sample.ims.data.*;

public class TestMultiSeg
{
 public static void main (String[] args)
 {
  byte[] segBytes = null;
  int srcPos = 0;
  int dstPos = 0;
  int totalLen = 0;
  int remainLen = 0;
  byte[] buff;
  short LL = 0;
  short ZZ = 0;

  try
  {
   // ------------------------------------------------
   // Populate the IMS transaction input message with
   // data.  Use the input message format handler method
   // getSize() to set the LL field of the input message.
   // ------------------------------------------------
   InputMsg input = new InputMsg();
   input.setIn__ll((short) input.getSize());
   input.setIn__zz((short) 0);
   //-----------------------------------------
   // find out the transaction code from your IMS
   // administrator
   //-----------------------------------------
   input.setIn__trcd("SKS6 ");
   input.setIn__data1("M2 SI1");
   input.setIn__data2("M3 SI1");

   // ------------------------------------------------
   // Run the IMS transaction.  The multi-segment output
   // message is returned.
   // ------------------------------------------------
   MSOImpl proxy = new MSOImpl();

   sample.ims.CCIBuffer output = proxy.runMultiSegOutput(input);

   // ------------------------------------------------
   // Retrieve the multi-segment output message as a
   // byte array using the output message format
   // handler method getBytes().
   // ------------------------------------------------
   System.out.println(
    "\nSize of output message is: " + output.getSize());
   segBytes = output.getBytes();

   srcPos = 0;
   dstPos = 0;
   totalLen = segBytes.length;
   remainLen = totalLen;

   // ------------------------------------------------
   // Populate first segment object from buffer.
   // ---------------------------------------
   buff = null;
   // Get length of segment.
   LL =
    MarshallIntegerUtils.unmarshallTwoByteIntegerFromBuffer(
     segBytes,
     srcPos,
     true,
     MarshallIntegerUtils.SIGN_CODING_TWOS_COMPLEMENT);
```

```
// Put segment in byte array.
buff = new byte[LL];
System.arraycopy(segBytes, srcPos, buff, dstPos, LL);
remainLen -= LL;

// Create and populate segment object from byte array.
OutputSeg1 S1 = new OutputSeg1();
S1.setBytes(buff);
System.out.println(
 "\nOutSeg1 LL is:    "
  + S1.getOut__ll()
  + "\nOutSeg1 ZZ is:    "
  + S1.getOut__zz()
  + "\nOutSeg1_DATA1 is: "
  + S1.getOut__data1());

// ------------------------------------------------
// Populate second segment object from buffer.
// ------------------------------------------------
srcPos += LL;
buff = null;
// Get length of segment.
LL =
 MarshallIntegerUtils.unmarshallTwoByteIntegerFromBuffer(
  segBytes,
  srcPos,
  true,
  MarshallIntegerUtils.SIGN_CODING_TWOS_COMPLEMENT);

// Put segment in byte array.
buff = new byte[LL];
System.arraycopy(segBytes, srcPos, buff, dstPos, LL);
remainLen -= LL;

// Create and populate segment object from byte array.

OutputSeg2 S2 = new OutputSeg2();
S2.setBytes(buff);
System.out.println(
 "\nOutSeg2 LL is:    "
  + S2.getOut__ll()
  + "\nOutSeg2 ZZ is:    "
  + S2.getOut__zz()
  + "\nOutSeg2_DATA1 is: "
  + S2.getOut__data1()
  + "\nOutSeg2_DATA2 is: "
  + S2.getOut__data2());
// ------------------------------------------------
// Populate third segment object from buffer.
// ------------------------------------------------
srcPos += LL;
buff = null;
// Get length of segment.
LL =
 MarshallIntegerUtils.unmarshallTwoByteIntegerFromBuffer(
  segBytes,
  srcPos,
  true,
  MarshallIntegerUtils.SIGN_CODING_TWOS_COMPLEMENT);

// Put segment in byte array.
buff = new byte[LL];
System.arraycopy(segBytes, srcPos, buff, dstPos, LL);
remainLen -= LL;

// Create and populate segment object from byte array.
OutputSeg3 S3 = new OutputSeg3();
```

```
    S3.setBytes(buff);
    System.out.println(
     "\nOutSeg3 LL is:    "
      + S3.getOut__ll()
      + "\nOutSeg3 ZZ is:    "
      + S3.getOut__zz()
      + "\nOutSeg3_DATA1 is: "
      + S3.getOut__data1()
      + "\nOutSeg3_DATA2 is: "
      + S3.getOut__data2()
      + "\nOutSeg3_DATA3 is: "
      + S3.getOut__data3());
   }
   catch (Exception e)
   {
    System.out.println("\nCaught exception is: " + e);
   }
  }
}
```

8. **Now you will test your application:** Expand the **MultiSegOutput** project and the **sample.ims** package.
9. Right-click the TestMultiSeg.java class and select Run. Select Run As > Java Application
10. You should see the following output on the console:

```
Size of output message is:      99
OutSeg1 LL is:                  16
OutSeg1 ZZ is:                  768
OutSeg1_DATA1 is:               *******M1S01

OutSeg2 LL is:                  31
OutSeg2 ZZ is:                  768
OutSeg2_DATA1 is:               *******M1S02
OutSeg2_DATA2 is:               *******M2S02

OutSeg3 LL is:                  52
OutSeg3 ZZ is:                  768
OutSeg3_DATA1 is:               *******M1S03
OutSeg3_DATA2 is:               *******M2S03
OutSeg3_DATA3 is:               *******M3S03
```

11. Congratulations! You have completed the MultiSegment Output tutorial.

Create a J2C application for an IMS transaction containing variable length and multiple segments: summary

This tutorial guided you through the detailed steps to generate a J2C application that processes variable length and multiple segment IMS transaction output messages.

Lessons learned

In this tutorial, you learned how to

- Use the J2C Java Bean wizard to create a J2C Java bean that runs an IMS transaction.
- Create a message buffer class, CCIBuffer.java, and edit this class using doclet annotations.
- Create a method for the J2C Java bean to run the IMS transaction and provide input and output data types for the method.
- Create Java data bindings for the segments of the output message.
- Create a test Java class, TestMultiSeg.java, to invoke the J2C Java bean method that runs the IMS transaction, then populate the output segments from the buffer of data returned by the IMS transaction.