Build a J2C application for a CICS COBOL copybook: Multiple possible output

Contents

Create a J2C application for a CICS transaction containing multiple possible outputs

This tutorial describes how to use the J2C Java™ Bean wizard to build a simple Web application that processes a transaction, which can process multiple possible outputs.

Learning objectives

This tutorial enables you to:

- Use the J2C Java bean wizard to create a J2C application that interfaces with a CICS transaction using an External Call Interface (ECI). Depending on the customer's classification, preferred customer, regular customer or bad customer, the program returns different output information about the customer. Create a JSP to deploy the application on a WebSphere application server.
- Create a Java method that accepts a customer number.
- Create a JSP to deploy the application on a WebSphere application server.

Time required

30 minutes

Related information

View the PDF version

Multiple Possible Outputs sample

Create a J2C application for a CICS transaction containing multiple possible outputs introduction

This tutorial describes how to use the J2C Java Bean wizard to build a simple Web application that processes a CICS® transaction with multiple possible outputs.

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 will lead you through the detailed steps to generate a J2C application that interfaces with a CICS transaction using an External Call Interface (ECI). The service is built from a CICS COBOL function, getCustomerInfo, which accepts a customer number. Depending on the customer's classification, preferred customer, regular customer or bad customer, the program returns different output information about the customer.

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 connect to a CICS ECI server. While completing the exercises, you will:

- Use the J2C Java bean wizard to create a J2C application that interfaces with a CICS transaction using an External Call Interface (ECI).
- Create a Java method, *getCustomerInfo*, which accepts a customer number. Depending on the customer's classification, preferred customer, regular customer or bad customer, the program returns different output information about the customer.
- Create a Java class, TestECIMPO.java to test your application.

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 CICS ECI 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
- **Connection to a CICS ECI server**: In this tutorial, your application interacts with a CICS program on a server. Specifically, you need to set up a CICS transaction gateway on a machine to access the server. You also need to perform some setup work on the CICS server machine, where you want the CICS to run. These steps are not covered.
- **A copy of the COBOL file taderc25.cbl**. You may locate this file in your product installation directory: <installdir>\IBM\SDP70Shared\plugins\com.ibm.j2c.cheatsheet.content_7.0.0\Samples\CICS\taderc25. If you wish to store it locally, you can copy the code from here: ../resources/taderc25.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
- COBOL programming language
- CICS ECI server technology

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 CICS ECI server.

Connecting to the CICS ECI 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 page, under **View by**, select **JCA version**. Expand **1.5**, and select **ECIResourceAdapter (IBM:6.0.2)**. Click **Next**.
7. In the Connection Properties page, select **Nonmanaged connection**. (For this tutorial, you will use the non-managed connection to directly access the CICS server, so you do not need to provide the JNDI name.) Accept the default **Connection class name** of `com.ibm.connector2.cics.ECIManagedConnectionFactory`. In the blank fields, provide connection information. Required fields, indicated by an asterisk (*), include the following:
   - **Server name**: (Not required) The name of the CICS Transaction Gateway server.
   - **Connection URL***: (Required) The server address of the CICS ECI server
   - **Port number**: (Not required) The number of the port that is used to communicate with the CICS Transaction Gateway. The default port is 2006.
   - **User name**: (Required) The user name for the connection.
   - **Password**: (Required) The password for the connection.
   - :

   You may obtain the connection information from your CICS Server system administrator.
8. Click **Next**.

taderc25.cbl

Here is the code from taderc25.cbl:

taderc25.cbl

```
      identification division.
       program-id. TADERC25.
       environment division.
       data division.
       working-storage section.
       01 tmp pic a(40).
       01  ICOMMAREA.
           02  ICustNo    PIC X(5).
           02  Ifiller    PIC X(11).
       01  GENCUST.
           02  GCUSTCODE PIC X(4).
           02  GFILLER PIC X(40).
       01  PREFCUST.
           02  PCUSTCODE PIC X(4).
           02  PCUSTNO    PIC X(5).
           02  ASSETS  PIC S9(6)V99.
       01  REGCUST.
           02  RCUSTCODE PIC X(4).
           02  RCUSTNO    PIC X(5).
           02  ACCOUNTNAME PIC A(10).
           02  BALANCE PIC S9(6)V99.
       01  BADCUST.
           02  BCUSTCODE PIC X(4).
           02  BCUSTNO    PIC X(5).
           02  DAYSOVERDUE PIC X(4).
           02  AMOUNT PIC S9(6)V99.
       LINKAGE SECTION.
       01 DFHCOMMAREA.
          02 inputfield pic x(50).
       procedure division.
       start-para.
           move DFHCOMMAREA to ICOMMAREA.
           IF ICustNo EQUAL '12345'
              move 'PREC' to PCUSTCODE
              move ICustNo to PCUSTNO
```

```
           move 43456.33 to ASSETS
           move PREFCUST TO DFHCOMMAREA
        ELSE IF ICustNo EQUAL '34567'
                move 'REGC' to RCUSTCODE
                move ICustNo to RCUSTNO
                move 'SAVINGS' TO ACCOUNTNAME
                move 11456.33 to BALANCE
                move REGCUST TO DFHCOMMAREA
             ELSE
                move 'BADC' to BCUSTCODE
                move ICustNo to BCUSTNO
                move '132' to DAYSOVERDUE
                move -8965.33 to AMOUNT
                move BADCUST TO DFHCOMMAREA
      *         END-IF.
           END-IF.
           EXEC CICS RETURN
           END-EXEC.
```

Lesson 1.2: Set up a Web project and Java interface and implementations

This lesson leads you through setting up a 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 store it in.
2. In the New J2C Java Bean page, type the value `Taderc25Sample` in the **Project Name** field.
3. Click **New** beside the **Project Name** field to create the new project.
4. In the New Source Project Creation page, select **Web project**, and click **Next**.
5. In the New Dynamic Web Project page, click **Show Advanced**.
6. Ensure that the following values are selected:
   a. **Project name**: Taderc25Sample
   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**: Taderc25SampleEAR
7. Click **Finish**.
8. A dialog box may appear asking if you would like to switch to the Dynamic Web perspective. Click **Yes**.
9. On the J2C Java Bean Output Properties page:
   a. In the **Package Name** field, click **Browse** and select the `Taderc25Sample` project. Click **OK**.
   b. Type `sample.ims` in the **Package Name** field.
   c. Type `CustomerInfoMO` in the **Interface Name** field.
   d. Type `CustomerInfoMOImpl` in the **Implementation Name** field.
   e. Leave the **Save session as Ant script** unchecked.
10. Click **Finish**.

Lesson 1.3: Create a Java method

Lesson 1.3 leads you through the creation of a Java method.

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 Java method
- Create the input and output data mapping between COBOL and Java

1. **First you will create a Java method:** You will now create a Java method that will use the COBOL importer to map the data types between the COBOL source and the data in your Java method.
2. Open the Snippets view by clicking on **Window > Show View > Snippets**. In the Snippets view, click on **J2C**.
3. Right click **Add Java method to J2C Java bean** and select **Insert**.
4. In the New Java Method page, click **Add**.
5. In the **Java method** name field, type `getCustomerInfo` for the name of the operation. Click **Next**.
6. **Next you will create the input parameter data mapping**: In this step, you will import the ../resources/taderc25.pdf (COBOL) file that is needed to create your application. The taderc25.cbl file is located in <installdir>\IBM\SDP70Shared\plugins\com.ibm.j2c.cheatsheet.content_7.0.0\Samples\CICS\taderc25, where <installdir> is the directory where this product is installed. The COBOL file contains the program that runs on the CICS server. It has the definition of the structure to be passed to the CICS server via the communications area (COMMAREA). This structure represents the customer records being returned from the CICS application. Before you can work with a file, you must import it from the file system into the workbench. In the **Specify the input/output type** field of the Java Method page, click **New**.
7. In the Data Import page, ensure that the **Choose mapping** field is **COBOL_TO_JAVA**. Click **Browse** beside the COBOL file
8. Locate the taderc25.cbl file in the file system, and click **Open**.
9. Click **Next**.
10. In the COBOL Importer page, select a **communication data structure**:
    a. Select **Win32** for **Platform Name**.
    b. Select **ISO-8859-1** for **Code page**.
    c. Click **Query**.
    d. Select **ICOMMAREA** for **Data structures**.
11. Click **Next**.
12. In the Saving properties page:
    a. Select **Default** for **Generation Style**.
    b. Click **Browse** to choose the Web project **Taderc25Sample**.
    c. In the **Package Name** field, type `sample.cics.data`
    d. In the **Class Name** field, the default value is **ICOMMAREA**; replace it with `InputComm`.
13. Click **Finish**.
14. **Next you will create the multiple possible outputs for the output parameter:** In the **Specify the input/output type** field in the Java Method page, click **New** beside the Output type area.
15. In the Data Import page, ensure that the Choose mapping field is **COBOL_MPO_TO_JAVA**.
16. Click **New** beside the multiple possible output area.
17. Click **Browse** beside the Cobol file name field, and locate the location of the **taderc25.cbl** file. Click **Open**.
18. Click **Next**.
19. In the COBOL Importer page, select a **communication data structure**:

a. Select **Win32** for **Platform Name**.
b. Select **ISO-8859-1** for **Code page**.
c. Click **Query**.
d. Select **PREFCUST**, **REGCUST**, and **BADCUST** for **Data structures**.

20. Click **Finish**. In the Specify data import configuration properties page, you will see the three data types listed.
21. Click **Next**.
22. **Next you will specify the saving properties:** In the Saving Properties page, you will see default values set for each of the customer type record. Ensure that **Taderc25Sample** appears in the **Project Name** field. Click **Browse** and choose the Web project **Taderc25Sample**.
    a. In the Specify the Saving properties page, highlight **COBOL MPO to Java Save Properties**.
       - Type `sample.cics.data` in the **Package Name** field
       - Type`OutputComm` in the **Class Name** field.
       - You can select **Overwrite existing class** .
    b. Expand **COBOL MPO to Java Save Properties**. The three data binding elements should appear.
    c. Highlight **COBOL To Java Save Properties For ″PREFCUST″ in File taderc25.cbl**
       - For **Generation Style**, select **Default**.
       - Type `sample.cics.data` in the **Package Name** field
       - Type`PrefCust` in the **Class Name** field.
       - You can select **Overwrite existing class** .
    d. Highlight **COBOL To Java Save Properties For ″REGCUST″ in File taderc25.cbl**.
       -
       - Type `sample.cics.data` in the **Package Name** field
       - Type`RegCust` in the **Class Name** field.
       - You can select **Overwrite existing class** .
    e. Highlight **COBOL To Java Save Properties For ″BADCUST″ in File taderc25.cbl**.
       - Type `sample.cics.data` in the **Package Name** field
       - Type`BadCust` in the **Class Name** field.
       - You can select **Overwrite existing class** .
23. Click **Finish**. Expand *OutputComm*, and you will see that it contains PrefCust, RegCust and BadCust in the **Output type** field.
24. On the Java Method page, click **Finish**.
25. In the Java methods page:
    a. Type `TADERC25` (the COBOL program id) in the **functionName** field.
    b. Select **Show Advanced**.
    c. Select **SYNC_SEND_RECEIVE(1)** in the **interactionVerb** field.
    d. Type `-1` in the **replyLength** field.
26. Click **Finish**.
27. **Now you will Add the recognition pattern tag to the generate Java output data mapping file:** Since the output coming back can be any one of the data types, the only way to match it is to have some pattern predefined in the data stream. The match method checks the recognition pattern.

```
@type-descriptor.external-decimal-td
@type-descriptor.float-td
@type-descriptor.initial-value
@type-descriptor.integer-td
@type-descriptor.level88
@type-descriptor.packed-decimal-td
@type-descriptor.recognition-desc
@type-descriptor.restriction
@type-descriptor.simple-instance-td
```

a. To add the recognition pattern for PrefCust:
   - Open the PrefCust.java file in a Java editor.
   - Navigate to the **getPcustcode()** method. The best way to do this is to open the Outline view and scroll down until you find the desired method.
   - In the method comment area , add the tag `@type-descriptor.recognition-desc pattern="PREC"` or you can use the content assist by pressing CTRL-space and navigate down the list to find the tag and then enter `"PREC"` as the pattern.
   - Save the changes and the code PrefCust.java will be regenerated.
   - Navigate to the match method to ensure that the change is there:

```
/**
  * @generated

  */
 public boolean match(Object obj) {
  if (obj == null)
   return (false);
  if (obj.getClass().isArray()) {
   byte[] currBytes = buffer_;
   try {
    byte[] objByteArray = (byte[]) obj;
    buffer_ = objByteArray;
    if (!("PREC".equals(getPcustcode().toString())))
     return (false);
   } catch (ClassCastException exc) {
    return (false);
   } finally {
    buffer_ = currBytes;
   }
  } else
   return (false);
  return (true);
 }
```

b. To add the recognition pattern for RegCust:
   - Open the RegCust.java file in a Java editor.
   - Navigate to the **getRcustcode()** method. Again, the best way to do this is to open the Outline view and scroll down until you find the desired method.
   - In the method comment area , add the tag `@type-descriptor.recognition-desc pattern="REGC"` or you can use the content assist by pressing CTRL-space and navigate down the list to find the tag and then enter `"REGC"` as the pattern.
   - Save the changes and the code RegCust.java will be regenerated.
   - Navigate to the match method to make sure the change is there:

```
/**
  * @generated

  */
 public boolean match(Object obj) {
  if (obj == null)
   return (false);
  if (obj.getClass().isArray()) {
   byte[] currBytes = buffer_;
   try {
    byte[] objByteArray = (byte[]) obj;
    buffer_ = objByteArray;
    if (!("REGC".equals(getRcustcode().toString())))
     return (false);
   } catch (ClassCastException exc) {
    return (false);
   } finally {
    buffer_ = currBytes;
   }
  } else
   return (false);
  return (true);
 }
```

c. To add the recognition pattern for BadCust:
   - Open the BadCust.java file in a Java editor.
   - Navigate to the **getBcustcode()** method. Again, the best way to do this is to open the Outline view and scroll down until you find the desired method.
   - In the method comment area , add the tag `@type-descriptor.recognition-desc pattern="BADC"` or you can use the content assist by pressing CTRL-space and navigate down the list to find the tag and then enter `"BADC"` as the pattern.
   - Save the changes and the code BadCust.java will be regenerated.
   - Navigate to the match method to make sure the change is there:

```
/**
  * @generated

  */
 public boolean match(Object obj) {
  if (obj == null)
   return (false);
  if (obj.getClass().isArray()) {
   byte[] currBytes = buffer_;
   try {
    byte[] objByteArray = (byte[]) obj;
    buffer_ = objByteArray;
    if (!("BADC".equals(getBcustcode().toString())))
     return (false);
   } catch (ClassCastException exc) {
    return (false);
   } finally {
    buffer_ = currBytes;
   }
  } else
   return (false);
  return (true);
 }
```

Lesson 1.4: Deploy your application

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

Before you begin, you must complete Lesson 1.3. In this lesson you will:

- Create a Java class to test your application.
- Run the test class.

1. **First you will create the TestECIMPO file:** Expand the **CustomerProj** project, expand the **Java Resources** section and select the **sample.cics** 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 `TestECIMPO`
4. Open **TestECIMPO** in the Java editor.
5. Replace all the code in the editor with the following:

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

```
/************************************************************
 * 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 C
 ************************************************************/
package sample.cics;

import sample.cics.data.*;
public class TestECIMPO
{



 public static void process(InputComm input)
 {

  System.out.println("processing....");
     try {
  //CustomerInfoMOImpl proxy = new CustomerInfoMOImpl();
  CustomerInfoMOImpl proxy = new CustomerInfoMOImpl();
      OutputComm output = proxy.getCustomerInfo (input);

  BadCust badCust = output.getBadCust();
  PrefCust prefCust = output.getPrefCust();
  RegCust regCust = output.getRegCust();

  if (regCust != null)
  {
   System.out.println("Reg Customer");
   System.out.println("account name:  " + regCust.getAccountname());
   System.out.println("balance:  " + regCust.getBalance());
   System.out.println("cust code:  " + regCust.getRcustcode());
   System.out.println("cust no:  " + regCust.getRcustno());
  }
  else if (prefCust != null)
  {
   System.out.println("Pref Customer");
   System.out.println("assets:  " + prefCust.getAssets());
   System.out.println("cust code:  " + prefCust.getPcustcode());
   System.out.println("cust no:  " + prefCust.getPcustno());
```

```
 }
 else if (badCust != null)
 {
  System.out.println("Bad Customer");
  System.out.println("amount:  " + badCust.getAmount());
  System.out.println("cust code:  " + badCust.getBcustcode());
  System.out.println("cust no:  " + badCust.getBcustno());
  System.out.println("days overdue:  " + badCust.getDaysoverdue());
 }
 else
  System.out.println("No match");
}
catch (Exception exc)
{
 System.out.println (exc);
 exc.printStackTrace();
}

}

public static void testPrefCust()
{
   System.out.println("===========testPreCust==============");
try {
 InputComm input = new InputComm();
 String prefC = "12345";
 input.setICustNo (prefC);
 process(input);
}
catch (Exception exc)
{
 System.out.println (exc);
 exc.printStackTrace();
}


}

public static void testRegCust()
{
   System.out.println("===========testRegCust==============");
try {
 InputComm input = new InputComm();
 String regC = "34567";
 input.setICustNo (regC);
 process(input);
}
catch (Exception exc)
{
 System.out.println (exc);
 exc.printStackTrace();
}

}

public static void testBadCust()
{

 System.out.println("===========testBadCust==============");
try {

 InputComm input = new InputComm();
 String badC = "123";
 input.setICustNo (badC);
 process(input);

}
```

```
catch (Exception exc)
{
 System.out.println (exc);
 exc.printStackTrace();
}
}


public static void main (String[] args)
{
 testPrefCust();
 testRegCust();
 testBadCust();

}
}
```

6. **Next you will test the application**
7. Right-click **TestECIMPO.java** and select **Run as> Java Application**.
8. The console should display the following output:

```
==========testPreCust========
processing. . . .
Pref Customer
assets:  43456.33
cust code: PREC
cust no: 12345
==========testRegCust========
processing. . . .
Reg Customer
account name: SAVINGS
assets:  11456.33
cust code: REGC
cust no: 34567
==========testBadCust========
processing. . . .
Bad Customer
assets:  -8965.33
cust code: BADC
cust no: 123
days overdue: 132
```

Congratulations! You have completed the CICS Taderc25 tutorial.

Create a J2C application for a CICS transaction containing multiple possible outputs summary

This tutorial has taught you how to use the J2C Java Bean wizard to build a simple web application that processes a CICS transaction with multiple possible outputs.

Lessons learned

From this tutorial, you have learned how to

- Use the J2C Java bean wizard to create a J2C application that interfaces with a CICS transaction using an External Call Interface (ECI).
- Create a Java method, **getCustomerInfo**, which accepts a customer number. Depending on the customer's classification, preferred customer, regular customer or bad customer, the program returns different output information about the customer
- Create a test Java class to test the application.
-