Develop an IMS Application using Java and Open Database

Skill Level: Beginner to Intermediate

Poonam Chitale (pchitale@us.ibm.com)
Software Engineer, IBM®

Joshua Newell (newelljo@us.ibm.com)
Software Engineer, IBM®

IBM Corporation
August 2013

Abstract

This tutorial takes you through the steps of using Rational® Developer for System z® Version 8.0.3 to write a Java™ application to access IMS databases through the IMS Universal DL/I driver and the IMS Universal JDBC driver.

About this tutorial

This tutorial will take you through the steps of writing a Java application to access IMS databases using the IMS Universal DL/I driver and the IMS Universal JDBC driver.

Customers who store business data in IMS databases want an easy way to access their data. They also want to be able to develop applications for IMS using modern and standardized programming solutions. The IMS Universal drivers, part of the IMS Version 12 Open Database solution, are software components that provide Java applications with connectivity to IMS databases from z/OS® and from distributed environments through TCP/IP.

The IMS Universal drivers are built on industry standards and open specifications. Java applications that use the IMS Universal drivers can reside on the same logical partition (LPAR) or on a different LPAR from the IMS subsystem. Two types of connectivity are supported by the IMS Universal drivers: local connectivity to IMS databases on the same LPAR (type-2 connectivity) and distributed connectivity through TCP/IP (type-4 connectivity).

This tutorial will help to familiarize you with using two of the IMS Universal drivers:

- IMS Universal DL/I driver, which provides a stand-alone Java application programming interface (API) for writing granular queries to IMS databases using programming semantics similar to traditional IMS DL/I calls

- IMS Universal JDBC driver, which provides a stand-alone Java Database Connectivity (JDBC) 3.0 driver for making structured query language (SQL)-based database calls to IMS databases.

In this tutorial, you will run Java applications in a Windows® environment and connect to the IMS database using type-4 connectivity mode.

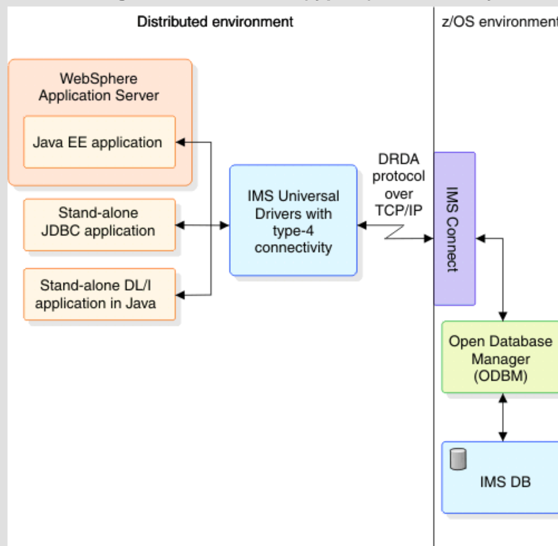
© Copyright IBM Corporation 2011, 2012. All rights reserved.

Distributed and local connectivity with the IMS Universal drivers

The IMS Universal drivers support distributed (type-4) and local (type-2) connectivity to IMS databases. The connectivity type is specified in the driverType connection property. In this tutorial exercise, you will use type-4 connectivity.

- **Type-4 connectivity:** With type-4 connectivity, the IMS Universal drivers can run on any platform that supports TCP/IP and a Java Virtual Machine (JVM), including z/OS. To access IMS databases using type-4 connectivity, the IMS Universal drivers first establish a TCP/IP-based socket connection to IMS Connect. IMS Connect is responsible for routing the request to the IMS databases using the Open Database Manager (ODBM), and sending the response back to the client application. The DRDA® protocol is used internally in the implementation of the IMS Universal drivers. You do not need to know DRDA to use the IMS Universal drivers.

Figure 1: Distributed (type-4) connectivity



- **Type-2 connectivity:** Local (or type-2) connectivity with the IMS Universal drivers is targeted for the z/OS platform and runtime environments. You would use type-2 connectivity when connecting to IMS subsystems in the same logical partition (LPAR). In this tutorial, you will not need type-2 connectivity.

Objectives

To understand and gain hands-on experience creating Java applications to access and manipulate enterprise data residing on the IMS database.

Upon completion of this study, you will be able to perform these tasks:

- Create a Java application to access IMS data by issuing IMS DL/I calls to the IMS database through the IMS Universal DL/I driver

- Create a Java application to access IMS data by issuing SQL calls to the IMS database through the IMS Universal JDBC driver

© Copyright IBM Corporation 2011, 2012. All rights reserved.

- Deploy and run a Java application in a Windows environment

System requirements for the tutorial:

Software installed on Windows

- Rational Developer for System z Version 8.0.3 (RDz)
- IMS Universal drivers libraries
  - imsudb.jar
- Sample Java project
  - IMSDBJavaApplicationLab.zip

System software installed on IBM z/OS

- IMS Version 12 configured with Open Database Manager (ODBM)
- IMS Connect Version 12

Checklist for first-time implementation

You may find it helpful have the following information and resources ready before proceeding with your first implementation of the Java applications using the IMS Universal drivers. The information and resources to run this tutorial is provided in the checklist below.

Table 1: Implementation checklist

Information or resource	Your environment	For this tutorial
IMS Connect host name (or IP address) and DRDA port number	Obtain this information from IMS system programmers.	**Host name:** ZSERVEROS.DEMOS.IBM.COM **DRDA** p**ort number:** 7001
IMS data store name (IMS ID)	Obtain this information from IMS system programmers.	**Datastore name:** IMSD
z/OS user ID and password	Obtain this information from IMS system programmers.	**Userid:** EM4ZIMS
MetadataURL to the Java metadata file generated by the IMS Explorer	Obtain this information from IMS application programmers.	**MetadataURL:(DatabaseName)** class://com.ibm.ims.db.databaseviews.DFSSAM09DatabaseView
Workspace directory and project name to be used when generating artifacts	A naming standard is recommended.	**Workspace directory:** C:\share\imsjavalab\workspace

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Overview of development tasks

To complete this tutorial, you will perform the following tasks:

1 Task 1 - Install the tutorial sample project

1.1 Import the IMSDBJavaApplicationLab sample project

1.2 Verify that the IMS Universal drivers library is located on the build path

2 Task 2 – Access IMS data with the IMS Universal DL/I driver

2.1 Connect to the IMS database through the IMS Universal DL/I driver

2.1.1 Open the DLIAPIAssigment.java sample application

2.1.2 Set the connection properties

2.2 Issue DL/I calls to access the IMS database

2.2.1 Exercise 1 - Retrieve data in an IMS database

2.2.2 Exercise 2: Retrieve batch data in an IMS database

2.2.3 Exercise 3: Create SSALists with multiple segments, specify qualifications, and mark specific fields for retrieval

2.2.4 Exercise 4: Utilize command codes for DL/I

3 Task 3 - Access IMS data with the IMS Universal JDBC driver

3.1 Connect to the IMS database through the IMS Universal JDBC driver

3.1.1 Open the JDBCApiAssignment.java sample application

3.1.2 Set the connection properties

3.2 Issue SQL calls to access the IMS database

3.2.1 Exercise 1 - Retrieve all fields of a segment

3.2.2 Exercise 2 - Retrieve fields of a segment based on a conditional statement

3.2.3 Exercise 3 - Order SQL query output by field values

3.2.4 Exercise 4 - Retrieve a specific field of a segment

3.2.5 Exercise 5 – Retrieve multiple fields from multiple segments

© Copyright IBM Corporation 2011, 2012. All rights reserved.

1 Task 1 - Install the tutorial sample project

In this task, you will import the tutorial sample project to IBM Rational Developer for System z (RDz), and verify that the Java library with the code to run the IMS Universal DL/I and JDBC drivers is installed.

1.1 Switch to the Java perspective

Switch from the default z/OS Projects perspective to the Java perspective.

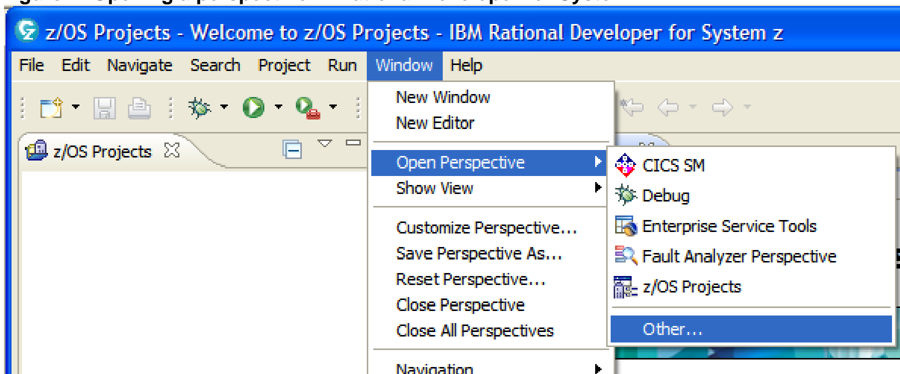
1. IBM Rational Developer for System z V 8.0.3 is started and you are using the C:\share\imsjavalab\workspaceas your workspace directory.

Important:
For this tutorial, you will use **C:\share\imsjavalab\workspace**as your workspace directory.

> **The Workspace**
> In RDz, a workspace is a directory that stores files for your projects. You can select your own directory or take the default directory. Artifacts created by RDz will be stored in this directory.

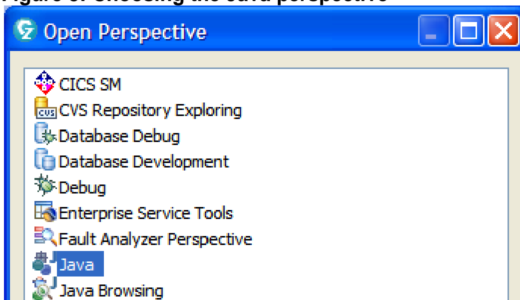
2. From the menu bar, select **Window** > **Open Perspective** > **Other.**

Figure 2: Opening a perspective in Rational Developer for System z



3. Scroll down and select **Java** from the Open Perspective dialog box.

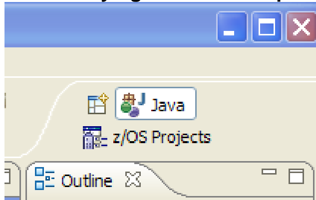
Figure 3: Choosing the Java perspective



© Copyright IBM Corporation 2011, 2012. All rights reserved.

4. Press **OK** to switch to the Java perspective.
5. To verify that you are in Java perspective, make sure that the Java button appears in the upper right corner of RDz, as shown in the figure below.

Figure 4: Verifying that the Java perspective is opened.



What is a perspective?
A perspective defines the initial set and layout of views in the Workbench window. Within the window, each perspective shares the same set of editors. Each perspective provides a set of functionality aimed at accomplishing a specific type of task or works with specific types of resources. For example, the Java perspective combines views that you would commonly use while editing Java source files, while the Debug perspective contains the views that you would use while debugging Java programs.

1.2 Import the IMSDBJavaApplicationLab sample project

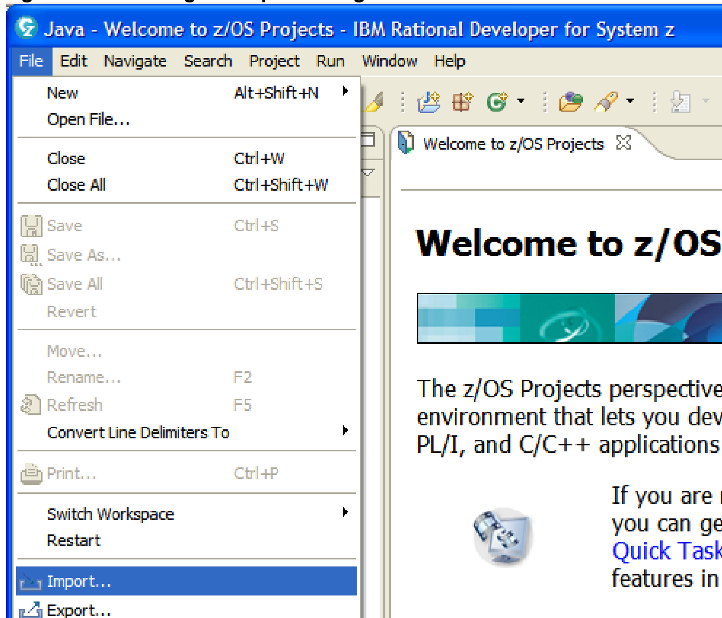
Import the files for the IMSDBJavaApplicationLab sample project into the RDz workspace.

The IMSDBJavaApplicationLab sample project
The sample project includes the Java library that contains the IMS Universal drivers required for this tutorial. The sample also includes sample Java application code that you will customize to connect to an IMS database and issue database access calls.

1. From the menu bar, click **File** > **Import** to open the Import dialog box.

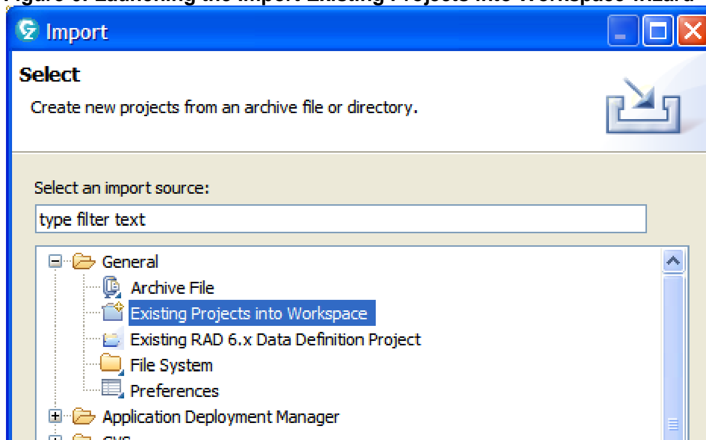
© Copyright IBM Corporation 2011, 2012. All rights reserved.

Figure 5: Launching the Import dialog box



2. From the Import dialog box, select **General** > **Existing Projects into Workspace** and click **Next**.

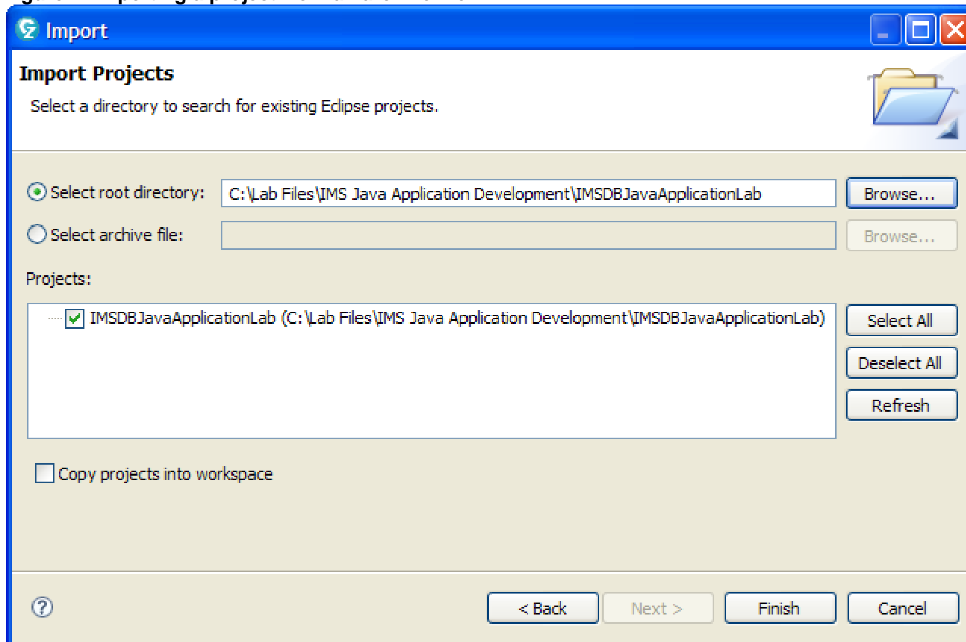
Figure 6: Launching the Import Existing Projects into Workspace wizard



3. From the Import Projects page, select **Select root directory** and click **Browse**.
4. Browse to the directory **C:\Lab Files\IMS Java Application Development\IMSDBJavaApplicationLab** and click **OK**.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

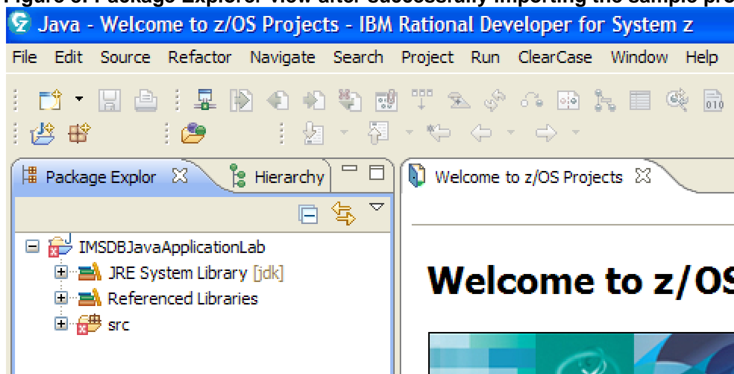
Figure 7: Importing a project from an archive file



5. Make sure that the checkbox for **IMSDBJavaApplicationLab** is selected and click **Finish**.

The sample project IMSDBJavaApplicationLab should appear in the Package Explorer view.

Figure 8: Package Explorer view after successfully importing the sample project



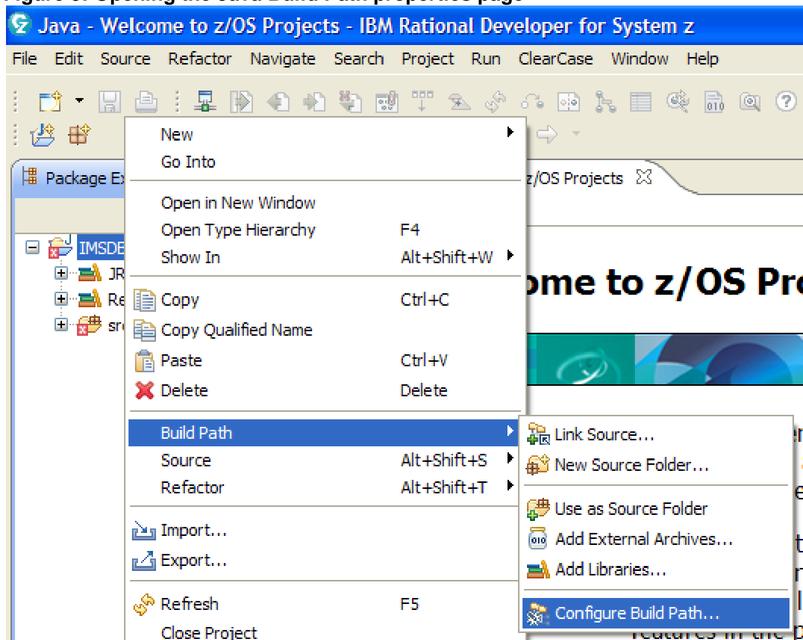
© Copyright IBM Corporation 2011, 2012. All rights reserved.

1.3 Verify that the IMS Universal drivers library is located on the build path

Verify that the Java archive file imsudb.jar is correctly located in the build path of this project.

1. Right click on the project in the Package Explorer view and select **Build Path** > **Configure Build Path**

Figure 9: Opening the Java Build Path properties page

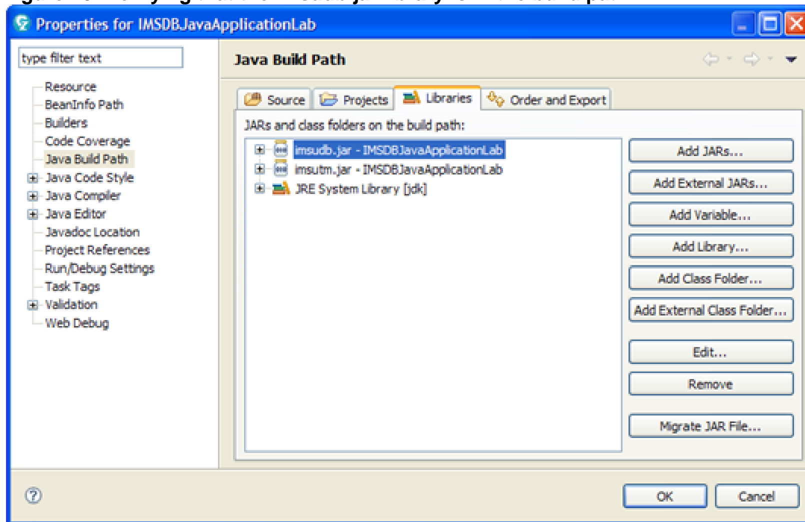


2. From the Java Build Path page, click on the **Libraries** tab. Verify that the file **imsudb.jar – IMSDBJavaApplicationLab** is present.

The imsudb.jar library
The imsudb.jar file contains the Java classes, interfaces, and metadata required to use the IMS Universal DL/I driver and the IMS Universal JDBC driver.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Figure 10: Verifying that the imsudb.jar library is in the build path



3. Click **OK** to save your changes and exit the Java Build Path page.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

2 Task 2 – Access IMS data with the IMS Universal DL/I driver

In this task, you will write a Java application to connect to an IMS database and manipulate data using a DL/I-based syntax with the IMS Universal DL/I driver.

What is DL/I?
Data Language/I (DL/I) is the IMS data manipulation language, which is a common high-level interface between a user application and IMS. DL/I calls are invoked from application programs written in languages such as Java, PL/I, COBOL, VS Pascal, C, and Ada. It also can be invoked from assembler language application programs by subroutine calls. IMS lets the user define data structures, relate structures to the application, load structures, and reorganize structures.

By using the IMS Universal DL/I driver, you can build segment search arguments (SSAs) and use the methods of the program communication block (PCB) object to read, insert, update, delete, or perform batch operations on segments. You can gain full navigation control in the segment hierarchy.

Basic programming model for a Java application using the IMS Universal DL/I driver
In general, to write a IMS Universal DL/I driver application, follow these steps:

1. Import the com.ibm.ims.dli package that contains the IMS Universal DL/I driver classes, interfaces, and methods.
2. Connect to an IMS database subsystem.
3. Obtain a program specification block (PSB), which contains one or more PCBs.
4. Obtain a PCB handle, which defines an application's view of an IMS database and provides the ability to issue database calls to retrieve, insert, update, and delete database information.
5. Obtain an unqualified segment search argument list (SSAList) of one or more segments in the database hierarchy.
6. Add qualification statements to specify the segments targeted by DL/I calls.
7. If retrieving data, mark the segment fields to be returned.
8. Execute DL/I calls to the IMS database.
9. Handle errors that are returned from the DL/I programming interface.
10. Disconnect from the IMS database subsystem.

2.1 Connect to the IMS database through the IMS Universal DL/I driver

Before you can execute DL/I calls from your IMS Universal DL/I driver application, you must connect to an IMS database.

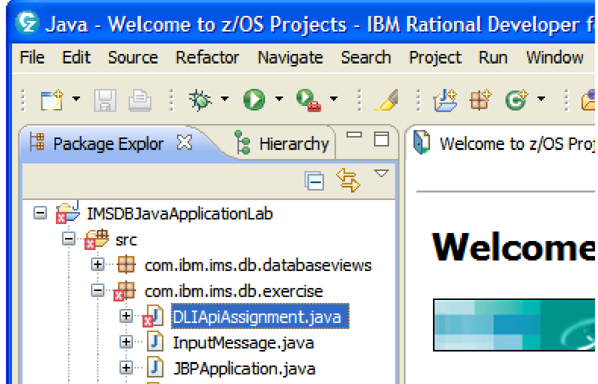
© Copyright IBM Corporation 2011, 2012. All rights reserved.

2.1.1 Open the DLIAPIAssigment.java sample application

The DLIApiAssignment.java sample application
This sample application contains skeleton Java code for connecting to the IMS database and issuing DL/I data access calls using the IMS Universal DL/I driver.

1. From the Package Explorer view, expand **IMSDBJavaApplicationLab** > **src** > **com.ibm.ims.db.exercise**.

Figure 11: Navigating to the DLIApiAssignment.java sample application

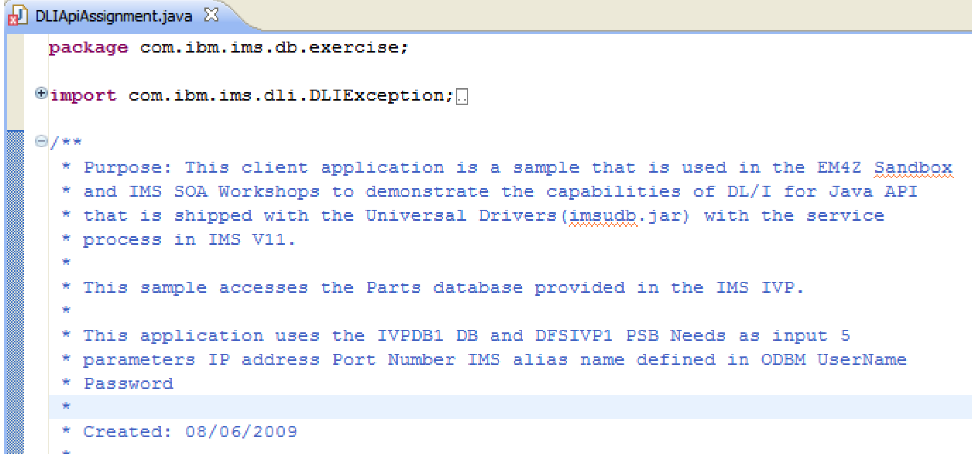


2. From the Package Explorer view, double click on the file **DLIApiAssignment.java** to open the sample application in the Java editor.

Maximizing a view is the ability to increase a view to the maximum possible size on the screen. This can be accomplished by double-clicking on the view tab. To go back to the original view size, double-click on the view tab again.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Figure 12: The opened DLIApiAssignment.java sample application in the Java editor

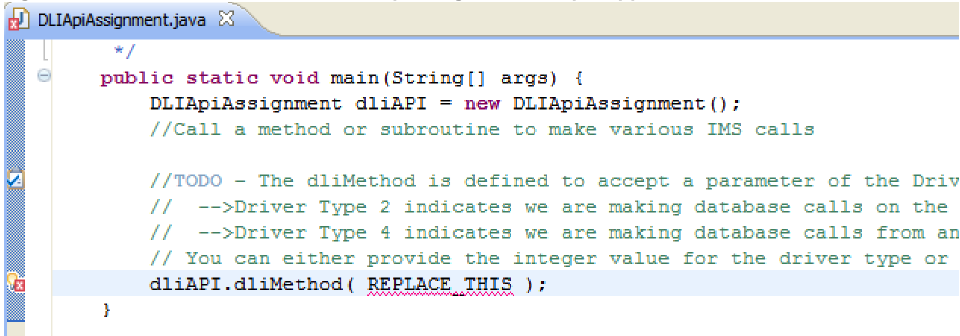


The Java editor
The Java editor provides specialized features for editing Java code. The editor includes support for syntax highlighting, content/code assist, code formatting, import assistance, and integrated debugging features.

2.1.2 Set the connection properties

1. In the Java editor, scroll down the application source code until you find the Java main method shown in the screenshot below.

Figure 13: Java `main` method in the DLIApiAssignment sample application



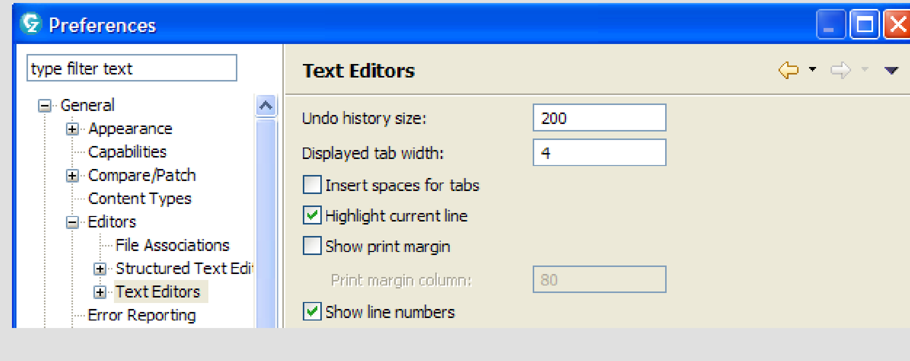
Moving your cursor to a specific line number
RDz provides a shortcut to move your cursor directly to a specific line in an editor. To go to a specific line, press **Ctrl + L** from the editor. Enter the line number and press **OK**.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Displaying line numbers in the editor

Line numbers can be displayed directly in the editor by going to **Windows** > **Preferences**. In the Preferences Dialog Menu navigate to **General** > **Editors** > **Text Editors** and check the box next to **Show line numbers**.

Figure 14: Configuring the editor to display line numbers



2. In line 44 of the code, delete the constant **REPLACE_THIS** and replace it with **IMSConnectionSpec.DRIVER_TYPE_4** to set the driver connectivity type.

Java code assist

RDz provides code assist for Java applications. By pressing CTRL **+ space**, the Java editor will display a list of possible commands variables for that line. Try specifying the driver type by typing `IMS` and pressing **CTRL + space** and scrolling to the constant `IMSConnectionSpec`. Alternatively, when you type a period (**.**) after a class, the Java code assist displays a menu of methods and variables that the class can invoke. Try it after `IMSConnectionSpec` and select `DRIVER_TYPE_4`.

3. In line 56 of the code, delete the string **"your.host.name.com"** and replace it with **"zserveros.demos.ibm.com"** to set the host.

The datastoreServer property

The `host` variable in the sample application is used to set the datastoreServer property. This connection property contains the name or IP address of the data store server (IMS Connect). You can provide either the host name (for example, dev123.svl.ibm.com) or the IP address (for example, 192.166.0.2). In this tutorial, the target IMS Connect has already been pre-configured for you. Use **zserveros.demos.ibm.com** as the dataStoreServer.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Figure 15: Setting the connection properties

```
DLIApiAssignment.java
52  public void dliMethod(int driverType) {
53      // TODO - assign the host name or IP address of the IMS Connect you a
54      // The IP address or host name of the IMS Connect that you are
55      // connecting to
56      String host = "your.host.name.com";
57      String datastoreName = ""; // The IMS alias name defined in ODBM
58      String username = "yourID"; // User Name
59      String password = Vault.getPassword(username);; // Password
60      int drdaPort = 7001; // the ICON DRDA port number
```

4. In line 57 of the code, delete the string **""** and replace it with "**IMSD"** to set the datastoreName.

The datastoreName property
This connection property contains the name of the IMS data store to access. When using type-4 connectivity, the datastoreName property must match either the name of the data store defined to Open Database Manager (ODBM) or be blank. In this tutorial, the target IMS data store has already been created for you and pre-populated with data. Use **IMSD** as the dataStoreName.

5. In line 58 of the code, delete the string **"yourID"** and replace it with **"EM4ZIMS"** to set the username.

The user and password properties
The user and password connection properties are the user name and password used for the connection to IMS Connect. This information can typically be obtained from your RACF® administrator.

6. In line 60 of the code, verify that the drdaPort value **7001** has already been set for you.

The portNumber property
The `drdaPort` variable in the sample application is used to set the portNumber property. This connection property is the TCP/IP server port number to be used to communicate with IMS Connect. The portNumber property is not required when using type-2 connectivity. In this tutorial, the target IMS Connect has already been pre-configured for you. Use **7001** as the drdaPort.

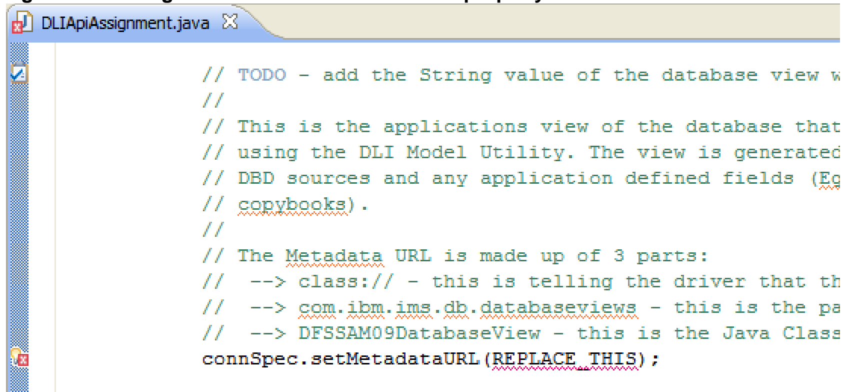
7. In line 82 of the code, delete the constant **REPLACE_THIS** and replace it with the string **"class://com.ibm.ims.db.databaseviews.DFSSAM09DatabaseView"** to set the metadataURL.

The metadataURL property
This connection property is the location of the database metadata representing the target IMS database. The metadataURL property is the fully qualified name of the Java metadata class generated by the IMS Enterprise Suite DLIModel utility plug-in, based on the PSB and DBD source files of the target IMS database. The Java metadata class must be generated before coding a Java application to access the target IMS database using the IMS Universal drivers. The format of the metadataURL is: `"class://packageName.className"`

In this tutorial, the Java metadata class has already been generated for you. Use **class:\\com.ibm.ims.db.databaseviews.DFSSAM09DatabaseView** as the metadataURL.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Figure 16: Setting the metadataURL connection property



8. Press Ctrl + S to save your code changes.

After completing this step, your Java application should be ready to connect using the Universal DL/I driver. Next, you will need to modify the Java application code to issue data access calls to IMS.

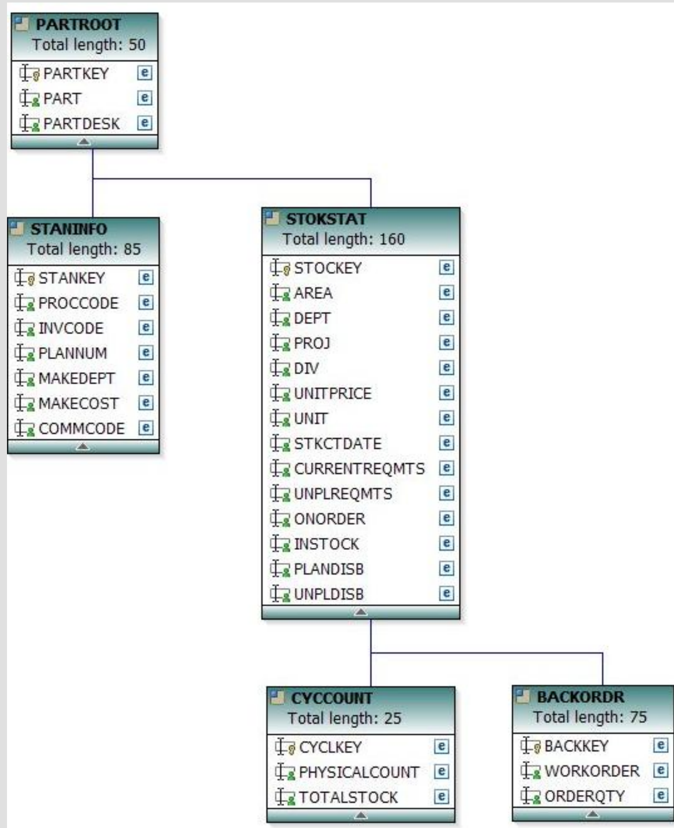
© Copyright IBM Corporation 2011, 2012. All rights reserved.

The Parts Order sample database

This tutorial uses the Parts Order database that is provided in the IMS Installation Verification Program (IVP). You can refer to this diagram when working on the exercises in this tutorial.

The diagram below shows the hierarchical structure of the segments in the Parts Order database. Each rectangle represents a database segment. PARTROOT is the root segment of this database, and STANINFO and STOKSTAT are its child segments. STOKSTAT has CYCCOUNT and BACKORDR as its child segments. Each segment contains one or more fields that contain data. For example, PARTKEY is a field in the PARTROOT segment.

Figure 17: Segments of the Parts Order database (reference only)



© Copyright IBM Corporation 2011, 2012. All rights reserved.

2.2 *Issue DL/I calls to access the IMS database*

The following exercises in this section will show you how to issue DL/I calls in your Java application to retrieve data from the IMS database using the IMS Universal DL/I driver.

Lab exercises

This task contains several programming exercises for you to complete. These exercises will help to familiarize you with basic data access operations using the IMS Universal DL/I driver. At certain points indicated in the instructions, you will be asked to provide the correct code. For your reference, we have provided the exercise solutions. You can find the code with the exercise solutions from the Package Explorer view by opening **IMSDBJavaApplicationLab** > **com.ibm.ims.db.exercise.solution** > **DLIApiAssignment.java**

2.2.1 Exercise 1 - Retrieve data in an IMS database

In this exercise, you will retrieve data in an IMS database by issuing DL/I Get Unique and Get Next calls through the IMS Universal DL/I driver.

Using the Get Unique (GU) and Get Next (GN) DL/I calls

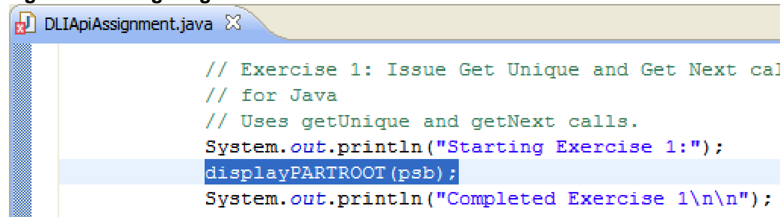
If an input message contains more than one segment, a Get Unique call retrieves the first segment of the message and Get Next (GN) calls retrieve the remaining segments.

When issued from the IMS Universal DL/I driver, the Get Unique call retrieves a specific segment or collection of segments on a hierarchic path from an IMS database. The GU call also establishes the position in the database from which additional segments can be processed in a forward direction.

The Get Next call retrieves the next segment or collection of segments on a hierarchic path from an IMS database. The GN call usually proceeds forward along the hierarchy of a database from the current database position to the next required segment. To modify the GN call to start at an earlier position than the current position in the database, you can use an IMS command code. The Get Next call returns a Path object representing the hierarchic path from the root segment to the segment the cursor is currently positioned on. The Path object includes the data stored in the segments along the hierarchic path.

Exercise 1 begins on line 103 of the **DLIApiAssignment.java** sample application, where the function `displayPARTROOT(psb)` is invoked. To go to the start of the function, go to line 107, move the mouse over the `displayPARTROOT(psb)` function invocation, and press **F3**.

Figure 18: Navigating to the start of Exercise 1



© Copyright IBM Corporation 2011, 2012. All rights reserved.

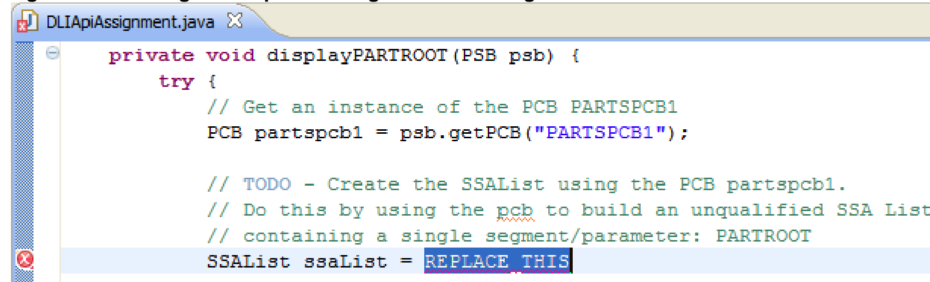
2.2.1.1 Exercise 1 - Step 1: Define an unqualified SSAList to specify the segments to retrieve

1. In line 146 of the code, delete the constant **REPLACE_THIS** and replace it with the Java code statement to get an unqualified SSAList for the PARTROOT segment. You can find the answer after Figure 19 below.

Hint:
Use the PCB object that has been created (**partspcb1**) to call the **getSSAList(String)** method. Pass in the segment name (**"PARTROOT"**) as the input parameter.

> **The `SSAList` interface**
>
> The `com.ibm.ims.dli.SSAList` interface represents a list of segment search arguments (SSAs) used to specify the segments to target in a particular database call. Use the `SSAList` interface to construct each segment search argument in the list, and to set the command codes and lock class for the segment search arguments. Each SSA in the `SSAList` can be qualified or unqualified. A SSA qualification can be used to filter the segments to update or retrieve on a hierarchic path.

Figure 19: Defining the unqualified segment search argument list



> **Verify your Java code statement:**
>
> In line 146, your Java code statement should look like this:

```
SSAList ssaList = partspcb1.getSSAList("PARTROOT");
```

2.2.1.2 Exercise 1 - Step 2: Issue a Get Unique DL/I call to retrieve segments

1. In line 164 of the code, delete the constant **REPLACE_THIS** and replace it with the Java code statement to issue a Get Unique DL/I call. You can find the answer after figure 20 below.

Hint:
Use the PCB object that was previously created (**partspcb1**) to call the **getUnique(Path, SSAList, boolean)** method.

- Pass in the Path object that was previously created (**path**) as the 1st input parameter.
- Pass in the SSAList object that was previously created (**ssaList**) as the 2nd input parameter.
- Pass in the boolean value **false** as the 3rd input parameter. False indicates that this DL/I call is not a Get Hold Unique call.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Figure 20: Insert code to issue the Get Unique call

```
DLIApiAssignment.java

        // TODO - Make a Get Unique call using partspcb1
        // The call takes 3 parameters: Path, SSAList, and a boolean
        // indicating if this is a HOLD call (GHU, GHN, GHNP).
        // The call will return true if the call is successful or false if
        // not.
        if (REPLACE_THIS) {
            // Format the output
            System.out.println("PARTKEY\t\tPART\t\tPARTDESC");
            System.out.println("--------------------------------------------
```

Verify your Java code statement:
In line 164, your Java code statement should look like this:

```
if (partspcb1.getUnique(path, ssaList, false)) {
```

2.2.1.3 Exercise 1 – Step 3: Issue a Get Next DL/I call

1. In line 178 of the code, delete the constant **REPLACE_THIS** and replace it with the Java code statement to issue a Get Next DL/I call. You can find the answer after Figure 21 below.

Hint:
Use the PCB object that was previously created (**partspcb1**) to call the **getNext(Path, SSAList, boolean)** method.

- Pass in the Path object that was previously created (**path**) as the 1st input parameter
- Pass in the SSAList object that was previously created (**ssaList**) as the 2nd input parameter
- Pass in the boolean value **false** as the 3rd input parameter, to indicate that this DL/I call is not a Get Hold Next call.

Figure 21: Insert code to issue a Get Next call

```
DLIApiAssignment.java

            // TODO - Make a get Next Call
            // Now lets continue to get more results by calling ge
            // (GN). The getNext Java call is used similarly to th
            // however this time we will use a while loop to go th
            // segments matching the SSAList are returned.
            while (REPLACE_THIS) {
                System.out.println(path.getString("PARTKEY").trim(
                        + "\t\t" + path.getString("PARTDESC").trim
            }
```

Verify your Java code statement:
In line 178, your Java code statement should look like this:

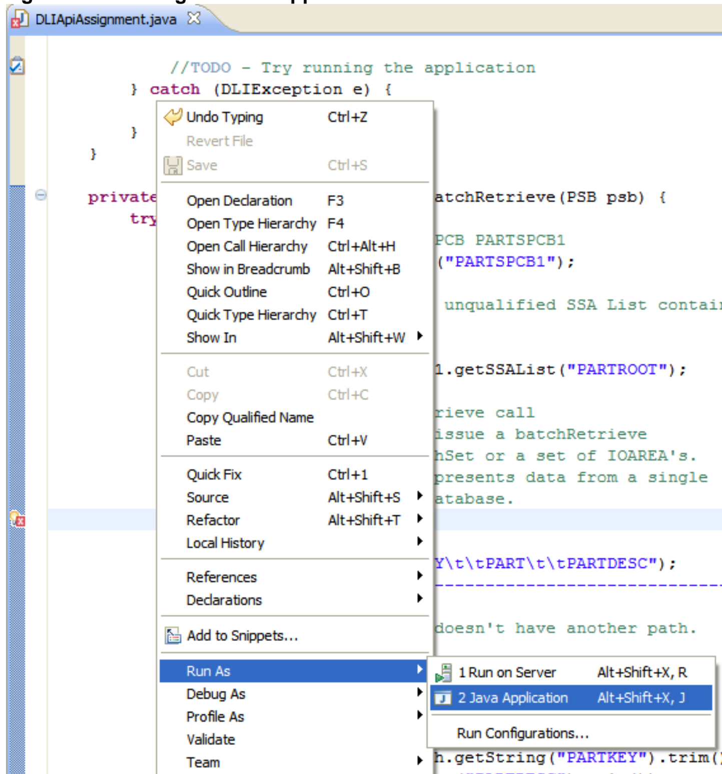
```
while (partspcb1.getNext(path, ssaList, false)) {
```


© Copyright IBM Corporation 2011, 2012. All rights reserved.

2.2.1.4 Exercise 1 – Step 4: Run the application and verify the output results

1. Press `Ctrl + S` to save your changes to the files.
2. Right click on the Java editor and select **Run As** > **Java Application**.

Figure 22: Running the Java application



3. An **Errors in Workspace** dialog box will appear but you can safely ignore it. Click on **Proceed** to continue.
4. In the **Console view**, verify that the output results look like the screenshots below.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Figure 23: Beginning of Exercise 1 result output in the Console view

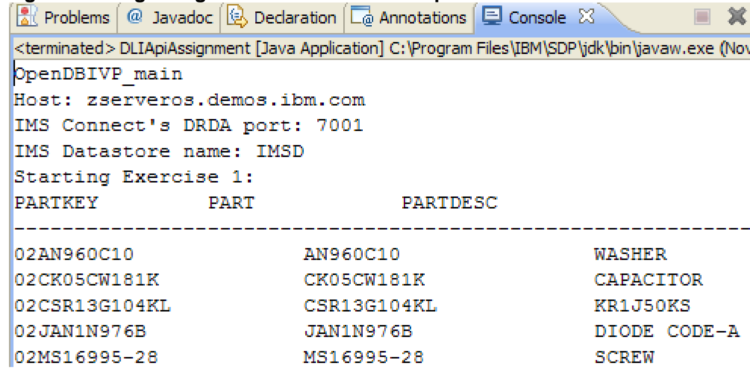
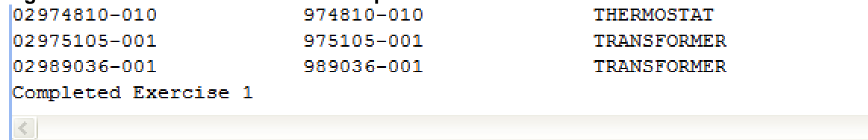


Figure 24: End of Exercise 1 result output in the Console view



2.2.2 Exercise 2: Retrieve batch data in an IMS database

In this exercise, you will retrieve batch data from an IMS database by issuing a Batch Retrieve call through the IMS Universal DL/I driver.

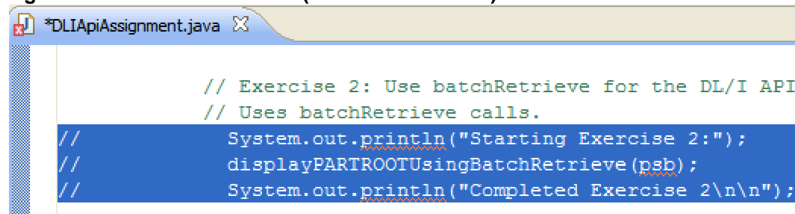
> **Batch Retrieve**
> You can use the batch retrieve call to retrieve multiple segments from an IMS database in a single call. Instead of a client application making multiple GU and GN calls, IMS performs all the GU and GN processing and returns the results back to the client in a single batch network operation. The fetch size property determines how much data is returned on each batch network operation.

Exercise 2 begins on line 110 of the **DLIApiAssignment.java** sample application. At the beginning of Exercise 2, the code for this exercise has been commented out.

2.2.2.1 Exercise 2 – Step 1: Uncomment the code for Exercise 2

1. In the **Java editor**, highlight lines 112 to 114 of the **DLIApiAssignment.java** sample application and press `Ctrl + /` to uncomment the code.

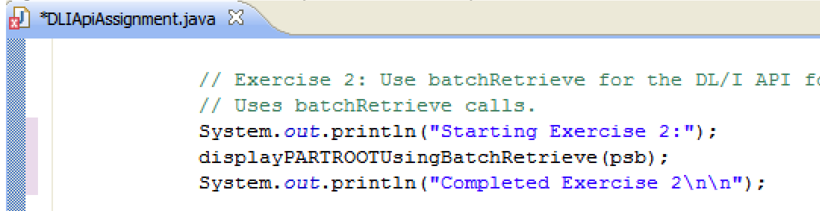
Figure 25: Code for Exercise 2 (before uncomment)



© Copyright IBM Corporation 2011, 2012. All rights reserved.

Code comments allows comment statements that will not be compiled and executed to be inserted directly into the application source code. In RDz, blocks of code can be commented and uncommented by highlighting that block and pressing **Ctrl + /.**

Figure 26: Code for Exercise 2 (after uncomment)



```
// Exercise 2: Use batchRetrieve for the DL/I API fo
// Uses batchRetrieve calls.
System.out.println("Starting Exercise 2:");
displayPARTROOTUsingBatchRetrieve(psb);
System.out.println("Completed Exercise 2\n\n");
```

2. The function `displayPARTROOTUsingBatchRetrieve` contains the Java code for the batch retrieval operation. In line 113, move your mouse over the `displayPARTROOTUsingBatchRetrieve(psb)` function invocation and press F3 to open the function declaration.

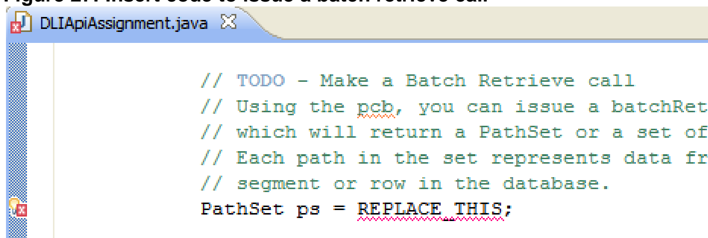
2.2.2.2 Exercise 2 – Step 2: Issue a Batch Retrieve call to retrieve multiple segments

1. In line 210 of the code, delete the constant **REPLACE_THIS** and replace it with the Java code statement to issue a Batch Retrieve call. You can find the answer after the figure below.

Hint:
Use the PCB object that was previously created (**partspcb1**) to call the **batchRetrive(SSAList)** method.

- Pass in the SSAList object that was previously created (**ssaList**) as the input parameter.

Figure 27: Insert code to issue a batch retrieve call



```
// TODO - Make a Batch Retrieve call
// Using the pcb, you can issue a batchRet
// which will return a PathSet or a set of
// Each path in the set represents data fr
// segment or row in the database.
PathSet ps = REPLACE_THIS;
```

Verify your Java code statement:
In line 210, your Java code statement should look like this:

```
PathSet ps = partspcb1.batchRetrieve(ssaList);
```


© Copyright IBM Corporation 2011, 2012. All rights reserved.

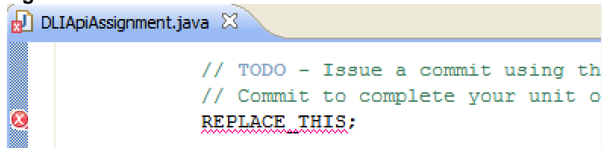
2.2.2.3 Exercise 2 – Step 3: Commit the unit of work

1. In line 227 of the code, delete the constant **REPLACE_THIS** and replace it with the Java code statement to commit your unit of work. You can find the answer after the figure below.

Hint:
Use the PSB object was previously created (**psb**) to call the **commit()** method.

> **Committing and rolling back DL/I transactions**
> The IMS Universal DL/I driver provides support for local transactions with the commit and rollback methods. A local transaction consists of a unit of work with several units of recovery. An IMS Universal DL/I driver application can commit or roll back changes to the database within a unit of recovery. In the IMS Universal DL/I driver, the local transaction is scoped to the `PSB` instance. No explicit call is needed to begin a local transaction. After the unit of work starts, the application makes DL/I calls to access the database and create, replace, insert, or delete data. The application commits the current unit of recovery by using the `PSB.commit` method. The commit operation instructs the database to commit all changes to the database that are made from the point when the unit of work started, or from the point after the last commit or rollback method call, whichever was most recent.

Figure 28: Insert code to issue a commit call



> **Verify your Java code statement:**
> In line 227, your Java code statement should look like this:
>
> ```
> psb.commit();
> ```

2.2.2.4 Exercise 2 – Step 4: Run the application and verify the output results

1. Press `Ctrl + S` to save your changes to the files.
2. Right click on the Java editor and select **Run As** section > **Java Application**, as shown in section 2.2.1.4.
3. An **Errors in Workspace** dialog box will appear. Click on **Proceed**.
4. In the **Console view**, verify that the output results look like the screenshots below.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Figure 29: Beginning of Exercise 2 result output in the Console view

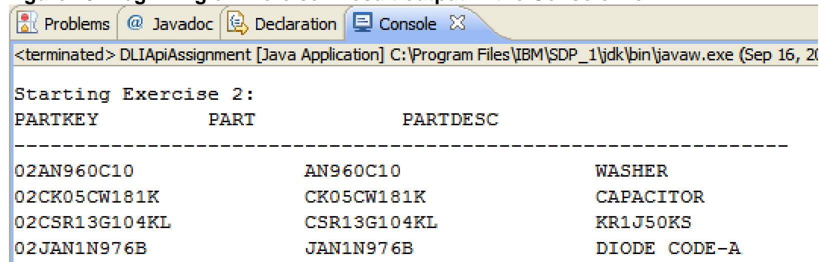
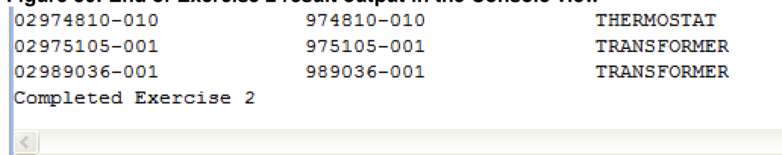


Figure 30: End of Exercise 2 result output in the Console view



2.2.3 Exercise 3: Create SSALists with multiple segments, specify qualifications, and mark specific fields for retrieval

In this exercise, you will mark specific segment fields for retrieval from the IMS database. You will also specify the number of rows of data for the IMS Universal DL/I driver to retrieve.

> **Marking segment fields for retrieval with the IMS Universal DL/I driver**
>
> In your Java application, you can specify which segment fields are to be returned from a database retrieve call by using the `markFieldForRetrieval` or the `markAllFieldsForRetrieval` methods. Following the IMS default, all of the fields in the lowest level segment specified by the `SSAList` are initially marked for retrieval.
>
> **The `markFieldForRetrieval` method**
>
> This `SSAList` method is used to mark a specific field for retrieval from the database. The `markFieldForRetrieval` method is used together with `getPathForRetrieveReplace`() and with the data retrieval methods in the `PCB` interface. When a retrieve call is made, the resulting `Path` object will contain all the fields that have been marked for retrieval.
>
> **The `markAllFieldsForRetrieval` method**
>
> This method is used to mark all fields in the specified segment for retrieval from the database. The markAllFieldsForRetrieval method is used together with `getPathForRetrieveReplace()` and with the data retrieval methods in the `PCB` interface. When a retrieve call is made the resulting `Path` object will contain only the fields marked for retrieval. Following the IMS default, all of the fields in the lowest level segment specified by the `SSAList` are initially marked for retrieval.

Exercise 3 begins on line 116 of the **DLIApiAssignment.java** sample application. At the beginning of Exercise 3, the code for this exercise has been commented out.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

2.2.3.1 Exercise 3 – Step 1: Uncomment the code for Exercise 3

1. In the **Java editor**, highlight lines 118 to 120 of the **DLIApiAssignment.java** sample application and press `Ctrl + /` to uncomment the code.

Figure 31: Code for Exercise 3 (before uncomment)

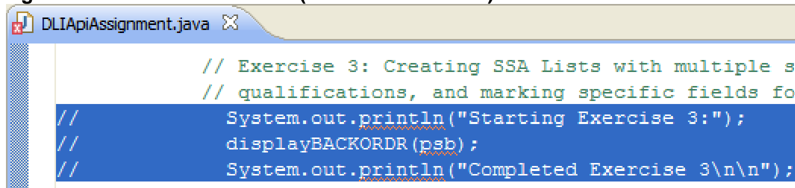
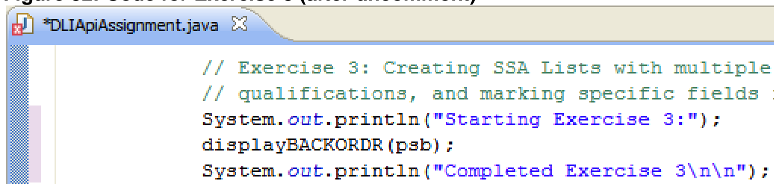


Figure 32: Code for Exercise 3 (after uncomment)



2. The function `displayBACKORDR` contains the Javacode for the retrieval. In line 119, move your mouse over the `displayBACKORDR(psb)` function invocation and press F3 to open the function declaration.

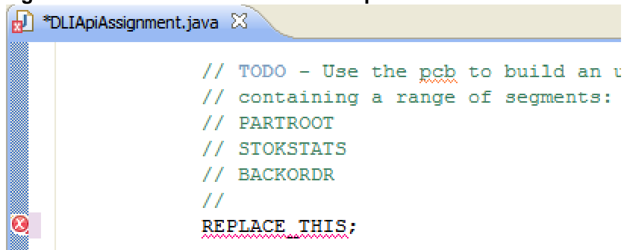
2.2.3.2 Exercise 3 – Step 2: Build an unqualified SSAList

1. In line 246 of the code, delete the constant **REPLACE_THIS** and replace it with the Java code statement to build an unqualified SSAList for a hierarchic path of segments ranging from the top-level PARTROOT segment to the bottom-level BACKORDR segment. You can find the answer after the figure below.

Hint:
Declare a new SSAList variable (**ssaList**). Use the PCB object that has been created (**partspcb1**) to call the **getSSAList(String, String)** method.

- Pass in the PARTROOT segment name (**"PARTROOT"**) as the 1st input parameter
- Pass in the BACKORDR segment name (**"BACKORDR")** as the 2nd input parameter

Figure 33: Insert code to build the unqualified SSAList



© Copyright IBM Corporation 2011, 2012. All rights reserved.

Verify your Java code statement:
In line 246, your Java code statement should look like this:

```
SSAList ssaList = partspcb1.getSSAList("PARTROOT", "BACKORDR");
```

2.2.3.3 Exercise 3 - Step 3: Mark the fields to retrieve

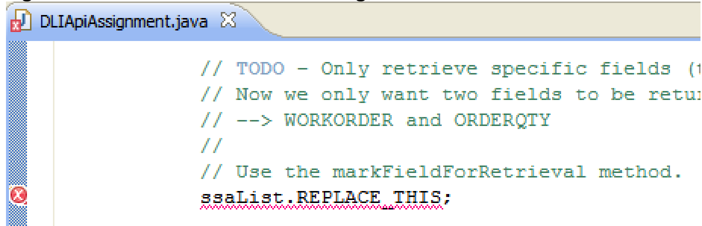
1. In line 262 of the code, delete the constant **REPLACE_THIS** and replace it with the Java code statement to mark the WORKORDER field for retrieval from the BACKORDR segment. In line 263, add the Java code statement to mark the ORDERQTY field for retrieval from the same segment. You can find the answer after the figure below.

Hint:
Use the SSAList object that has been created (**ssaList**) to call the **markFieldForRetrieval(String, String, boolean)** method.

- Pass in the segment name (**"BACKORDR"**) as the 1st input parameter, to indicate the name of the segment in the SSAList containing the field
- Pass in the field name (**"WORKORDER")** as the 2nd input parameter, to indicate the name of the field to be marked for retrieval from the database
- Pass in the boolean value true as the 3rd input parameter, to indicate that this field should be retrieved from the database

In the next line, create a similar statement to mark the ORDERQTY field for retrieval.

Figure 34: Insert code to mark the segment fields to retrieve



Verify your Java code statement:
In line 262 and 263, your Java code statements should look like this:

```
ssaList.markFieldForRetrieval("BACKORDR", "WORKORDER", true);
ssaList.markFieldForRetrieval("BACKORDR", "ORDERQTY", true);
```

2.2.3.4 Exercise 3 – Step 4: Specify the number of rows to fetch per network call

1. In line 274 of the code, delete the constant **REPLACE_THIS** and replace it with the Java code statement to set the fetch size property to 30. You can find the answer after the figure below.

Hint:
Use the PCB object that has been created (**partspcb1**) to call the **setFetchSize(int)** method.

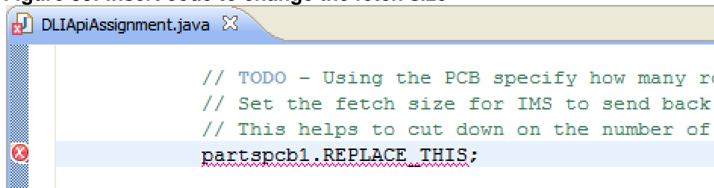
- Set the number of rows to fetch (**30**) as the 1st input parameter

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Fetch size property

The fetch size is the number of rows physically retrieved from the IMS database per network call. A list of rows is represented by a `Path` instance containing one or more segments that match the segment search argument criteria specified by an `SSAList`. This is set for you internally. You can also set the fetch size using the `setFetchSize` method from the `PCB` interface. Setting the fetch size allows a single request to return multiple rows at a time, so that each application request to retrieve the next row does not always result in a network request.

Figure 35: Insert code to change the fetch size



Verify your Java code statement:

In line 274, your Java code statement should look like this:

```
partspcb1.setFetchSize(30);
```

2.2.3.5 Exercise 3 – Step 5: Print the retrieved segment fields from the path

1. In line 287 of the code, delete the 1st instance of the constant REPLACE_THIS and change the `System.out.println` statement to print the value of the WORKORDER field returned by IMS. In the same line, delete the 2nd instance of the constant REPLACE_THIS and change the Java code statement to print the value of the ORDERQTY field returned by IMS. You can find the answer after the figure below.

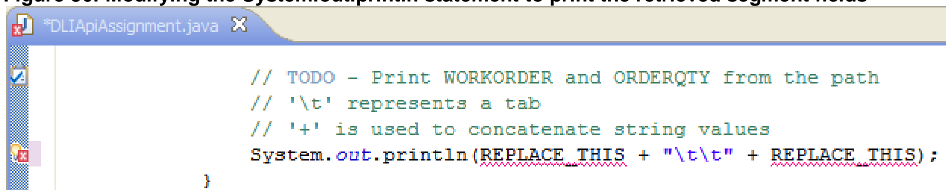
Hint:

Use the Path object that has been created (**path**) to call the **getString(string)** method.

- Set the 1st parameter to the field name (**"WORKORDER")**, to retrieve the value of this field.

Use a similar method call to retrieve the value of the ORDERQTY field.

Figure 36: Modifying the System.out.println statement to print the retrieved segment fields



Verify your Java code statement:

In line 287, your Java code statement should look like this:

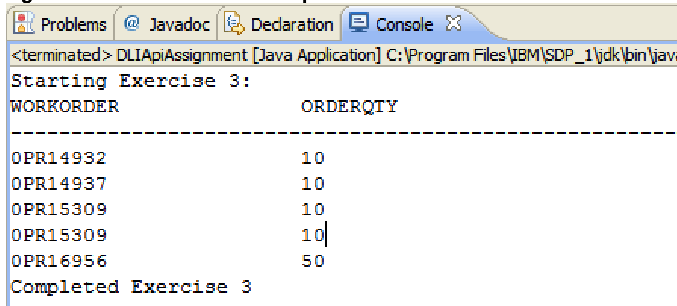
```
System.out.println(path.getString("WORKORDER") + "\t\t" +
path.getString("ORDERQTY"));
```


© Copyright IBM Corporation 2011, 2012. All rights reserved.

2.2.3.6 Exercise 3 – Step 6: Run the application and verify the output results

1. Press `Ctrl + S` to save your changes to the files.
2. Right click on the Java editor and select **Run As** > **Java Application**, as shown in section 2.2.1.4.
3. An **Errors in Workspace** dialog box will appear. Click on **Proceed**.
4. In the **Console view**, verify that the output results look like the screenshots below.

Figure 37: Exercise 3 result output in the Console view



2.2.4 Exercise 4: Utilize command codes for DL/I

In this exercise, you will add a command code in the SSAList to retrieve a sequence of segments.

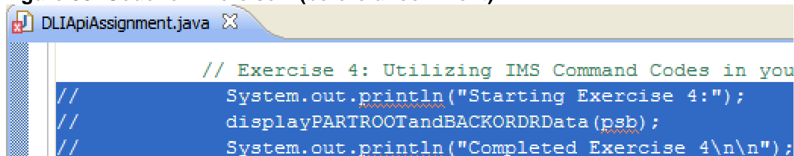
> **Command codes for DL/I**
> SSAs can also include one or more command codes, which can change and extend the functions of DL/I calls. For example, you can use the D command code to retrieve or insert a sequence of segments in a hierarchic path with one call rather than retrieving or inserting each segment with a separate call. A call that uses the D command code is called a path call.

Exercise 4 begins on line 122 of the **DLIApiAssignment.java** sample application. At the beginning of Exercise 4, the code for this exercise has been commented out.

2.2.4.1 Exercise 4 – Step 1: Uncomment the code for Exercise 4

1. In the **Java editor**, highlight lines 123 to 125 of the **DLIApiAssignment.java** sample application and press `Ctrl + /` to uncomment the code.

Figure 38: Code for Exercise 4 (before uncomment)



© Copyright IBM Corporation 2011, 2012. All rights reserved.

Figure 39: Code for Exercise 4 (after uncomment)

```
*DLIApiAssignment.java
        // Exercise 4: Utilizing IMS Command Codes in y
        System.out.println("Starting Exercise 4:");
        displayPARTROOTandBACKORDRData(psb);
        System.out.println("Completed Exercise 4\n\n");
```

2. The function `displayPARTROOTandBACKORDER` contains the code for the batch retrieval operation. In line 124, move your mouse over the `displayPARTROOTandBACKORDRData(psb)` function invocation and press F3 to open the function declaration.

2.2.4.2 Exercise 4 – Step 2: Add a command code to the SSAList

1. In line 333 of the code, delete the constant **REPLACE_THIS** and replace it with the Java code statement to add the D command code. You can find the answer after the figure below.

Hint:
Use the SSAList object that has been created (**ssaList**) to call the addCommandCode (String, byte) method.

- Set the name of the segment (**"PARTROOT")** as the 1st input parameter
- Set the command code (**SSAList.CC_D)** as the 2nd input parameter

Figure 40: Insert the code to add an IMS command code

```
DLIApiAssignment.java
        // TODO - Add an IMS command code to the SS
        // We want to retrieve data from both PARTR
        // order to do this in IMS you need to add
        // to the PARTROOT because the way the SSAL
        // you will only get data from the leaf seg
        //
        // After adding the command code D the SSA 
        // PARTROOT *D(PARTKEY = 027618032P101 )
        // STOKSTATS
        // BACKORDR
        //
        // Use the addCommandCode method.
        REPLACE_THIS
```

Verify your Java code statement:
In line 333, your Java code statement should look like this:

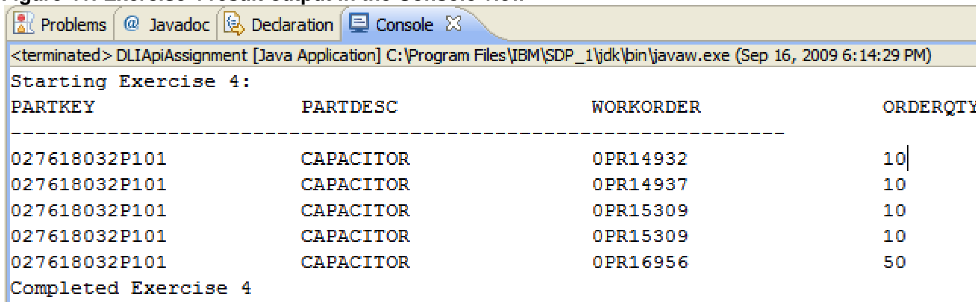
```
ssaList.addCommandCode("PARTROOT", SSAList.CC_D);
```


© Copyright IBM Corporation 2011, 2012. All rights reserved.

2.2.4.3 Exercise 4 – Step 3: Run the application and verify the output results

1. Press `Ctrl + S` to save your changes to the files.
2. Right click on the Java editor and select **Run As** > **Java Application**, as shown in section 2.2.1.4.
3. In the **Console view**, verify that the output results look like the screenshots below.

Figure 41: Exercise 4 result output in the Console view



© Copyright IBM Corporation 2011, 2012. All rights reserved.

3 Task 3 - Access IMS data with the IMS Universal JDBC driver

In this task, you will write a Java application to connect to an IMS database and manipulate data using structured query language (SQL) with the IMS Universal JDBC driver.

What is JDBC?
Java Database Connectivity (JDBC) is an application programming interface (API) that Java applications use to access relational databases or tabular data sources. The JDBC API is the industry standard for database-independent connectivity between the Java programming language and any database that has implemented the JDBC interface. The client uses the interface to query and update data in a database.

IMS support for JDBC lets you write Java applications that can issue dynamic SQL calls to access IMS data and process the result set that is returned in tabular format. The IMS Universal JDBC driver is designed to support a subset of the SQL syntax with functionality that is limited to what the IMS database management system can process natively. Its DBMS-centric design allows the IMS Universal JDBC driver to fully leverage the high performance capabilities of IMS. The IMS Universal JDBC driver also provides aggregate function support, and ORDER BY and GROUP BY support.

Basic programming model for a Java application using the IMS Universal JDBC driver
The IMS Universal JDBC driver supports the standard programming model for using JDBC drivers. For more information about the JDBC programming model, see the JDBC Basics tutorial by SUN.

3.1 Connect to the IMS database through the IMS Universal JDBC driver

Before you can execute SQL calls from your IMS Universal JDBC driver application, you must connect to an IMS database.

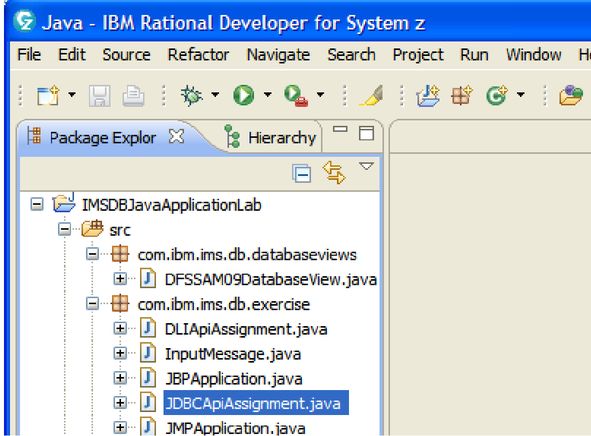
3.1.1 Open the JDBCApiAssignment.java sample application

The JDBCApiAssignment.java sample application
This sample application contains skeleton Java code for connecting to the IMS database and issuing SQL data access calls using the IMS Universal JDBC driver.

1. In the Package Explorer view, expand **IMSDBJavaApplicationLab** > **src** > **com.ibm.ims.db.exercise**

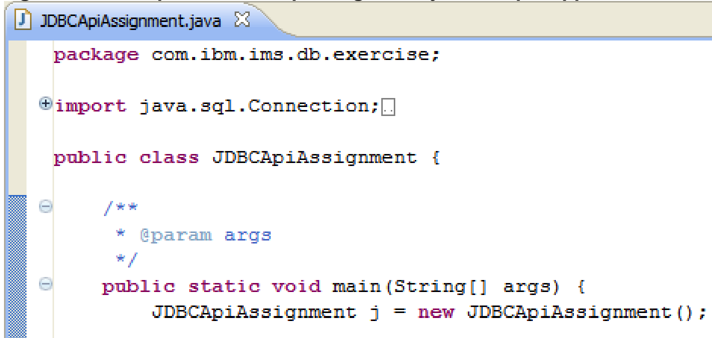
© Copyright IBM Corporation 2011, 2012. All rights reserved.

Figure 42: Navigating to the JDBCApiAssignment.java sample application



2. From the Package Explorer view, double click on **JDBCApiAssignment.java** to open the sample application in the Java editor.

Figure 43: The opened JDBCApiAssignment.java sample application in the Java editor



3.1.2 Set the connection properties

1. In line 22 of the code, verify that this application will connect using type-4 connectivity.
2. In line 35, verify that **7001** is set as the IMS Connect DRDA port number.
3. In line 38, verify that **"class://com.ibm.ims.db.databaseviews.DFSSAM09DatabaseView"** is set as the metadataURL.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

3.2 Issue SQL calls to access the IMS database

The following exercises in this section will show you how to issue SQL calls in your Java application to retrieve data from the IMS database using the IMS Universal JDBC driver.

How do IMS database elements map to relational database elements?

The IMS Universal JDBC driver performs the necessary translation between IMS and relational database elements. The table below summarizes the database element mappings.

Hierarchical database elements in IMS	Equivalent relational database elements
Segment name	Table name
Segment instance	Table row
Segment field name	Column name
Segment unique key	Table primary key
Virtual foreign key field	Table foreign key

3.2.1 Exercise 1 - Retrieve all fields of a segment

In this exercise, you will retrieve all the fields of a segment by issuing a SELECT statement using the IMS Universal JDBC driver.

Using the SELECT keyword

Use the SELECT statement to retrieve data from one or more tables. The result is returned in a tabular result set. The syntax for a simple SELECT query is:

SELECT *column_name(s)* **FROM** *table_name*

An asterisk * can be used in place of *column_name* to represent all columns of that table. Because IMS is a hierarchical database, *column_name* maps to *field_name* and *table_name* maps to *segment_name*.

When using the SELECT statement with the IMS Universal JDBC driver:

- If you are selecting from multiple tables and the same column name exists in one or more of these tables, you must table-qualify the column or an ambiguity error will occur.
- The FROM clause must list all the tables you are selecting data from. The tables listed in the FROM clause must be in the same hierarchic path in the IMS database.
- In Java applications using the IMS JDBC drivers, connections are made to PSBs. Because there are multiple database PCBs in a PSB, queries must specify which PCB in a PSB to use. To specify which PCB to use, always qualify segments that are referenced in the FROM clause of an SQL statement by prefixing the segment name with the PCB name. You can omit the PCB name only if the PSB contains only one PCB.

1. Move your cursor to line 20 of the sample application. You will modify this statement to issue different SQL queries in this task.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Figure 44: Modify the query string to issue different SQL queries in the sample application

```
JDBCApiAssignment.java

        //TODO Add a SQL statement between the quotes
        String query = "";

        String result = j.sqlMethod(query, IMSConnectio
```

2. In line 20, construct a SQL query to retrieve all of the fields of the **PARTSPCB1.PARTROOT** segment. Note that the segment name must start with the PCB qualifier.
    - Set the query string to **"SELECT * FROM PARTSPCB1.PARTROOT"**
3. Press Ctrl + S to save your changes to the files.
4. Right click on the Java editor and select **Run As** > **Java Application**, as shown in section 2.2.1.4.
5. In the Console view, verify that the result output looks like the screenshots below.

Comment [U1]:

Figure 45: Beginning of Task 3 - Exercise 1 result output

```
Problems  @ Javadoc  Declaration  Annotations  Console
<terminated> JDBCApiAssignment [Java Application] C:\Program Files\IBM\SDP\jdk\bin\javaw.exe
PARTKEY                 PART                    PARTDESC
02AN960C10              AN960C10        WASHER
02CK05CW181K            CK05CW181K      CAPACITOR
02CSR13G104KL           CSR13G104KL     KR1J50KS
02JAN1N976B             JAN1N976B       DIODE CODE-A
02MS16995-28            MS16995-28      SCREW
02N51P3003F000          N51P3003F000    SCREW
02RC07GF273J            RC07GF273J      RESISTOR
02106B1293P009          106B1293P009    RESISTOR
02250236-001            250236-001      CAPACITOR
02250239                250239          TRANSISTOR
02250241-001            250241-001      CONNECTOR
```

Figure 46: End of Task 3 - Exercise 1 result output

```
<terminated> JDBCApiAssignment [Java Application] C:\Program Files\IBM\SDP\jdk\bin\
02930331-123            930331-123      FILTER
02930333-001            930333-001      DISCRIMINATO
02946325-086            946325-086      PIN
02950060-006            950060-006      RELAY
02954017-001            954017-001      RESISTOR
02958007-180            958007-180      RESISTOR
02960528-067            960528-067      RESISTOR
02968534-001            968534-001      SOCKET
02974810-010            974810-010      THERMOSTAT
02975105-001            975105-001      TRANSFORMER
02989036-001            989036-001      TRANSFORMER
```


© Copyright IBM Corporation 2011, 2012. All rights reserved.

3.2.2 Exercise 2 - Retrieve fields of a segment based on a conditional statement

In this exercise, you will retrieve specific fields of a segment based on a conditional statement by issuing a SELECT statement with a WHERE clause using the IMS Universal JDBC driver.

> **Using the WHERE keyword**
>
> Use the **WHERE** keyword in SQL to select data conditionally. The syntax for a conditional select query is:
>
> **SELECT** *column_name(s)* **FROM** *table_name* **WHERE** *column_name operator value*
>
> Note that for text values, the *value* must be enclosed in quotes. Operators on text values perform binary comparisons. The IMS Universal JDBC driver converts the WHERE clause in an SQL query to a segment search argument (SSA) list when querying a database. SSA rules restrict the type of conditions you can specify in the WHERE clause.

1. In line 20, construct a SQL query that will display all fields of the **PARTSPCB1.PARTROOT** segment where the **PARTKEY** field is greater than a '025'. Note that the PARTKEY field contains data that is alphanumeric.
   - Set the query string to **"SELECT * FROM PARTSPCB1.PARTROOT WHERE PARTKEY > '025'"**
2. Press Ctrl + S to save your changes to the files.
3. Right click on the Java editor and select **Run As** > **Java Application**, as shown in section 2.2.1.4.
4. In the Console view, verify that the beginning of the result output looks like the screenshot below.

Figure 47: Beginning of Task 3 - Exercise 2 result output

Problems | Javadoc | Declaration | Annotations | Console

<terminated> JDBCApiAssignment [Java Application] C:\Program Files\IBM\SDP\jdk\bin\javaw.exe (

PARTKEY	PART	PARTDESC
0256134-016	56134-016	NAS671C1 NUT
0260003-118	60003-118	7734304P8661T0 RES
02652540-002	652540-002	WIRE WRAP
02652799	652799	PULSE TRANSFORMER
0268663-102	68663-102	CM05C100K03
0268663-104	68663-104	CM05D200J03
0269857-635	69857-635	CP09A1KE153K3 CAPAC
027060654P001	7060654P001	ELE TUBE
027438995P002	7438995P002	NUT
027454949P001	7454949P001	LAMP HOLDER
027618032P101	7618032P101	CAPACITOR

© Copyright IBM Corporation 2011, 2012. All rights reserved.

3.2.3 Exercise 3 - Order SQL query output by field values

In this exercise, you will retrieve data from a segment in sorted order issuing a SELECT statement with an ORDER BY clause using the IMS Universal JDBC driver.

> **Using the ORDER BY keyword**
>
> Use the **ORDER BY** clause in SQL to sort the results of a SQL query in ascending or descending order. The syntax for a ordered select query is:
>
> **SELECT** *column_name(s)* **FROM** *table_name* **ORDER BY** *column_name* **ASC|DESC**
>
> Note that **ASC** is used for ascending order and **DESC** is used for descending order. The field names that are specified in an ORDER BY clause must match exactly the field name that is specified in the SELECT statement.

1. In line 20, construct a SQL query that will retrieve all fields of the **PARTSPCB1.PARTROOT** segment and sort the results by the **PART** field in descending order.
   - Set the query string to **"SELECT * from PARTSPCB1.PARTROOT ORDER BY PART DESC"**
2. Press Ctrl + S to save your changes to the files.
3. Right click on the Java editor and select **Run As** > **Java Application**, as shown in section 2.2.1.4.
4. In the Console view, verify that the result output looks like the screenshots below.

Figure 48: Beginning of the Task 3 - Exercise 3 result output

Problems | Javadoc | Declaration | Annotations | Console

<terminated> JDBCApiAssignment [Java Application] C:\Program Files\IBM\SDP\jdk\bin\javaw.

PARTKEY	PART	PARTDESC
02RC07GF273J	RC07GF273J	RESISTOR
02N51P3003F000	N51P3003F000	SCREW
02MS16995-28	MS16995-28	SCREW
02JAN1N976B	JAN1N976B	DIODE CODE-A
02CSR13G104KL	CSR13G104KL	KR1J50KS
02CK05CW181K	CK05CW181K	CAPACITOR
02AN960C10	AN960C10	WASHER
02989036-001	989036-001	TRANSFORMER
02975105-001	975105-001	TRANSFORMER
02974810-010	974810-010	THERMOSTAT
02968534-001	968534-001	SOCKET

Figure 49: End of the Task 3 - Exercise 3 result output

02252252-003	252252-003	COUPLING
02250891	250891	SERVO VALVE
02250796	250796	SWITCH
02250794	250794	RESISTOR
02250241-001	250241-001	CONNECTOR
02250239	250239	TRANSISTOR
02250236-001	250236-001	CAPACITOR
02106B1293P009	106B1293P009	RESISTOR

© Copyright IBM Corporation 2011, 2012. All rights reserved.

3.2.4 Exercise 4 - Retrieve a specific field of a segment

In this exercise, you will retrieve a specific field in a segment. By querying only specific fields instead of selecting all the fields in a particular segment, you can reduce network overhead when using the IMS Universal JDBC driver.

1. In line 20, construct a SQL query that will display only the **PART** field from the **PARTSPCB1.PARTROOT** segment.
   - Set the query string to **"SELECT PART FROM PARTSPCB1.PARTROOT"**
2. Press Ctrl + S to save your changes to the files.
3. Right click on the Java editor and select **Run As** > **Java Application**, as shown in section 2.2.1.4.
4. In the Console view, verify that the result output looks like the screenshots below.

Figure 50: Beginning of Task 3 - Exercise 4 result output

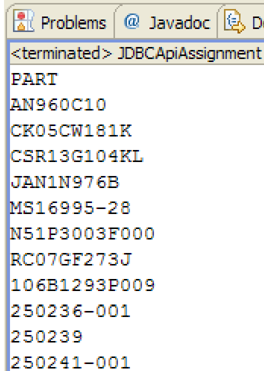
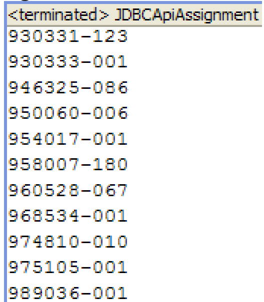


Figure 51: End of Task 3 - Exercise 4 result output



3.2.5 Exercise 5 – Retrieve multiple fields from multiple segments

In this exercise, you will issue a SELECT query to retrieve data from segments that are on the same and on different hierarchical paths in the IMS database. Note that the PARTROOT and the BACKORDR segments are on the same hierarchic path, while the CYCCOUNT segment is on a separate hierarchic path.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Retrieving fields from multiple segments

In SQL queries to relational databases, the **JOIN** keyword is typically used to query data from multiple tables based on a relationship between the tables. IMS does not support using the **JOIN** keyword explicitly, because IMS is a hierarchical database and it is possible that two segments are unrelated to each other. However, an implicit join will be performed if the segments fall within the same hierarchical path. The syntax for this is the same as for a SELECT query. Note that multiple column names and table names can be specified as long as a comma is used to separate them. IMS allows issuing a SELECT call to retrieve data from segments that are not on the same hierarchical path, if a logical relationship has been defined between them.

1. In line 20, construct a SQL query that will display the **PART** field from the **PARTSPCB1.PARTROOT** segment and the **BACKKEY** field from the **PARTSPCB1.BACKORDR** segment.
   - Set the query string to **"SELECT PART, BACKKEY FROM PARTSPCB1.PARTROOT, PARTSPCB1.BACKORDR"**
2. Press Ctrl + S to save your changes to the files.
3. Right click on the Java editor and select **Run As** > **Java Application**, as shown in section 2.2.1.4.
4. In the Console view, verify that the result output looks like the screenshot below.

Figure 52: Result output of Task 3 - Exercise 5 part 1

```
Problems  @ Javadoc  Declaration  Ar
<terminated> JDBCApiAssignment [Java Application]
PART                    BACKKEY
JAN1N976B        30PR237942
250236-001       30PR265943
250236-001       30PR347921
250236-001       30PR426134
3003806          30SO536609
3003806          30SO536610
7618032P101      30PR149329
7618032P101      30PR149376
7618032P101      30PR153096
7618032P101      30PR153098
7618032P101      30PR169566
7736847P001      30PR135640
925363-136       30PR729437
```

5. In line 20, construct a SQL query that will display the **PHYSICALCOUNT** field from the **PARTPCB1.CYCCOUNT** segment and the **WORKORDER** field form the **PARTSPCB1.BACKORDR** segment.
   - Set the query string to **"SELECT PHYSICALCOUNT, WORKORDER FROM PARTSPCB1.CYCCOUNT, PARTSPCB1.BACKORDR"**
6. Press Ctrl + S to save your changes to the files.
7. Right click on the Java editor and select **Run As** > **Java Application**, as shown in section 2.2.1.4.
8. In the Console view, verify that the query fails with this error message: **"The tables BACKORDR and CYCCOUNT specified in the query cannot be joined together. They are not along the same hierarchic path in the database"**.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

Resources

Learn

- View the Information Management Software for z/OS Solutions Information Center for the latest information and educational resources available for Information Management System (IMS), including the Java API reference for the IMS Universal drivers.

- Visit the Rational software area on developerWorks for technical resources and best practices for Rational Software Delivery Platform products.

- Explore Rational computer-based, Web-based, and instructor-led online courses. Hone your skills and learn more about Rational tools with these courses, which range from introductory to advanced. The courses on this catalog are available for purchase through computer-based training or Web-based training. Additionally, some "Getting Started" courses are available free of charge.

- Subscribe to the Rational Edge e-zine for articles on the concepts behind effective software development.

- Check out the Information Management IMS zone on developerWorks®.

- Subscribe to the IBM developerWorks newsletter, a weekly update on the best of developerWorks tutorials, articles, downloads, community activities, webcasts and events.

- Browse the technology bookstore for books on these and other technical topics.

- Use Java in IMS dependent region: http://www.ibm.com/developerworks/data/library/techarticle/dm-1011javaimsregions/index.html?ca=drs-

- IMS Universal Drivers programming guide: http://pic.dhe.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.ims12.doc.apr/ims_javaprogrammingreference.htm

- Java programming reference: http://pic.dhe.ibm.com/infocenter/dzichelp/v2r2/topic/com.ibm.ims12.doc.apr/ims_javaprogrammingreference.htm

- Find code sample for cast-to-DLI approach for JDBC here: https://www.ibm.com/developerworks/community/blogs/8a337b1c-3c0c-48a5-b7cc-7f805884dbb9/entry/new_example_available_cast_to_dl_i_from_the_ims_universal_jdbc_driver?lang=en

Get products and technologies

- Download trial versions of IBM Rational software.

- Download a trial version of IBM Rational Developer for system z.

Discuss

- Participate in the discussion forum.

- Check out developerWorks blogs and get involved in the developerWorks community.

© Copyright IBM Corporation 2011, 2012. All rights reserved.

About the authors

Poonam Chitale is a Software Engineer for IMS Open Database solution

Joshua Newell is a Software Engineer for IMS Open Database solution and level 2 support

Trademark notice

IBM, the IBM logo and ibm.com are trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at www.ibm.com/legal/copytrade.shtml.

© Copyright IBM Corporation 2011, 2012. All rights reserved.