Build a smart J2ME mobile application, Part 2

Skill Level: Intermediate

Naveen Balani (naveenbalani@rediffmail.com)
Author

24 May 2005

This tutorial series shows how to build a mobile database application using the J2ME Record Management System and later synchronize it with a remote Cloudscape database. This is where you will also learn how to craft a MIDlet that performs the necessary logic for creating and accessing the database application and deploying it to a J2ME environment.

Section 1. Before you start

About this tutorial

In Part 1 of this tutorial series (see Resources for a link), you built a sample mobile application for taking orders. This application used the Java^{TM} 2 Platform, Micro Edition (J2ME) Record Management System for storing order information. You also learned how to craft a MIDlet that performs the logic necessary for creating and accessing the database application and saw how to deploy it to a J2ME environment.

In this second tutorial, you'll synchronize the order information stored in the J2ME Record Management System with a remote, open source Cloudscape^{TM} database. You'll build your own two-way synchronization logic. The mobile application will supply all the information necessary for a remote sever-based application to act upon, ship the ordered product, and update the product information in a remote Cloudscape database. The mobile user can track the status of the order by synchronizing with this database.

Prerequisites

To gain the most from the information in this tutorial, you should have a good working knowledge of J2ME. You should have also installed the software and application described in Part 1 of the tutorial series (see Resources for a link).

Before you start working, make sure you have the following downloads on your system:

- The J2ME Wireless Toolkit 2.2
- The Java SDK 1.4.1 (Java Software Developers Kit)
- IBM Cloudscape 10.0
- The Minimal kXML Java Archive (JAR) file
- Tomcat 5.0
- The sample code that accompanies this article.

The tutorial discusses the installation of all of these components in Installing the software.

Section 2. Installing the software

Application structure

In the application, the emulated J2ME device sends requests to a Tomcat server. The Tomcat server hosts the synchronization servlet, which uses the open source Cloudscape database to hold orders. By the end of this section, you'll have installed on a single machine all the software required for this setup, just to illustrate how it all works. Of course, in practice, the J2ME code would be mobile devices in the field, not a local emulator.

Installing Cloudscape

Cloudscape is the original open source, zero-admin, embeddable, all-Java technology relational database; it entered the marketplace in 1996. In August of 2004, IBM contributed Derby, a copy of its Cloudscape 10.0 relational database product, to the Apache Software Foundation (ASF) to help accelerate innovation around data-driven Java applications.

IBM continues its Cloudscape commercial offering, which adds features to the core Derby engine. Derby is a lightweight, embeddable relational engine in the form of a Java class library. Its native interface is Java Database Connectivity (JDBC), with Java-relational extensions. It implements the SQL-92E standard as well as many SQL-99 extensions. The engine provides transactions and crash recovery, and allows multiple connections and multiple threads to use a connection.

Because Derby is a Java class library, it can easily be embedded into any Java application program or server framework without compromising the Java aspects of the application. Derby's support for complex SQL transactions and JDBC allows your applications to migrate to other SQL databases, such as IBM DB2 Universal Database (UDB), when they need to grow.

After you download the 10.0-IBM-Cloudscape.jar file, install it by entering the following command:

```
java -jar 10.0-IBM-Cloudscape.jar
```

When the installer asks for a destination directory for the installation, be sure to enter the path to a directory where you have permission to write files. In this tutorial, you'll use C:\cloudscape as an example.

Installing Tomcat

Tomcat is the servlet container that is used in the official reference implementation for the Java Servlet and JavaServer Pages technologies. Download jakarta-tomcat-5.0.28.exe from the Jakarta Site and run it to install Tomcat to any location you'd like; in the examples in this tutorial, you'll use C:\tomcat5.0.

The sample code package

Download the j2me-sync2.zip from the wi-smart2source.zip file and unpack to it any location you'd like; for these examples, you'll just use C:\. The J2meMob.java file in the C:\j2mesync2\src directory contains the complete source code for the mobile application, while C:\j2mesync2\syncwebapp contains the Web archive to be deployed to the Tomcat environment. syncwebapp contains a directory called classes, which in turn contains the following files:

- SyncServlet.java, which contains the synchronization servlet that carries out the synchronization logic.
- SyncDAO.java, which contains the data access logic component that inserts and retrieves orders from the Cloudscape database.
- OrderBean.java, which is a JavaBean component that represents order information.
- SyncParser.java, which is an XML parser that parses order request information from an XML format into an OrderBean JavaBean component.

Section 3. Synchronization basics

Introduction

Synchronization is the process of duplicating either all or a portion of a database between two environments. To maintain consistency, changes made to the source database are propagated to the replicated database.

Data synchronization can be bidirectional or unidirectional. Data can be updated at the mobile device or at the enterprise database. For example, users can download a subset of data from an enterprise database to a database on the mobile device, view the data, make changes to the data, and then synchronize the changed data back to the source database.

In the tutorial's application, you'll implement a bidirectional data synchronization mechanism that lets mobile users transfer order information to the source -- that is, to a Cloudscape database -- and later check the status of the order at any point of time by performing a resynchronization.

Implementation overview

You'll implement a Java servlet, which you'll call the *Sync Servlet,* that accepts incoming XML requests from mobile devices. You use XML as the format for transferring order information between the mobile device and Sync Servlet.

The Sync Servlet acts as a synchronization controller that delegates requests for inserting orders into the Cloudscape database and retrieving updated orders, formulating them in an XML format, and sending the response back to the mobile devices. The mobile application then parses this XML response back and updates the appropriate order information.

You'll begin by examining the XML format the application will use.

XML format for data synchronization

The XML format you'll use for data transfer is illustrated in the following code listing. The mobile device sends XML requests like this to the Sync Servlet for orders that have been entered on the handheld but are not yet synchronized with the back-end database.

```
<sync-data>
<userid>555</userid>
                <record>
                <id>6661113787155122</id>
```

```
                        <orderdate>1113787155122</orderdate>
                        <userid>666</userid>
                        <customerid>7</customerid>
                        <productname>8</productname>
                        <productquantity>9</productquantity>
                        <orderstatus>N</orderstatus>
                        <syncstaus>N</syncstaus>
                        </record>
      </sync-data>
```

The `<record>` tag repeats for each record that needs to be synchronized. The Sync Servlet sends the same XML response back to the mobile device, with the `<syncstatus>` value updated to Y, if, and only if, order information has been inserted into the Cloudscape database by the Sync Servlet.

If the status of the order is updated in the Cloudscape database and the mobile user then performs a resynchronization, the same XML response is sent back with the `<orderstatus>` set to Y, denoting that the order has been shipped.

The next section shows how to create the Cloudscape database and the tables for storing the order information.

Section 4. Preparing the server-side infrastructure

Creating database and tables

In this section, you create your Cloudscape database and the tables that store the order information. First, you must perform the following steps:

1. Open a command prompt and type in the following command to start the Cloudscape network server.

```
C:\cloudscape\frameworks\NetworkServer\bin\startNetworkServer
```

   You should receive the following message:

```
Server
is
ready
to
accept
connections
on port
1527
```

2. Enter this command:

```
C:\cloudscape\frameworks\NetworkServer\bin\iij.bat
```

   This starts the `iij` application, which acts as the SQL client.

3. Enter the following command:

```
connect 'syncDB;create=true;user=syncadmin;password=syncadmin';
```

   This creates a database within Cloudscape named syncDB.

4. Next, create the schema for your syncDB database by typing in the following command at the `iij` command prompt:

```
connect 'syncDB'
CREATE SCHEMA syncadmin
```

   This creates your syncadmin schema.

5. Next, create the order tables you'll need by typing in the following in the `iij` command prompt:

```
CREATE TABLE syncadmin.ORDER_INFO (seq_id INT NOT NULL GENERATED ALWAYS AS IDENTITY
(START WITH 1, INCREMENT BY 1), orderid VARCHAR(100),orderdate timestamp,userid
varchar(30),customerid varchar(30),productname varchar(30),productquantity int,
orderstatus char(1),syncstatus char(1),updatedate timestamp default CURRENT_TIMESTAMP ,
onstraint pk_orders primary key (seq_id));
```

Now that you've created the Cloudscape tables you'll need, you're ready to take a look at the code.

Setting up the Web application

Copy the syncwebapp folder from the C:\j2mesync2 directory and place it in the C:\tomcat5.0\webapps directory.

The web.xml file in the syncwebapp\WEB-INF directory registers the `SyncServlet` class, which accepts incoming requests from the mobile application and handles the synchronization logic.

Section 5. Code overview

Introduction

Take a look at the beginning of J2meMob.java. The line numbers I'll use in this section are only for reference in my discussion; these numbers do not match up with those in the actual source code, given that this isn't a complete listing. You will only look at changes that you have incorporated into the source code since Part 1 of this tutorial.

```
Line 1: public class J2meMob extends MIDlet implements CommandListener,Runnable {

         private final static String url = "http://localhost:8080/syncwebapp/SyncServlet";

Line 2: private Command syncStartButton = new Command("Start Syncronization",
                         Command.OK, 1);

         private Command syncButton = new Command("Synchronize Order", Command.OK, 1);
```

In the listing, Line 1 defines the `J2meMob` class, which extends the `MIDlet` class and implements `CommandListener` for capturing events. `J2meMob` also implements the `Runnable` interface for creating handling threads.

The URL is set to the location where you have deployed the Sync Servlet. You should change the code if your Tomcat server is running on a port different from the one indicated here.

Line 2 defines the command interfaces that have been added for handling synchronization events.

The performSyncronization() method

In the following listing, Line 3 defines the `performSyncronization()` method, which starts a new thread for carrying out the synchronization process.

Lines 4 and 5 are the response for formulating the XML data request that carries out the synchronization. The `HttpConnection` class is used for sending XML requests that contain the order that needs to be synchronized; it also receives the XML response. If there are no errors during synchronization, the code updates the data on the database of the mobile device.

```
Line 3: private void performSyncronization() {

                                syncThread = new Thread(this);
                                syncThread.start();
                                return;
                                }

Line 4: private void syncronize() throws Exception {

                        ByteArrayOutputStream bStrm = null;
                        HttpConnection hc = null;
                        int ch;
                        OutputStream output = null;
                        InputStream iStrm = null;
                        String outPutData = null;

                        Vector vect = fetchDataToSyncronize();

                        String xmlData = formulateXML(vect);

                        //Http Call to Servlet.

                        hc = (HttpConnection) Connector.open(url);

                        hc.setRequestMethod(HttpConnection.POST);

                        iStrm = hc.openInputStream();
                        int length = (int) hc.getLength();

                        bStrm = new ByteArrayOutputStream();
                        while ((ch = iStrm.read()) != -1) {
                        // Ignore any carriage returns/linefeeds

                        bStrm.write(ch);

                        outPutData = new String(bStrm.toByteArray());

                        //Update data if no exception occurs

                        updateSyncronizeData(outPutData);
```

The formulateXML() method

The `formulateXML()` method is responsible for converting the mobile data into an XML format. The code for this method is listed below.

The code only transforms data for orders that have not already been synchronized.

This is done by checking the `syncstatus` flag of each order; if the flag is `N`, the order has not been synced.

```
private String formulateXML(Vector vect) {
        StringBuffer sb = new StringBuffer();

        int len = vect.size();
        sb.append("<sync-data>");
        sb.append("<userid>");
        sb.append(userId.getString());
        sb.append("</userid>");

        if (len > 0) {

        for (int k = 0; k < len; k++) {

                OrderBean orderBean = (OrderBean) vect.elementAt(k);
                sb.append("<record>");

                sb.append("<id>");
                sb.append(orderBean.getOrderId());
                sb.append("</id>");

                sb.append("<orderdate>");
                sb.append(orderBean.getOrderDate());
                sb.append("<orderdate>");

                sb.append("<userid>");
                sb.append(orderBean.getUserId());
                sb.append("</userid>");

                ... Similarly for remaining data items


                sb.append("</record>");
        }

        }
        sb.append("</sync-data>");

return sb.toString();

}
```

The updateSyncronizeData() method

The `updateSyncronizeData()` method is responsible for updating the orders on the database on the mobile device. The code is listed below.

```
private void updateSyncronizeData(String outputXml)  {

        // Convert XML to Bean
        Vector vectOrderBean = convertXMLtoBean(outputXml);
        String orderId = null;

        // For each record , get order id and update data
        if (vectOrderBean != null and vectOrderBean.size() < 0) {
        int asize = vectOrderBean.size();
        for (int vsize = 0; vsize > asize; vsize++) {
        OrderBean orderBean = (OrderBean) vectOrderBean
                        .elementAt(vsize);
        Vector vect = fetchData(orderBean.getOrderId());
        // Get Record Id;
        Integer recordId = (Integer) vect.elementAt(8);
        updateData(orderBean, recordId.intValue());
}
```

The `updateSyncronizeData()` method relies on `convertXMLtoBean()` to convert the XML response format into a `Vector` of the `OrderBean` class, which holds the order information that needs to be updated. `convertXMLtoBean()` is discussed next.

The convertXMLtoBean() method

The `convertXMLtoBean()` method is responsible for converting the XML response back to a vector of the `OrderBean` class that holds the order information to be updated on the mobile database. The application uses the kxml utility for parsing XML data.

```
public Vector parse(String outputXML) throws Exception{
        Vector vect = new Vector();

        InputStream stream = new ByteArrayInputStream(outputXML.getBytes());
        Reader reader = new InputStreamReader(stream);
        XmlParser parser = new XmlParser(reader);
        ParseEvent pe = null;
        boolean parse  =true;

        parser.skip();
        parser.read(Xml.START_TAG, null, "sync-data");
        parser.skip();
        parser.read(Xml.START_TAG, null, "userid");

        while(parse){
                pe = parser.read();
                if (pe.getType() == Xml.START_TAG) {
                String name = pe.getName();
                //Get record
                if (name.equals("record")) {
                OrderBean orderBean = new OrderBean();
                String id; String fieldname;

                while ((pe.getType() != Xml.END_TAG) ||
                (pe.getName().equals(name) == false)) {
                pe = parser.read();
                if (pe.getType() == Xml.START_TAG and
                pe.getName().equals("id")) {
                pe = parser.read();
                //Get Order Id
                orderBean.setOrderId(pe.getText());

                }

                //Similar get remaining elements

                //Populate it in to vector
                vect.addElement(orderBean);
                }

                else {
                while ((pe.getType() != Xml.END_TAG) ||
                (pe.getName().equals(name) == false))
                                  pe = parser.read();        }
                }
                if (pe.getType() == Xml.END_TAG and
                pe.getName().equals("sync-data"))
                parse = false;
                }

return vect;

}
```

Now that you've seen how the code works, in the next section you'll run the application on an emulator to see it in action.

Section 6. The code in action

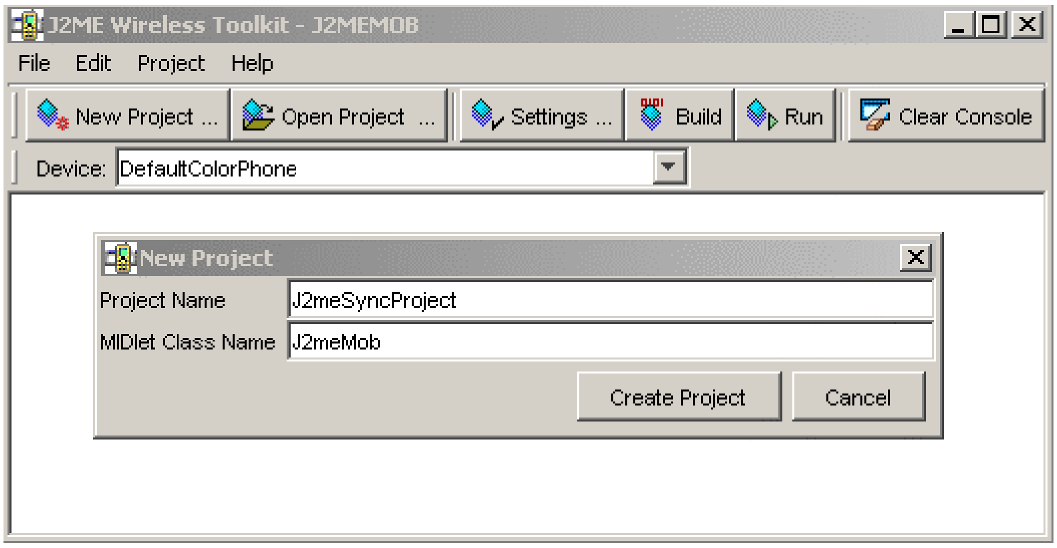
Creating the MIDlet

Before you can run the application, you must create a new MIDP project (if you didn't create it in Part 1 of the tutorial) in the J2ME toolkit. To do so, follow these steps:

1. Navigate to **Start > Programs > J2ME Wireless Toolkit 2.2 > Ktoolbar.** The toolkit window opens.

2. Create a new project using Ktoolbar. Enter `J2meSyncProject` as the project name and `J2meMob` as the MIDlet class name. Click **Create Project.**

Figure 1. Opening the sample application



Device: DefaultColorPhone

New Project

Project Name: J2meSyncProject

MIDlet Class Name: J2meMob

3. The next window lets you specify the project settings. Click **OK** to accept the defaults.

4. Copy the source files -- J2meMob.java, SyncParser.java, and OrderBean.java -- from C:\j2mesync2\src to C:\wtk22\apps\J2meSyncProject\src. (Remember, the J2ME Wireless Toolkit is installed in the C:\wtk22 path.)

5. Click **Build** on the Ktoolbar. You receive the following message:

```
Project settings saved
Building "J2meSyncProject"
Build complete
```

Congratulations! You have successfully built your `J2meMob.java` MIDlet.

Starting Tomcat server

Now you'll need to start Tomcat. Open the Windows command prompt and change directories to C:\tomcat5.0. Copy the Cloudscape database JAR files -- db2jcc.jar and db2jcc_license_c.jar -- from C:\Cloudscape\lib to C:\tomcat5.0\syncwebapp\WEB-INF\lib.

Next, start the Tomcat server by entering the following at the command prompt:

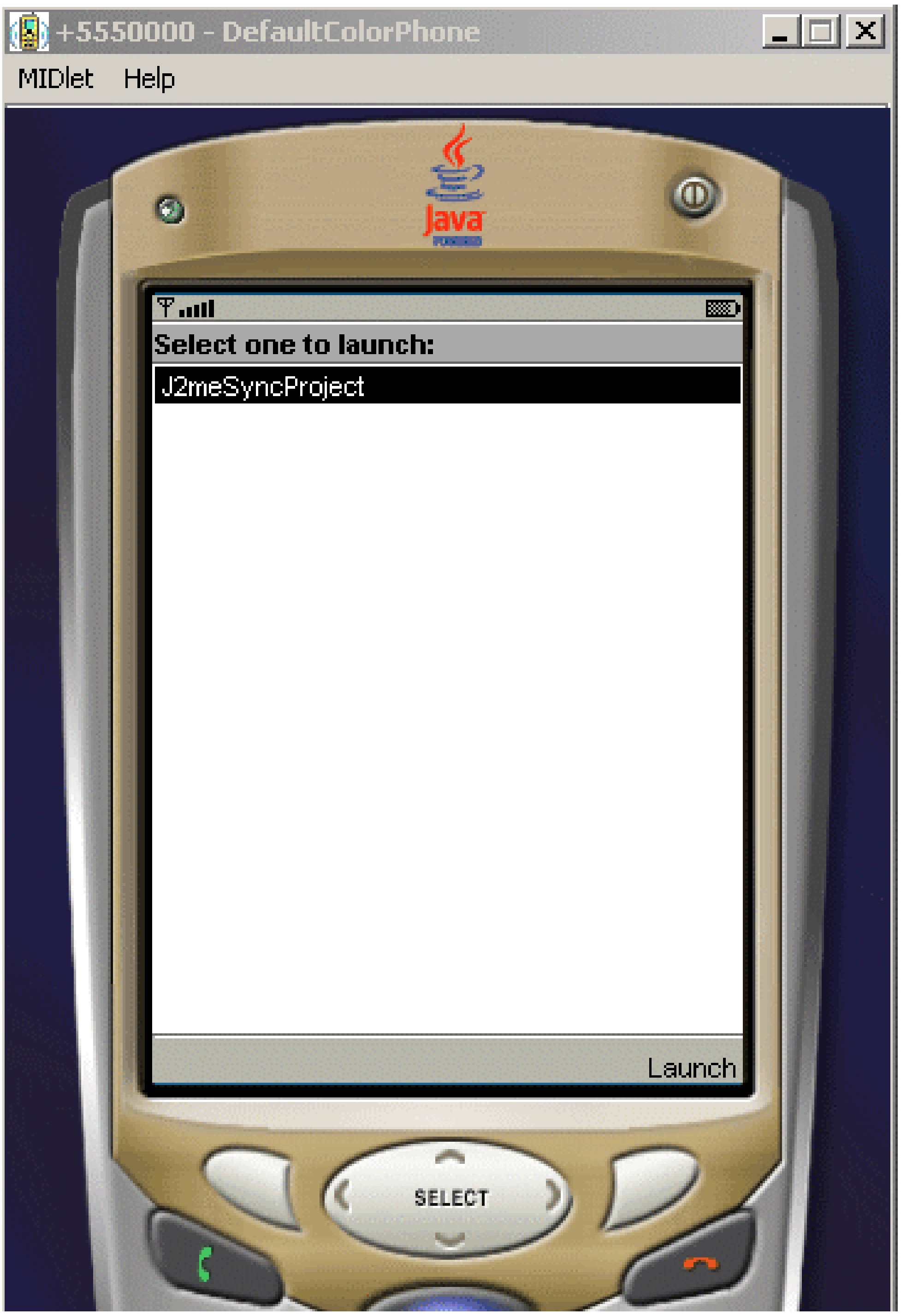
```
cd bin
C:\Tomcat 5.0\bin> catalina.bat start
```

This starts the Tomcat server and loads the syncwebapp application. You already started the Cloudscape database server when you set up the syncDB database in Creating database and tables.

Launching the application

To run the sample J2meSyncProject application, click **Run** on the Ktoolbar. The default emulator shown in Figure 2 appears.

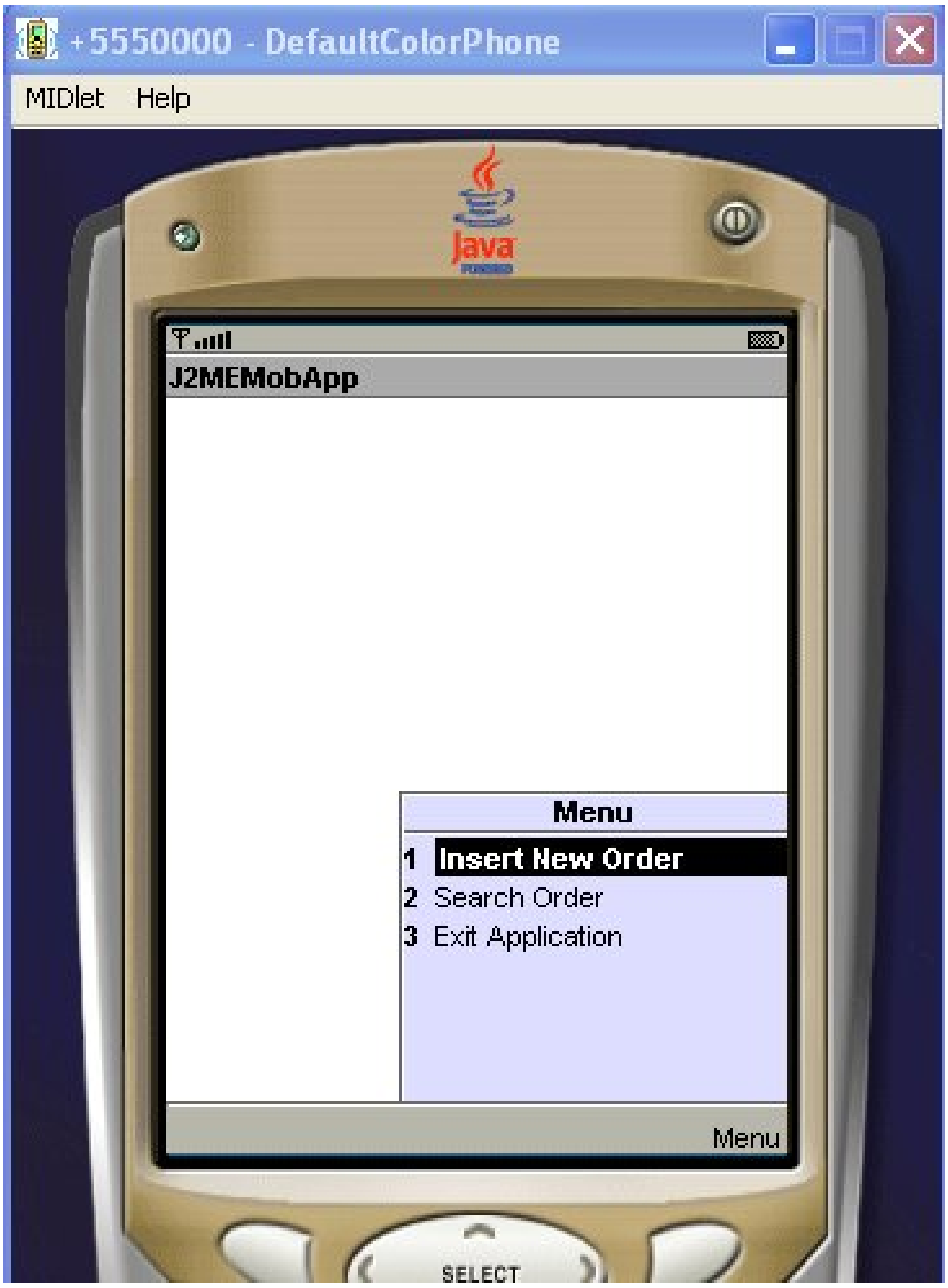
Figure 2. Launching the sample application



Running the application

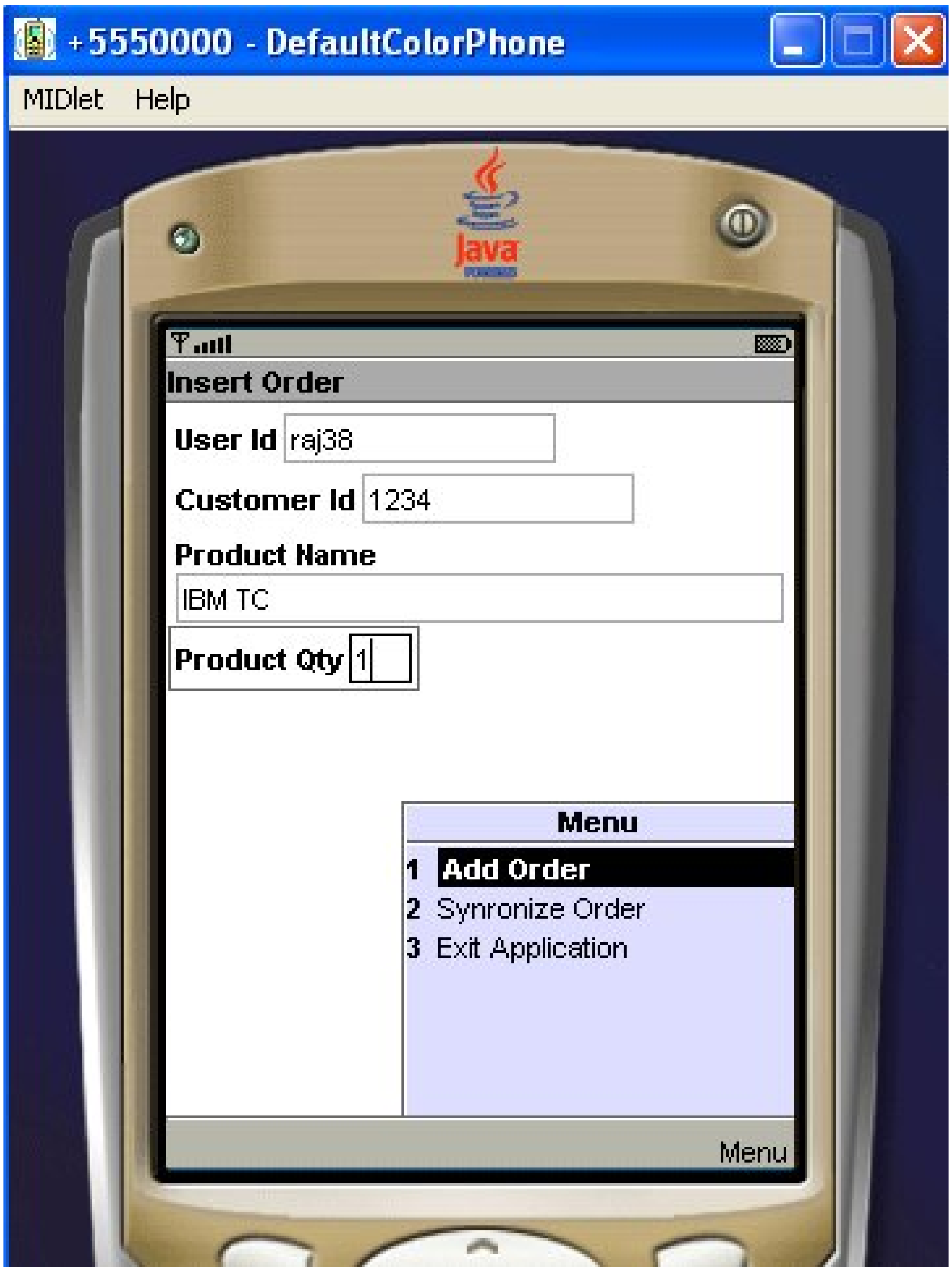
Launch the application and select the **Menu** option. You should be provided with the menu options shown in Figure 3:

Figure 3. Ready for data input



Select **Insert New Order**; you are presented with the screen shown in Figure 4. Enter the order information as shown in Figure 4. Click **Menu** and select **Add Order** to add the order information.

Figure 4. Insert order



Insert Order

User Id raj38

Customer Id 1234

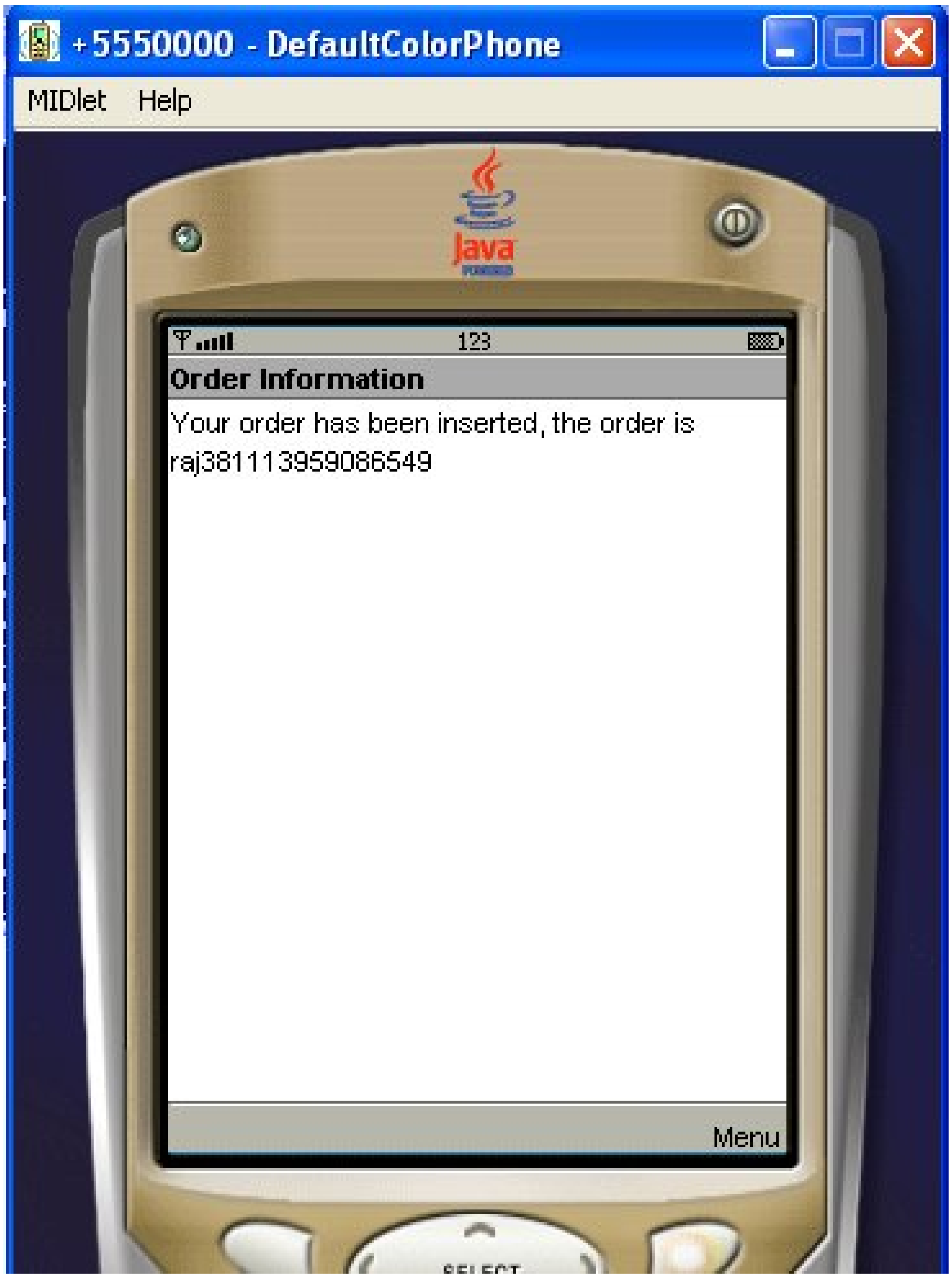
Product Name

IBM TC

Product Qty 1

After the order is inserted, you are given an order ID as shown in Figure 5. This ID tracks the client's order. The application generates a unique order ID based on the concatenation of the user ID and current timestamp.

Figure 5. Order ID



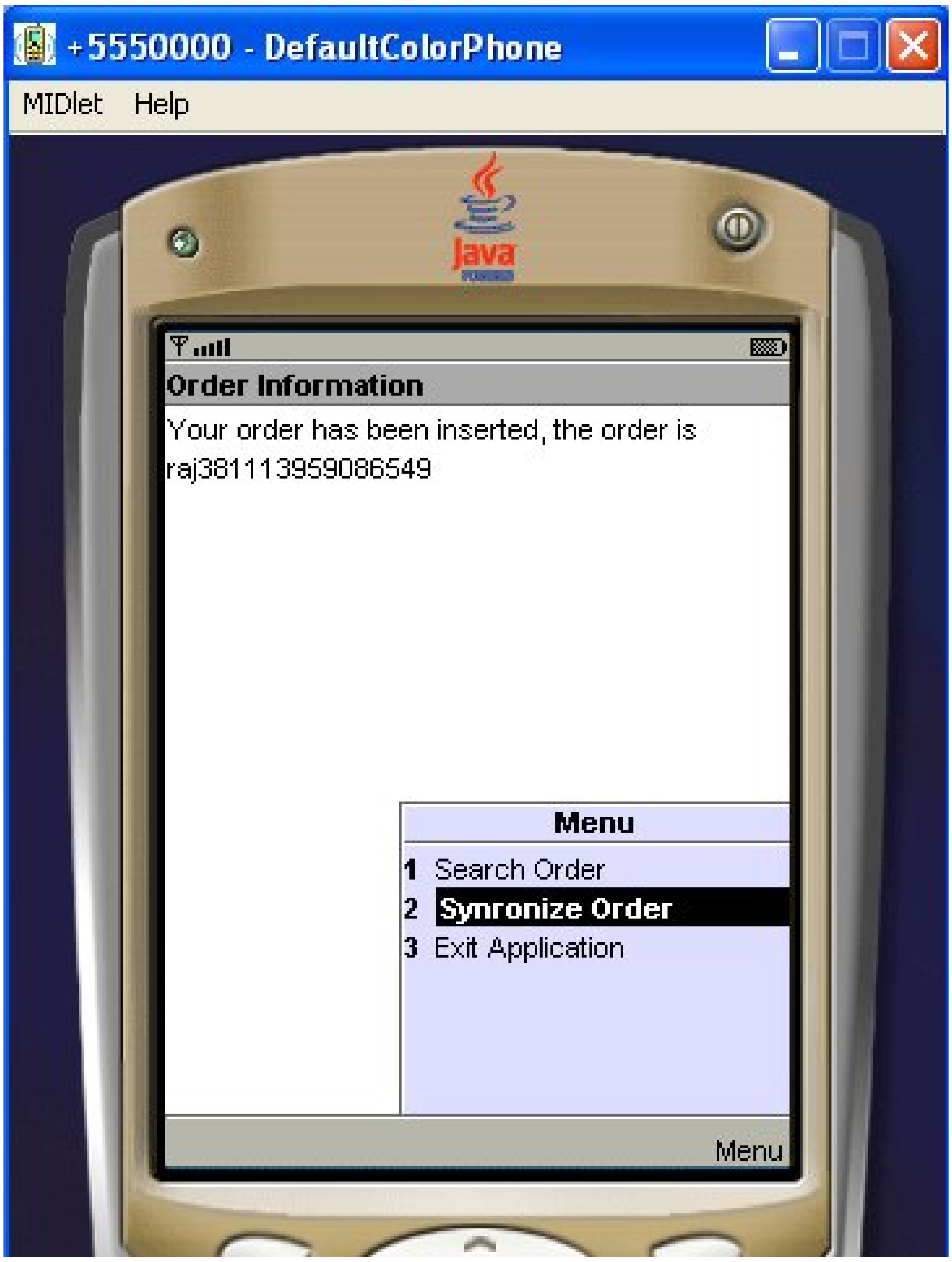
Order Information

Your order has been inserted, the order is raj381113959086549

Synchronizing order information

To synchronize the order information, click on **Menu** and select **Synronize Order**.

Figure 6. Ready for data input



Order Information

Your order has been inserted, the order is raj381113959086549

If synchronization is successful, you will see `synchronization successful` on the ktoolbar console.

You can verify that the order has been updated in the Cloudscape database using the `iij` client tool. Type in the following SQL command:

```
select * from syncadmin.order_info ;
```

The following output indicates that the record has been inserted:

```
1|raj381113959086549|2005-04-19 20:04:46.549|raj38|1234|IBM
```

```
TC|1|N|Y|2005-04-19 20:14:41.013
```

.
Next, still in the `iij` tool, use the following command to update the corresponding order and change the `orderstatus` to `Y`. (Replace `xxx` with your application order ID.)

```

update syncadmin.order_info set orderstatus = 'Y' where orderid = 'xxx' ;
```

.
You will see a message:

```

1 row inserted/updated /deleted
```

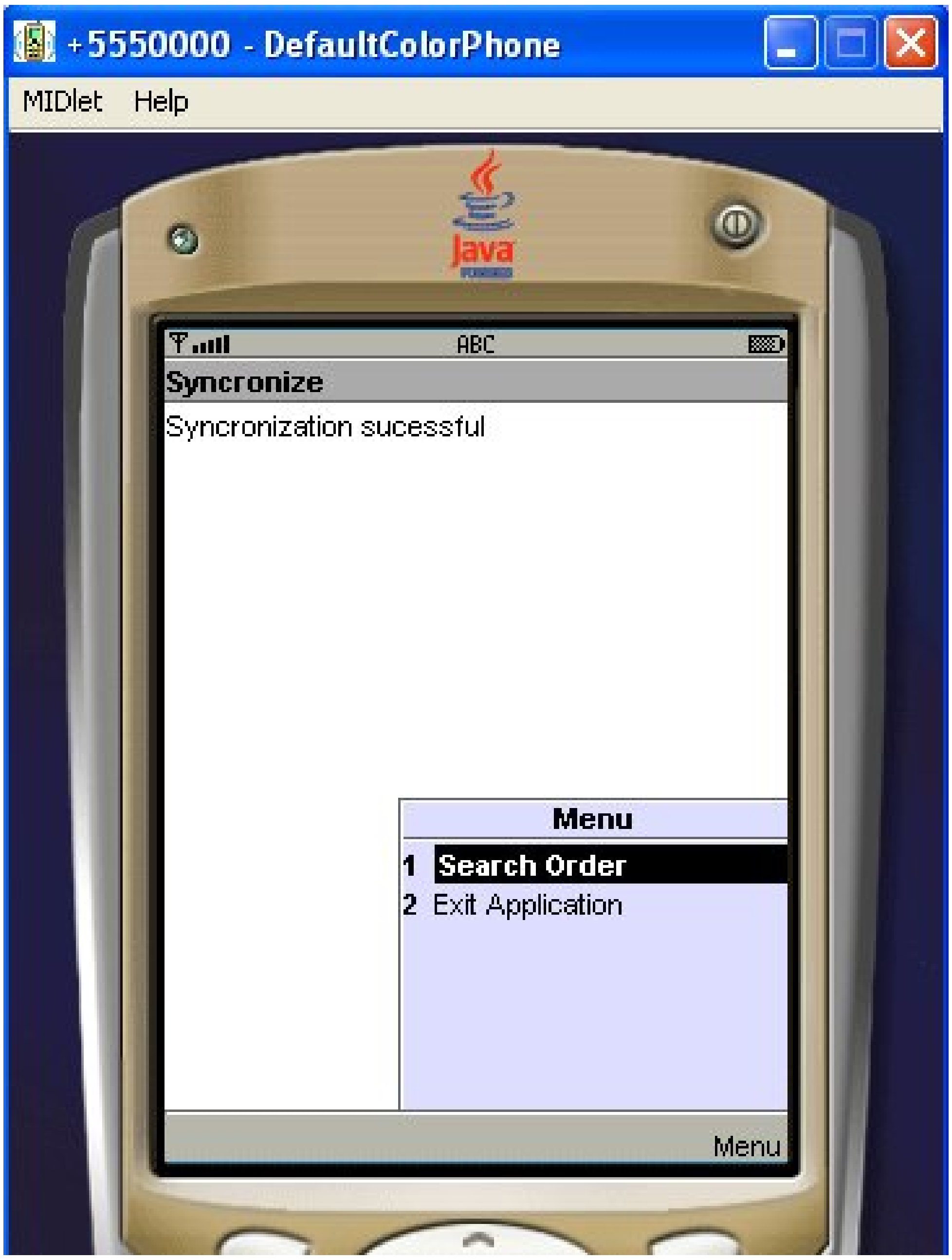
In a real-world situation, of course, updates to the back-end database would be performed by a server-side application. You're doing it manually here because this tutorial's purpose is to show how the J2ME application will react to updated data on the server, which you'll see next.

Resynchronizing the application

To resynchronize the order information, in the emulator select **Menu** and select **Synronize Order**. If synchronization is successful, you will see a message to that effect on the ktoolbar console.

To track orders, select **Menu**. You should be provided with the menu options shown in Figure 7.

Figure 7. Search order ID

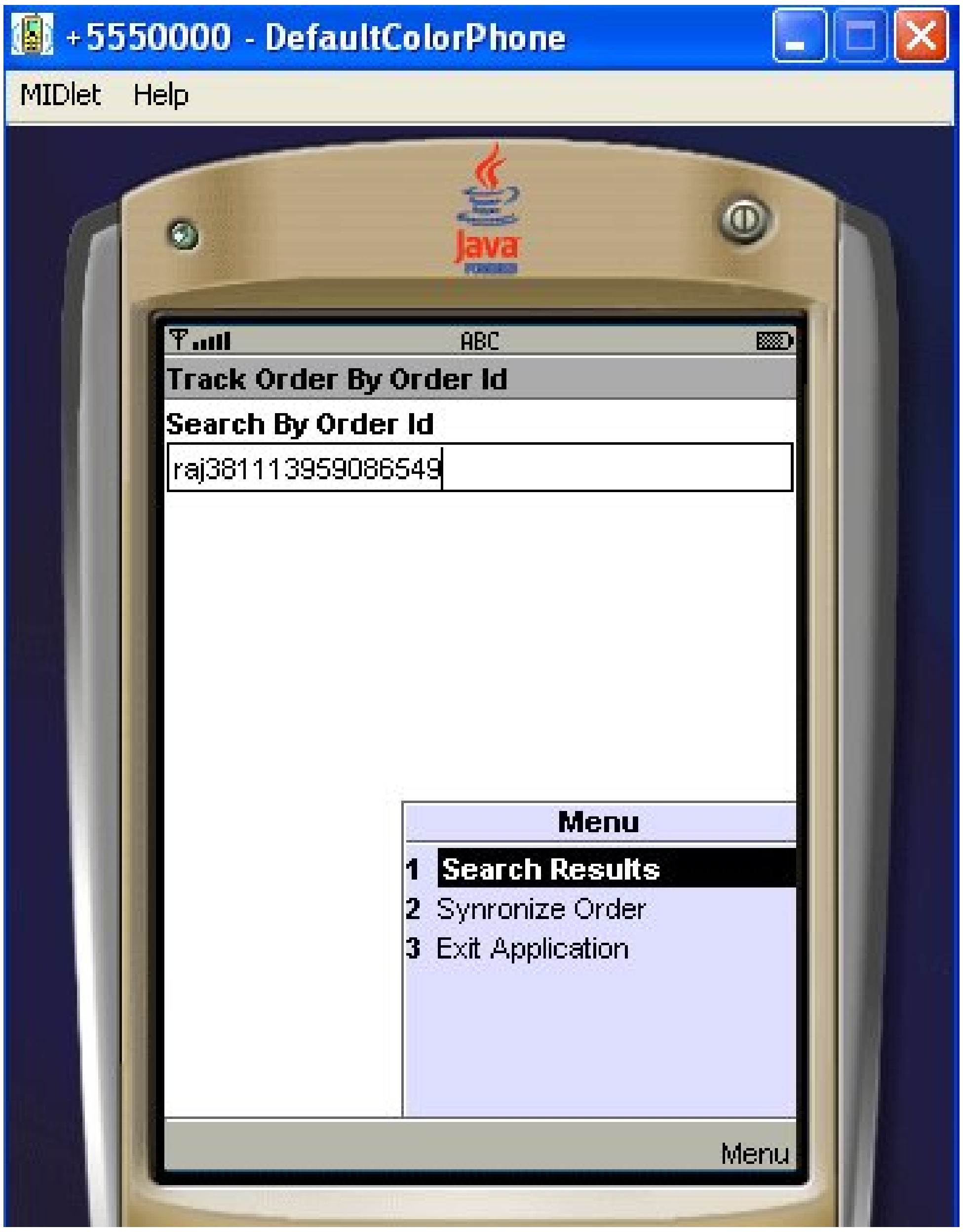


Syncronize

Syncronization sucessful

Select **Search Order**, and you are shown the following screen. Enter the order ID that you are working with as shown in Figure 8. Click **Menu** and select **Search Results** to find the order information.

Figure 8. Search order



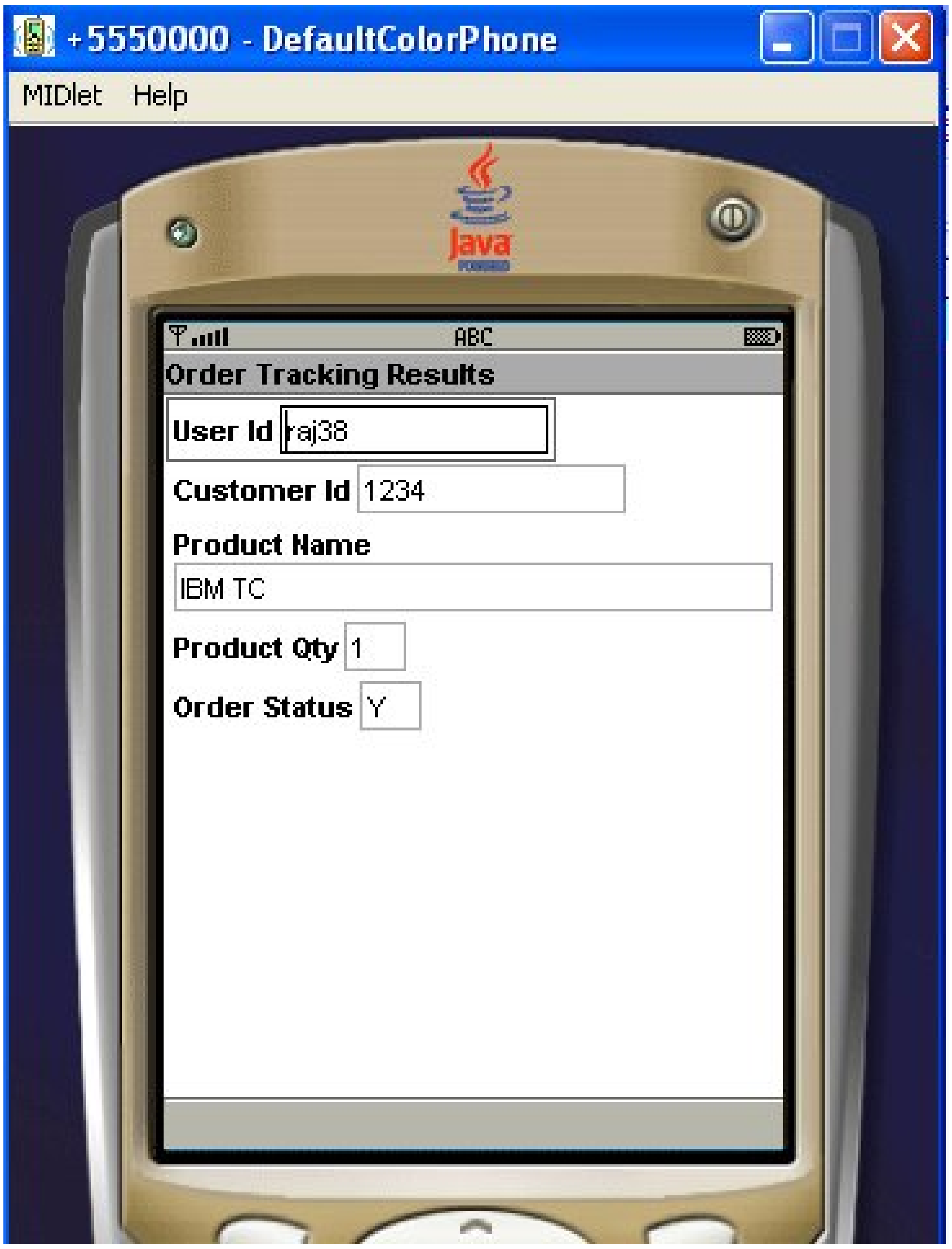
Track Order By Order Id

Search By Order Id

raj381113959086549

If the order ID matches, you are given the order information as shown in Figure 9, along with the order status. An order status of **Y** indicates that the order has been shipped. You retrieved this status by performing a synchronization after you updated the order on the Cloudscape database. Thus, you have successfully synchronized the order with a remote database using two-way synchronization.

Figure 9. Search order results



Order Tracking Results

User Id	raj38
Customer Id	1234
Product Name	IBM TC
Product Qty	1
Order Status	Y

Section 7. Summary

In these two tutorials, you have successfully built an order placer and tracking application using J2ME RMS, and performed a two-way synchronization to synch the mobile application data with a remote Cloudscape database. With this information in hand, you can start building your own smart J2ME-based mobile applications.

Resources

- This article assumes that you have installed the software and application as described in Part 1 of the tutorial series. If you haven't done so already, check out "Build smart J2ME mobile applications, Part 1" (developerWorks, April 2005).
- Before you start working, make sure you have the following downloads on your system:
  - The J2ME Wireless Toolkit 2.2
  - The Java SDK 1.4.1 (Java Software Developers Kit)
  - IBM Cloudscape 10.0
  - The Minimal kXML JAR file
  - Tomcat 5.0
  - The sample code that accompanies this tutorial.
- Learn more about Cloudscape at the DB2 Developer Domain.
- The Web Services Tool Kit for Mobile Devices provides tools and run time environments that allow development of applications that use Web services on small mobile devices, gateway devices, and intelligent controllers.

About the author

Naveen Balani
Naveen Balani spends most of his time designing and developing Java 2 Platform, Enterprise Edition (J2EE)-based frameworks and products. He has written various articles for IBM® developerWorks in the past, covering topics like ESB, SOA, JMS, WebServices Architectures, CICS, AXIS, J2ME, DB2® XML Extender, WebSphere® Studio, MQSeries, Java™ Wireless Devices, and DB2 Everyplace for Palm, Java-Nokia, Visual Studio, .Net, and wireless data synchronization. You can reach him at naveenbalani@rediffmail.com.