

Push the limits of Java UDRs in Informix Dynamic Server V10

Extend IDS features with Java

Skill Level: Intermediate

[Halit \(Hal\) M. Maner \(hmaner@msystemsintl.com\)](mailto:hmaner@msystemsintl.com)
Chief Technology Officer
M Systems International, Inc.

[Jean Georges Perrin \(jgp@jgp.net\)](mailto:jgp@jgp.net)
Director
IIUG

05 Oct 2006

Learn how to write server side Java™ code in the form of a User Defined Routine (UDR), running inside the Informix® Dynamic Server (IDS). Also, learn how to set up your project using modern tools such as Eclipse.

Section 1. Before you start

Learn what to expect from this tutorial, and the technical environment.

Introduction

Although IDS is one of the best general purpose and online transaction processing (OLTP) databases out there, you may need to extend its function set. IDS provides a very open interface for doing so: UDRs. UDRs can be developed in Java, making portability a non-issue, helping you benefit from your existing Java knowledge and letting you reuse libraries and components already available in your applications.

Objectives

In this tutorial, you first discover the fundamentals then directly jump to a real-life example. You also call your Java UDR from an Informix 4GL application. You then see how you can include this example into Eclipse, the famous development environment. Finally, you study the deployment strategy to use your UDR in production.

Prerequisites

It is assumed that you have development experience with IDS. No Java or Informix 4GL background is required.

System requirements

In this tutorial, you will use the following:

- IDS V10.0.TC4
- J2SE Development Kit (JDK) V5.0 (update 6)
- and Eclipse V3.2 RC7

This example uses Windows XP, but the examples are easily portable to Linux or any other UNIX system.

Section 2. Fundamentals

IDS, like in many other areas, has been the innovator and technology leader in embedding Java Virtual Machine (JVM) technology in a database server. This powerful feature, called J/Foundation, has been part of IDS since Version 9 (then called Informix-Universal Server) was released following the 1996 acquisition of Illustra by the Informix Corporation.

A UDR is a routine that you create and register in the system catalog tables and then invoke within an SQL statement. A UDR can be either a function (can accept arguments and can return values) or a procedure (can accept arguments but does not return values).

Note:The test bed in this example is called scoobidoo. It is a Windows XP Professional SP2 machine running IDS 10.0.TC4 as ol_scoobidoo. The Informix user has the Informix password. The engine is installed on C:\Program Files\IBM\Informix (%INFORMIXDIR% as this is on Windows). You will work in C:\UDRProject.

In order to activate J/Foundation in IDS, certain onconfig parameters must be set and a JVM must be available. On your system, the onconfig configuration file is ONCONFIG.ol_scoobidoo, available in %INFORMIXDIR%\etc.

Onconfig parameters

The default system smart blob space (sbospace) is sbospace1. This is where your database server stores your Java UDR Java archive (JAR) files and your smart blobs, unless a specific dbospace was provided for them.

```
SBSPACENAME      sbospace1
```

If you have not set up a smart blob space during the initial installation, you can do so by using the onspaces command:

```
onspaces
-c
-S sbospace1
-p C:\IFMXDATA\ol_scoobidoo\sboospace1_dat.000
-o 0
-s 10000
```

The appendix details the use of onspaces. The number of Java Virtual Processors (JVPs) that the database server should start. It is commented by default.

```
VPCLASS      jvp,num=1
```

The following is the installation directory for the database's Java Runtime Environment (JRE). The default is \$INFORMIXDIR/extend/krakatoa/jre.

```
JVPJAVAHOME C:\PROGRA~1\IBM\Informix\extend\krakatoa\jre
```

The following is the directory where the J/Foundation classes and JDBC classes are installed. The default is \$INFORMIXDIR/extend/krakatoa.

```
JVPHOME C:\PROGRA~1\IBM\Informix\extend\krakatoa
```

Warning: The Java UDRs JDBC driver is not the same as the JDBC driver found in the client. The UDRs JDBC driver complies to JDBC V2.0, whereas the driver coming with the client package is compliant to JDBC V3.0. Be careful about that when deploying. The JDBC driver in the client package does not support the Informix-direct protocol.

Java traces, output, and stack dumps are written to this file by the database server. On this system, the default was `ol_scoobidoo_jvp.log`.

```
JVPLOGFILE C:\PROGRA~1\IBM\Informix\extend\krakatoa\jvp.log
```

This is an optional parameter. When set, it points to the path of the JVP properties file. On this system, the default was `.jvpprops_ol_scoobidoo`. Trace level settings, trace verbosity settings, monitor port, and a few other properties are set in this file.

```
JVPPROFILE C:\PROGRA~1\IBM\Informix\extend\krakatoa\.jvpprops
```

This is the major JDK version supported by the database server. For IDS V10, the valid values are 1.4, 1.3, and 1.2. Unless you have a specific reason to use an older version, leave 1.4.

```
JDKVERSION 1.4
```

Note: IDS V10.00.TC3 and V10.00.TC4 for Windows come with the IBM JRE Version 1.4.2.

The following is the path to the JVM libraries relative to `JVPJAVAHOME`, the default value is `fine`.

```
JVPJAVALIB \bin\
```

The following is a list of the JVM libraries that the database server loads. Unless you have very specific requirements, leave the default values.

```
JVPJAVAVM jsig;dbgmalloc;hpi;jvm;java;net;zip;jpeg
```

This parameter sets the `CLASSPATH` for the J/Foundation JAR files (`krakatoa.jar` and `jdbc.jar`) for the database server to use during startup. The default value is `fine`.

```
JVPCLASSPATH C:\PROGRA~1\IBM\Informix\extend\krakatoa\krakatoa.jar;  
C:\PROGRA~1\IBM\Informix\extend\krakatoa\jdbc.jar
```

The following are the memory settings for the JVM:

```
JVPARGS -Xms128m; -Xmx128m
```

Very important: This parameter is not set for you and it may not even be in the `onconfig.std` file, yet it is a critical parameter. This is where you set the options for the JVM, including the amount of memory it can use.

The `onconfig` line shown above allocates 128 MB to the JVM. If you do not set this parameter, you only get 16 MB by default.

You should also be aware of the `JAR_TEMP_PATH` environment variable. Set this if you want temporary JAR files to be stored somewhere other than `C:\tmp` (Windows) or `/tmp` (UNIX).

```
JAR_TEMP_PATH
```

Very important: If this environment variable is not set, and the directory does not exist, you get an error when you try to install your file(s).

You should restart the Informix engine (the Windows service) before you can use the Java UDRs. To check that everything works fine, try the following:

```
onstat -g glo | grep jvp
```

You should get the following:

```

jvp      1      0.00      0.00      0.00
3      6052      jvp      0.00      0.00      0.00

```

Note: Start the Informix engine from the command line on Windows. For more information, refer to [Restart the Informix engine from the command line](#).

Note: If you do not have a `grep` command on your Windows system, use `more` (or `get one`): `onstat -g glo | more`.

Section 3. Advantages

The following are some of the advantages of UDRs:

Application speed and reduced bandwidth load

The primary reason to deploy application functionality in the form of a UDR should be to keep this functionality where it belongs, for example, in the server. The increasing use of the client or server and especially Web applications, bring with them the need to minimize the network traffic between the client and the server. This is true no matter how fast the connection in between may be. The less traffic there is, the faster the application runs. It also positively impacts the performance of other applications that depend on the same pipe, because of the reduced load on the bandwidth. Server disk I/O intensive applications are the perfect candidate.

Those of you who have developed or supported a thick client application that makes heavy use of cursors over a Wide Area Network (WAN) connection, know how important it is to have a high speed network bandwidth. And also how much network traffic such an application creates. A UDR allows almost all of that to take place inside the database server, without slowing down due to network bandwidth capacity. It should be mentioned that we are not advocating Java UDRs as the best way to write applications. They have a specific purpose, they are a good tool and technology to use in certain applications. Mainly, where your application deployment need is on the server side and also if it is somewhere between writing a stored procedure and writing a client based or server based function or executable.

Ease of deployment

Server-side deployment is fast and much easier when compared to client-side installation. Within a few minutes, and from a central location, new application functionality can be added in the form of a JAR file.

Rich, standard full programming language

Instead of being limited to SQL or a proprietary stored procedure language, you can take advantage of a modern object-oriented standard programming language for server-side routines. The code is nearly 100 percent portable because it is written in an industry standard programming language, Java, which works with most industry leading databases and all industry leading operating system platforms.

Application layer flexibility

Thanks to the use of an industry standard programming language, if you decide to run your code outside the server, you can do so with minimal work. This is not possible to do with a proprietary stored procedure language.

Function centralization

In environments where multiple applications, perhaps each written in a different language, frequently have to perform a certain task in a common way (for example, calculate the total cost of a certain manufacturing part), implementing this common function as a UDR, can enable all these applications to use the same routine with an SQL call.

Section 4. Hello world example

The best example of all is always the same: the indefatigable hello world from our masters, Kernighan & Ritchie.

Now see how it applies to Java UDRs within IDS. You are going to create a simple function that returns the traditional message. In order not to interfere with the rest of the example, work in C:\ProjectUDR\hw.

The following is a step-by-step process to create a hello world Java UDR.

Step 1: Design and code your Java classes

Listing 1. The hello world UDR (HelloWorld.java)

```
public class HelloWorld {  
    public static String getHelloWorld() {  
        return "Hello, world...";  
    }  
}
```

Step 2: Compile your code

Here are the simple commands to build the UDR.

Listing 2. Compiling the hello world example

```
del *.class  
javac -source 1.4 -target 1.4 HelloWorld.java
```

Note: If you get an error message: 'javac' is not recognized as an internal or external command, operable program or batch file, make sure that your JDK bin directory is in the path (C:\Program Files\Java\jdk1.5.0_05\bin on the example system). For more information, refer to the [Appendix](#).

Step 3: Build a JAR file

```
erase hw.jar
jar cvf hw.jar HelloWorld.class
```

Note: For Advanced Java users, it is not necessary to include a manifest file in the JAR file.

Step 4: Install your JAR file

Install the JAR file in the engine and create its corresponding SQL function (this assumes that your database is logged -- you do not have to use BEGIN WORK/COMMIT WORK if you do not want to). Run DB-Access and connect to the stores_demo as a user having the EXTEND privilege (the Informix user has it).

Listing 3. SQL install script for hello world

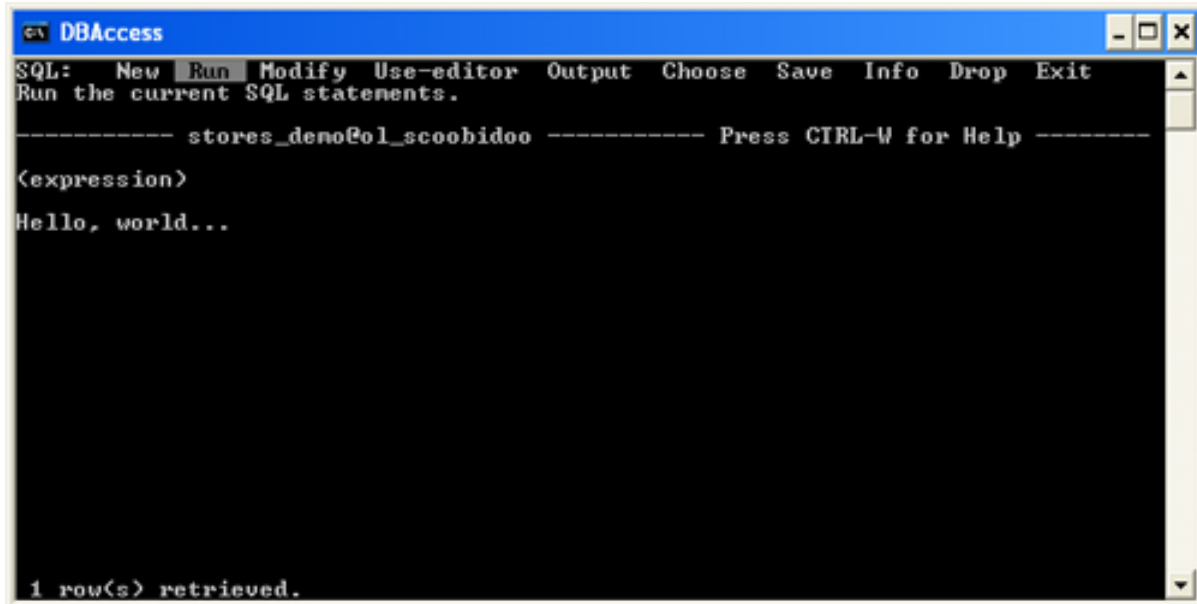
```
BEGIN WORK;
EXECUTE PROCEDURE sqlj.install_jar
('file:\C:\UDRProject\hw\hw.jar',
'hw_jar',
0);
CREATE FUNCTION getHelloWorld()
RETURNS CHAR(50)
EXTERNAL NAME 'hw_jar:HelloWorld.getHelloWorld'
LANGUAGE JAVA;
GRANT EXECUTE ON FUNCTION getHelloWorld() TO ALL;
COMMIT WORK;
```

Step 5: Run

Now you can run it.

```
EXECUTE FUNCTION getHelloWorld();
```

Figure 1. DB-Access running your first UDR

A screenshot of a Windows-style application window titled "DBAccess". The window has a menu bar with options: "SQL:", "New", "Run", "Modify", "Use-editor", "Output", "Choose", "Save", "Info", "Drop", "Exit". Below the menu bar, it says "Run the current SQL statements." followed by a separator line and "stores_demo@ol_scoobidoo" and "Press CTRL-W for Help". The main area of the window is black with white text. It shows "(expression)" followed by "Hello, world...". At the bottom, it says "1 row(s) retrieved.".

```
SQL:  New Run Modify Use-editor Output Choose Save Info Drop Exit
Run the current SQL statements.
----- stores_demo@ol_scoobidoo ----- Press CTRL-W for Help -----
(expression)
Hello, world...

1 row(s) retrieved.
```

Section 5. A real-life example

The UDR example, `PriceChanger.java`, is built for the superstores demo database that comes with IDS. Given an effective date and a percentage, the UDR goes through the orders in the database and changes the prices if they meet the date criteria. This example demonstrates that it is possible (and practical) to build object-oriented, server-based applications running inside the database server. The applications can utilize multiple classes and practically all the features and functionality of the Java language.

Building the UDR step-by-step, follow the same steps you did previously.

Step 1: Design

The main design goal was to have a reusable component, called `PriceChangerGeneric`. It contains all the processing and business logic. `PriceChangerGeneric` needs two things to do its job:

- A database connection (`Connection`)
- A logging system (`LogInterface`)

In this implementation, `PriceChangerGeneric` gets a direct connection to IDS and the logging system uses the UDR logging system. For more information, refer to the

[Troubleshoot through tracing and logging](#) section. In other words, you have isolated the business logic and it could work with an engine that does not have Java UDRs or any other piece of software. It is reusable.

The added value resides in `PriceChanger`. It calls `SuperStoresConnection` to build the connection to the engine, instantiates `LogUDR` to have a `LogInterface` compliant logging system.

You are using four classes and an interface:

- **PriceChanger.java:** Sets-up the interface to the engine.
- **PriceChangerGeneric.java:** Does the processing.
- **SuperStoresConnection.java:** The connection class to the engine.
- **LogInterface.java:** Defines a basic logging interface (or contract).
- **LogUDR.java:** Implements the `LogInterface` so it can be used in various places.

Listing 4. PriceChanger.java

```
package net.jsp.lab.udr.pricechanger;

import java.sql.Connection;
import java.sql.SQLException;

/**
 * Price Changer class
 *
 * The role of this class is to enable the "interface" between the
 * business logic and both the database engine and the logging system.
 */
public class PriceChanger {
    /**
     * Database connection
     */
    private static Connection udrConn;

    /**
     * This is the "entry point" of the UDR. The SQL function that can be
     * called from client applications corresponds to this method.
     *
     * @param effectiveDate Effective date of change.
     * @param percentChange Percent of change.
     * @return 0 when everything's ok, 1 if error.
     * @throws Exception If something goes wrong.
     */
    public static int changePriceByPercent(String effectiveDate,
        int percentChange) throws Exception {

        int resultCode = 0;

        // Instantiates the logging mechanism
        LogInterface li = new LogUDR();
        li.log("PriceChanger.changePriceByPercent method started with "
            + effectiveDate + " and " + percentChange + "%.");
    }
}
```

```

try {
    // Connection to the database
    udrConn = SuperStoresConnection.connect();
    if (udrConn == null) {
        li.log("Connection to the database failed.");
        return resultCode;
    }

    // Then we ask the driver to do the actual processing.
    PriceChangerGeneric priceChangerGeneric = null;

    // We give the driver its required elements, the connection to the
    // database and the logging system
    priceChangerGeneric = new PriceChangerGeneric(udrConn, li);

    li.log(3,
        "PriceChanger.changePriceByPercent established connection.");

    // We give the driver its business related parameters
    priceChangerGeneric.setParameters(effectiveDate, percentChange);

    // The job will now be done
    resultCode = priceChangerGeneric.drivePriceChanges();

    // We are closing the connection
    SuperStoresConnection.close();
}
catch (SQLException ex) {
    ex.printStackTrace();
    throw ex;
}

// we return the result code
return resultCode;
}
} // end class

```

Java does not provide the return of multiple values, your UDR does not support multiple return values either.

Important: Do not forget to close your ResultSet and Statement objects. Otherwise, you will run into problems and you may even cause IDS to crash.

Listing 5. PriceChangerGeneric.java

```

package net.jgp.lab.udr.pricechanger;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;

/**
 * Price Changer Generic class.
 *
 * This class does the effective work. It applies the business rules to
 * the data in the database. At this level, we do not care about the
 * database: we receive a connection and we do the processing. The
 * connection can come from a UDR (direct connection) or from an
 * indirect connection to database, making this piece of code highly
 * reusable.
 */
public class PriceChangerGeneric {
    /**

```

```

    * Database connection
    */
private Connection dbConn;

/**
 * Effective date
 */
private String effectiveDate;

/**
 * Percent change
 */
private int percentChange;

/**
 * Log
 */
private LogInterface log;

/**
 * Update stock unit price statement
 */
private PreparedStatement updateStockUnitPriceStatement;

/**
 * Update items for price change statement
 */
private PreparedStatement updateItemsForPriceChangeStatement;

/**
 * The constructor expects the two required parameters.
 */
public PriceChangerGeneric(Connection databaseConnection,
    LogInterface logSystem) {
    this.dbConn = databaseConnection;
    this.log = logSystem;
}

/**
 * Sets the effective date and price change percentage of this object.
 *
 * @param effectiveDate
 * @param percentChange
 */
public void setParameters(String effectiveDate, int percentChange) {
    this.effectiveDate = effectiveDate;
    this.percentChange = percentChange;
}

/**
 * This is the main driver method for the price changes. It first
 * updates stock.unit_price for all stock items. It then updates
 * items.item_subtotal in orders whose dates (orders.order_date) are
 * on or after the price change effective date that have not yet been
 * paid (orders.paid_date is null).
 *
 * @return resultCode 0 if successful, 1 if error
 */
public int drivePriceChanges() throws SQLException {
    int resultCode = 0;
    String sqlStatement = null;

    try {
        this.log.log(3, "Driving with drivePriceChanges method...");
        this.log.log(3, "percentChange=" + this.percentChange);
        this.log.log(3, "effectiveDate=" + this.effectiveDate);

        // first, update the prices in the stock table
        sqlStatement = "UPDATE stock "

```

```

        + "SET unit_price = unit_price + ((unit_price * ?) / 100) "
        + "WHERE unit_price IS NOT NULL";

this.updateStockUnitPriceStatement = dbConn
    .prepareStatement(sqlStatement);
this.updateStockUnitPriceStatement.setInt(1, this.percentChange);
this.updateStockUnitPriceStatement.executeUpdate();

// then, find the orders that meet the effective date criteria and
// update their items
sqlStatement = "UPDATE items"
    + " SET item_subtotal = item_subtotal +"
    + " ((item_subtotal * ?) / 100)" + " WHERE order_num IN "
    + " (SELECT order_num FROM orders"
    + " WHERE paid_date IS NULL AND order_date >= ?)";
this.updateItemsForPriceChangeStatement = dbConn
    .prepareStatement(sqlStatement);
this.updateItemsForPriceChangeStatement.setInt(1,
    this.percentChange);
this.updateItemsForPriceChangeStatement.setString(2,
    this.effectiveDate);
this.updateItemsForPriceChangeStatement.executeUpdate();

// Commits the transaction
this.dbConn.commit();
}
catch (SQLException ex) {
    this.dbConn.rollback();
    resultCode = 1;
    ex.printStackTrace();
    throw ex;
}

// Cleaning
this.updateStockUnitPriceStatement.close();
this.updateItemsForPriceChangeStatement.close();

return resultCode;
}
} // end class

```

Listing 6. SuperStoresConnection.java

```

package net.jgp.lab.udr.pricechanger;

import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;

/**
 * Database connection class
 */
public abstract class SuperStoresConnection {

    /**
     * Database connection
     */
    private static Connection udrConn = null;

    /**
     * Performs the actual connection to the database.
     *
     * @return Connection object.
     */
    public static Connection connect() throws SQLException {

```

```
if (udrConn != null) {
    System.err
        .println("Connection is already open, use close() first.");
    return null;
}

try {
    // Loads the driver
    Class.forName("com.informix.jdbc.IfxDriver");

    // Connect to the database. Note that the URL to connect to the
    // database is not the same as the usual one used by the JDBC
    // driver, here we have a *direct connection* to the engine.
    udrConn = DriverManager
        .getConnection("jdbc:informix-direct:superstores_demo");

    // Set some parameters
    udrConn.setAutoCommit(false);
}
catch (SQLException sqlEx) {
    sqlEx.printStackTrace();
    throw sqlEx;
}
catch (ClassNotFoundException ex) {
    System.err.println("Class not found exception...");
}

return udrConn;
} // end connect()

/**
 * Closes the connection.
 */
public static void close() {
    if (udrConn != null) {
        System.err
            .println("Connection is not open, use connect() first.");
        return;
    }
    try {
        udrConn.close();
    }
    catch (SQLException ex) {
        ex.printStackTrace();
    }
} // end close()
} // end class
```

As you can see, the connection is done through Informix-direct, the direct connection is an internal connection, so it does not open a new session to the database. It also remains within the same transaction.

The source code for the log files is found in the [Troubleshoot through tracing and logging](#) section.

Step 2: Compilation

Use the provided batch file, compile.bat. It contains a little more than Listing 2:

- Packages
- Elements from the Informix libraries

As you use a package (net.jgp.lab.udr.pricechanger), it physically maps to the net\jgp\lab\udr\pricechanger directory. Use an environment variable (workdir) to make the compilation process easier.

You will see two libraries (JARs) coming from the Informix package: krakatoa.jar and jdbc.jar, which are located in C:\Progra~1\IBM\Informix\extend\krakatoa, thus the use of the INFORMIXKRAKATOADIR environment variable.

Listing 7. Compiling the PriceChanger example (compile.bat)

```
set workdir=net\jgp\lab\udr\pricechanger
set INFORMIXKRAKATOADIR=C:\Progra~1\IBM\Informix\extend\krakatoa
set CLASSPATH=%CLASSPATH%;.%INFORMIXKRAKATOADIR%\krakatoa.jar;
%INFORMIXKRAKATOADIR%\jdbc.jar;
del %workdir%\*.class
javac -source 1.4 -target 1.4 %workdir%\PriceChanger.java
javac -source 1.4 -target 1.4 %workdir%\PriceChangerGeneric.java
javac -source 1.4 -target 1.4 %workdir%\SuperStoresConnection.java
javac -source 1.4 -target 1.4 %workdir%\LogUDR.java
```

Step 3: Build a JAR for deployment

Building the JAR file remains a very easy process.

```
erase pc.jar
jar cvf pc.jar .
```

Step 4: Install

Using DB-Access, log in to the superstores_demo database as informix.

Listing 8. SQL install script for PriceChanger (install.sql)

```
BEGIN WORK;
EXECUTE PROCEDURE sqlj.install_jar(
  'file:\C:\UDRProject\pc\pc.jar',
  'pc_jar',
  0);
CREATE FUNCTION changePriceByPercent(
  effectiveDate CHAR(10),
  percentChange INTEGER)
RETURNS INTEGER
EXTERNAL NAME
  'pc_jar:net.jgp.lab.udr.pricechanger.PriceChanger.changePriceByPercent'
LANGUAGE JAVA;
```

```
GRANT EXECUTE ON FUNCTION changePriceByPercent(CHAR, INTEGER) TO ALL;  
COMMIT WORK;
```

You are likely to need to remove your UDR as well (to install a new one perhaps), and here is how to do that.

Listing 9. SQL uninstall script for PriceChanger (uninstall.sql)

```
BEGIN WORK;  
DROP FUNCTION changePriceByPercent;  
EXECUTE PROCEDURE sqlj.remove_jar('pc_jar', 0);  
COMMIT WORK;
```

Listing 10. SQL update script for PriceChanger (update.sql)

```
BEGIN WORK;  
EXECUTE PROCEDURE sqlj.replace_jar(  
  'file:\C:\UDRProject\pc\pc.jar',  
  'pc_jar');  
COMMIT WORK;
```

Step 5: Run

You are now ready to execute the UDR. The following paragraphs illustrate how to run your new UDR from DB-Access and a 4GL application.

Call from DB-Access

The example UDR can be called from DB-Access as follows:

```
EXECUTE FUNCTION changePriceByPercent('2006-05-01', 10);
```

Call from a 4GL application

Calling the UDR from 4GL is similar to calling it interactively.

Listing 11. Call to PriceChanger from I-4GL

```
LET udr_statement_string = "EXECUTE FUNCTION changePriceByPercent(?, ?)"  
PREPARE udr_statement  
  FROM udr_statement_string  
EXECUTE udr_statement  
  USING eff_date, chg_percent  
  INTO udr_result_code  
FREE udr_statement
```

Important tip for 4GL users: If you are calling your UDR from Informix-4GL, be

sure to FREE the statement. Otherwise, the virtual memory segment will not clear properly.

Section 6. Troubleshoot through tracing & logging

UDR tracing can generate messages written to the JVP log file specified by the JVPLOGFILE onconfig parameter. com.informix.udr.UDRTraceable is the name of the Java interface for tracing. UDR tracing uses “trace zones” -- this is a “logical” concept where you can name multiple zones in one UDR or one zone for all your UDRs. Within each zone, you can have six levels of detail, from zero (off) to five (superfine, or the most detailed).

You can see this in the example code.

Listing 12. LogInterface.java

```
/**
 * An interface defines a "contract" between the user and the
 * implementor. In this case, it means that all implementations
 * of
 * LogInterface must match all the functions listed here.
 * However, the
 * implementation is free to do whatever it wants: log via the
 * UDR
 * logging system, log using Log4J (a popular Java logging
 * system), log
 * using web services on a remote host, send SNMP events, etc.
 */
package net.jsp.lab.udr.pricechanger;

public interface LogInterface {

    /**
     * Logs a simple message.
     *
     * @param logMessage The message to log.
     */
    void log(String logMessage);

    /**
     * Logs a simple message associated to a level.
     *
     * @param logLevel Level to log.
     * @param logMessage The message to log.
     */
    void log(int logLevel, String logMessage);

    /**
     * Logs a simple message associated to a level and a zone.
     *
     * @param logZone Zone to redirect the log message to.
     * @param logLevel Level to log.
     * @param logMessage The message to log.
     */
}
```

```
    void log(String logZone, int logLevel, String logMessage);  
}
```

The interface is the contract given to all classes in need of logging. The real implementation is found in Listing 13.

Listing 13. LogUDR.java

```
package net.jgp.lab.udr.pricechanger;  
  
import com.informix.udr.UDREnv;  
import com.informix.udr.UDRLog;  
import com.informix.udr.UDRManager;  
import com.informix.udr.UDRTraceable;  
  
/**  
 * This is a basic logging system that implements the LogInterface and  
 * relays all the logging work to the Informix UDR logging classes.  
 */  
public class LogUDR implements LogInterface {  
  
    /**  
     * The default zone to log to.  
     */  
    private static final String DEFAULT_LOGZONE = "sSTZ";  
  
    /**  
     * UDR Log  
     */  
    private UDRLog log;  
  
    /**  
     * UDR Traceable  
     */  
    private UDRTraceable traceable;  
  
    /**  
     * The constructor establishes the link to the UDR API.  
     *  
     * @throws Exception  
     */  
    public LogUDR() throws Exception {  
        UDREnv env;  
  
        env = UDRManager.getUDREnv();  
        this.log = env.getLog();  
        this.traceable = env.getTraceable();  
        // used log here because the superStoresConnection has not yet been  
        // established  
    }  
  
    /**  
     * Implementation of basic logging.  
     */  
    public void log(String logMessage) {  
        this.log.log(logMessage);  
    }  
  
    /**  
     * Implementation of basic logging with zone and level.  
     */  
    public void log(String logZone, int logLevel, String logMessage) {  
        this.traceable.tracePrint(logZone, logLevel, logMessage);  
    }  
}
```

```
/**
 * Implementation of basic logging with level. Note that this method
 * will simply forward the call to the log() function supporting the
 * zone parameter. The method will simply use the default zone.
 */
public void log(int logLevel, String logMessage) {
    this.log(DEFAULT_LOGZONE, logLevel, logMessage);
}
```

Section 7. Automate with Eclipse

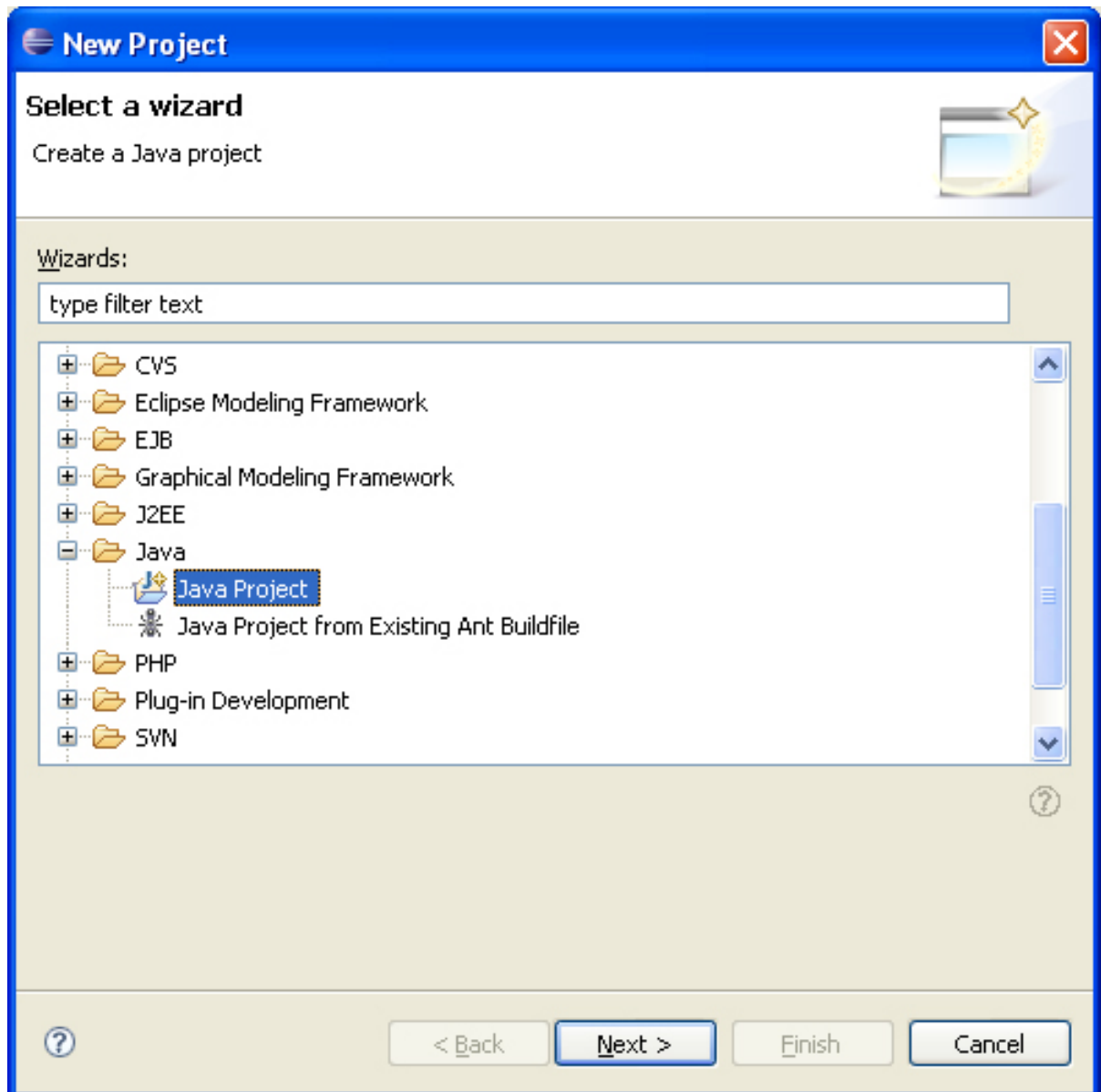
It is agreed, there must be an easier way to develop a UDR by utilizing modern tools like Eclipse. First, create a project within Eclipse, and then define some actions within the Integrated Development Environment (IDE) using external tools.

Create the project within Eclipse

Create a new project

Start Eclipse and select your workspace if applicable. My workspace is available in the C:\UDRProject\workspace directory. You should come quickly to the Welcome screen. Close the Welcome screen. Go to **File > New > Project**.

Figure 2. Create a new Java project



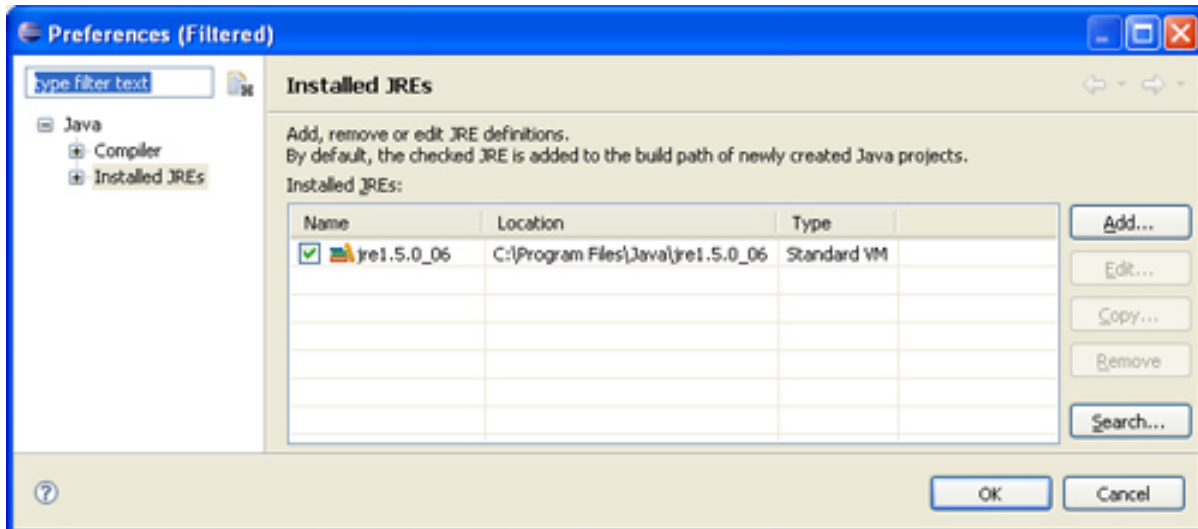
Click **Next**. Enter your project name.

Advanced Note: I usually name my projects like my root packages, which in this case will be net.jgp.lab.udr.pricechanger.

Configure Eclipse for Java 1.4

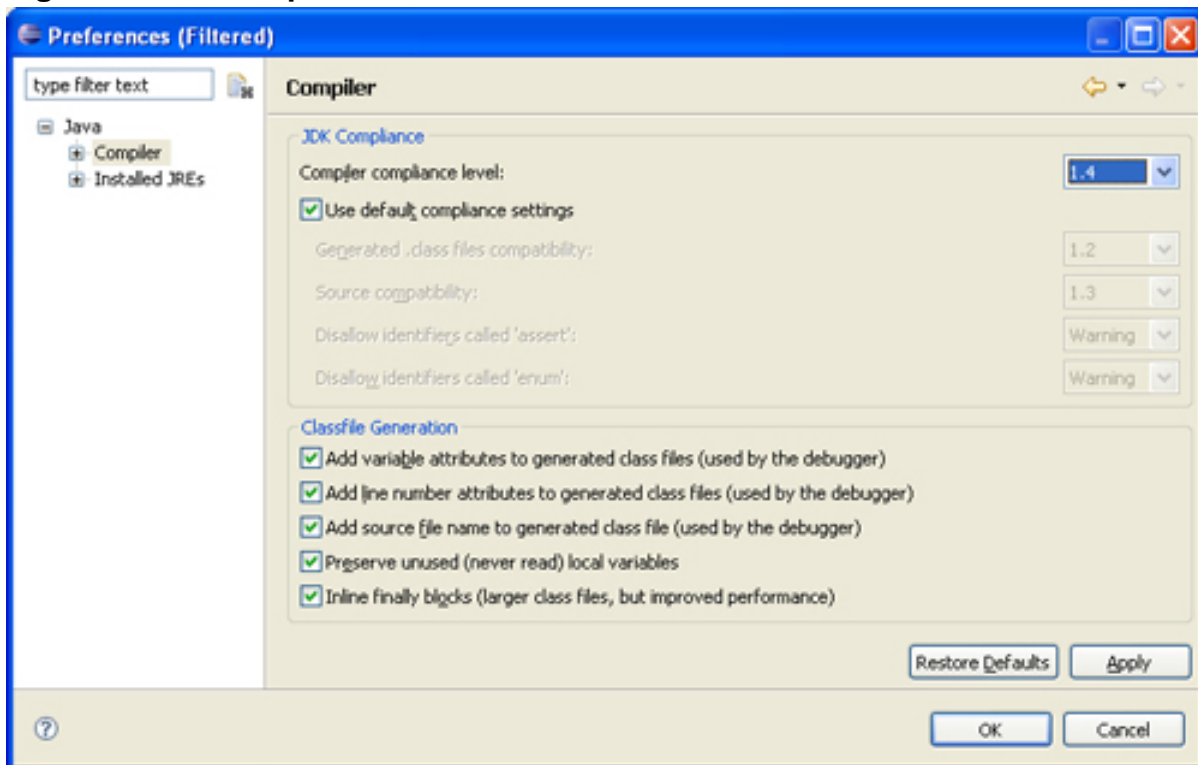
The JVM embedded in IDS is Version 1.4.2. This means that you need to provide it with a Java 1.4 byte code (the result of the Java compilation). Although you are using Java 5, you can produce Java 1.4 compliant byte code. Click **Configure JREs....** You get a Preferences panel, as shown in Figure 3.

Figure 3. Preferences panel



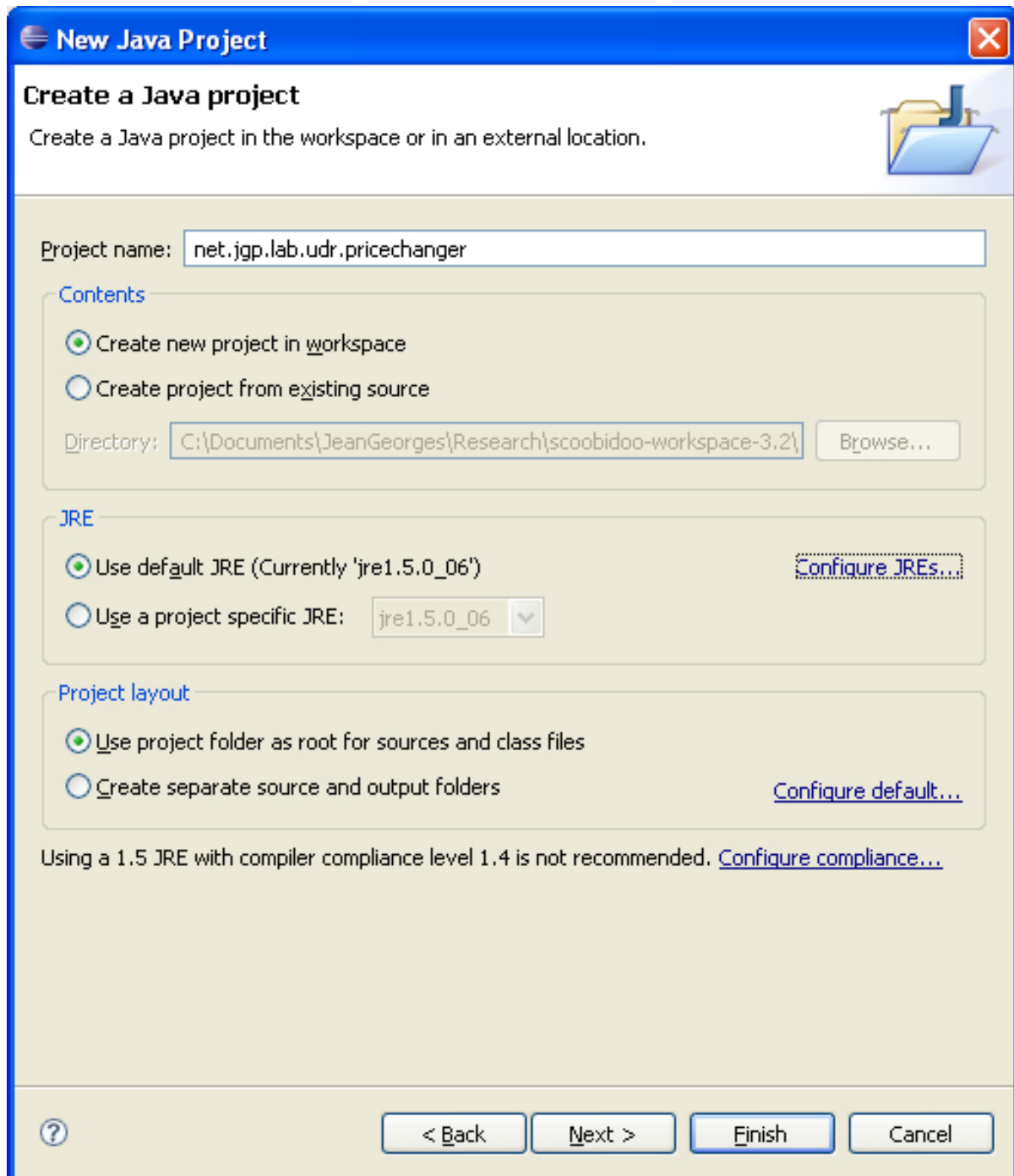
Click **Compiler**. In the Compiler compliance level drop-down, choose **1.4**. Your preferences should look like Figure 4.

Figure 4. JDK compliance level



Click **Ok**. Confirm by clicking **Yes**.

Figure 5. New Java project



Click **Finish**.

Add the krakatoa libraries

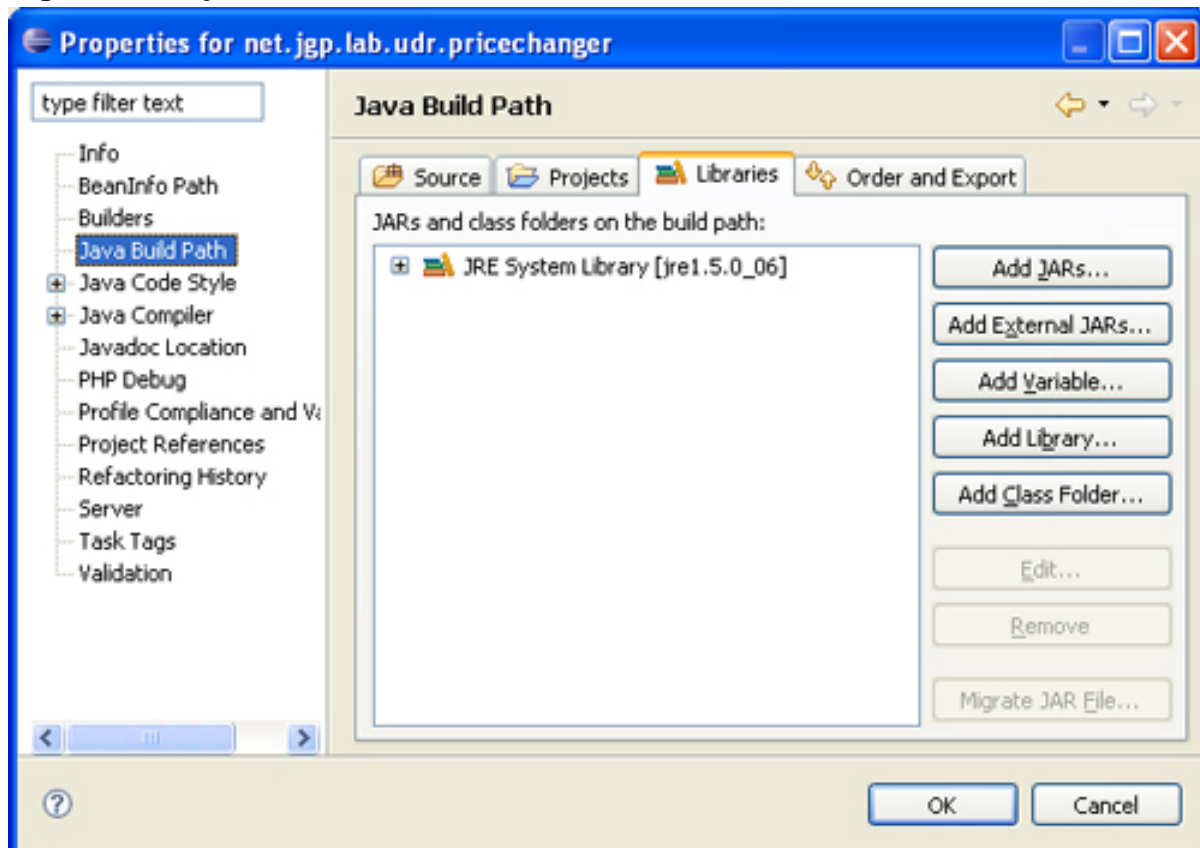
Before you can add your code to the project, you need to add the Informix libraries required for the development of your UDR.

Warning: You need to use the engine's library, as found in the krakatoa subdirectory; not the one coming from the JDBC driver.

There are several ways of adding libraries to an Eclipse project. It is recommend that you use one based on variables. This is easier when you share projects or work in a team where installation directories may differ.

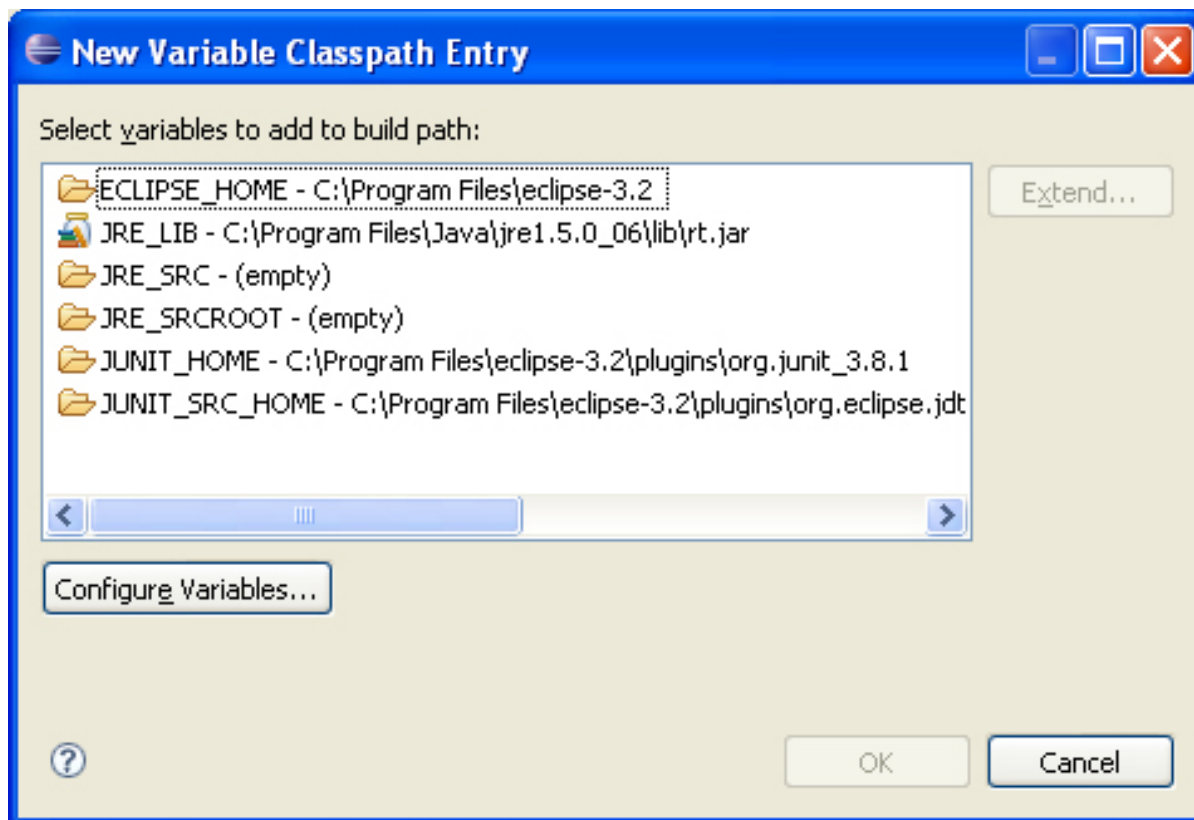
Right-click on your project in the Package Explorer, select the last element of the contextual menu, **Properties**. Highlight **Java Build Path**, and select the Libraries tab. You should see something similar to Figure 6.

Figure 6. Project libraries



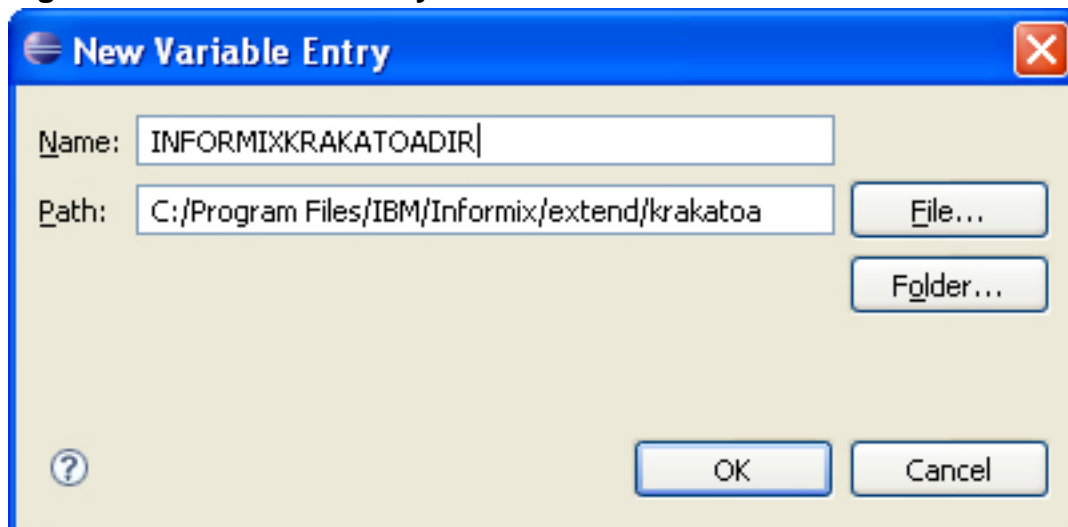
Don't click on Add JARs, select **Add Variable**.

Figure 7. Project libraries



Click **Configure Variables > New**. Enter a name for your variable (INFORMIXKRAKATOADIR is used in this article), then click **Folder** and point to %INFORMIDIR%\extend\krakatoa. Do not use an environment variable, use the full path, like C:/Program Files/IBM/Informix/extend/krakatoa. It should look like Figure 8.

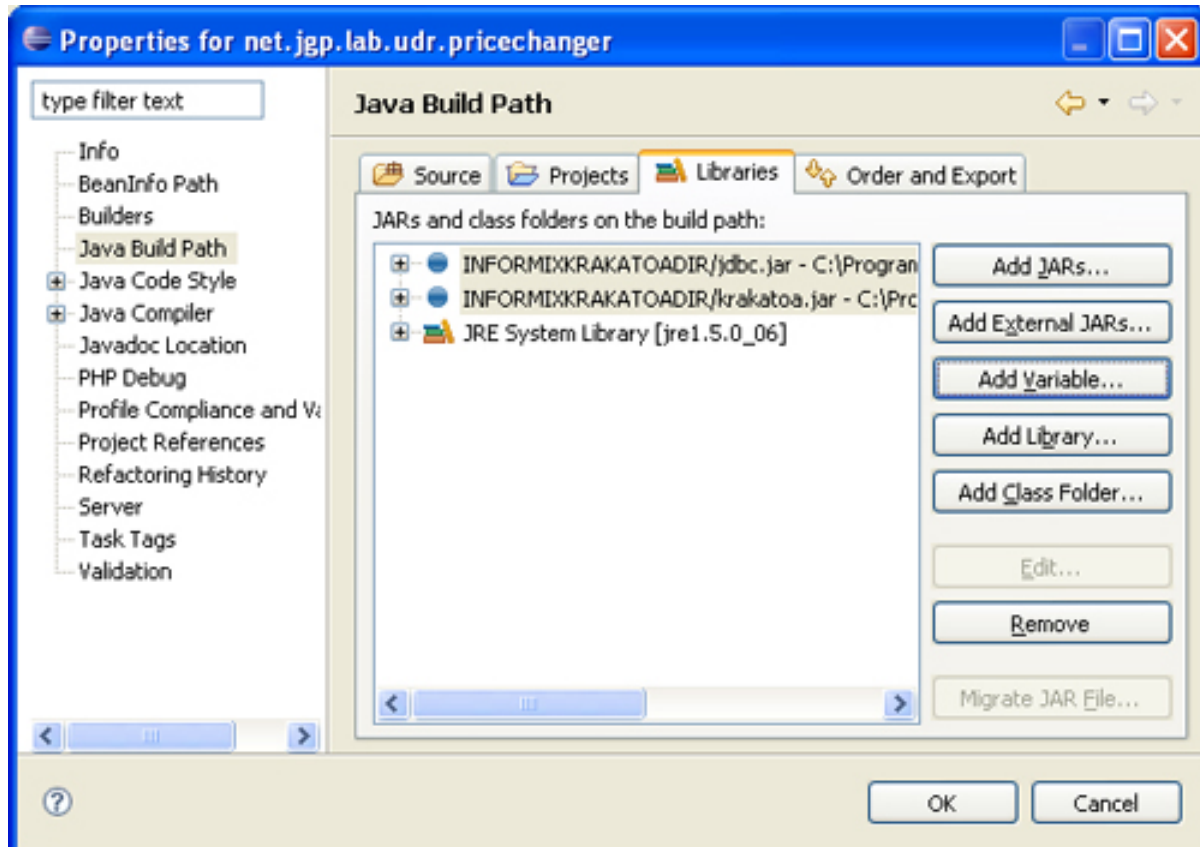
Figure 8. New variable entry



Double-click **OK**. In the New Variable Classpath Entry, select your new entry

(INFORMIXKRAKATOADIR in this example), and click **Extend**. Select **jdbc.jar** and **krakatoa.jar** in the list, and click **OK** (you can select both libraries by pressing the Ctrl key). Click **OK**. You should see something similar to Figure 9.

Figure 9. Project properties are now set

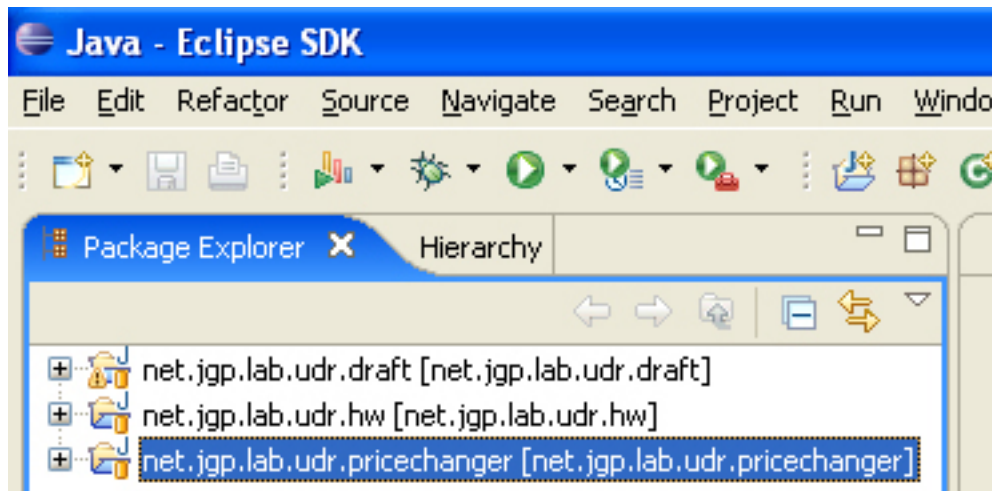


Click **OK**. You are now ready to code.

Add Code

Now that your project is ready, you can add the source code. In the Package Explorer, highlight the project (shown in Figure 10).

Figure 10. Highlight project



Go to **File > New > Class**. The package used here is `net.jgp.lab.udr.pricechanger`. The class is the same as before: `PriceChange`. Before clicking **Finish**, check that it looks like Figure 11.

Figure 11. Create the PriceChange class

New Java Class

Java Class
Create a new Java class.

Source folder:

Package:

Enclosing type:

Name:

Modifiers: public default private protected
 abstract final static

Superclass:

Interfaces:

Which method stubs would you like to create?

public static void main(String[] args)

Constructors from superclass

Inherited abstract methods

Do you want to add comments as configured in the [properties](#) of the current project?

Generate comments

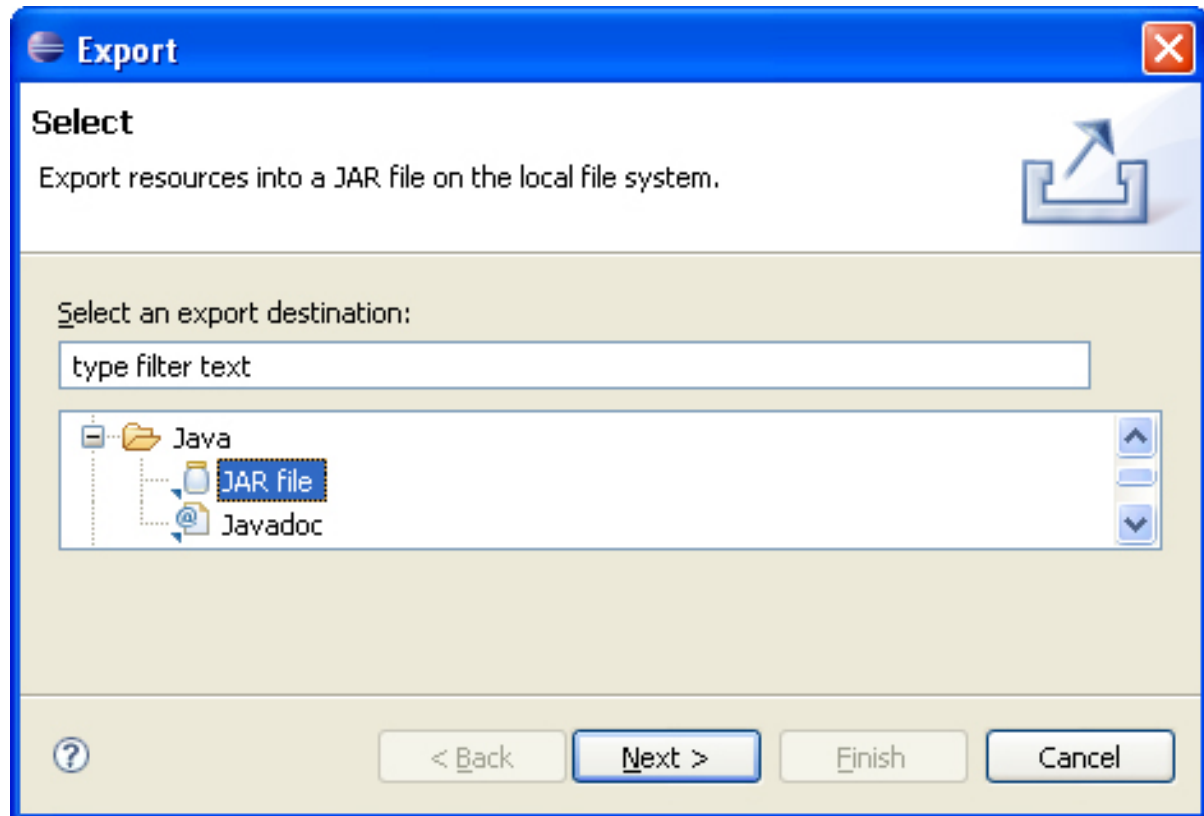
You can repeat the operation for the two other classes; PriceChangerGeneric and SuperStoresConnection. You can then copy (or type) the source from the PriceChanger.java, PriceChangerGeneric.java, and SuperStoresConnection.java files.

Define actions

Package the files

The first action you need to define is how to package your class files in a JAR. Eclipse provides a convenient wizard to build the archive. Right-click on the project's name, and click **Export**. Select **JAR file** in the Java folder, as shown in Figure 12.

Figure 12. Export

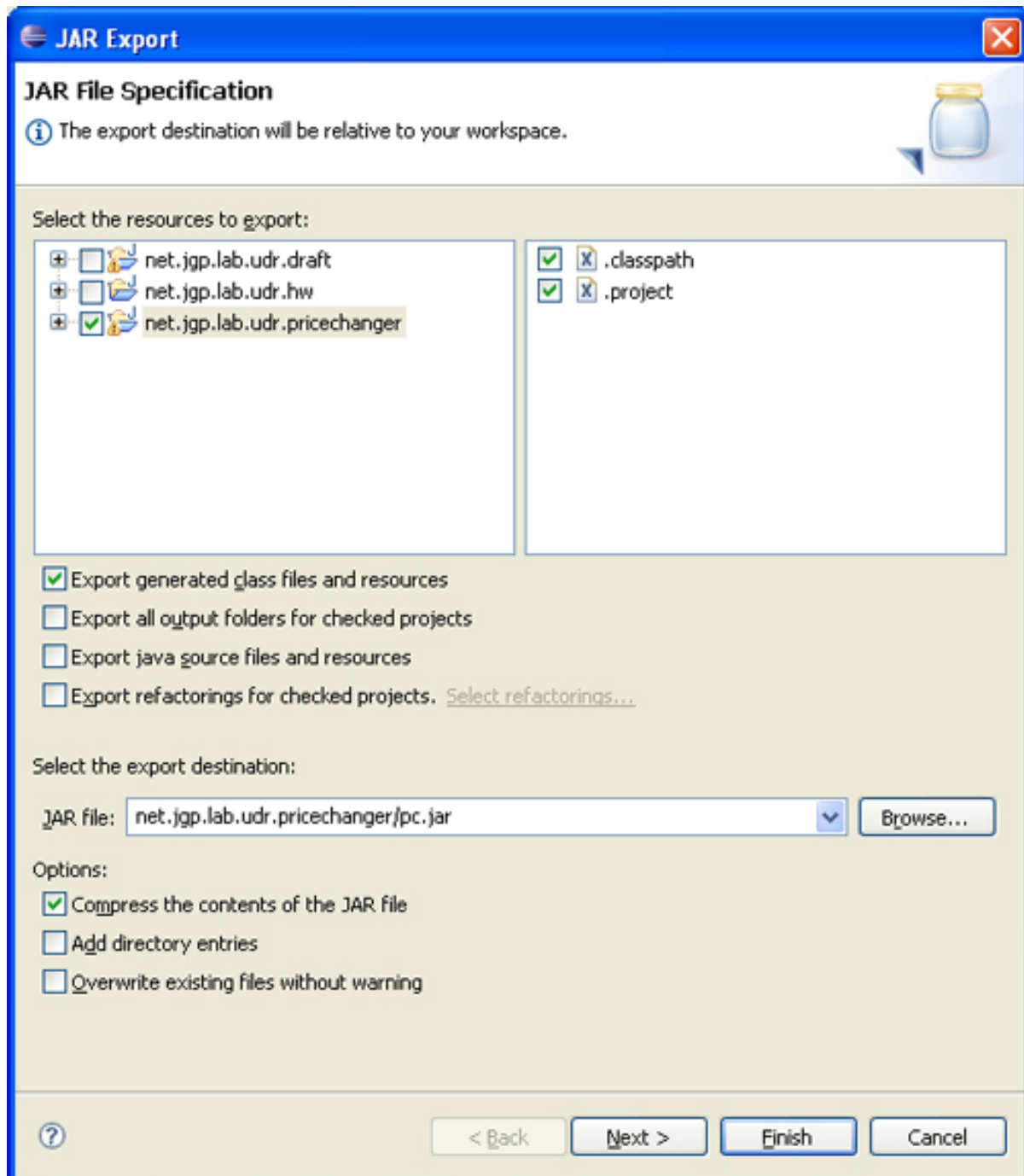


You need to specify a few parameters:

- **What:** Leave the default (you can tune that later if you wish). Be sure to have **Export generated class files and resources** checked, and leave all the other check boxes unchecked.
- **Where:** The project directory has been selected, naming it pc.jar. As a result, the JAR file is saved relatively to the workspace directory in net.jsp.lab.udr.pricechanger/pc.jar.

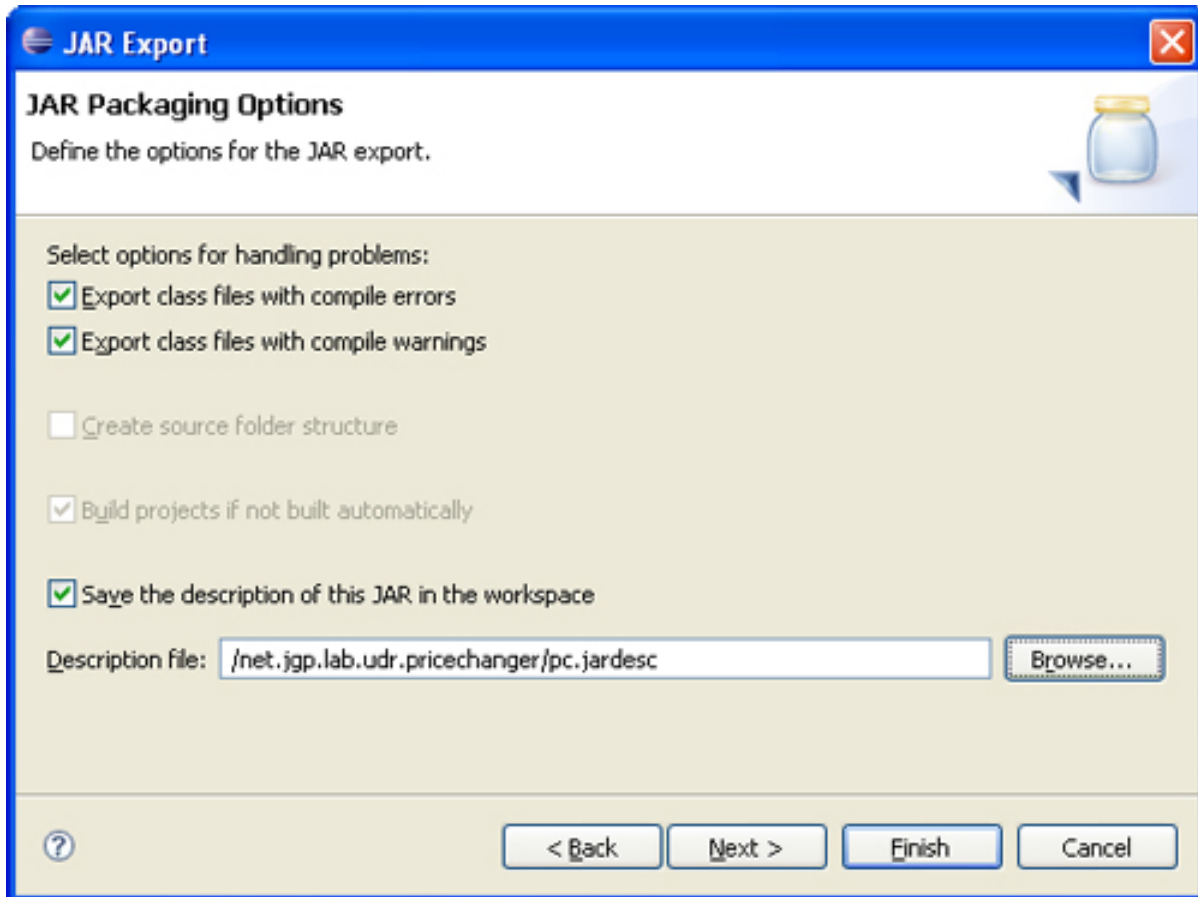
Before clicking **Next**, check that it looks like Figure 13.

Figure 13. JAR export step 1



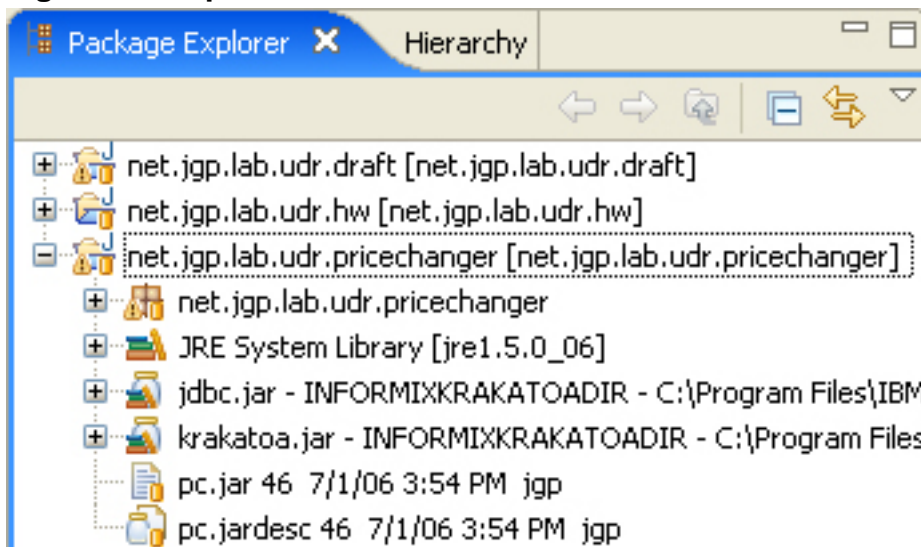
Be sure to save the description in your project, as displayed in Figure 14. The Description file has a jardesc extension. The description is saved in /net.jgp.lab.udr.pricechanger/pc.jardesc. It allows you to reuse it in the future.

Figure 14. JAR export step 2



Click **Finish**. As illustrated in Figure 15, you can see pc.jardesc and pc.jar in the Project Explorer.

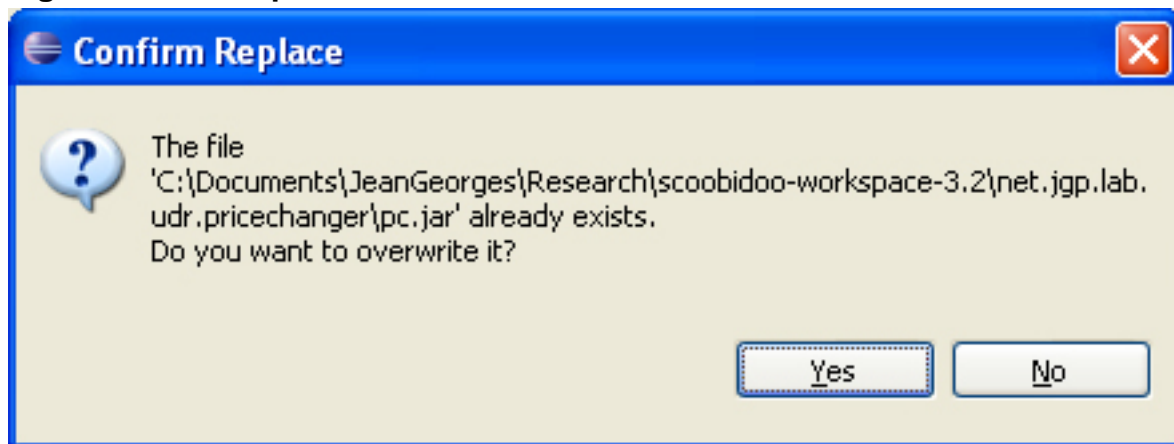
Figure 15. Export result



Refresh the package

Now, every time you have modified your source files and saved them (Eclipse automatically compiles them), you can right-click on pc.jardesc and select **Create JAR**. JAR is built automatically. Eclipse does prompt you to replace the existing JAR file (as shown in Figure 16).

Figure 16. JAR replacement

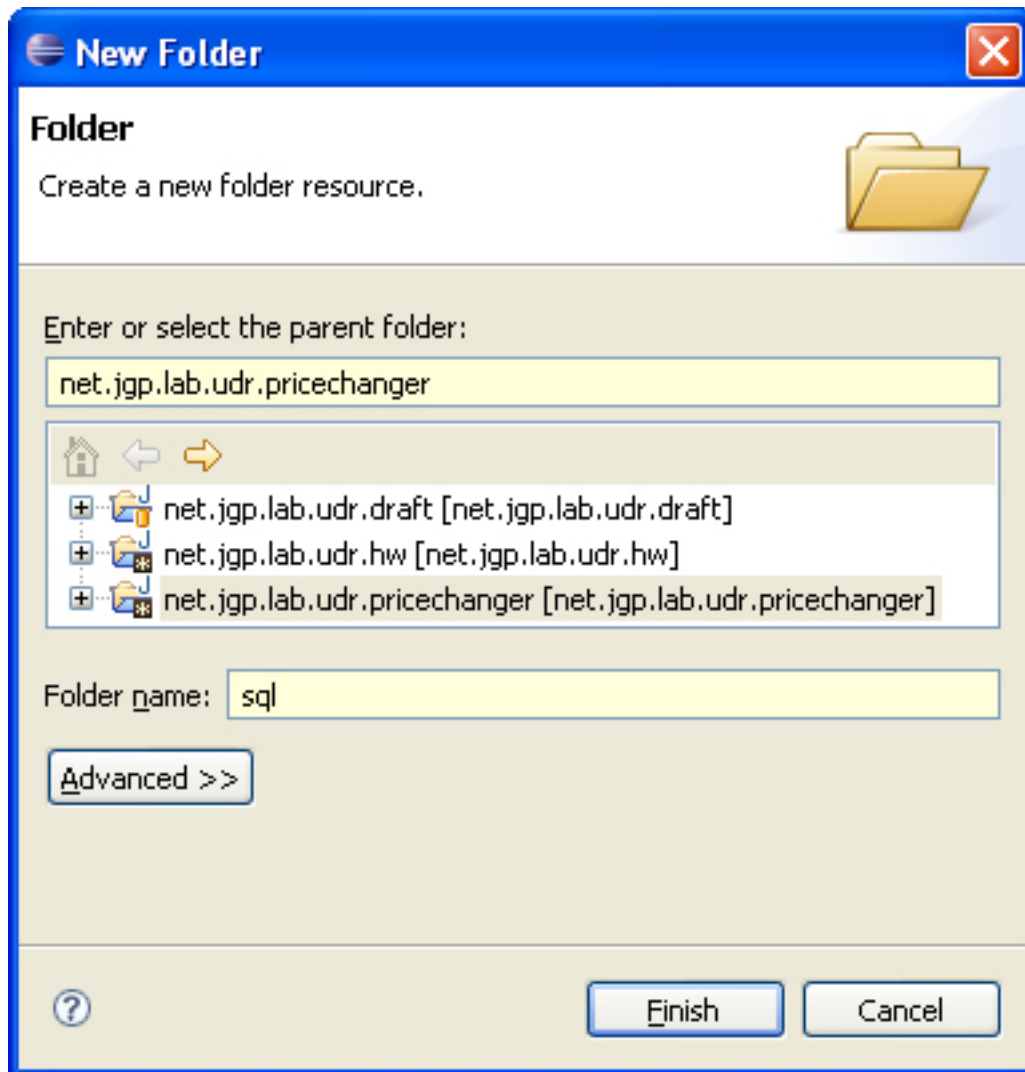


Install the package in the engine

Now you need to install pc.jar in the engine. Simply ask Eclipse to run DB-Access with the install.sql script you used earlier.

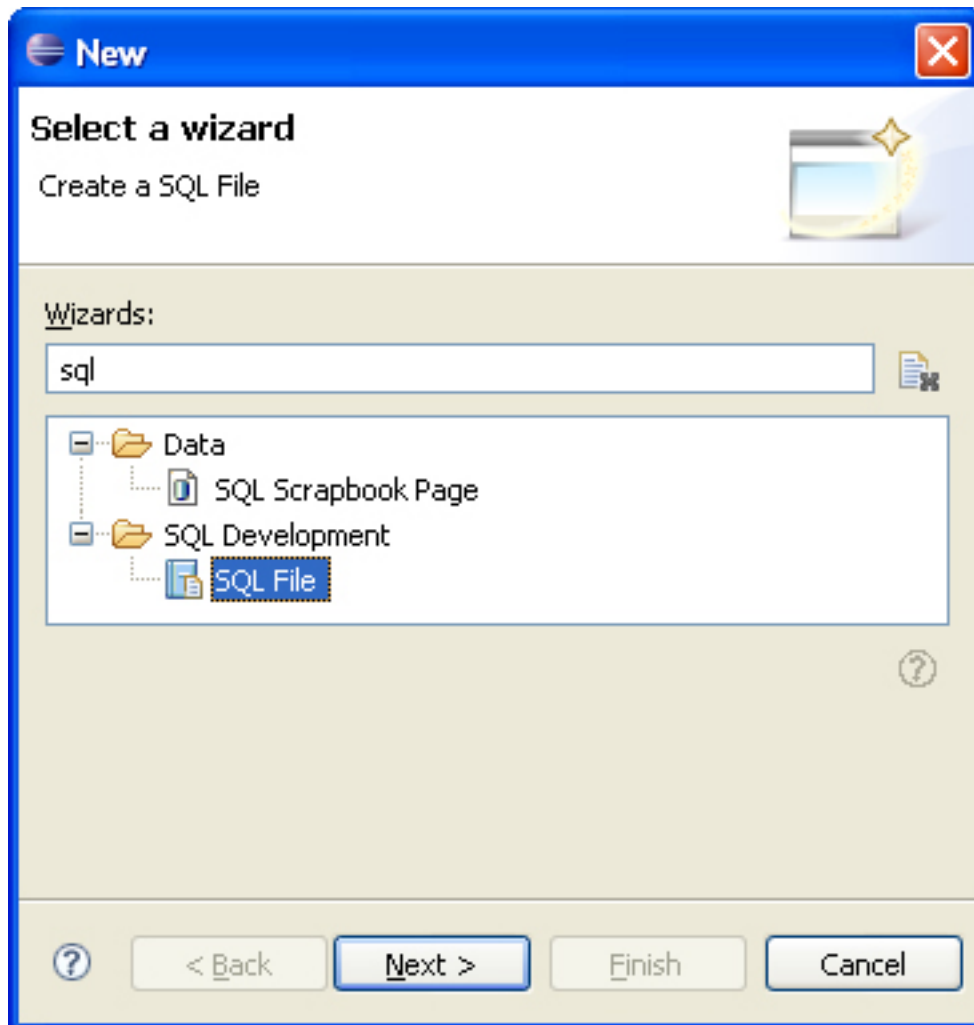
Integrate your SQL scripts in Eclipse. To organize things a little, first create an sql directory under the root project. Right-click on the project (net.jgp.lab.udr.pricechanger), and select **New > Folder**. In the New Folder dialog box, enter sql, and click **Finish** (as shown in Figure 17).

Figure 17. New SQL folder



Right-click on the sql directory, and select **New > Other**. To ease navigation in the New dialog box, you can type sql in the filter area. Select the **SQL File** under SQL Development, and click **Next**.

Figure 18. New SQL file



Check that the sql folder is highlighted. Type **install.sql** as the file name, and click **Finish**.

Figure 19. New install.sql



In the editor, type the SQL install script (as shown in Listing 14).

Listing 14. SQL install script

```
CONNECT TO 'superstores_demo' USER 'informix' USING 'informix';

BEGIN WORK;
EXECUTE PROCEDURE sqlj.install_jar(
  'file:\C:\UDRProject\workspace\net.jgp.lab.udr.pricechanger\pc.jar',
  'pc_jar',
  0);
CREATE FUNCTION changePriceByPercent(
  effectiveDate CHAR(10),
  percentChange INTEGER)
RETURNS INTEGER
EXTERNAL NAME
  'pc_jar:net.jgp.lab.udr.pricechanger.PriceChanger.changePriceByPercent'
```

```
LANGUAGE JAVA;
GRANT EXECUTE ON FUNCTION changePriceByPercent(CHAR, INTEGER) TO ALL;
COMMIT WORK;
```

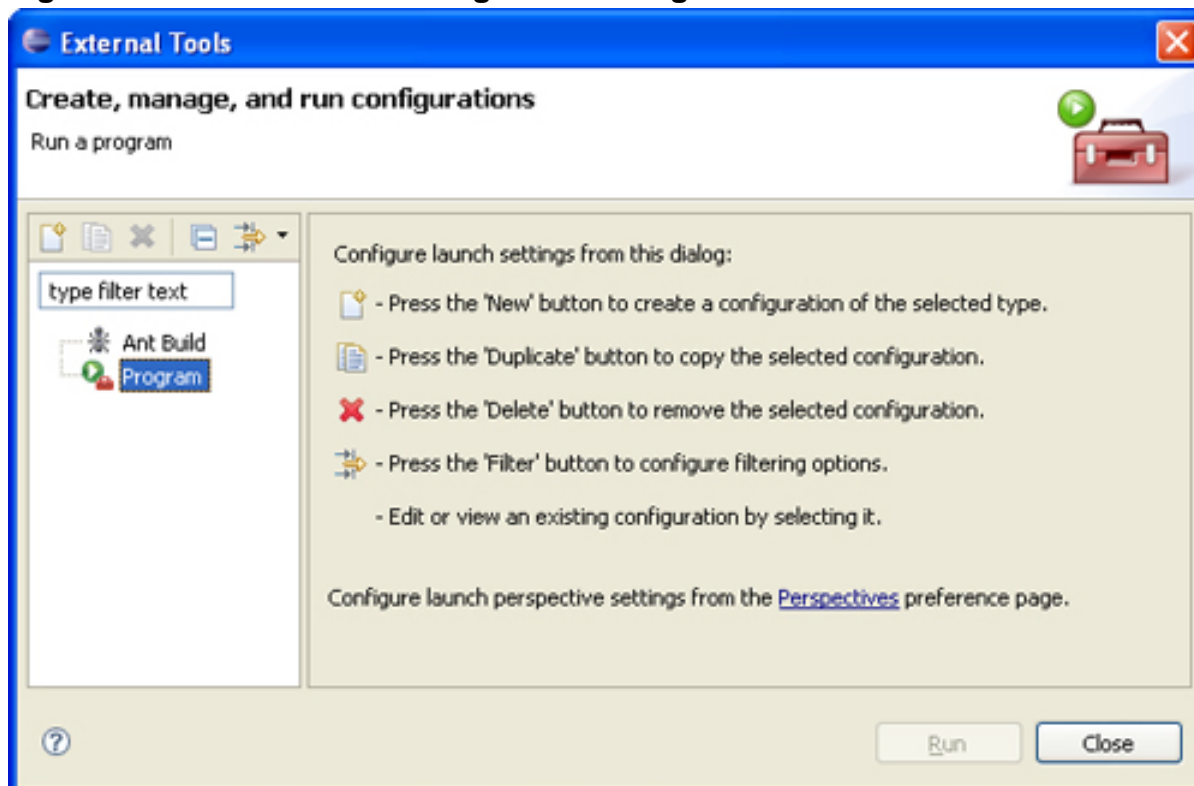
Note that the install script is very similar to the one created previously. As you are going to run it without being previously connected to the database, you simply need to add a connect statement:

```
CONNECT TO 'superstores_demo' USER 'informix' USING 'informix';
```

The path is also different as you are using the path in the workspace.

Click **Run > External Tools > Select Program**. You should see something similar to Figure 20.

Figure 20. External tools management dialog box



As indicated, click **New**.

Table 1. Set up the Install UDR external tool

Tab	Field	Comment	Default Value
	Name	Name of the tool.	Install UDR
Main	Location	Location of DB-Access.	C:\Program Files\IBM\Informix\bin\dbaccess.exe

Main	Arguments	Arguments to DB-Access. Note that there is a space between the dash and the dollar sign.	- \${project_loc}/sql/install.sql
Environment	Variable	Value for INFORMIXDIR.	C:\PROGRA~1\IBM\Informix
Environment	Variable	Value for INFORMIXSERVER.	ol_scoobidoo

To add an environment variable, click on the Environment tab, then click **New**, and type the name and then the value. Click **OK**. Do it for INFORMIXDIR and INFORMIXSERVER. You should see something similar to Figures 21 and 22.

Figure 21. External tools configuration: Main tab

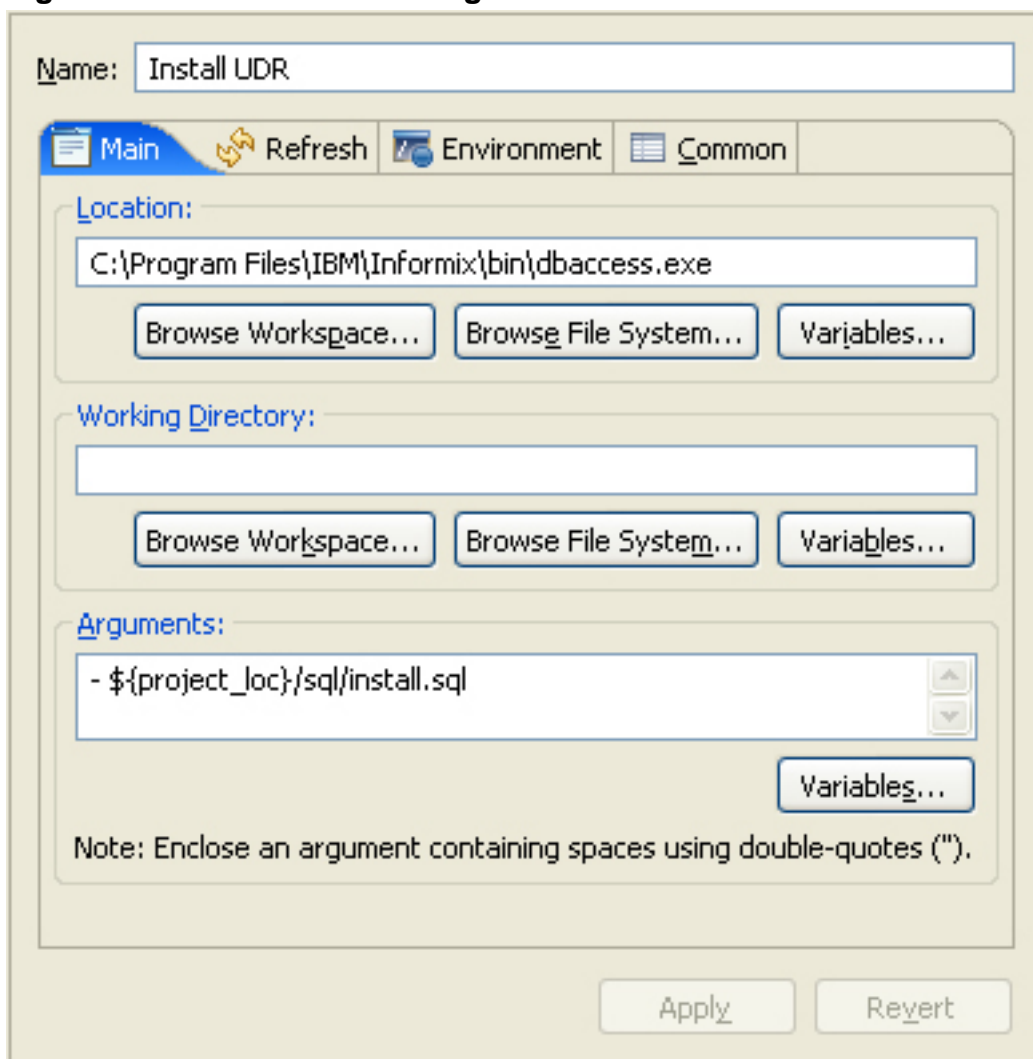
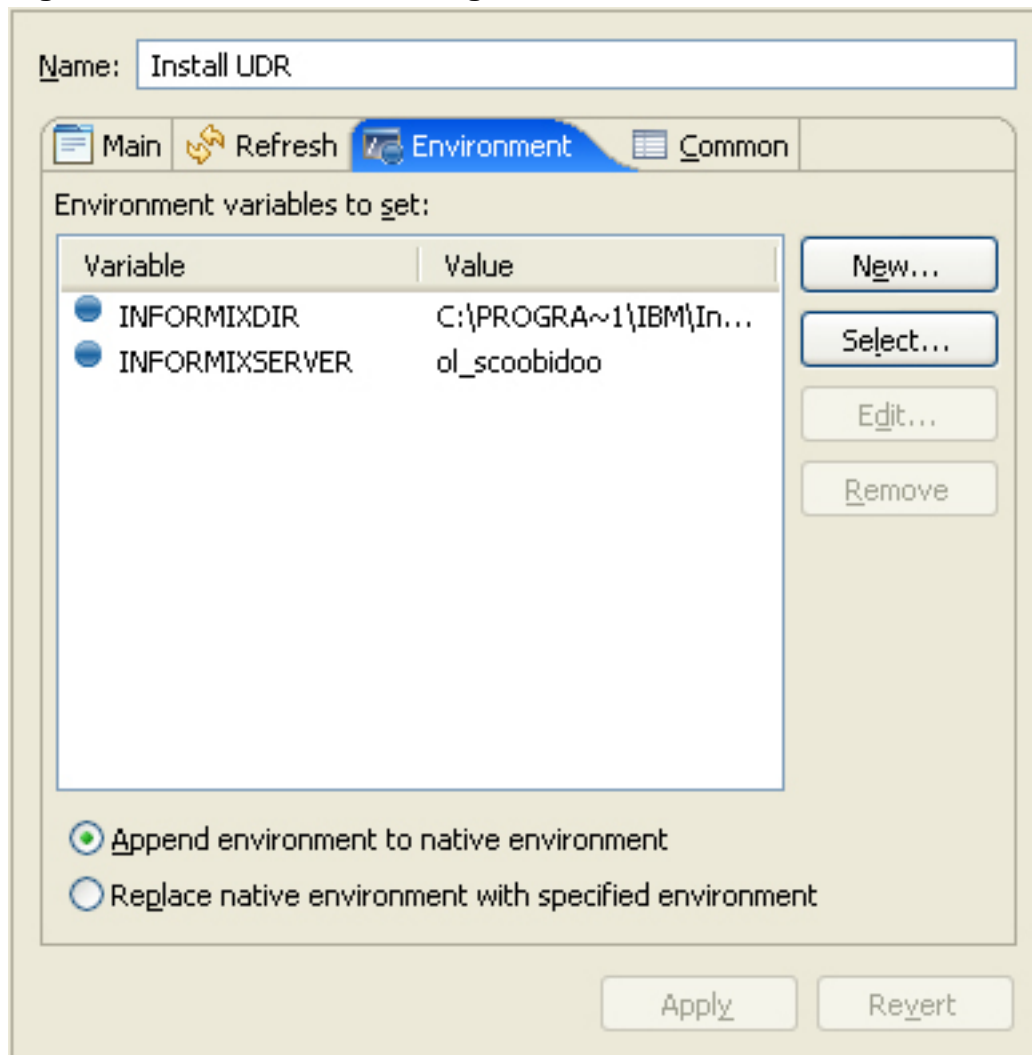


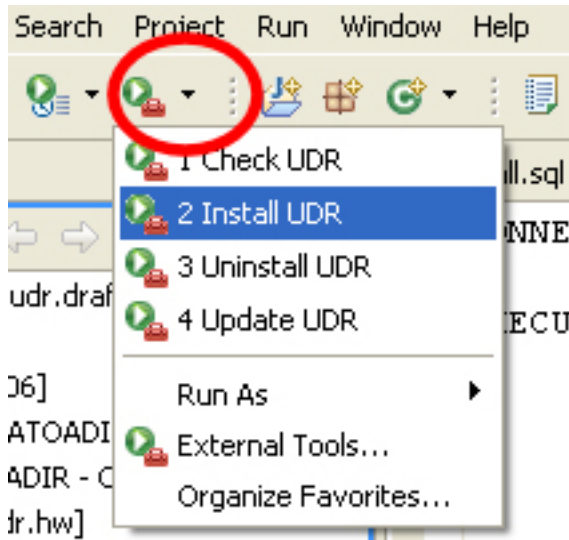
Figure 22. External tools configuration: Environment tab

The actions you have defined are dependent on the current project. You have two projects, `net.jsp.lab.udr.pricechanger` and `net.jsp.lab.udr.hw`. If your current project is `net.jsp.lab.udr.hw`, then, when you run the `Install UDR` action, you will use `install.sql` from `net.jsp.lab.udr.hw`, which will install `hw.jar` and the `getHelloWorld` function.

To run it, click **Run**.

Once you have run it, it stays in the list of recent run configurations. To run it later, click **Run > External Tools**, if it is in the list of recent runs, it will be available there. Otherwise, click **External Tools...**. Alternatively, you can run it from the toolbar, as displayed in Figure 23.

Figure 23. Run external tools from the toolbar

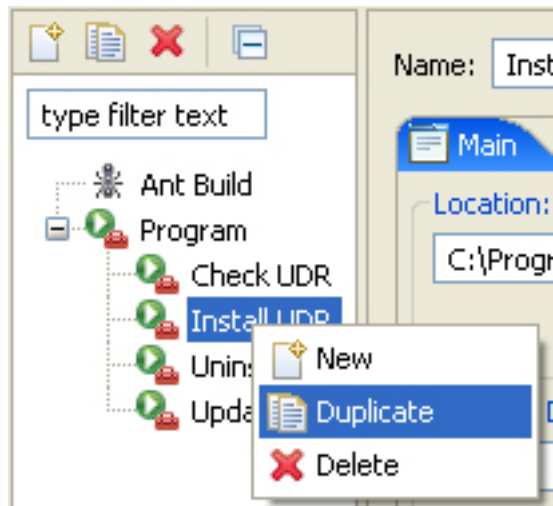


Update, check, and uninstall the package

You can now define the actions for checking, uninstalling, and updating the UDR. You can either duplicate the Install UDR entry or reproduce and process it for each external tool. Duplicating is the recommended method.

Click **Run > External Tools > External Tools...**. In the left column, you have Install UDR (under Program), right-click on it and select **Duplicate**, as shown in Figure 24.

Figure 24. Duplicate an external tool



Change the name to **Uninstall UDR**. In the arguments field, simply change install.sql with **uninstall.sql**. Click **Apply**.

You can reproduce the operation for Check UDR with check.sql and Update UDR with update.sql. Here are the required SQL scripts for the various operations:

Listing 15. SQL uninstall script

```
CONNECT TO 'superstores_demo' USER 'informix' USING 'informix';

BEGIN WORK;
DROP FUNCTION changePriceByPercent;
EXECUTE PROCEDURE sqlj.remove_jar('pc_jar', 0);
COMMIT WORK;
```

Listing 16. SQL check / execute script

```
CONNECT TO 'superstores_demo' USER 'informix' USING 'informix';

EXECUTE FUNCTION changePriceByPercent('2005-05-01', 10);
```

Listing 17. SQL update script

```
CONNECT TO 'superstores_demo' USER 'informix' USING 'informix';

BEGIN WORK;
EXECUTE PROCEDURE sqlj.replace_jar(
  'file:\C:\UDRProject\workspace\net.jgp.lab.udr.pricechanger\pc.jar',
  'pc_jar');
COMMIT WORK;
```

Section 8. Conclusion

Java UDRs are a good option to consider when you have to create server-based, database-intensive applications.

Reduced network traffic and the use of a portable and fully object-oriented language are some of the advantages of Java UDRs.

Setting up J/Foundation in IDS requires the availability of a JVM and the configuration of several onconfig parameters.

It is possible to create simple, single class UDRs as well as more complicated ones, consisting of multiple classes and an object-oriented application.

J/Foundation provides zone- and level-based tracing capabilities useful for debugging during development and troubleshooting during production.

You need to be careful and meticulous about closing and freeing your statements and result sets in Java UDRs in order to avoid memory problems.

Finally and most important, by the time you've run all the examples, you'll notice that your baseball glove, which used to cost \$30, is now well over \$60. If pricing could be that easy.

Section 9. Appendix

Creating a Smart Blob Space (sbspace)

You have been advised do the following steps under the informix user.

When running:

```
onspaces -c -S sbspace1 -p
C:\IFMXDATA\ol_scoobidoo\sbspace1_dat.000
-o 0 -s 10000
```

The result should be:

```
Verifying physical disk space, please wait ...
Space successfully added.

** WARNING ** A level 0 archive of Root DBSpace will need to
be done.
```

If you get:

```
The file C:\IFMXDATA\ol_scoobidoo\sbspace1_dat.000 does not
exist.
```

Create the file first, by calling:

```
copy con C:\IFMXDATA\ol_scoobidoo\sbspace1_dat.000
```

Press **Ctrl+Z** (or **F6**) then **Enter** to exit. The file is created and you can call the onspaces utility again.

Install the Super Stores database

You are advised do the following steps under the informix user.

This example relies on the Super Store database, which is not installed by default on the system.

First you need to create an empty file that will contain the Smart Blob Space.

```
copy con C:\IFMXDATA\ol_scoobidoo\s9_sbSPc_dat.000
```

Press **Ctrl+Z** (or **F6**) then **Enter** to exit. You then need to run onspaces to create the Smart Blob Space.

```
onspaces -c -S s9_sbSPc -g 2 -p C:\IFMXDATA\ol_scoobidoo\s9_sbSPc_dat.000  
-o 0 -s 2000
```

Finally, you can install it:

```
dbaccessdemo_ud -log-dbSPc dbS_scoobidoo superstores_demo
```

Restart the Informix engine from the command line.

It is sometimes easier to do things through the command line rather than through clicking. Here is a way to restart your IDS instance without using the mouse.

To stop the engine, type:

```
net stop "Informix IDS - ol_scoobidoo"
```

You should get:

```
The Informix IDS - ol_scoobidoo service is stopping...  
The Informix IDS - ol_scoobidoo service was stopped successfully.
```

To start the engine, type:

```
net start "Informix IDS - ol_scoobidoo"
```

You should get:

```
The Informix IDS - ol_scoobidoo service is starting.
```

The Informix IDS - ol_scoobidoo service was started successfully.

Listing the Virtual Processors

To be sure that your JVP is running, you can use the onstat command.

```
onstat -g glo
```

This is a typical result when the JVP is not running.

```
IBM Informix Dynamic Server Version 10.00.TC4
-- On-Line -- Up 1 days 04:57:56 -- 21568 Kbytes

MT global info:
sessions threads vps lngspins
0 14 8 11

total: sched calls thread switches yield 0 yield n yield forever
per sec: 0 6343727 650771 5692983 208988 167929
0 0 0 0 0 0

Virtual processor summary:
class vps usercpu syscpu total
cpu 1 2.92 1.00 3.92
aio 1 0.00 0.00 0.00
lio 1 0.00 0.00 0.00
pio 1 0.00 0.00 0.00
adm 1 0.25 0.20 0.45
soc 2 0.00 0.00 0.00
msc 1 0.00 0.07 0.07
total 8 3.17 1.27 4.43

Individual virtual processors:
vp pid class usercpu syscpu total
1 2472 cpu 2.92 1.00 3.92
2 3576 adm 0.25 0.20 0.45
3 3580 lio 0.00 0.00 0.00
4 3680 pio 0.00 0.00 0.00
5 3716 aio 0.00 0.00 0.00
6 3720 msc 0.00 0.07 0.07
7 3724 soc 0.00 0.00 0.00
8 3728 soc 0.00 0.00 0.00
tot 3.17 1.27 4.43
```

This is a typical result when the JVP is running.

```
IBM Informix Dynamic Server Version 10.00.TC4
-- On-Line -- Up 00:01:27 -- 21568 Kbytes

MT global info:
sessions threads vps lngspins
0 14 9 0

total: sched calls thread switches yield 0 yield n yield forever
per sec: 0 4829332 210947 4618410 129 105396
0 0 0 0 0 0
```

```
Virtual processor summary:
class      vps      usercpu   syscpu    total
cpu        1        2.18      0.72      2.90
aio        1        0.00      0.00      0.00
lio        1        0.00      0.00      0.00
pio        1        0.00      0.00      0.00
adm        1        0.00      0.00      0.00
soc        2        0.00      0.00      0.00
msc        1        0.00      0.00      0.00
jvp       1        0.00      0.00      0.00
total      9        2.18      0.72      2.90
```

```
Individual virtual processors:
vp  pid      class      usercpu   syscpu    total
1   2888     cpu        2.18      0.72      2.90
2   2260     adm        0.00      0.00      0.00
3   6052     jvp        0.00      0.00      0.00
4   3784     lio        0.00      0.00      0.00
5   2316     pio        0.00      0.00      0.00
6   5452     aio        0.00      0.00      0.00
7   4876     msc        0.00      0.00      0.00
8   2928     soc        0.00      0.00      0.00
9   3644     soc        0.00      0.00      0.00
    tot        2.18      0.72      2.90
```

Add your JDK in the path

If you want to use the JDK on the command line, you should ensure that javac (the Java compiler) is in your system path.

To check that it is available, start a command prompt and type javac. If you get the following error message, continue reading.

```
'javac' is not recognized as an internal or external command,
operable program or batch file.
```

Locate javac on your system. Typically it is in C:\Program Files\Java\jdk1.5.0_05\bin. The minor version might defer on your system.

Click **Start > Control Panel**. Double-click on the **System** icon. Click on the **Advanced** tab, and then click **Environment Variables**. In the system variables block, find and select the Path entry. Click **Edit**.

Before typing anything, ensure that your cursor is at the end of the field. Press either the **right arrow** key or the **End** key on your keyboard.

Type:

```
;C:\Program Files\Java\jdk1.5.0_05\bin
```

Do not forget the semicolon.

Click **OK** three times.

To check that it works, open a new command prompt and type javac, you should get its usage.

Troubleshooting – Error #25785 (Informix error)

If you get:

```
25785: Cannot create external routine (gethelloworld) without the EXTEND role.
```

It means that you cannot create an external function or a procedure without the privileges of the EXTEND role. You can either grant the role to the user or log in as the Informix user. DB-Access lets you connect as a specific user: Connection, Connect, Select your database server (here ol_scoobidoo), enter informix as the username and then your password, and you can select the database.

Troubleshooting – Error #46002 (SQL State)

If you get:

```
(46002) - Attempt to remove or alter path for non-existing jar:  
(superstores_demo.informix.hw_jar).
```

Symptom: You are probably trying to update a JAR, which does not exist (like launching the update script before the install script or just after the uninstall script).

Solution: Install the JAR first.

If you get:

```
(46002) - Attempt to install an existing jar: (superstores_demo.informix.pc_jar).
```

Symptom: You are probably running the install script but the JAR is already installed.

Solution: Uninstall the JAR first or use the update script.

Troubleshooting – Error #46003 (SQL State)

If you get:

```
(46003) - Invalid jar removal. All dependent UDRs not dropped.
```

You will have a hard time finding where the error is documented. It is a SQL State error: “46003 Java DDL - Invalid class deletion”.

I originally made a typo when I installed the Hello World function and called it “getHellowWorld” (notice the two ‘w’s). The function was installed OK (with the typo), was running OK (with the typo) but uninstallation did not work as I was dropping the function without the typo, and IDS did not want to remove the JAR file, and thus the error #46003.

We used AGS’ ServerStudio and it listed all the functions available to me and we discovered the wrongly-spelled function in the database.

Troubleshooting – Function not found

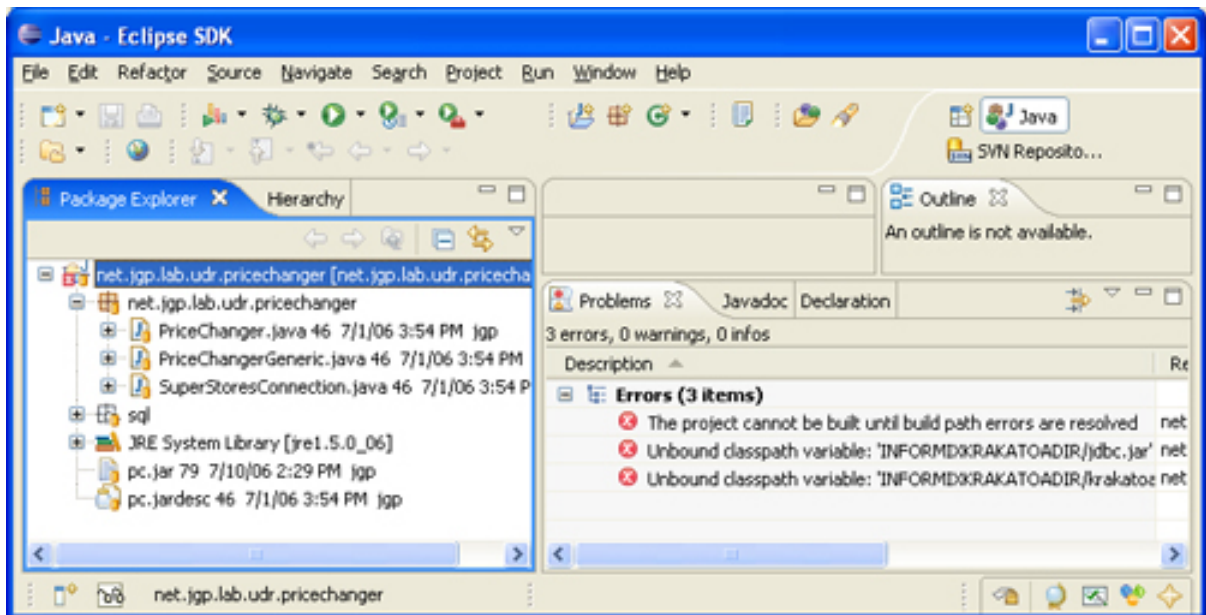
Another common problem is that the function is not found by the engine, simply because the user does not have the rights to run it. Think about granting the right to your users.

```
GRANT EXECUTE ON FUNCTION getHelloWorld() TO ALL;
```

Problems when installing the source code

You may encounter some problem when installing the source code on your system, as illustrated in Figure 25.

Figure 25. Problem during integration of the project



The error Unbound classpath variable: 'INFORMIXKRAKATOADIR/jdbc.jar' in project net.jsp.lab.udr.pricechanger, simply means that your INFORMIXKRAKATOADIR variable is not defined within your workspace. To solve the problem, refer to an earlier section of the tutorial where the INFORMIXKRAKATOADIR variable was set.

Resources

Learn

- ["Getting Started with IBM Informix J/Foundation on Linux"](#) (developerWorks, September 2002): Find step-by-step directions for configuring IBM IDS J/Foundation for a 9.3x server on the Linux platform.
- [DB2 Information Center](#): Find a list of valid SQLSTATE error codes.
- [J/Foundation Developer's Guide, Version 10.0. G251-2291-00](#): Discover more information about how to write UDRs in Java for IDS with J/Foundation.e.
- [IBM Informix Dynamic Server Administrator's Guide, Version 10.0. G251-2267-02](#): Use this user guide and reference manual to find features of IDS.
- [IBM Informix Dynamic Server Administrator's Reference, Version 10.0. G251-2268-01](#): Find reference material for IDS.
- [IBM Informix DB-Access User's Guide, Version 10.0. G251-2277-00](#): Discover how to use the DB-Access utility to access, modify, and retrieve data from IBM Informix databases.
- [IBM Informix Guide to SQL: Syntax, Version 10.0. G251-2284-02](#). Find detailed descriptions of the syntax for all IBM Informix SQL and SPL statements.
- Visit the [developerWorks Informix Zone](#) to read articles and tutorials and connect to other resources to expand your Informix skills.
- Learn more about Informix at the [IBM Informix Dynamic Server v10.00 Information Center](#).
- Browse the [technology bookstore](#) for books on these and other technical topics.

Get products and technologies

- Download a free trial version of [IDS](#).

Discuss

- [Participate in the discussion forum for this content](#).
- [The IIUG forums](#): Check them out to get more information and discuss this article within the Informix developer community.

About the authors

Halit (Hal) M. Maner



Halit (Hal) Maner is the President and Chief Technology Officer of M Systems International, Inc., a solutions developer based in the Research Triangle Park in North Carolina, USA. M Systems specializes in database application development and support, database administration, and network administration in the Windows, Unix, and Linux platforms. Hal is certified in Informix, DB2, and Microsoft SQL Server databases. He has a Bachelor's of Science in Automatic Control and Computer Engineering as well as an MBA, plus 22 years of experience as a programmer, DBA, software architect, and team leader.

Jean Georges Perrin



Jean Georges Perrin has been involved in software engineering for the last decade, using many development languages ranging from Informix-4GL to Java, through Visual Basic, C, C++ and PHP. Jean Georges has been working on the Web and related technologies since 1994. He has been elected to the Board of Directors of IIUG since 2002. He headed various R&D and project lead positions and lives near Strasbourg, France, with his wife and two sons.