
A JDBC persistence framework

JdbcStore Programmer's Guide

CUSTOMER

Classes **Attributes** JDBC Mappings Inheritance Keys Relationships Cataloged SQL Caching

Attributes

- Lname
- City
- Phone
- CompanyName**
- Id
- Address
- Zip
- Fname
- State

Java Class java.lang.String ▼

Java Field Name companyName

Java Getter getCompanyName

Java Setter setCompanyName

☐ Key Attribute ☒ Updateable ☐ Concurrency Field

Add Delete

Cancel Apply

Close

(c) LPC Consulting Services, Inc.

This manual was produced using *Doc-To-Help*®, by WexTech Systems, Inc.

WEXTECH

WexTech Systems, Inc.
310 Madison Avenue, Suite 905
New York, NY 10017
1-800-WEXTECH
(212) 949-9595
Fax: (212) 949-4007

Contents

Installation	5
Installing JdbcStore.....	5
Setting the CLASSPATH	5
Directory structure	5
Introduction	7
JdbcStore overview.....	7
What is JdbcStore.....	7
Sample application	7
Features.....	9
JdbcStore concepts and terminology	12
Object type	12
Model.....	13
Attribute.....	13
Relationship	13
User class	13
Persistent extension	14
Factory	14
Class generation	15
SQL generation	15
Concurrency control attribute	15
Generated keys.....	16
Cataloged (or Loading) SQLs	16
Using the Workbench	17
Overview	17
Starting the workbench	17
Working with settings	19
Connection settings	19
Directories settings	20
Class generation settings	21
SQL generation settings	22
Working with a model.....	23
Defining types from a SQL schema	23
Loading a JDBC driver.....	24
Connecting to a database	24
Closing the database connection	25
Retrieving SQL catalog information	25
Displaying table information	26
Generating object types.....	27
Working with types.....	27

Adding a type.....	28
Editing a type.....	28
Editing class information	29
Editing an attribute	29
Defining the JDBC mappings.....	31
Handling inheritance.....	32
Defining keys.....	34
Defining relationships	35
Defining cataloged SQL statements	38
Choosing a caching strategy	39
Generating classes	40
Generating SQL.....	41
Working with existing classes	42
Defining attributes	43
Obtaining version information	44

Basic Programming 46

Overview	46
Using the LPCSqlDriver	46
Loading a driver.....	46
Getting a list of loaded drivers.....	46
Displaying driver information	47
Working with models.....	47
Loading from a file.....	47
Loading from an URL	48
Using a LPCSqlOrb	48
Connecting a broker directly	48
Connecting a broker using LPCConnectionInfo.....	49
Disconnecting a broker.....	49
Transaction control.....	50
Turning autocommit on/off	50
Committing transactions	50
Rolling back transactions	50
Working with factories.....	50
Creating factories	51
Creating new persistent instances	51
Fetching objects	51
Fetching all instances	52
Fetching by key	52
Fetching using arbitrary search values (=)	53
Ordering result sets	53
Fetching using arbitrary operators	54
Fetching using SQL queries	55
Using AdHoc SQL	55
Using persistent instances (LPCPersistence protocol).....	56
Updating and inserting new instances	56
Deleting instances	57
Working with relationships.....	58
Fetching related objects	58
Adding a relationship instance (one to many).....	59
Removing a relationship instance (one to many).....	60
Adding/Setting a relationship instance (one to one).....	60
Removing a relationship instance (one to one).....	60

Caching	61
Caching overview	61
Dynamic cache availability	61
Displaying cache statistics	62
Flushing the cache	62
Beans	65
JdbcStore Beans	65
Advanced Programming	67
Accessing JDBC connection information	67
Using model and type meta-data	67
Retrieving types	67
Getting type information	68
Using attribute information	68
Using attribute information	68
Using relationships information	68
Sample application	69
Swing (JFC) components	71
Customizing code templates	71
Delivering JdbcStore Applications	73
Java applications	73
Java Applets	73
Working with the Examples	74
Examples overview	74
Customizing the examples	77
Changing the example settings	77
Creating the sample database	78
Using SQL anywhere	78
Using Access	78
Other databases	78
Using the sample model	79
Recompiling the examples	79

Installation

Installing JdbcStore

To install JdbcStore, type the following:

```
java JdbcStore
```

The default installation path is set to /lpc.

Setting the CLASSPATH

After installing JdbcStore, you should include the following directories and/or files in your classpath:

- c:\lpc\symantec\symbeans.jar
- c:\lpc

If you have chosen to install JdbcStore in a directory other than c:\lpc, substitute your installation directory instead of c:\lpc in the above.

Directory structure

The JdbcStore directories are structured as follows:

batch	command files to recompile the sample and example classes
com.lpc	the JdbcStore classes and examples directories
jdbctest	classes subdirectories

examples	example classes
classes	generated classes for the examples
models	sample models for examples and sample application
run	JdbcStore runtime classes
sampleApplication	sample application classes
swingUI	LPC classes for JFC components
workbench	JdbcStore workbench classes
lpctemplate\default	default templates for code generation
_jdbcstore_source	contains Java source subdirectories
examples	Java source for example classes
classes	Java source for generated classes
sampleApplication	sample application source files
swingUI	source files for LPC's JFC components

Introduction

JdbcStore Overview

What is JdbcStore

JdbcStore is a Java to Relational DBMS **persistence framework**. It enables you to store Java objects in a relational DBMS .

JdbcStore **reduces the impedance mismatch** between Java's object orientation and the relational representation of data used by RDBMSs.

You can write an entire Java application and store your data in a relational database **without any SQL coding or calls to the JDBC API**. JdbcStore transparently interfaces to JDBC and generates the required SQL to store and retrieve your objects from the database.

JdbcStore consists of:

- a **workbench** to define mappings between Java classes and relational tables
- a **runtime environment** to transparently store and retrieve objects from JDBC compliant databases

JdbcStore is **flexible**. It does not require you to implement your persistent classes as a subclass of a specific class. Persistent classes can be implemented as subclasses of Object or any other classes.

JdbcStore does not modify or pre-process Java source code. You can even define persistent classes for class files (i.e. files for which you do not have source code).

Sample application

The following example is an extract of a small JdbcStore application that fetches and displays all customers, their sales orders and sales order line items. It also increases the quantity of items for each sales order item.


```

// Create a factory ...
// its responsibility is to create and return objects
System.out.println("Creating the factory");
customerFactory = new CustomerFactory();
customerFactory.broker(broker);

// Fetch all the customers
System.out.println("Fetching all the rows");
try {
    lpcCustomer e;
    Vector v = customerFactory.fetchAll();
    for (int i=0; i < 2; i++) {
        e = (lpcCustomer) v.elementAt(i);

        // fetch the customer sales orders
        Vector so = e.getSalesOrders();
        System.out.println(e.getLname()+" "+e.getFname());

        for (int j=0; j < so.size() ; j++) {
            lpcSalesOrder s = (lpcSalesOrder) so.elementAt(j);
            System.out.println("\t"+ s.getId() + " " +
                               s.getOrderDate());

            // fetch the sales order items
            Vector si = s.getSalesOrderItemss();
            for (int k = 0; k < si.size() ; k++) {
                lpcSalesOrderItems oi =
                    (lpcSalesOrderItems) si.elementAt(k);
                System.out.println("\t\t" +
                                   oi.getProduct().getDescription()
                                   + " " + oi.getQuantity());

                // update the sales order item quantity
                oi.setQuantity(
                    new Integer(oi.getQuantity().intValue() + k) );

                // store it
                oi.store();
            }
        }
        System.out.println();
    }
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("Broker fetch error - " + e.getMessage());
}

// commit the changes
System.out.println("Commit transaction");
try {
    broker.commit();
} catch (Exception e) {
    System.out.println("Broker commit error - " + e.getMessage());
}

// Close the connection
System.out.println("Disconnecting the broker");
try {
    broker.close();
} catch (Exception e) {
    System.out.println("Broker disconnect error - " +
                       e.getMessage());
}
}

```

Note that there is no need to code any SQL or to interface to JDBC in this application.

Features

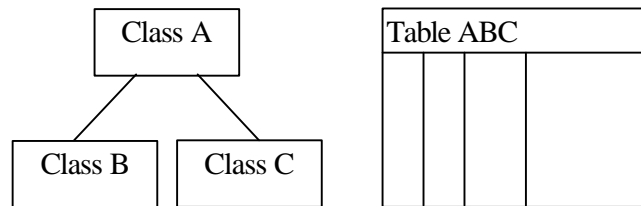
JdbcStore supports the following main features:

Support for inheritance

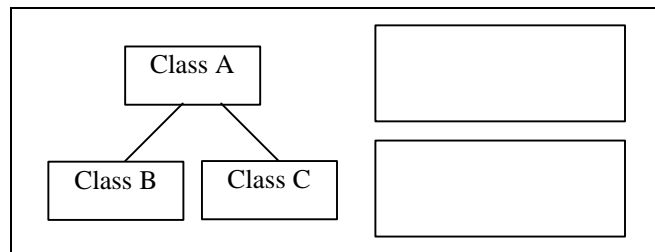
Inheritance relationships can be defined and maintained in the relational DBMS. It supports both abstract and concrete supper classes.

An inheritance hierarchy can be implemented in various ways:

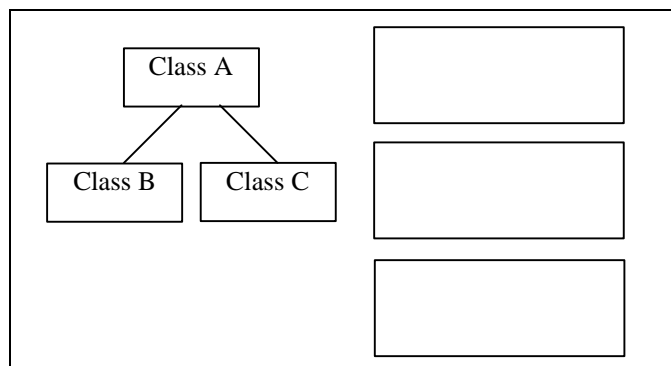
- single table
the table contains the attributes of the superclass and all of its subclasses



- subclass tables
the table contains each subclass and its superclass attributes



- a table for each class in the hierarchy
each table contains its own attributes



Support for composition (associations)

Complex objects can be modeled and stored using JdbcStore. One-to-one and one-to-many associations are supported. At runtime, associated objects are fetched and instantiated transparently.

Lazy and immediate instantiation of complex objects

When working with a large number of objects, JdbcStore enables you to instantiate an object and its associated objects immediately (instead of using lazy instantiation). This improves performance by reducing network traffic and the number of database calls.

Bean support

JdbcStore public classes such as model, broker and factories were developed for Java Beans and include BeanInfo classes.

Generated classes not only adhere to Java Beans conventions (getXXX, setXXX) but can also generate BeanInfo classes.

JFC support

JdbcStore includes a generic JFC table panel that enables you to display objects in tabular format. It also includes a table data model customized for JdbcStore persistent classes.

Full source code for all JFC-based UI components is provided, so that you can implement your own component by either subclassing or implementing new widgets.

Class and bean generation from a relational database schema

JdbcStore can generate classes and BeanInfo classes based on existing relational schemas. The classes, methods and fields will be generated with names conforming to established Java and Java Beans naming conventions.

SQL Data Definition Language (DDL) statement generation from existing class files

Alternatively, JdbcStore can generate the SQL required to define tables in order to store existing class instances into a database. This feature uses introspection to discover the non-private attributes of the classes and does not require Java source code for the classes.

Instance and Object cache management

One of the difficulties of using a relational database in an object oriented application, is to ensure that only one instance of a persistent object exists in the running application. Multiple queries to the database may return different instances (but not identical) of the same object.

JdbcStore can transparently ensure that only a single instance of an object ever exists in your application. This is handled by an object cache.

As well, the cache optimizes performance by reducing the number of database calls.

The cache can use weak references when supported by the runtime Java VM to allow unreferenced cached instances to be garbage collected by the VM.

100% Pure Java

JdbcStore is written entirely in Java with no calls to the Java Native Interface API. Therefore, both the workbench and the runtime environments can be deployed on any Java supported platforms.

Applet and Application Deployment

Because JdbcStore is written entirely in Java and does not use any feature that would violate a browser's normal security settings, it can be used to develop both applications and applets. The JdbcStore based applets do not need to be defined as trusted.

User friendly workbench

JdbcStore includes a workbench to define the mappings between persistent classes and databases. The workbench includes facilities to define project settings (e.g. directories, package names, etc.) and define and edit the mappings.

The workbench is also used for generation of Java and SQL code.

There is no need to learn a new Data Definition Language, since the persistence model is defined entirely through the workbench.

Flexible architecture

JdbcStore does not make any assumptions about persistent classes as there is no need to inherit from a persistent root class. The generated business classes can be modified to add behavior.

JdbcStore implements the persistence system by subclassing your business objects. Business objects do not contain any SQL or persistence code.

As well, JdbcStore implements the JDBC persistence through a broker class. This broker class can easily be replaced in the future to provide a non-JDBC persistent broker such as RMI or CORBA.

Designed for performance

JdbcStore uses the reflection API heavily to transparently provide the persistent behavior required for the generated classes. To optimize performance, dynamic method lookups are kept to a minimum. Once a method has been looked up, it is cached so that no dynamic method and lookup resolution is required.

As well the Object cache minimizes database calls and network traffic.

Meta level programming

JdbcStore stores the persistence mapping information in types. These types contain such information as the class name, instance factory class name, attributes names and types, constructors, accessors and getters, association attributes, table mappings, column mappings, etc.

These types are in turn stored into a Dictionary structure called a model. When you save a model in the workbench, it is serialized and stored in a file.

Loading the model is one of the first tasks you must complete when running a JdbcStore information. The model and type metadata is available to your application through a public protocol and this can be used to develop generic components. JdbcStore itself includes such a component to display any persistent object in tabular format.

JdbcStore Concepts and Terminology

Object type

LPCObjectType

An ObjectType defines the mappings between a Java Class and one or more relational tables. It includes the following definitions:

- tables or tables used to store the type
- attributes – those that defined the mappings between the Java fields and table columns
- fields or columns uniquely identifying the type
- relationships between the object types. The relationship defines the relationships between the Java classes and also the database tables
- caching strategy used by the object type
- inheritance relationships between the object types and how these are implemented by the database
- Cataloged SQL statements for complex queries

The ObjectType can be generated from a database table, a Java class, or defined from scratch. It can then be used to generate classes, SQL database definition statements, or both.

Since ObjectTypes contain all of the information required to map classes to tables, they can be used to write generic components using the mapping information provided by the types (meta level programming).

Model

LPCModel

A model is just a collection of types. It should include all of the types required by an application.

Models are stored in serialized instances. The model can then be loaded by an applet or an application at runtime.

Attribute

LPCAttribute

An attribute of a type defines the mappings between the Java fields and the table columns.

It includes the following information:

- table columns including name, size, type, etc.
- Java fields including name and type
- Java getters and setters for the Java fields

Relationship

LPCRelationship

Relationships are used to represent aggregation and composition associations.

JdbcStore support one-to-one and one-to-many relationships. In addition the relationships may be implemented by uni-directional or bi-directional links.

A bi-directional implementation enables you to navigate the relationship from any of the related objects. For example, we have a relationship where a customer has zero or more sales orders (a one-to-many relationship):

- If we implement it using a bi-directional link, we can first fetch sales orders and then instantiate the related customer. Alternatively, we could first fetch the customers and then obtain their sales orders. If we choose to implement the relationship uni-directional from the customer, then we would not be able to instantiate a customer from its sales order.

When defining a relationship between Object types, you must also define the type controlling the assignment of the key. For example, if we have a relationship defined between a Sales Order Item and a Product, the product key should get stored in the Sales Order Item (and not the reverse) when associating a Product to a Sales Order Item. Therefore, the Product type controls the key of the relationship.

User class

A user class represents a business object. These classes are the main classes of your application. In the sample provided with JdbcStore, Customer, SalesOrder, etc. are user classes.

JdbcStore makes these classes persistent.

These classes can either be generated by JdbcStore or be defined previously. JdbcStore does not require you to have source for these classes.

When working with classes, you should keep in mind the following:

- only fields with public protected (and possibly package) visibility can be made persistent if no accessors are defined for them. Since private fields are not visible outside the class in which they are defined, JdbcStore cannot store them
- not all fields of a class must be defined as persistent
- public or protected (and possibly package) getters or setters must be defined for the fields. When the class is generated by JdbcStore, getters or setters are generated automatically. If none are defined, you must generate them in the persistent extension (class generation option).
- you must use the setters to update the value of the fields at runtime. JdbcStore overrides the getter to keep track of whether an object has changed or not. Otherwise you must mark the object dirty yourself (LPCPersistense.dirt(boolean flag).
- You must create new instances using the associated factory class.

You can modify a user class to add any behavior that you require. However, if the class is generated after you have made modifications, your changes will be lost. You should therefore backup the old version and re-apply your changes

Persistent extension

A persistent extension is a subclass of a user class. JdbcStore uses this class to implement persistence.

The persistent extension implements the store method and overrides (or implements) setters and getters for the Java fields.

Every persistent class has a persistent extension. These classes are always named `lpcUserClassName` where `UserClassName` is the name of your class.

In the current version, the prefix ‘lpc’ cannot be changed. In a future release, you will be able to assign your own prefix.

Factory

A factory is a class generated by JdbcStore. Every persistent class must have an associated factory. The default name of a factory is `UserClassNameFactory` where `UserClassName` is the name of the persistent class.

The factory is used for creating new instances of the persistent class. These new instances can either be created as the result of a query against the database or through the use of the `newInstance` method to create new objects.

Therefore the factory implements the following behavior:

- creation of new instance (`newInstance` method)
- retrieval from the database (fetch and associated methods)

Class generation

JdbcStore can generate the following classes:

- User Class
- persistent extension Class
- Factory Class
- BeanInfo class

You generate a class from the workbench. However, you must always generate a persistent extension and a factory.

You can also generate a BeanInfo class. The BeanInfo class is generated for the persistent extension class (since it is the class that implements the persistent behaviour).

SQL generation

JdbcStore can generate SQL Data Definition Language statements from the Object Types. The generated SQL is saved to a file.

Since the SQL generated is generic, you may have to edit it to implement features specific to your chosen database.

The generated SQL can include:

- Drop table statements
- Create table statements
- Primary Key constraints statements
- Create unique index statements

Concurrency control attribute

JdbcStore handles concurrency using optimistic concurrency control. Since applications may interface to a wide variety of databases, it must use a portable mechanism for concurrency control.

The implementation chosen is database independent. When you define a type, you can designate one attribute (an Integer type) to represent a concurrency control attribute. When a table row is updated or deleted, the value of the attribute is selected to ensure it has not changed since the row was fetched. If it has changed an exception is raised.

To detect whether a row was successfully updated or deleted, JdbcStore checks the rowCount returned by the execute.

Generated keys

In some cases, you may want to have keys assigned automatically to instances of your persistent classes.

You can use these assigned keys either as Object Identifiers or as regular database keys.

JdbcStore supports two types of generated keys:

- assigned by JdbcStore
- assigned by the database

JdbcStore assigned keys

When a key is generated by JdbcStore, it is generated using the following technique:

- JdbcStore issues a select Max(generatedAttribute) where column 1 = ? and column 2 = ?, etc. where generatedAttribute is the name of the generated column and column 1, column 2, etc. are the other key attributes which are not generated.

Therefore you can use this technique on objects which have one or more key columns. When used with a single column key, it can be used to generate Object Identifiers for your objects.

This technique is database independent.

Database assigned keys

This is only available when instances of your class or table row are identified by a single attribute.

In this case, the database is responsible for assigning a value to the attribute. For example, you could use this technique with Oracle sequence, Sybase identity and Access counter datatypes.

After a successful Insert statement, JdbcStore retrieves the assigned key value by issuing a fetch using all the attributes values of instances in its 'where' clause.

Cataloged (or Loading) SQLs

In some cases you may want to use complex SQL statements (e.g. correlated subqueries, etc.) to filter the objects returned from the database or to instantiate instances of a class and its related objects simultaneously (e.g. Customer with sales orders and sales order items).

To do so, you must provide SQL statements to the factories. You can either provide SQL strings at runtime or defined statements for the Object type that will be stored in the model. These stored statements are called Cataloged or (Loading) SQLs.

Using the Workbench

Overview

You use the workbench to define the mappings between your database and your Java classes. These mappings are stored in a model. The model is stored in a serialized class instance that is then used by the runtime environment or loaded in the workbench to modify the mappings.

You use the workbench to:

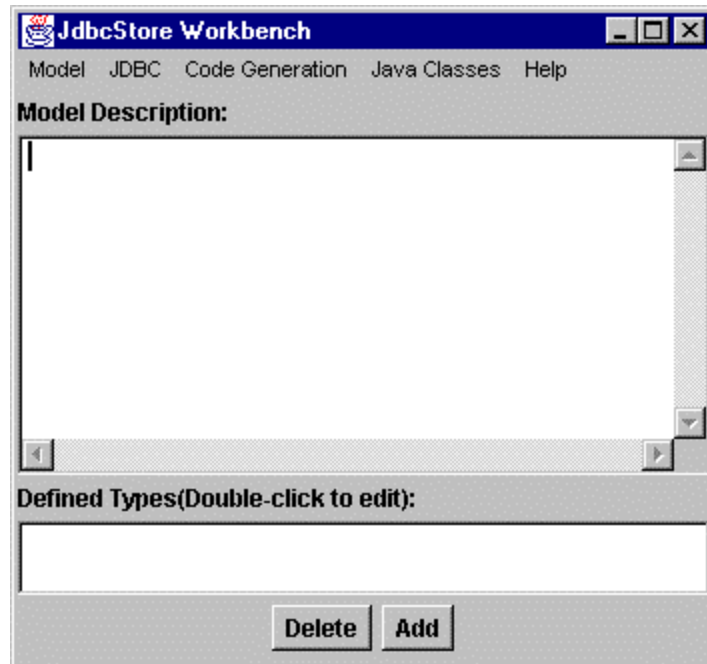
- define project settings (e.g. paths, JDBC URL, etc.)
- generate a model based on a relational schema
- generate a model based on existing Java classes
- define the OO to RDBMS mapping
- generate Java classes
- generate SQL DDL to define tables

Starting the workbench

To start the workbench, enter:

```
java com.lpc.jdbcstore.workbench.LPCMainFrame
```

The workbench main window will display and open on a new and empty model.



The directory where you installed jdbcstore must be in your CLASSPATH. Alternatively you can also have the current directory (i.e. ".") in your path.

Before starting the workbench, you should change the current directory to the jdbcstore directory. JdbcStore looks for the default settings file based on the current directory.

The main window of the workbench includes the following menus:

Model

- New - creates a new model
- Open - displays file dialog to load a previously created model
- Save - saves the currently opened model
- Save as - displays a file dialog to save the currently opened model
- Settings - opens the settings dialog
- Exit - exits the workbench

JDBC

- Load Driver - loads a JDBC driver in the Java VM
- Connect - connects the workbench to a JDBC URL
- Disconnect - disconnects from a previously connected URL
- Tables - displays the catalog dialog (only after a successful connect)

Code generation

- **Classes** - opens a class generation dialog to generate classes
- **SQL DLL statements** - opens a SQL generation dialog to generate and create table statements

Java classes

- **Class Loader** - opens a Class Loader dialog to load Java classes in the VM so that you can define mappings based on these classes

Help

- **About** - displays an about dialog with the current version information

Working with settings

When you start the workbench, the default settings from the current directory are loaded. These settings are stored in a serialized instance in a file called: defaultSettings.lpc.

You can modify the default settings and create settings files that can be loaded in the workbench.

The settings establish defaults for directories, the JDBC URL, JDBC driver, code generation options, etc.

Although the default values are read from the settings, you can always modify the various options before performing a task.

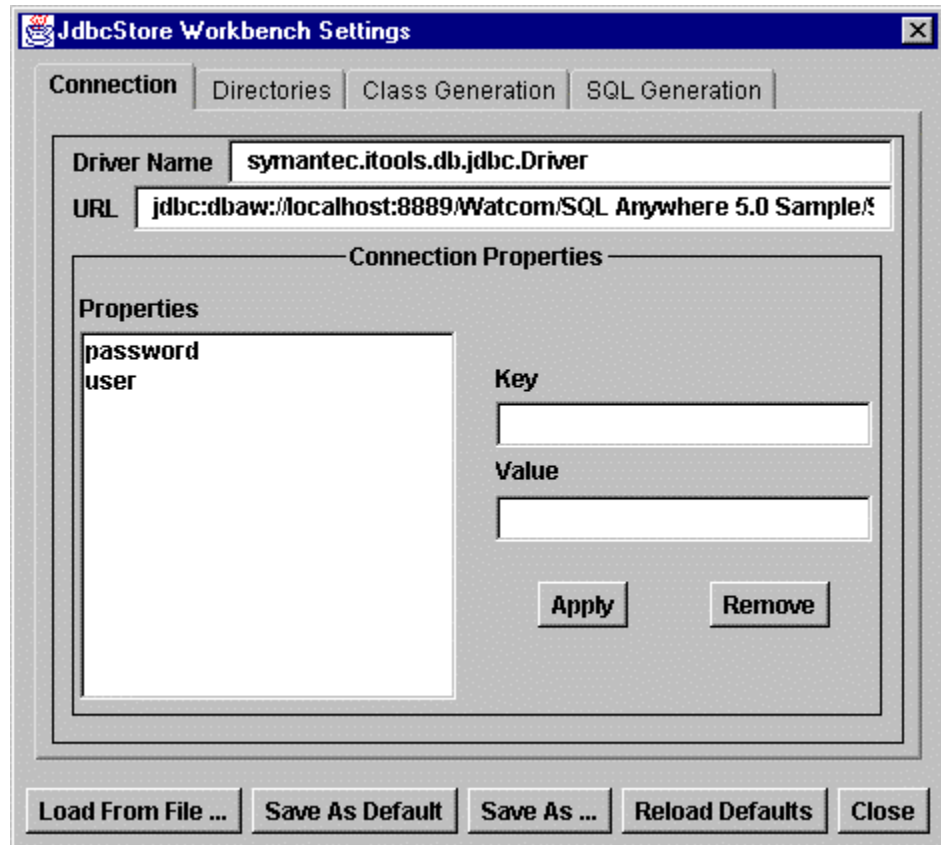
The settings dialog is invoked by selecting Settings from the **Model** menu.

There are five buttons at the bottom of the dialog that will show no matter which panel is displayed. The actions performed by the buttons are as follows:

- **Load from file** - opens a file dialog to load a settings file
- **Save As Default** - loads settings from the file “defaultSettings.lpc”.
- **Save As** - opens a file dialog to save a settings file
- **Reload Defaults** - loads settings from file “defaultSettings.lpc”
- **Close** - closes the settings dialog and any changes made will be in effect for the current session

Connection settings

When you invoke the dialog, the first tab panel is displayed. This panel is used to establish connection options.



Connection Panel

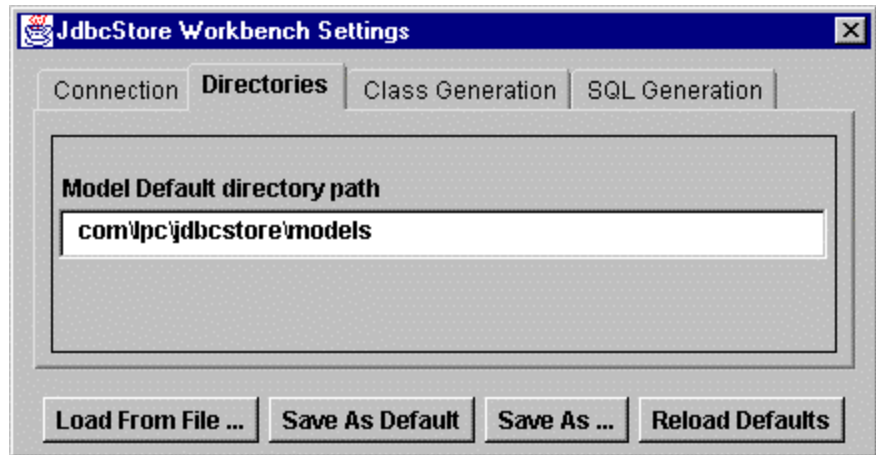
The selections on the panel are used as follows:

- **Driver Name** – specify a class name for the default JDBC driver (this should include the package name)
- **URL** – specify the JDBC URL of your database
- **Key** - specify a connection properties key
- **Value** - specify a connection properties value
- **Apply** - after you have specified a connection properties key and value, click on the Apply button to add them to the connection information
- **Remove** - select a property from the Properties list and click on the Remove button to delete the property key and value

JDBC Connection properties are specific to each JDBC driver. Please consult your JDBC driver documentation for details.

Directories settings

The Directories settings enable you to specify default directories for storing models.

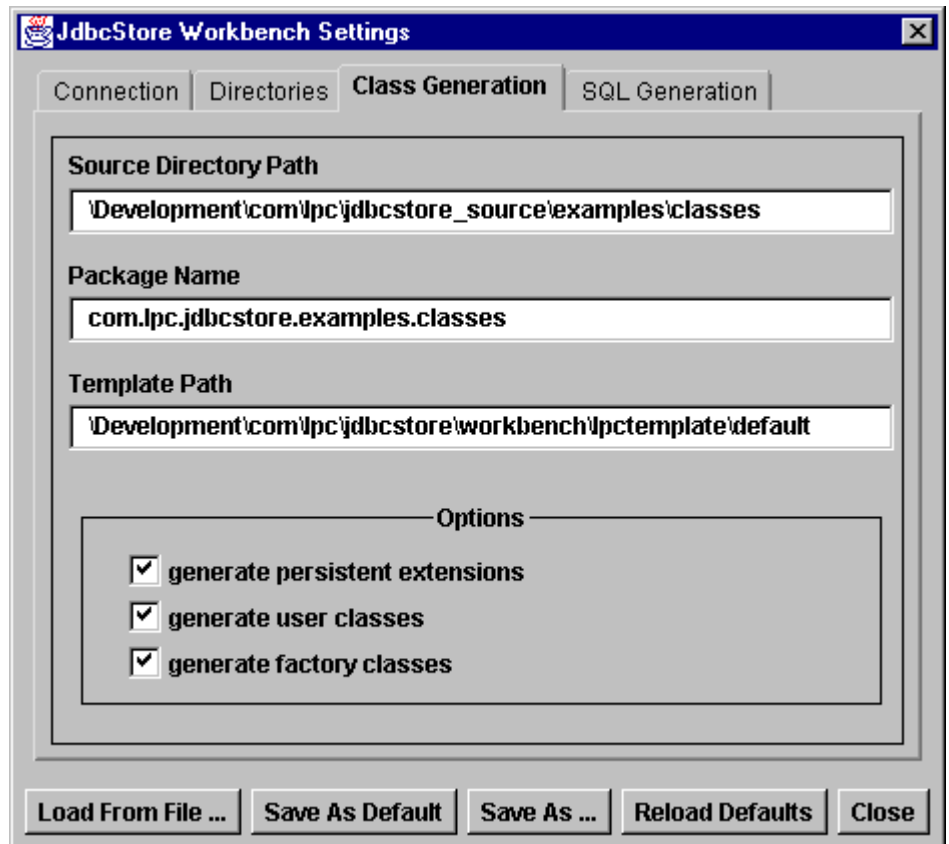


Directories Settings

Specify the default directory where your models will be saved.

Class generation settings

The Class generation settings allow you to specify default options for generating Java classes.



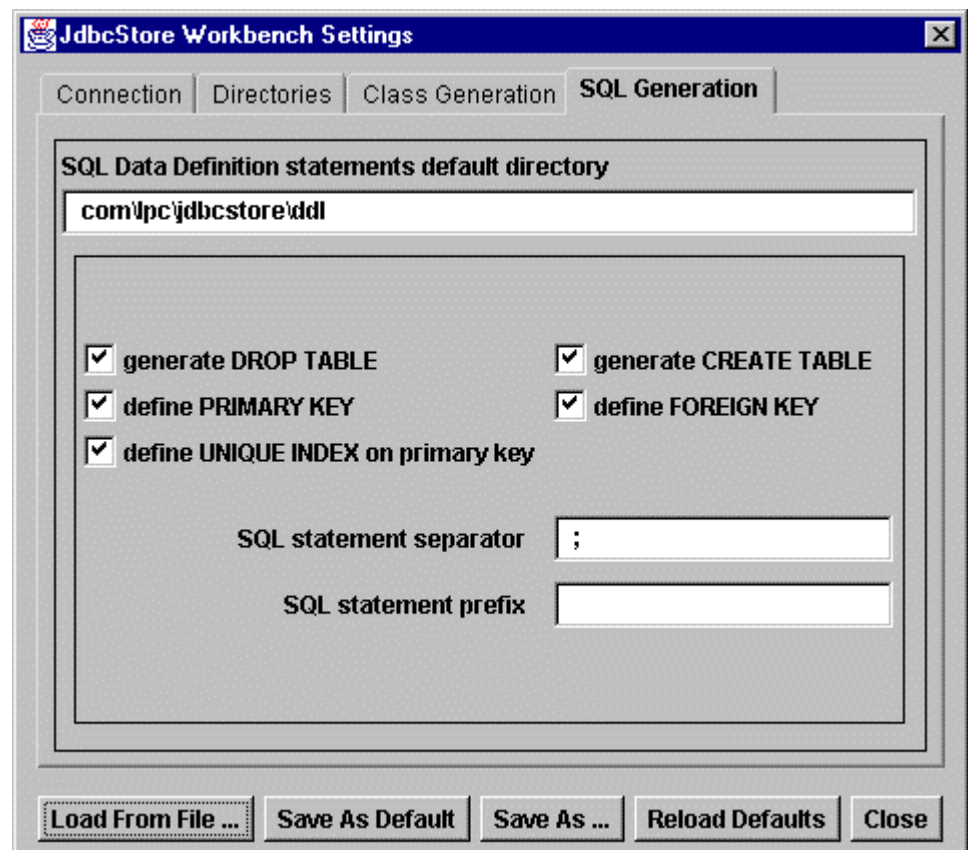
Class Generation settings

The settings are:

- **Source directory path** - the directory where generated Java files will be stored
- **Package name** – the default package name that will be used (all generated classes must be in a package)
- **Template path** - the directory where code generation templates are located
- **Generate persistent extension** - if selected, the persistence code will be generated
- **Generated user classes** - if selected, business object classes will be generated
- **Generate factory classes** - if selected, factory classes will be generated

SQL generation settings

The SQL generation settings specify default options for generating SQL Data Definition Language (DDL) statements.



SQL Generation settings panel

The default settings are:

- **SQL default directory** - the directory where SQL statement files will be stored
- **Generate Drop Table** - if selected, Drop Table statements will be generated
- **Generate Create Table** - if selected, Create Table statements will be generated
- **Define Primary KEY** - if selected, Primary Key constraints will be generated
- **Define Foreign Key** - if selected, Foreign Key constraints will be generated key
- **Define Unique Index** - if selected, Create Unique Index statements will be generated (on the primary key columns of the tables)
- **SQL statement separator** - the string used to separate SQL statements (e.g. “;”, “go”)
- **SQL statement prefix** - the string used to prefix SQL statements (e.g. DB2)

Working with a model

To:

- load an existing model, select Open from the **Model** menu
- to save a model, select Save from the **Model** menu
- to save a model with a different file name, select Save As from the **Model** menu

Defining types from a SQL schema

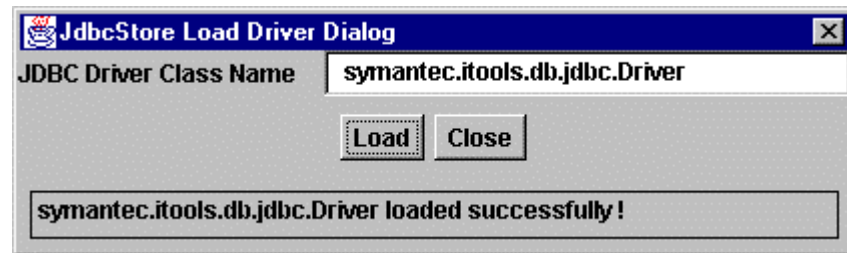
One of the most common tasks faced by developers is to build an application for which databases already exist or have been defined.

To define a model based on a schema, you need to perform the following tasks:

- load the appropriate JDBC Driver
- connect to your database
- retrieve the SQL Catalog
- generate Object Types for the relevant tables

Loading a JDBC driver

To load a driver, select JDBC driver from the **JDBC** Menu. The following dialog will be displayed (values are retrieved from the default settings and can be overridden here):



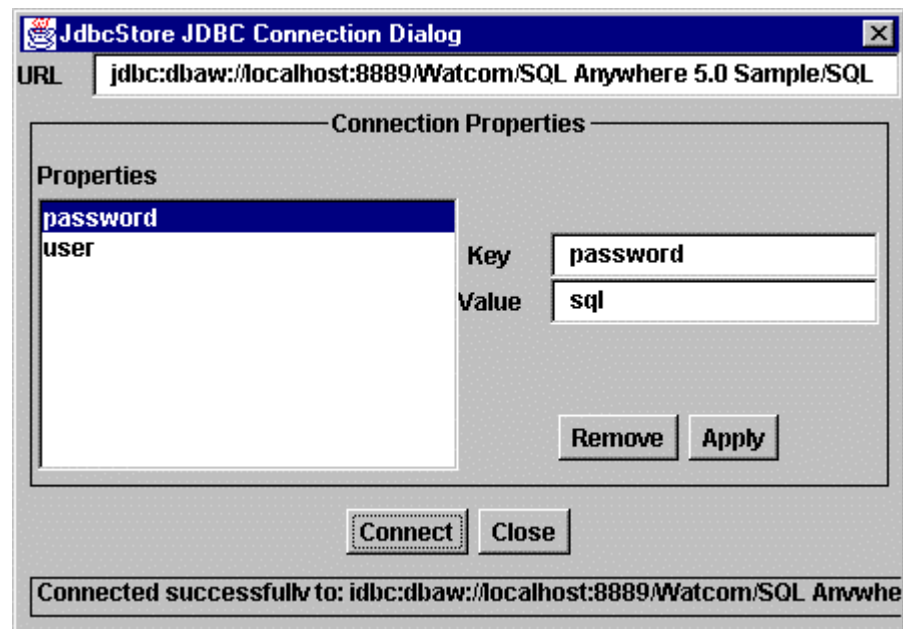
Load driver dialog

Specify the name of your JDBC driver and click on the Load button. A message will be displayed to indicate whether the driver was loaded or not. After loading a driver, you can close the dialog.

Your CLASSPATH must include the directory in which the package for your JDBC driver is located.

Connecting to a database

After you have loaded a driver, you can connect to a database. Select Connect from the **JDBC** menu. The following dialog will be displayed:



Connection dialog

The default values are selected from the workbench settings and can be overridden. Click on the Connect button to connect to the URL. A message will be displayed to indicate whether the connection was successful or not.

After connecting, close the dialog.

Closing the database connection

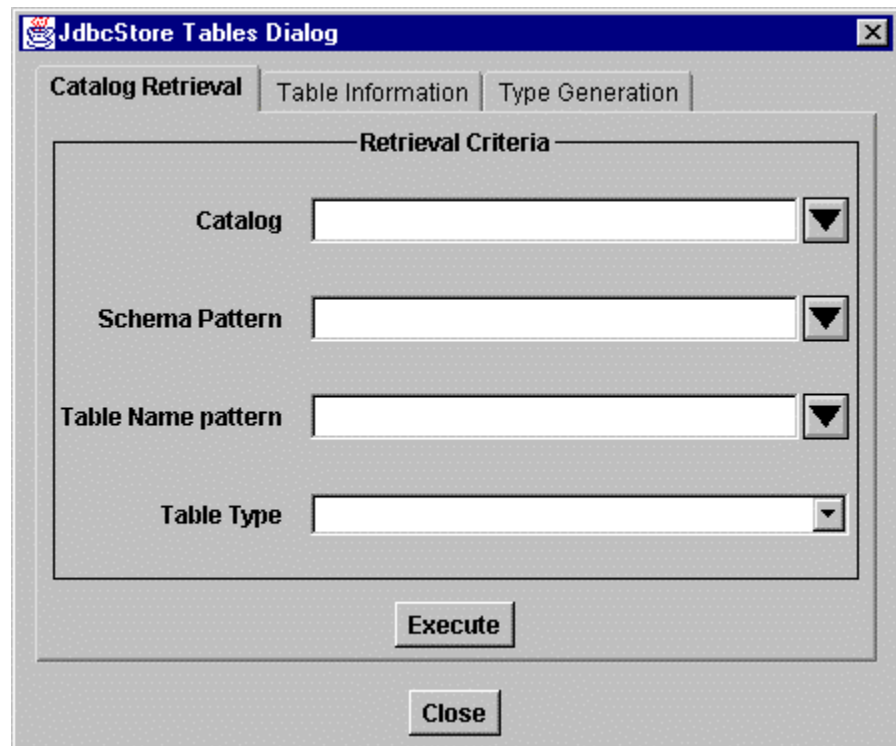
The database connection will be closed automatically when you close the workbench.

If you want to disconnect before exiting the session, select **Disconnect** from the **JDBC** menu.

Retrieving SQL catalog information

After Tables has been selected, the Tables Dialog is displayed. From this dialog, you may obtain information from the catalog and generate Object types based on table definitions in the open model.

The catalog retrieval panel is where you specify conditions to filter the information from the SQL catalog.



Tables Dialog

On this dialog you may specify:

- **Catalog** - a String specifying the database catalog. You may also specify <null> or <empty string>

- **Schema Pattern String** – usually the owner. You may also specify <null> or <empty string>
- **Table Name Pattern** - a string pattern for the table name. You may also specify <null> or <empty string>
- **Table Type** - select either Table, View or System Table

Not all values may be supported. Some databases may require <empty string> or <null> in some specific fields. You may have to experiment until you find a valid combination of values.

Click on the Execute button to fetch the catalog information.

Displaying table information

Select a table and a column to display the tabel and column information.

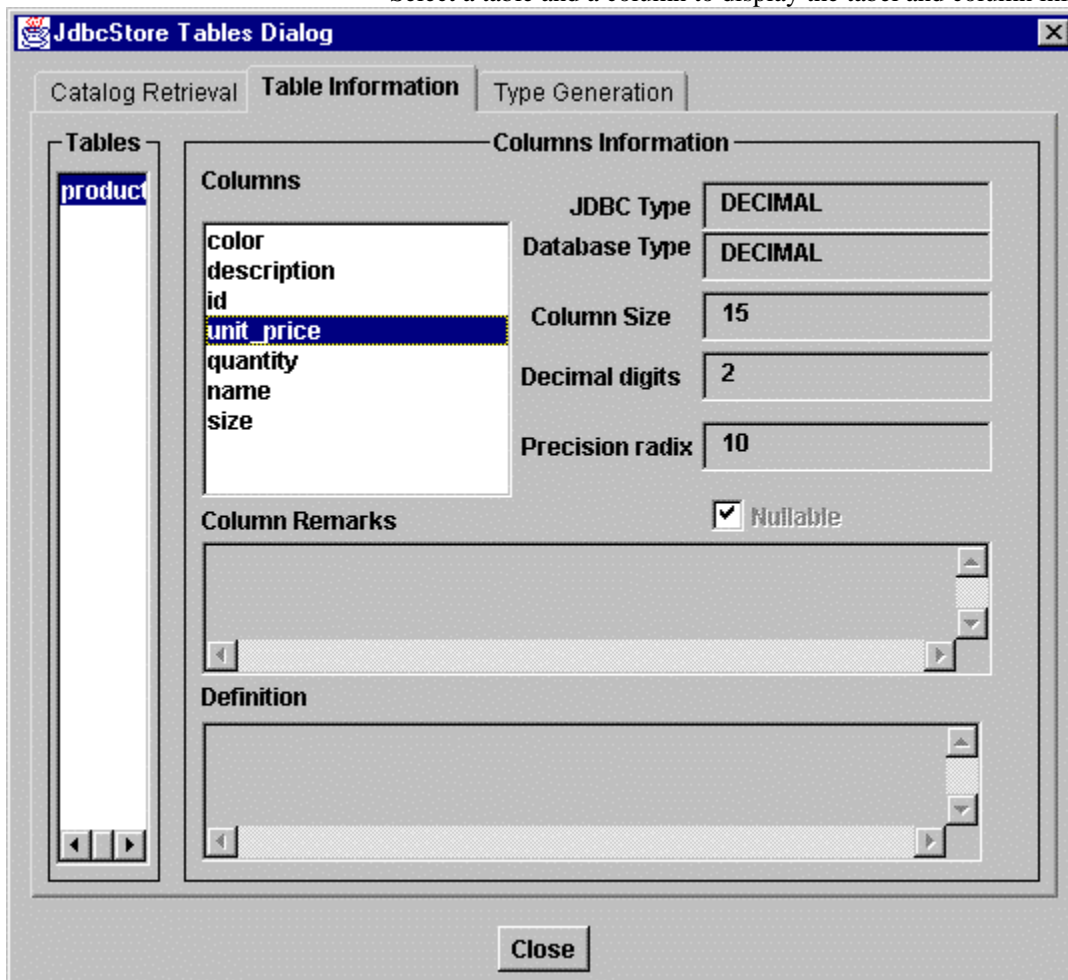
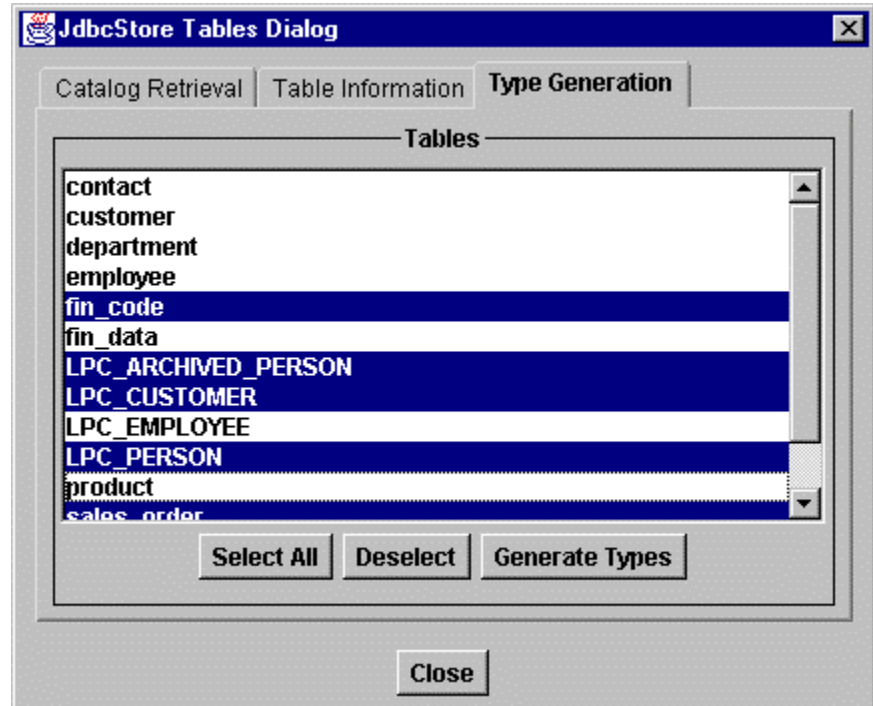


Table information panel

Generating object types

The Type Generation panel generates object types and adds them to the model.



Type Generation panel

Select one or more items from the Tables list and click on the Generate Types button. The generated types will be added to the model.

If a type with the same name already exists in the model, it will not be replaced. To change an Object type, first remove it from the model and then generate it again.

Your model should now contain the types you have generated.

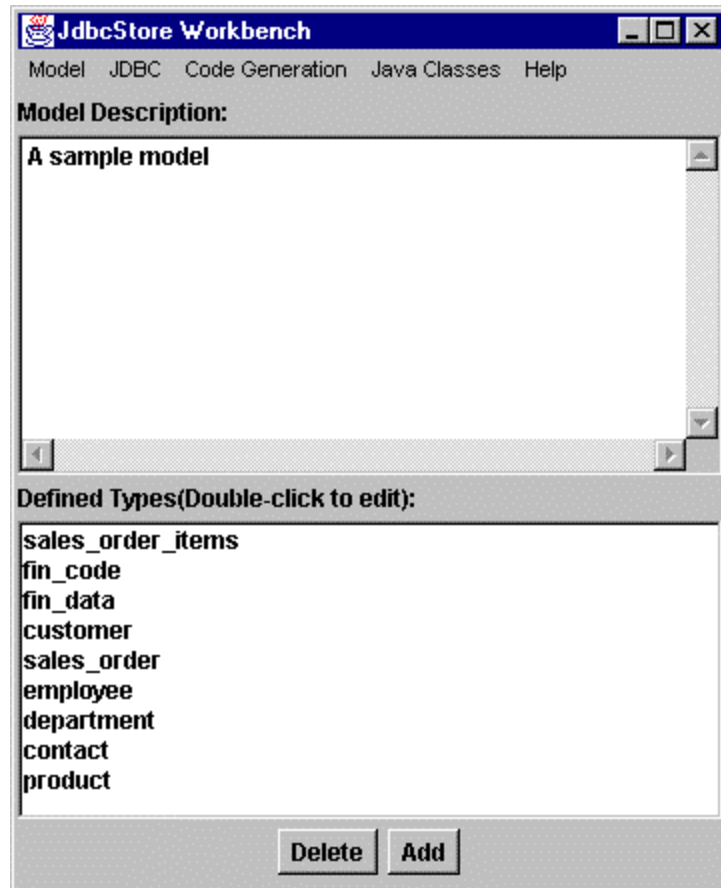
Working with types

The workbench main window displays a list of the types contained in your model.

To:

- delete a type - select a type and click on the Delete button
- add a type - click on the Add button
- edit a type - select a type and double click on the selection

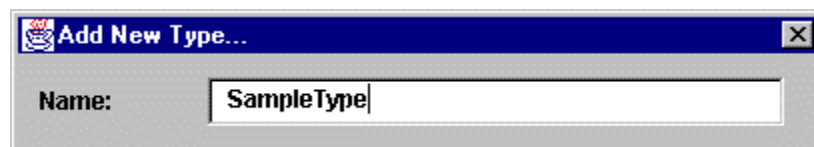
There is no confirmation for deleting a type.



Workbench Main Window

Adding a type

After clicking on the Add button, a dialog will be displayed to enable you to enter the name of the type. After the name is specified, the type can be edited.



Add new type dialog

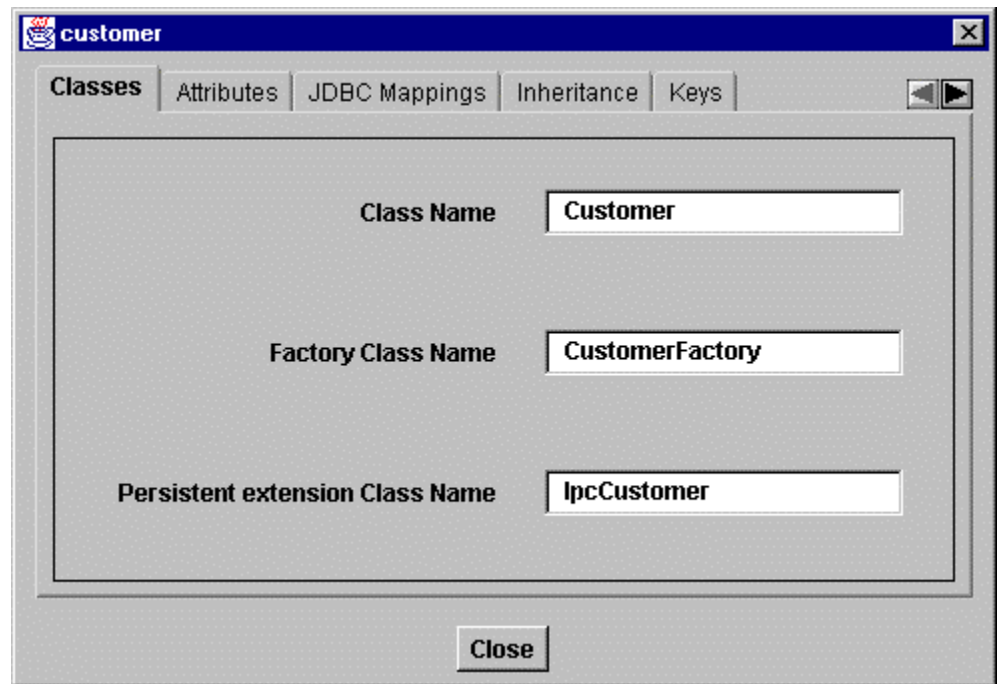
Editing a type

The type dialog enables you to specify a type's Java attributes and JDBC mappings, associations and caching strategy.

After double clicking on a type from the list, the Type Dialog is displayed. If you have generated the type from a table or a Java class, most of its data will be filled in. Otherwise, you will have to define it yourself.

Editing class information

The first panel display is the class information panel.



Class information panel

In this panel, you can change the default class name assigned by JdbcStore. The fields are as follows:

- **Class Name** - the name of the user class
- **Factory Class Name** - the name of the factory associated with the user class
- **Persistent extension Class Name** - the name of the persistent extension class (a subclass of the user class)

Editing an attribute

The next panel is the Attributes panel. It allows you to define and edit attribute information.



Attributes panel

You must click on the Apply button for changes made to the attribute to be applied. Otherwise the changes will be ignored.

To modify or view an attribute, select it from the list. You can then specify or change the following:

- **Java Class** - the Java class of the attribute. Arrays, wrapper and primitive types are supported

Unless you are using a Class for which you have no source, DO NOT USE PRIMITIVE TYPES. Primitive Java data types cannot represent null values. Therefore null in numeric fields will be converted to zero in primitive fields (on a subsequent store of the object, zero will be stored as well).

- **Java Getter** - the name of the method to retrieve the field value
- **Java Setter** - the name of the method to set the field value

There must be a getter/setter pair for each persistent attribute. JdbcStore uses these to keep track of whether an object has changed and set the values of fields in your object.

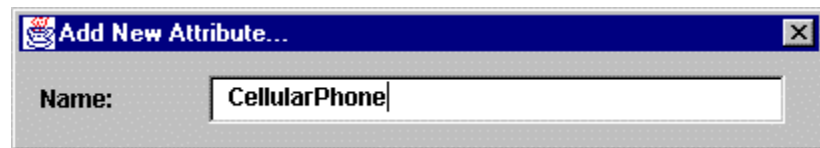
- **Key Attribute** - if selected, this attribute is use to identify the object
- **Updateable** - if selected, this attribute can be modified
- **Concurrency Field** - if selected, this attribute is used for concurrency control

Only one attribute can be marked as a concurrency attribute. This attribute must be a `java.lang.Integer`.

The restrictions on the concurrency attribute were designed to ensure the portability of the concurrency control mechanism across different databases

Adding an attribute

To add an attribute, click on the Add button. The following prompt will be displayed to enter the attribute name:



New attribute Name Dialog

You can then edit the attribute.

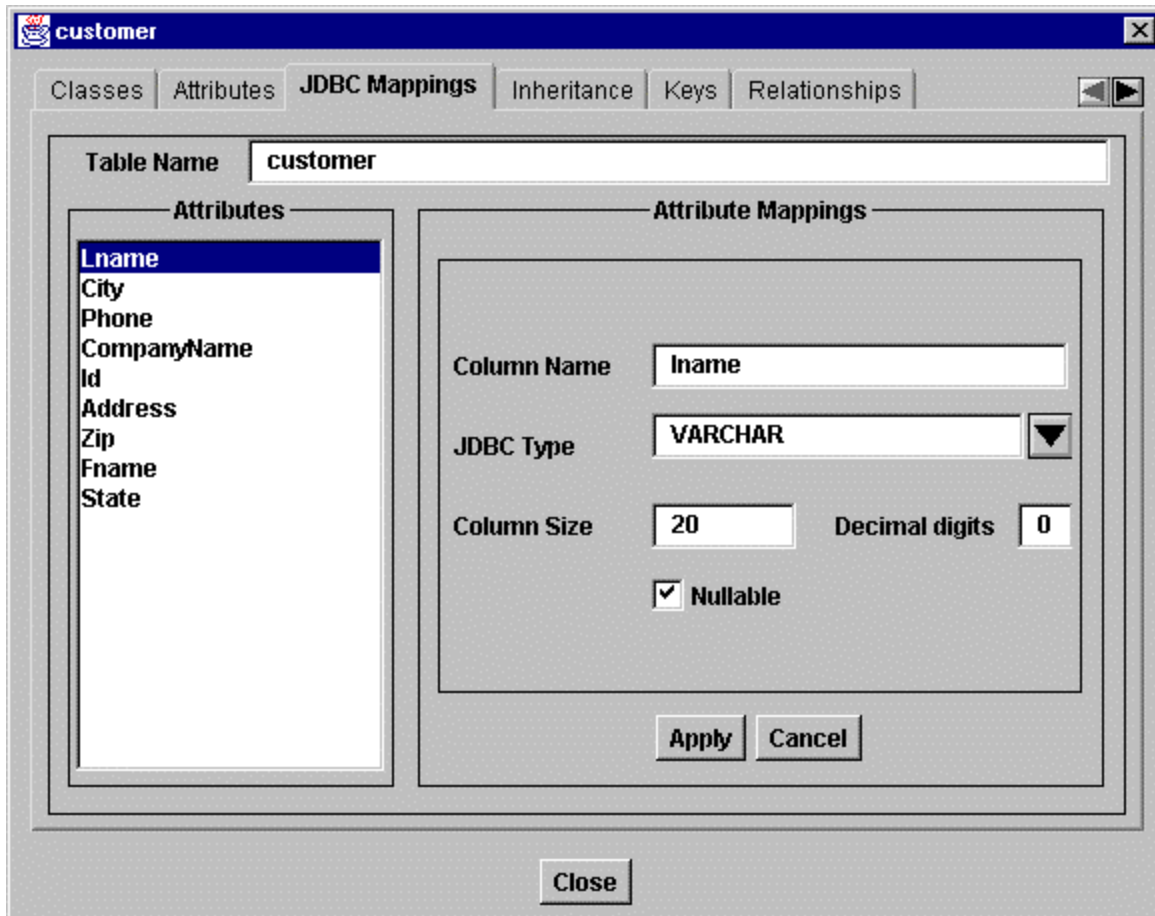
Deleting an attribute

To delete an attribute, click on the Delete button.

Defining the JDBC Mappings

The JDBC Mappings panel allows you to define the mappings between the Object type and a database table.

If you have generated the type from a table, all of the information will be completed. Otherwise, if you have generated a type from an existing class or defined one from scratch, you will need to supply the information.



JDBC Mappings Panel

The following information must be completed:

- **Table Name** - the name of the table
- **Column Name** - the name of the column in the table
- **JDBC Type** - the type of the column (this is restricted based on the Java type of the attribute)
- **Column Size** - the size of the column
- **Decimal digits** - the number of decimal digits (for numeric types)
- **Nullable** - if selected, the column will accept null values

Handling inheritance

Although inheritance is not explicitly supported by the relational model, it is possible to represent it using Type/Subtype relationships.

JdbcStore enables you to define Type/Subtype relationships in your types and map them to a Java hierarchy.

The Inheritance panel is where you specify the inheritance hierarchy.

The screenshot shows a window titled "LPC_EMPLOYEE" with a tabbed interface. The "Inheritance" tab is active, displaying the "Subtype Data" section. This section includes a checked checkbox for "has Supertype", a dropdown menu for "Supertype" currently showing "LPC_PERSON", a text field for "Discriminant value" containing "E", and two more checked checkboxes: "Supertype attributes stored in same table" and "Supertype cascades deletes". Below this is a section for "Discriminant Attribute (for Supertypes)" with an empty dropdown menu. A "Close" button is located at the bottom right of the window.

Inheritance Panel

The following information must be specified:

- **has Supertype** - select it if the Object type is a subtype (the generated class for this type will be inherited from its parent)
- **Supertype** - select the name of the Supertype
- **Discriminant value** - this must be a String and is the value of the discriminant attribute (in the parent type) that identifies the instance as a specific subtype (e.g. "E" for employee, "C" for customer).
- **Supertype attributes stored in the same table** - select it if the attributes of the parent will be stored in the subtype table.
- **Supertype cascades deletes** - select it if Cascade Delete is implemented by the supertype table.
- **Discriminant Attribute** - specify the attribute, if any, that is used to identify the subtypes of the Object type (specified in the Supertype).

Single table implementation

If you want to implement a single table containing the attributes of the Supertype and all of its subtypes, do the following:

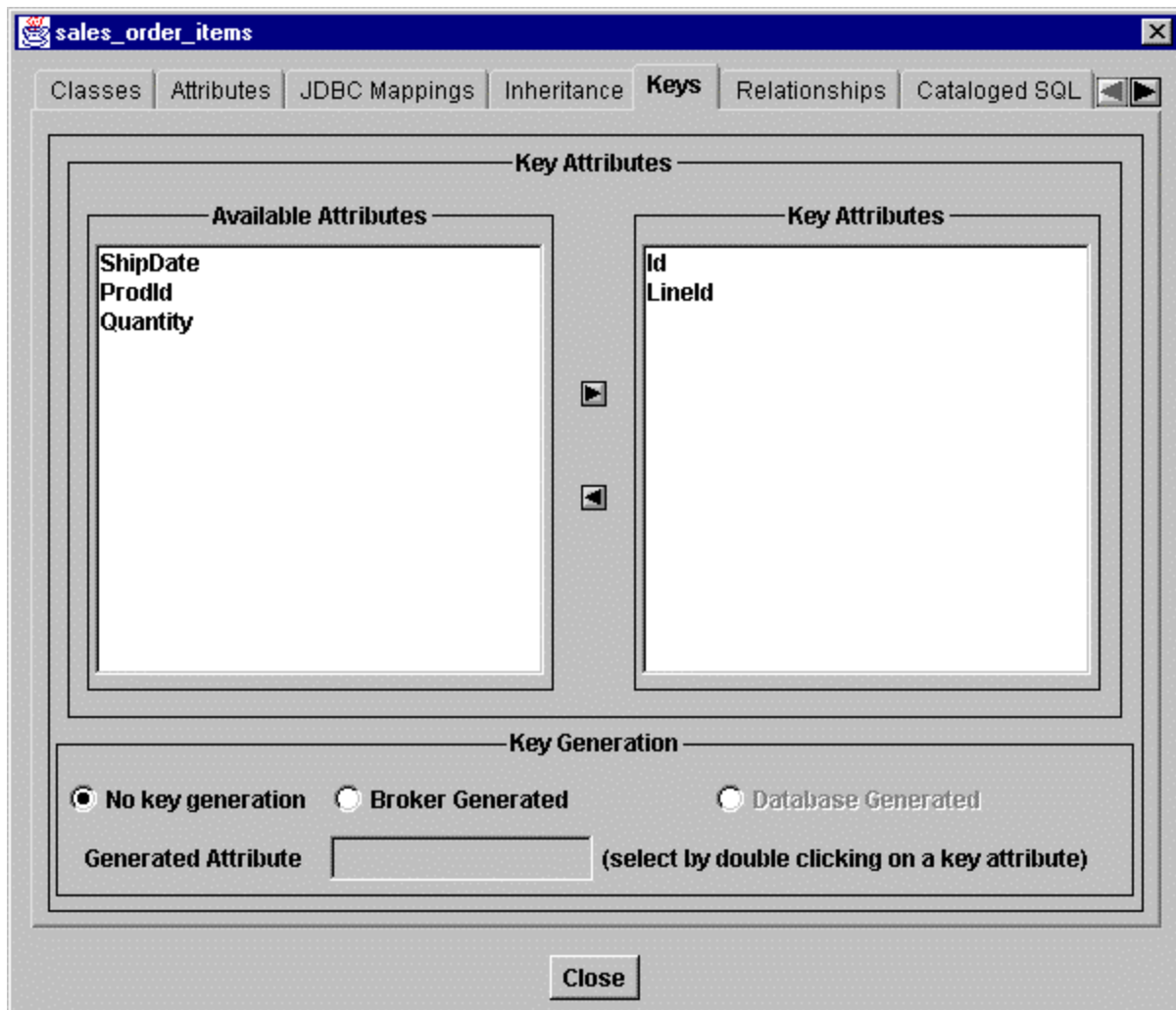
- define the Supertype mapped to the appropriate table. Its attributes should only contain the Supertype attributes (if you have generated the type from the table, you will need to remove attributes).
- define the subtypes. Each subtype will be mapped to the same table as the supertype. Add the required attributes (applicable only to the subtype).

The Supertype should implement an attribute to discriminate between the subtypes. Otherwise, the framework cannot recognize the different subtypes (since they are implemented by the same table).

Defining keys

The keys panel enables you to specify the table and object type keys as well as options for the generation of keys.

The keys defined in an object type are used to uniquely identify an instance. They may or may not correspond to the primary keys of the table. However, the combination of values must be unique in the database.



Keys Panel

The key attributes are displayed on the right. Use the arrow buttons to add or remove attributes from the key.

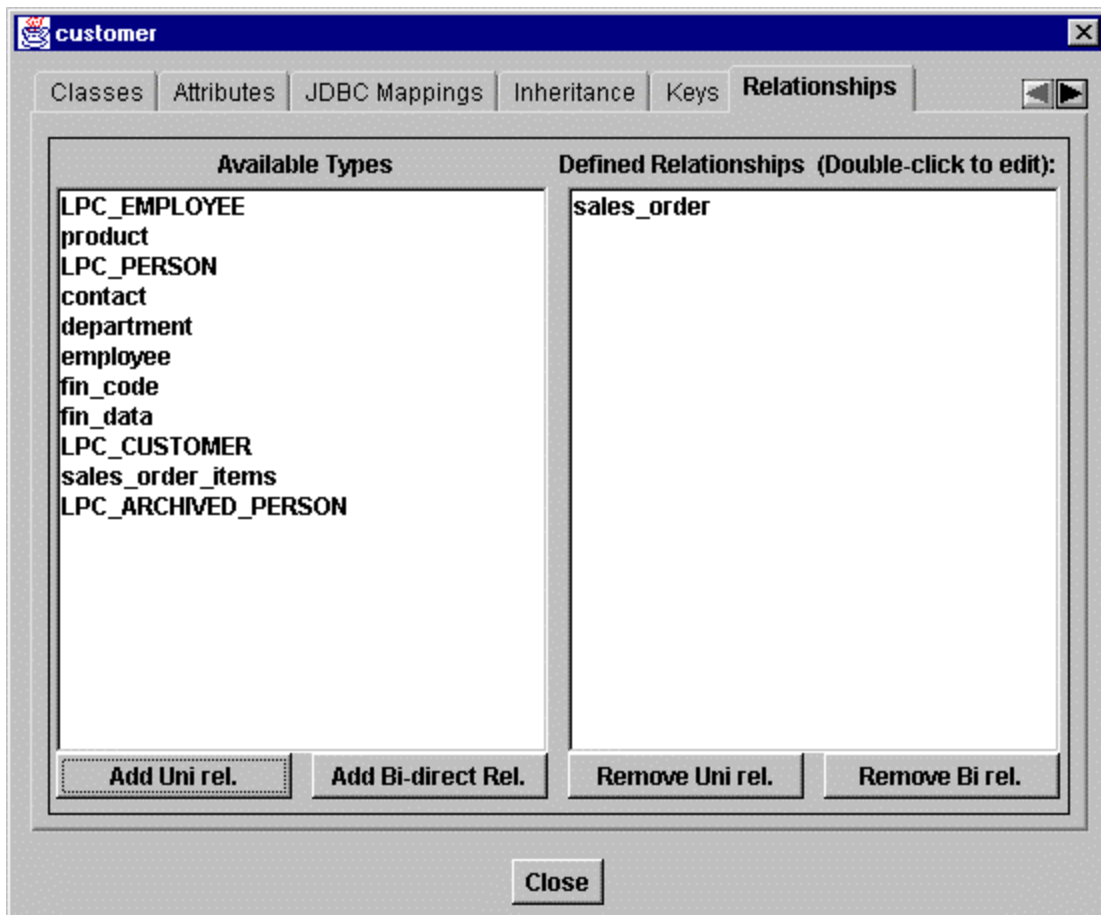
The following key generation options are supported:

- No key generation - your application is responsible for supplying unique key values
- Broker Generated - the framework will generate the key. This key is generated by using a MAX function on the generated attribute (double click on a key attribute to select it). The 'where' clause of the select MAX statement is the remaining key attribute.

The Database Generated button is only enabled for single attribute keys.

Defining relationships

The Relationship panel enables you to define relationships between types.



Relationships panel

- To add a uni-directional relationship, select an available type and click on the Add Uni rel. button.
- To add a bi-directional relationship, select an available type and click on the Add Bi-direct rel. button
- To remove a relationship, select a relationship and click on the Remove Uni rel. or the Remove Bi rel. button.

If Remove Uni rel. is used against a bi-directional relationship, only the side of the relationship from the type being edited is removed. Therefore, the other side of the relationship is maintained. If Remove Bi rel. is selected, both sides of the relationship are removed.

For uni-directional relationships, clicking on any of the Remove buttons will delete the relationship.

Editing a relationship

You must also define the cardinality and the attributes participating in the relationship. Double click on a relationship to edit it.

The Edit Relationship panel is where you further specify the relationship attributes.

Edit Relationship

Key control

From: To:

☒ controls key ☐ controls key

Cardinality ☐ One-To-One ☒ One-To-Many

Implementation ☒ bidirectional

Attributes

Edit Relationship dialog

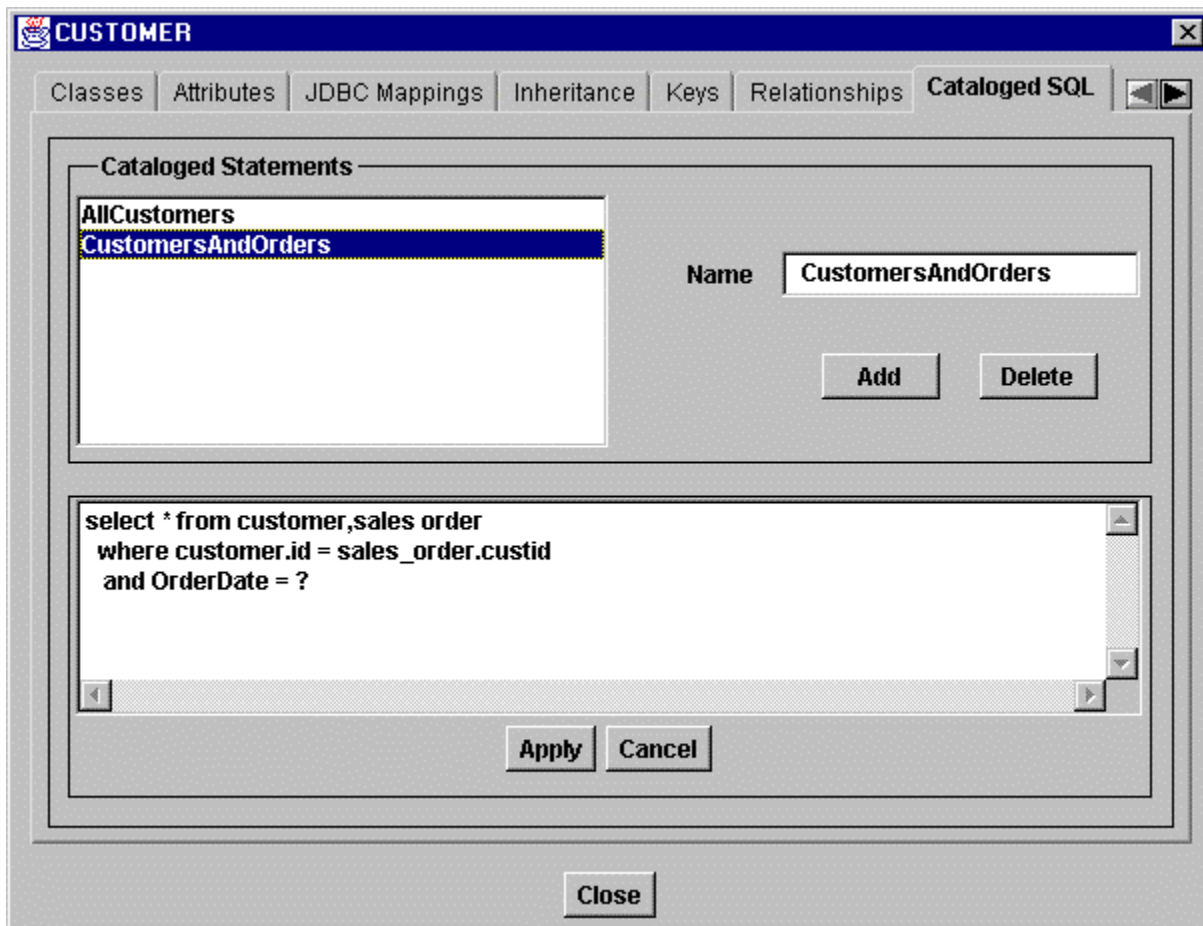
In this dialog you specify the following information:

- controlling type – click on the controls key button under the type that will control the keys of the relationship. In the sample screen, any time a sales order is added to a customer, the sales order attribute referring to the customer will be set from the value of the customer key (since the customer type controls the key).
- cardinality - select one-to-one or one-to-many
- implementation - select the implementation box if it is a bi-directional relationship (the value will have to be set based on whether Add Uni rel. or Add Bi-direct rel. was selected).
- attributes - to map the relationship attributes, select an attribute from each combination box and click on the Add Mapping button. Multiple attributes may be mapped in a relationship.

- To unmap attributes, select the mapping and click on the Remove Mapping button.

Defining cataloged SQL statements

The Cataloged SQL statements panel is where you can specify cataloged (or Loading) SQL statements.



Cataloged SQL panel

Adding a statement

To add a statement, type in the name of the statement in the Name field and click on the Add button.

Removing a statement

To remove a statement, select a statement in the list and click on the Delete button.

Editing a statement

To add or edit the SQL associated with a statement, type or modify the text in the bottom text area and click on the Apply button.

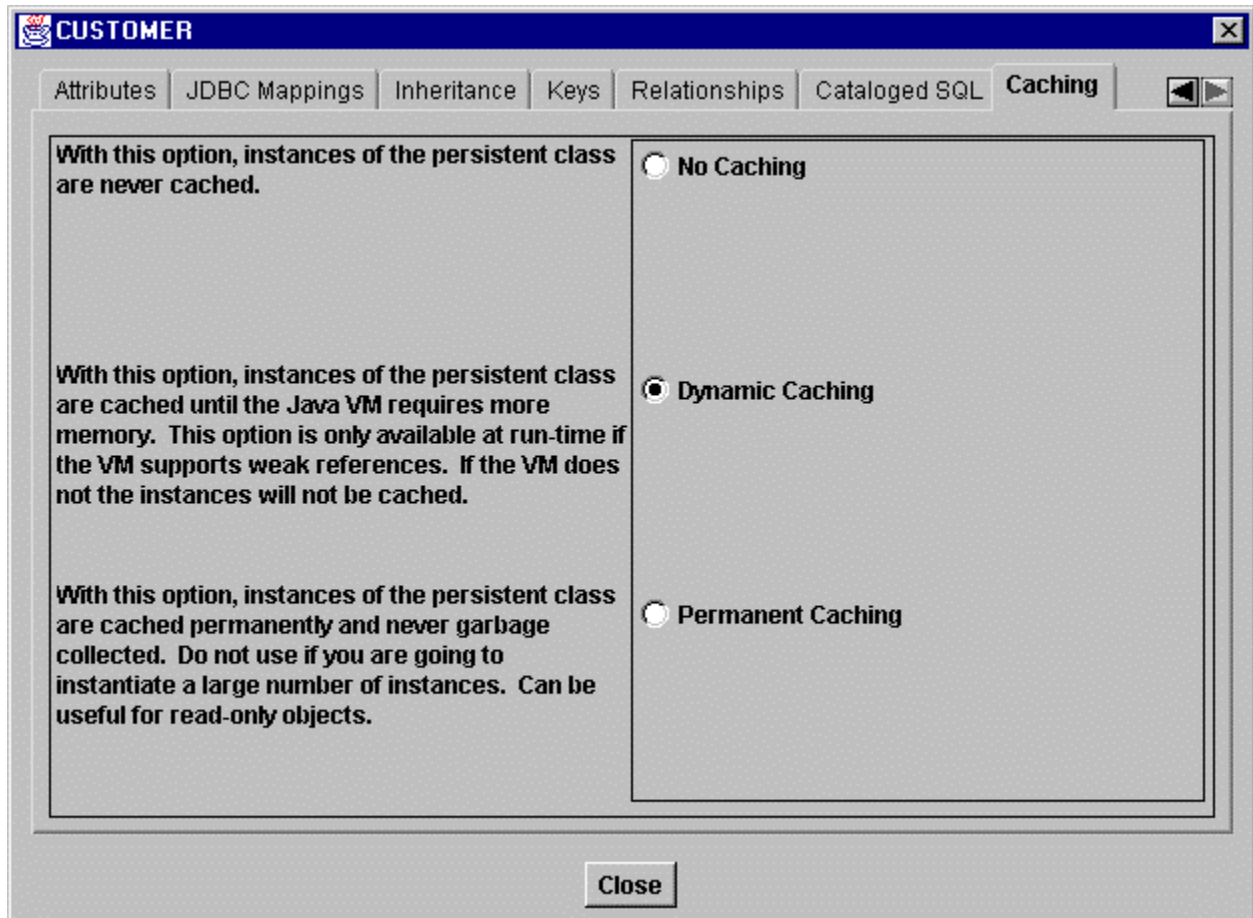
To undo the changes click on the Cancel button.

You must click on the Apply button after modifying the SQL text.

You can use parameter markers in your SQL statements. You would then supply the parameter values at runtime.

Choosing a caching strategy

You can define a caching strategy for each type. This is done in the Caching panel.



Caching panel

Click on the appropriate button to select a caching strategy for the object type.

You can modify the caching strategy at runtime.

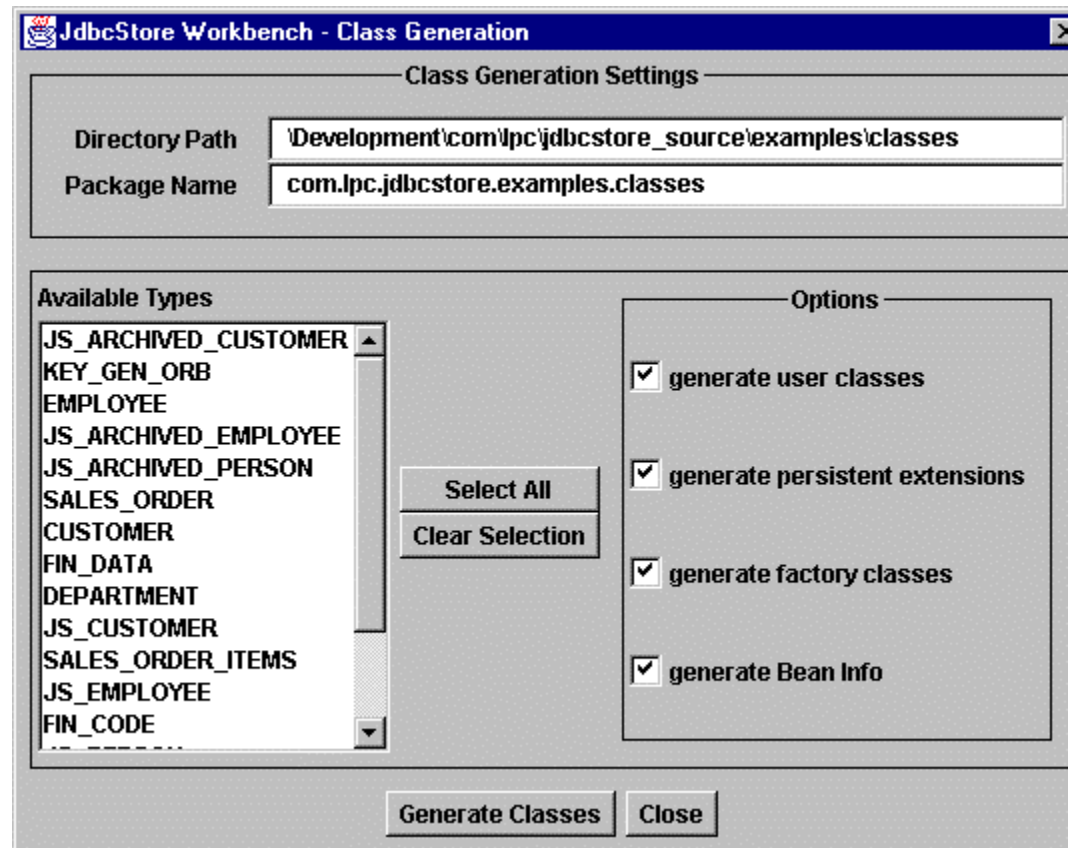
If the runtime Java VM is not Sun's JVM, types using dynamic caching will not use a cache.

Generating classes

Once you have defined some types in your model, you can generate Java source code for your persistent classes.

To do so, select Classes from the **Code Generation** menu.

The following dialog will be displayed:



Class Generation dialog

The fields on the dialog should be filled as follows:

- Directory Path - the directory where all the generated source files for the classes will be stored
- Package Name - the name of the package for the generated classes
- generate user classes - select this if you want to generate your business classes (generally, the only time you will not want to generate them is when you generate a type from an existing class)
- generate persistent extensions - select this if you want to generate persistent extensions for the business classes (this should always be selected)
- generate factory classes - select this if you want factories for your classes (this should always be selected)

- generate Bean Info - select this if you want to generate Beans for the persistent classes

Select the types for which you want to generate classes and click on the Generate Classes button. Check the Java console for messages.

A package name must always be specified for the generated classes. If you have generated types from existing classes, use the package name of these classes.

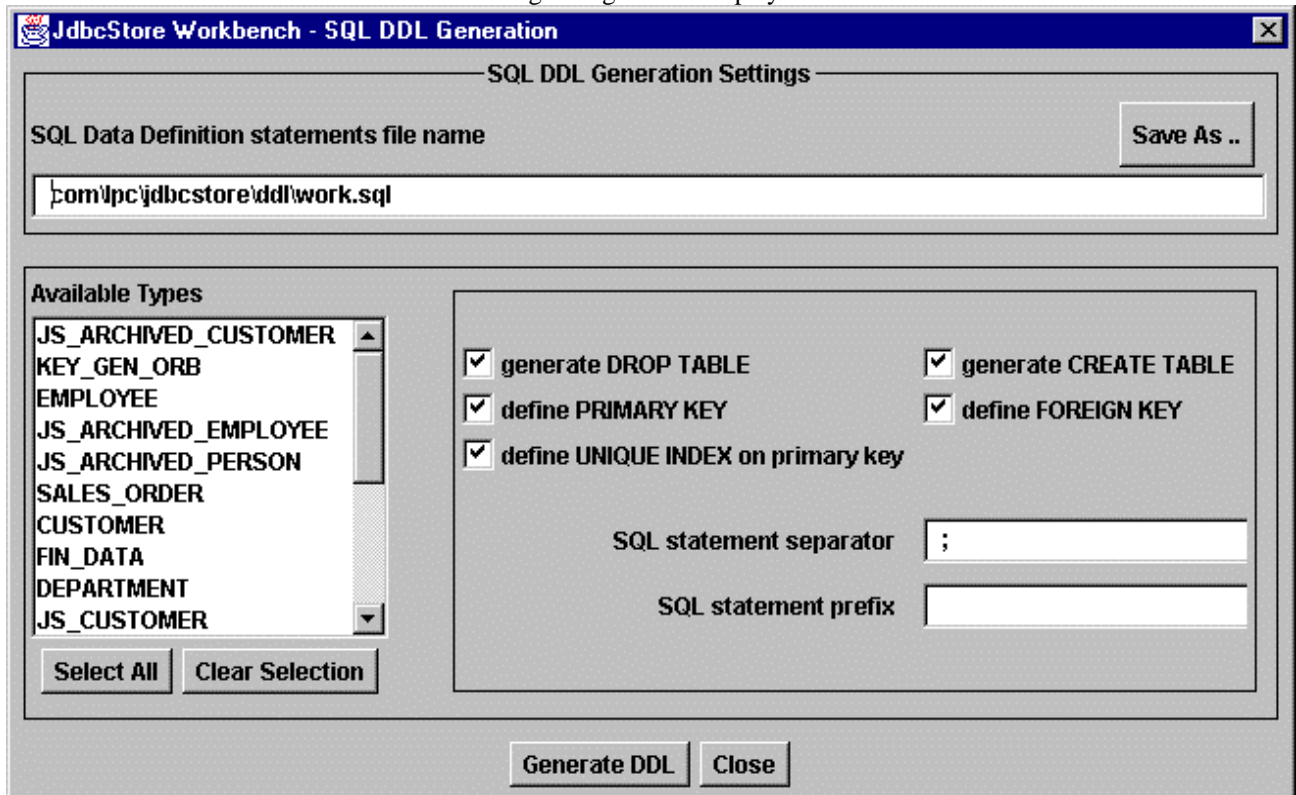
Generating SQL

If you have created types from scratch or generated them from classes, you can generate SQL to create tables.

The SQL generation is accessed by selecting SQL DDL statements from the **Code Generation** menu.

SQL DDL statements is only enabled after you connect to a JDBC data source (it requires access to the data source to obtain information about SQL type names, etc.)

The following dialog will be displayed.



SQL DDL Generation dialog

In this dialog you can specify:

- file name: the name of the file where the SQL will be stored (click on the Save As button to display a File Dialog)
- generate DROP TABLE - if selected, DROP TABLE statements will be generated
- generate CREATE TABLE - if selected, CREATE TABLE statements will be generated
- define PRIMARY KEY - if selected, PRIMARY KEY constraints will be generated
- define FOREIGN KEY - if selected, FOREIGN KEY constraints will be generated
- define UNIQUE INDEX on primary key - if selected, CREATE UNIQUE INDEX statements will be generated (on the primary key columns of the tables)
- SQL statement separator - the string used to separate SQL statements (e.g. “;”, “go”)
- SQL statement prefix - the string used to prefix SQL statements (e.g. DB2)

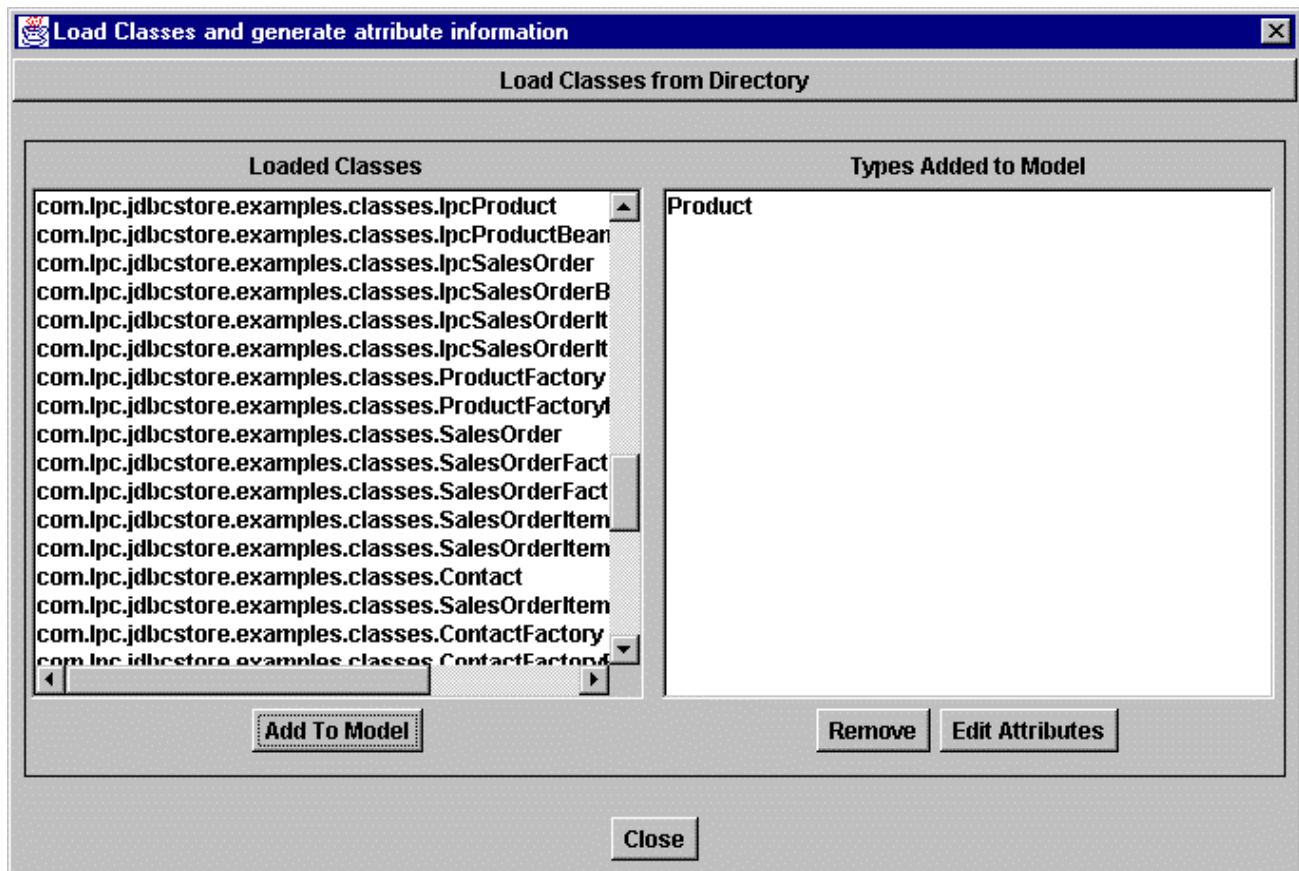
Select the types for which you want to generate table definitions and click on the Generate DDL button. You can then review and edit the generated SQL file and submit the SQL to your database.

Because of database peculiarities, you may have to modify the generated SQL for your database.

Working with existing classes

JdbcStore can generate types from existing Java classes. You use the Class Loader dialog to load existing classes. It is accessible by selecting Class Loader from the **Java classes** menu.

After selecting Class Loader, the following dialog will be displayed:



Class Loader dialog

- To load classes, click on the Load Classes from Directory button. A file dialog will be displayed. Select a file and click on the OK button. All of the Class files from the directory will be loaded.

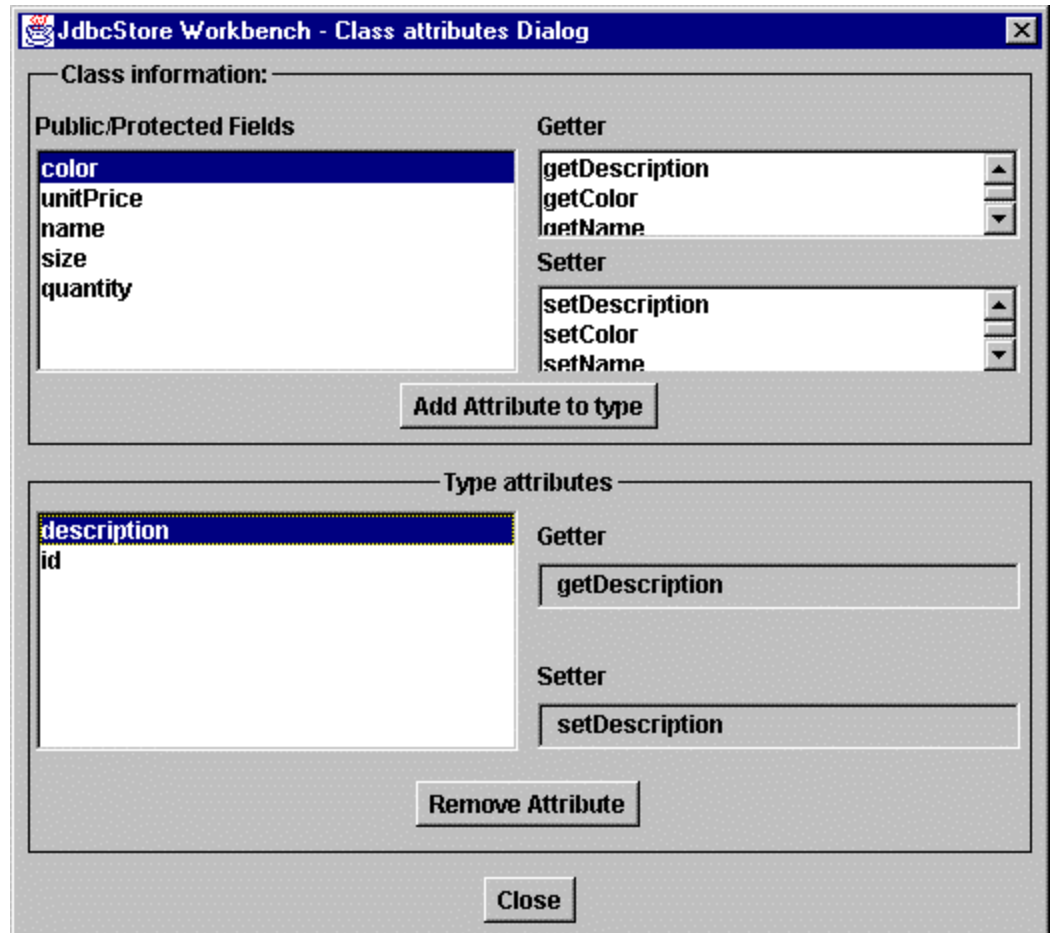
You must select a file from the directory. This will load all of the classes from the directory.

- To generate a type and add it to the model, select a class and click on the Add To Model button
- To remove a type just added to the model, select the type and click on the Remove button
- To define the fields, setters and getters that the type should contain, select a type and click on the Edit Attributes button

If a type with the name of the class already exists in the model, you must first remove it before regenerating the type from the class.

Defining attributes

After clicking on the Edit Attributes button, the following dialog will be displayed:



Edit Attributes Dialog

- to add an attribute to the type, select a field, a getter, a setter and click on the Add Attribute to type button
- to remove an attribute, select it and click on the Remove Attribute button

In the current version, only fields with getters and setters can be made persistent. The next version of JdbcStore will remove the restriction to allow all public and protected fields to be made persistent.

Obtaining version information

The About selection of the **Help** menu displays an information dialog. Please report the version information when contacting LPC Consulting Services, Inc.



About Dialog

Basic Programming

Overview

To write a JdbcStore application, you need to complete the following steps:

- load the appropriate JDBC driver
- load the model containing the Java to JDBC mappings for your application
- create and connect a broker to the JDBC URL
- manipulate your persistent objects
- close the connection to the broker

Using the LPCSqlDriver

The LPCSqlDriver class encapsulates the JDBC Driver class.

Loading a driver

Before you can connect to a database, you must load the appropriate JDBC driver class.

The following example shows how to load a driver:

```
System.out.println("Loading dbAnywhere driver");
try {
    LPCSqlDriver.loadDriver("symantec.itools.db.jdbc.Driver");
} catch (Exception e) {
    System.out.println("*** Class not loaded - " + e.getMessage());
}
```

Getting a list of loaded drivers

The LPCSystem static method drivers() returns a vector of loaded (i.e. registered) java.sql.Driver instances.

To display a list of all the registered driver names, you can use:

```
Vector drivers = LPCSqlDriver.drivers();
System.out.println("Registered JDBC drivers");
for (int i = 0 ; i < drivers.size() ; i++ ) {
    System.out.println(drivers.elementAt(i).getClass().getName());
}
```

Displaying driver information

You can obtain version and JDBC compliance information from a driver.

The following is how you would display the version and JDBC compliance of the driver you are using:

```
static LPCSqlDriver driver ;
static String driverName = "symantec.itools.db.jdbc.Driver";
...
driver = LPCSqlDriver.loadDriver(driverName );

// Display the driver info
System.out.println("Info for:" + driverName);
System.out.println("Major Version: " + driver.majorVersion());
System.out.println("Minor Version: " + driver.minorVersion());
System.out.println("JDBC Compliant?: " + driver.jdbcCompliant());
```

Working with models

Your model contains all of the mapping information (including relationships) between your Java classes and your database schema.

You must always load your model in your JdbcStore application.

You load your model either from a file or an URL.

If you are using Java Beans, you can use the model directly as a Bean, since models are stored as serialized instances of LPCModel .

Loading from a file

To load your model from a file use the LPCSystem static method `loadModelFromFile(String fileName)`.

The following example shows how to load a model from a file:


```
// Load the model
static String fileName =
    "\\Development\\com\\lpc\\jdbcstore\\models\\SampleModel.ser";

try {
    LPCSystem.loadModelFromFile(fileName);
    System.out.println("Model loaded" );
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("Error loading model: " + e.getMessage());
}
```

Loading from an URL

To load your model from a file, use the LPCSystem static method `loadModelFromURL(URL url)`.

The following example shows how to load a model from an URL:

```
try {
    url = new URL (
        "file:///e:/Development/com/lpc/jdbcstore/models/SampleModel.ser");

    LPCSystem.loadModelFromURL(url);
    System.out.println("Model loaded" );
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("Error loading model: " + e.getMessage());
}
```

Using a LPCSqlOrb

The broker is the class that allows you to interface to a JDBC compliant data base.

To create a broker, you need to create a new instance, as in the following:

```
broker = new LPCSqlOrb();
```

Connecting a broker directly

To connect to a JDBC compliant database, you need to supply some information such as URL, userid and password. In addition, there may be other information required by the JDBC driver.

In `JdbcStore`, all of this information is encapsulated by a `LPCConnectionInfo` instance. When the driver you are using requires standard information, you may use the `LPCBroker` directly to specify the information.

The following example shows how to connect a broker:


```

static public String url =
    "jdbc:dbaw://localhost:8889/Watcom/SQL Anywhere 5.0 Sample/SQL
Anywhere 5.0 Sample";

static public LPCSqlOrb broker;
System.out.println("Connecting the broker");

try {
    broker = new LPCSqlOrb();
    broker.setUrl(url);
    broker.setUser("dba");
    broker.setPassword("sql");
    broker.connect();
} catch (Exception e) {
    System.out.println("Broker not connected - " + e.getMessage());
}

```

Connecting a broker using LPCConnectionInfo

If you need to supply more information than an URL, userid and password, use a `LPCConnectionInfo` to supply the connection keys and values.

The following example shows how to use the `LPCConnectionInfo`:

```

static public String url =
    "jdbc:dbaw://localhost:8889/Watcom/SQL Anywhere 5.0 Sample/SQL
Anywhere 5.0 Sample";

static public String user = "dba";
static public String password = "sql";
static public LPCSqlOrb broker;

System.out.println("Connecting the broker");
try {
    LPCConnectionInfo connectionInfo = new LPCConnectionInfo();
    connectionInfo.url(url);
    connectionInfo.property("user",user);
    connectionInfo.property("password",password);

    broker = new LPCSqlOrb();
    broker.setConnectionInfo(connectionInfo);
    broker.connect();
} catch (Exception e) {
    System.out.println("Broker not connected - " + e.getMessage());
}

```

Disconnecting a broker

You should disconnect the broker before exiting your application. The following example shows how to disconnect a broker:

```

// disconnect the broker
System.out.println("Disconnecting the broker");
try {
    broker.close();
} catch (Exception e) {
    System.out.println("Broker disconnect error - " + e.getMessage());
}

```

Transaction control

In the JDBC API, transaction control is handled by a `java.sql.Connection`. In `JdbcStore`, it is the responsibility of the `LPCSqlBroker`.

Turning autocommit on/off

To turn autocommit on or off, use `setAutoCommit`:

```
// turn auto commit off
System.out.println("turn auto commit off");
try {
    broker.setAutoCommit(false);
} catch (Exception e) {
    System.out.println("Broker auto commit error - " + e.getMessage());
}
```

Autocommit should always be turned off when updating databases through `JdbcStore`.

Committing transactions

Use `commit` to commit a transaction:

```
// commit work
System.out.println("committing transaction");
try {
    broker.commit();
} catch (Exception e) {
    System.out.println("commit error - " + e.getMessage());
}
```

Rolling back transactions

Use `rollback` to rollback a transaction:

```
// commit work
System.out.println("committing transaction");
try {
    broker.rollback();
} catch (Exception e) {
    System.out.println("rollback error - " + e.getMessage());
}
```

Working with factories

Factories are generated `JdbcStore` classes that allow you to create new instances of your persistent Java classes and to fetch existing ones from your JDBC compliant databases.

A factory is created for every object type defined in your model.

Creating factories

To create a factory, create a new instance and associate the newly created factory with a broker:

```
System.out.println("Creating the factory");
customerFactory = new CustomerFactory();
customerFactory.broker(broker);
```

Before attempting to store or fetch objects through a factory, it must be associated with a broker.

Creating new persistent instances

To create a new persistent instance, you use the factory method `newInstance`. This will create an instance of the persistent extension and will mark it as new. Therefore when the new instance executes `store`, a new row will be inserted in the database.

The following example shows the creation of a new instance and its insertion in the database:

```
System.out.println("Creating customer");
try {
    lpcCustomer c = customerFactory.newInstance(new Object[0]);
    c.setLname("Charpentier");
    c.setCity("Don Mills");
    c.setId(new Integer(9001));
    c.setCompanyName("LPC Consulting Services, Inc.");
    c.setPhone("(416)510-2015");
    c.setFname("Luc");
    c.setZip("M3A 1N1");
    c.setAddress("7 Geraldine Ct");
    c.setState("ON");
    c.store();
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("Factory creation error - " + e.getMessage());
}
```

The `newInstance` method takes an array of objects (`Object []`) as an argument. You should supply the arguments required by your class constructor or an empty array for a no-argument constructor.

Note that you can use your Java class (e.g. `Customer`) instead of its persistent extension. In this case, you must type cast the instance back to its persistent extension (e.g. `lpcCustomer`) before using any of the methods defined in the `LPCPersistent` interface.

Fetching objects

Factory instances implement the fetching protocol in `JdbcStore`. This protocol offers a rich set of options that will meet most of your requirements for retrieving objects from your database.

In addition, you can use arbitrary SQL statements for retrieving objects. This allows you not only to fetch using SQL features not explicitly supported by the factory protocol (e.g. correlated subqueries, joins, etc.) but also to instantiate related objects immediately.

All fetch methods return a vector of instances.

Fetching all instances

To fetch all instances of a class from a database, use `fetchAll`.

In the following example, we fetch all of the customers and display their ID, first name and last name:

```
System.out.println("Creating the factory");
customerFactory = new CustomerFactory();
customerFactory.broker(broker);

System.out.println("Fetching all the rows");
try {
    lpcCustomer e;
    Vector v = customerFactory.fetchAll();
    for (int i=0; i < v.size(); i++) {
        e = (lpcCustomer) v.elementAt(i);
        System.out.println(
            e.getId() + " " + e.getLname() + " " + e.getFname());
    }
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("Broker fetch error - " + e.getMessage());
}
```

Since the `fetchAll` method returns a vector of object, you must cast each object to the appropriate persistent type.

Fetching by key

All persistent instances must be uniquely identified by a key. This is required to guarantee the uniqueness of the stored instances in the database as well as the maintenance of relationships.

This key can be generated automatically by the database (e.g counter, sequences or identity), by `JdbcStore` or assigned by the application.

The `fetchForKey` method returns an instance of a persistent object.

In the following example we fetch a customer (the key is the ID):


```

System.out.println("Fetching...");
try {
    lpcCustomer e;
    Object[] key = new Object[1];
    key[0] = new Integer(101);
    e = (lpcCustomer) customerFactory.fetchForKey(key);
    System.out.println(
        e.getId() + " " + e.getLname() + " " + e.getFname());
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("Broker fetch error - " + e.getMessage());
}

```

The argument of the `fetchForKey` method is an array of objects (`Object[]`). In this array you should store the key value(s).

Since the `fetchForKey` method returns an object (`Object`), you must cast the object to the appropriate persistent type.

If caching is turned on for the type, `fetchForKey` will first check the cache. If a matching instance is found, then it will return the cached instance.

Fetching using arbitrary search values (=)

When you want to fetch instances using equality on arbitrary fields use `fetchAllUsingAttributesValues`.

This method requires two arguments:

- an array of attribute names (`String []`)
- an array of attribute values (`Object []`)

The instances having matching values (using equality) will be returned.

In the following example, we fetch all customers residing in California:

```

System.out.println("Fetching...");
try {
    lpcCustomer e;
    String attribs[] = { "State" };
    Object values[] = new Object[1];
    values[0] = "CA";
    Vector v =
        customerFactory.fetchAllUsingAttributesValues(attribs, values);

    for (int i=0; i < v.size(); i++) {
        e = (lpcCustomer) v.elementAt(i);
        System.out.println(
            e.getId() + " " + e.getLname() + " " + e.getFname());
    }
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("Broker fetch error - " + e.getMessage());
}

```

Ordering result sets

If you want to order the results, use `fetchAllUsingAttributesValuesOrderBy`.

In addition to the array of attribute names and values, the third argument specifies the attributes that will be used to sort the result set.

Therefore, the method takes three arguments:

- an array of attribute names (String [])
- an array of attribute values (Object [])
- an array of {attribute name, sorting order ("asc" or "desc")} for sorting (String [] [])

In the following examples, the returned vector of customers will be ordered by the attributes Fname and description:

```
System.out.println("Fetching...");
try {
    lpcCustomer e;
    String attribs[] = { "State" };
    Object values[] = new Object[1];
    values[0] = "CA";
    String orderBy[][] = { {"Fname", "desc" } };
    Vector v = customerFactory.fetchAllUsingAttributesValuesOrderBy(
        attribs, values, orderBy);

    for (int i=0; i < v.size(); i++) {
        e = (lpcCustomer) v.elementAt(i);
        System.out.println(
            e.getId() + " " + e.getFname() + " " + e.getLname());
    }
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("Broker fetch error - " + e.getMessage());
}
```

Fetching using arbitrary operators

If you need to fetch instances using an operator other than equality, use `fetchAllUsingAttributesValuesOperatorsOrderBy`.

This method requires four arguments:

- an array of attribute names (String [])
- an array of attribute values (Object [])
- an array of operators (String [])
- an array of {attribute name, sorting order ("asc" or "desc")} for sorting (String [] [])

In the following example, we fetch all customers whose first name begins with A and whose ID > 102 (the results are ordered by first name):


```

System.out.println("Fetching...");
try {
    lpcCustomer e;
    String attribs[] = { "Fname", "Id" };
    Object values[] = new Object[2];
    values[0] = "A%";
    values[1] = new Integer(102);
    String operators[] = { "like", ">" };
    String orderBy[][] = { { "Fname", "asc" } };
    Vector v =
        customerFactory.fetchAllUsingAttributesValuesOperatorsOrderBy(
            attribs, values, operators, orderBy);

    for (int i=0; i < v.size(); i++) {
        e = (lpcCustomer) v.elementAt(i);
        System.out.println(
            e.getId() + " " + e.getFname() + " " + e.getLname());
    }
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("Broker fetch error - " + e.getMessage());
}

```

Fetching using SQL queries

JdbcStore includes a facility (called the row broker) to handle fetches using arbitrary SQL queries. There are three main reasons you may want to use this facility:

- using complex selection criteria not handled by the fetch methods (e.g. correlated and nested subqueries, etc.)
- immediate instantiations of object and their associations by using join queries
- performance

When you use joins in arbitrary SQL queries, JdbcStore will instantiate the main objects of the query as well as any associated mapped objects retrieved by the query. For example, if you need to get sales orders as well as their sales order items (let's say for a report) it is much more efficient to use a join query. If you do not use one, then for each sales order JdbcStore will issue a query when accessing its sales order items. When using the join query, one call to the database is required to fetch both sales orders and their associated items.

Arbitrary SQL queries can either be submitted through a string in the application or statements can be defined at design time in the workbench (cataloged SQL).

Just like the fetch method, a factory implements the fetching using arbitrary SQL statements. In addition, JdbcStore requires you to specify three items:

- the SQL query
- the parameter values (use ? in the SQL statement as place holder for a parameter)
- the related types which the query should instantiate

Using AdHoc SQL

To execute an Adhoc SQL statement, use the factory method `fetchUsingSql`.

The following example fetches and displays sales orders and their order items:

First we create the factories


```

System.out.println("Creating the factory");
SalesOrderFactory salesOrderFactory = new SalesOrderFactory();
salesOrderFactory.broker(broker);
SalesOrderItemsFactory salesOrderItemsFactory = new
SalesOrderItemsFactory();
salesOrderItemsFactory.broker(broker);

```

Then we define the related object types. In this case the main object type is SalesOrder. We will then invoke fetch on its factory. The related object types consist of the SalesOrderItems.

```

LPCObjectType[] relatedTypes = new LPCObjectType[1];
relatedTypes[0] = salesOrderItemsFactory.baseType();

```

We then invoke fetchUsingSql. Since the query has no parameters, we pass the empty array values for parameter values.

```

lpcSalesOrder order;
lpcSalesOrderItems item;
Vector items;
Object[] values = new Object[0];
Vector result;
result = salesOrderFactory.fetchUsingSql(sql, values, relatedTypes);

System.out.println("number of orders: " + result.size());

```

Now JdbcStore has instantiated both the SalesOrder and the SalesOrderItems. No further calls to the database are required from this point on.

```

for(int i=0;i<result.size();i++) {
    order = (lpcSalesOrder)result.elementAt(i);
    System.out.println(
        "id: " + order.getId() + " region: " + order.getRegion());
    items = order.getSalesOrderItemss();
    for(int j=0;j<items.size();j++) {
        item = (lpcSalesOrderItems)items.elementAt(j);
        System.out.println(
            "    id: " + item.getLineId() + " product: "
            + item.getProdId());
    }
}

```

Using persistent instances (LPCPersistence protocol)

Updating and inserting new instances

Persistent objects know whether they have been modified or created. To store an object, you need to store it. If the object has been modified or created, the appropriate calls will be issued to update the database.

In the following example, we create and store a customer:


```

lpcCustomer c = customerFactory.newInstance(new Object[0]);
c.setLname("Charpentier");
c.setCity("Don Mills");
c.setId(new Integer(9001));
c.setCompanyName("LPC Consulting Services, Inc.");
c.setPhone("(416)510-2015");
c.setFname("Luc");
c.setZip("M3A 1N1");
c.setAddress("7 Geraldine Ct");
c.setState("ON");

// store
c.store();

```

In this example we update all the sales orders.

```

Vector si = s.getSalesOrderItems();
for (int k = 0; k < si.size() ; k++) {
    lpcSalesOrderItems oi = (lpcSalesOrderItems) si.elementAt(k);

    // update quantity
    oi.setQuantity(new Integer(oi.getQuantity().intValue() + k) );

    // store
    oi.store();
}

```

Deleting instances

Deleting persistent instances is almost as simple as updating or inserting. Since the framework cannot get the objects that should be deleted, first mark them for deletion using the `markDeleted` method.

The following example illustrates deletions. We remove the customer and the sales orders created in the previous example:

```

System.out.println("Creating the factory");
customerFactory = new CustomerFactory();
customerFactory.broker(broker);
lpcCustomer c;

System.out.println("Fetching...");
try {
    Object[] key = new Object[1];
    key[0] = new Integer(9001);
    c = (lpcCustomer) customerFactory.fetchForKey(key);
    System.out.println("to be deleted: " + c.getId() + " " +
        c.getLname() + " " + c.getFname());
    c.markDeleted();

    // get the sales orders
    Vector salesOrders = c.getSalesOrders();
    for (int i = 0 ; i < salesOrders.size() ; i++) {
        lpcSalesOrder s = (lpcSalesOrder) salesOrders.elementAt(i);
        s.markDeleted();
        s.store();
    }

    c.store();
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("Broker fetch/store error - " + e.getMessage());
}

```

Working with relationships

When you define a model, you define the relationships between your objects. These relationships may be defined as one-to-one or one-to-many and may be implemented by uni-directional or bi-directional links.

In our sample database, we have customers who have one or more sales orders. Each sales order is associated with only one customer. In turn, a sales order has zero or more sales order items. Each sales order item is associated with one product.

Fetching related objects

To fetch related objects, you need to use the accessor provided for the relationship. This generated accessor is named as follows:

- `getClassName` - for one-to-one relationships where `ClassName` is the name of the class for the associated object
- `getClassNames` - for one-to-many relationship (plural form of the one-to-one relationship)

One-to-many relationships return a vector; one-to-one relationships return an object.

In the following example we display all of the customers, their sales orders with sales order items and the product description.

The relationships are accessed using:

- `getSalesOrders()`: returns a vector of sales orders
- `getSalesOrderItems`: returns a vector of sales order items
- `getProduct()`: returns a product


```

System.out.println("Fetching all the rows");
try {
    lpcCustomer e;
    Vector v = customerFactory.fetchAll();

    // for each customer
    for (int i=0; i < v.size(); i++) {
        e = (lpcCustomer) v.elementAt(i);
        Vector so = e.getSalesOrders();
        System.out.println(e.getLname()+" "+e.getFname());

        // for each sales order
        for (int j=0; j < so.size() ; j++) {
            lpcSalesOrder s = (lpcSalesOrder) so.elementAt(j);
            System.out.println(
                "\t"+ s.getId() + " " + s.getOrderDate());
            Vector si = s.getSalesOrderItemss;

            // for each sales order item
            for (int k = 0; k < si.size() ; k++) {
                lpcSalesOrderItems oi =
                    (lpcSalesOrderItems) si.elementAt(k);
                System.out.println(
                    "\t\t" + oi.getProduct().getDescription()
                    + " " + oi.getQuantity());
            }
        }
        System.out.println();
    }
} catch (Exception e) {
    e.printStackTrace();
    System.out.println("Broker fetch error - " + e.getMessage());
}

```

That's all there is to it. You need to use the relationship accessor to fetch related objects. If the relationship has already been instantiated, the objects are returned. Otherwise, they are fetched from the database and then returned.

Adding a relationship instance (one-to-many)

JdbcStore creates methods to add and remove new instances of a relationship. These methods are generated in the extension class.

To add an instance of a one-to-many relationship, use `addClassName` where `ClassName` is the name of the class (e.g. the many side of the relationship).

In the following example, we add a sales order to our newly created customer.

```

lpcCustomer c
...
// creating sales order
lpcSalesOrder s = salesOrderFactory.newInstance(new Object[0]);

// add it to the customer
c.addSalesOrder(s);

s.setCustId(c.getId());
s.setId(new Integer(1));
s.setOrderDate(new java.sql.Date( System.currentTimeMillis()));
s.setFinCodeId("r1");
s.setRegion("Central");
s.setSalesRep(new Integer(195));
s.store();

```


Removing a relationship instance (one-to-many)

The generated method to remove a relationship instance is `removeClassName` where `className` is the name of the related class (e.g. the many side of the relationship).

In the following example, we remove the relationship we added in the previous example:

```
c = (lpcCustomer) customerFactory.fetchForKey(key);
Vector salesOrders = c.getSalesOrders();
for (int i = 0 ; i < salesOrders.size() ; i++ ) {
    lpcSalesOrder s = (lpcSalesOrder) salesOrders.elementAt(i);
    if (s.getId().equals(new Integer(1))) {
        // remove from relationship and delete
        c.removeSalesOrder(s);
        s.markDeleted();
        s.store();
    }
}
```

Note that removing a relationship instance and storing the object from which the instance was removed, DOES NOT STORE (AND DELETE) the removed object. You must explicitly invoke store on the removed object to remove it from the database

Adding/Setting a relationship instance (one-to-one)

To add an instance of a one-to-one relationship, use `setClassName` where `ClassName` is the name of the other class involved in the relationship.

In the following example, we add the product to a sales order item:

```
lpcSalesOrderItems i = salesOrderItemsFactory.newInstance(
    new Object[0]);
i.setProdId(new Integer(300));
Object[] o = new Object[1];
o[0] = new Integer(300);
// fetch and set the product
lpcProduct p = (lpcProduct) productFactory.fetchForKey(o);
i.setProduct(p);
i.store();
```

Removing a relationship instance (one-to-one)

To remove an instance of a one-to-one relationship, just invoke the setter with no arguments.

```
Vector salesOrderItems = s.getSalesOrderItemss();
for (int j = 0 ; j < salesOrderItems.size() ; j++) {
    lpcSalesOrderItems soi = (lpcSalesOrderItems)
        salesOrderItems.elementAt(j);
    // remove product relationship
    soi.setProduct();
    soi.setProdId(null);
    soi.markDeleted();
    soi.store();
}
```

Caching

Caching overview

JdbcStore allows you to cache your objects in caches. There are two main reasons to use the cache:

- performance
retrieval by key will first check whether the object is in the cache. If so it will be retrieved from the cache and no calls will be issued to the database. If not the object will be fetched from the database and cached.
- managing instances
when you do not use a cache, two retrievals of the same objects will produce two (different) instances (i.e. you will end up with two objects, whose attribute values are equal). In this case, you are responsible for managing these duplicate instances.
When using the cache for any retrieval, JdbcStore will check if the object is already present in the cache (for non-key retrieval, this occurs after the fetch). If the object is already in the cache, the cached object is substituted in the result vector. Therefore, you will always have one single instance of an object in your application.

JdbcStore supports both a permanent and a dynamic cache:

- permanent cache
the static cache is implemented using a hash table. Therefore, after an object is cached, it will not be garbage collected until the cache is flushed.
This type of cache is appropriate for small sets of non-volatile objects.
- dynamic cache
the dynamic cache is implemented using Sun's VM extensions (i.e. `sun.misc.Cache`). If you have specified this cache and it is available in the runtime environment of your application, then it is used.
Otherwise, no cache is used.
When you use this cache, unreferenced cached objects will be garbage collected as needed by the Java VM.

You can detect whether dynamic caching is supported at runtime. If it is not supported, you may set the cache type of the objects to a static cache instead of a dynamic cache. If you do so remember to flush the cache periodically.

Caching and cache options are defined for each object type in your model. You could therefore create an application using a static cache for small static sets of objects, dynamic caching for your business objects, and no caching for descriptive objects.

Dynamic cache availability

There are three constants defined in `LPCCache`. These can be used to set the type of cache used by an object type.

- NULL_CACHE - no caching
- DYNAMIC_CACHE - cache using sun.misc.Cache
- PERMANENT_CACHE - cache using a hashtable

You can check whether DYNAMIC_CACHING is available by invoking `LPCSystem.sunCacheAvailable()`.

The following example checks the dynamic cache availability and sets the appropriate caching for the Customer object type:

```
customerFactory = new CustomerFactory();
customerFactory.broker(broker);

// check if Dynamic caching is available
if (LPCSystem.sunCacheAvailable()) {
    System.out.println("Dynamic caching is available");
    customerFactory.baseType().cacheType(LPCCache.DYNAMIC_CACHE);
} else {
    System.out.println("Dynamic caching is not available");
    customerFactory.baseType().cacheType(LPCCache.PERMANENT_CACHE);
}
```

Displaying cache statistics

You can obtain the number of cache hits and misses from the object type.

The following example displays the cache hits and misses for the customer type.

```
// Display cache hits and misses
System.out.println("Before flush - Cache hits: " +
    customerFactory.baseType().cache().cacheHits());
System.out.println("Before flush - Cache misses: " +
    customerFactory.baseType().cache().cacheMisses());
```

Flushing the cache

You can empty the cache by using `flush()`.

```
// flush the cache
customerFactory.baseType().cache().flush();
```

When you flush the cache, non-referenced instances will be candidates for garbage collection. However, referenced instances will be removed from the cache but will not be candidates for garbage collection. When using a permanent cache, you should ensure that no references point to any persistent cached objects in a cache before flushing it.

There is no need to flush the dynamic cache.

Beans

JdbcStore Beans

JdbcStore supplies the following BeanInfo classes

- LPCFactoryBeanInfo - a BeanInfo class used by all of the generated factories
- LPCSqlOrbBeanInfo: a BeanInfo class for the LPCSqlOrb class.

JdbcStore also generates BeanInfo classes for all of the generated factories (if selected in the workbench).

Java Beans support will be refined in future versions of the product.

Advanced Programming

Accessing JDBC connection information

A `java.sql.Connection` provides a number of methods to access information regarding a JDBC connection. Information can be obtained either directly through the `java.sql.Connection` (e.g. `getTransactionIsolation`, `getCatalog`, etc.) or through a `java.sql.DatabaseMetaData` obtained through `getMetaData`.

The `LPCSqlConnection` class encapsulates the `java.sql.Connection` class.

Use the accessor connection to access the instance of `LPCSqlConnection` associated with a broker.

Use the `LPCSqlConnection` method `getMetaData()` to obtain an instance of `DatabaseMetaData` (see the JDBC API documentation for its protocol).

Using model and type meta-data

The `JdbcStore` model and type protocols allow you to access all of the mapping information at runtime. You can use this information to develop generic code to handle any type and its relationships.

The `SampleApplication` uses this information to display any type, its relationships and handle the creation, deletion and modification of persistent objects.

Retrieving types

To retrieve the type names contained in a model, you can use the model method `typeNames()`.

```
String[] typeNames = model.typeNames();
for (int i= 0 ; i < typeNames.length; i++) {
    typeList.addItem(typeNames[i]);
}
```

Once you have the name of the type, you can use the method `typeNameed` to obtain the instance of the `LPCObjectType`.

```
LPCObjectType o = model.typeNamed(typeList.getSelectedItem());
```


Getting type information

From a type, you can obtain information about its attributes and relationships.

The following methods can be used:

- `allAttributes` - returns all of the attributes (including inherited ones) from the type. Instances of `LPCAttribute` are returned.
- `allRelationships` - returns all of the relationships (including inherited ones) from the type. Instances of `LPCObjectRelationship` are returned.
- `attributes` - similar to `allAttributes` but including only the ones defined in the type (no inheritance).
- `relationships` - similar to `allRelationships` but including only the ones defined in the type (no inheritance).

Using attribute information

An `LPCAttribute` contains information about its Java and database types.

You can use the following methods:

- `columnName` - the name of the mapped column
- `fieldName` - the name of the mapped Java field
- `getter` - the method (`java.lang.reflect.Method`) used to retrieve the field value
- `javaType` - an instance of `LPCJavaType` describing the mapped Java field
- `jdbcType` - and instance of `LPCJdbcType` describing the mapped JDBC column
- `setter` - the method (`java.lang.reflect.Method`) used to set the field value

With a `LPCJavaType`, you can obtain the class of the Java field by using the method: `typeClass()`.

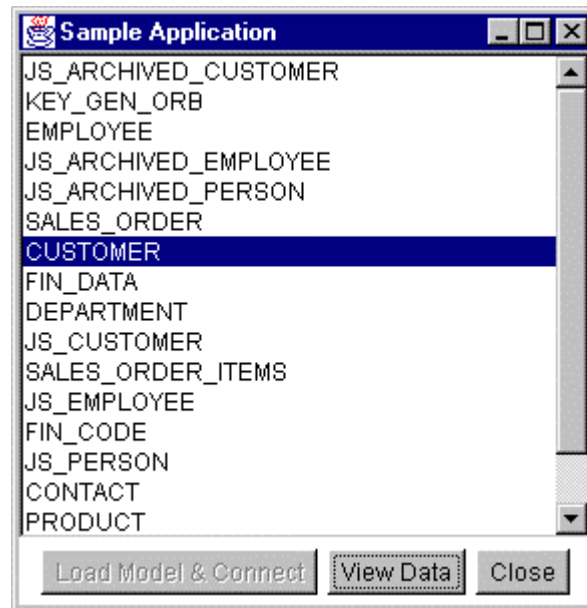
Using relationships information

When working with a `LPCObjectRelationship`, you can use the following methods:

- `isOneToMany()` - answers true if the relationship is one-to-many
- `isOneToOne()` - answers true if the relationship is one-to-one
- `relatedObjectType()` - the `LPCObjectType` participating in the relationship

Sample application

The JdbcStore sample application allows you to edit any persistent types defined in a model.



Sample Application

To load and connect the model, click on the Load Model & Connect button

To view and edit instances, select a type and click on the View Data button.

The sample application uses the example settings. See “Changing the example settings” in the “Customizing the examples” chapter of this manual.

A type is displayed using the LPCTablePanel.

CUSTOMER				Relationships
Lname	City	Phone	CompanyN...	SALES_ORDER
Devlin	Rutherford	2015558966	The Power ...	
Reiser	New York	2125558725	AMF Corp.	
Niedringhaus	Paoli	2155556513	Darling Ass...	
Mason	Knoxville	6155555463	P.S.C.	
McCarthy	Carmel	3175558437	Amo & Sons	
Phillips	Middletown	2035553464	Ralston Inc.	
Colburn	Raleigh	9195555152	The Home ...	
Goforth	Chattanooga	6155558926	Raleigh Co.	
Gagliardo	Hull	8195559539	Newton Ent.	
Agliori	Columbus	6145552496	The Pep Sq...	
Ricci	Syracuse	3155554486	Dynamics I...	
McDonough	Brooklyn Park	6125555603	McManus Inc.	
Kaiser	Minneapolis	6125553409	Lakes Inc.	
Chopp	St Paul	6125556453	Howard Co.	
Phillips	St Paul	6125556425	Sterling & Co.	
Gugliuzza	Mamaroneck	9145553817	Sampson &...	
Morgan	Westerville	6145558989	Square Spo...	
Sanford	Raleigh	9195555152	Raleigh Acti...	

Add Modify Delete Refresh Print

Sample table display

The relationships and attributes are retrieved dynamically from the type.

You can then edit the instances:

Modify CUSTOMER	
Lname	Goforth
City	Chattanooga
Phone	6155558926
CompanyName	Raleigh Co.
Address	11801 Wayzata Blvd.
Zip	37421
Fname	Matthew
State	TN
OK Cancel	

Sample type edit dialog

From the table display, you can also print out the table entries by clicking on the Print button.

The sample application demonstrates the power of using type meta -data to develop generic components.

Swing (JFC) components

The current version of JdbcStore supports the JFC JTable. Other JFC components will be supported in the future.

The following classes are implemented:

- `com.lpc.jdbcstore.run.LPCJTableDataModel` - this extends the JFC `AbstractTableModel`
- `com.lpc.jdbcstore.swingUI.LPCTablePanel`

The table panel is used to display persistent instances.

To use it, do the following:

```
lpcTable = new LPCTablePanel();
LPCJTableDataModel dataModel = lpcTable.getDataModel();
dataModel.setFactory(factory);
dataModel.setRows(rows);
```

First we create a new `LPCTablePanel`. This `tablePanel` will then create its own instance of a table model.

Then we specify in the Table data model the factory class and the rows of the type to be displayed.

That's all there is to it. With very few lines of code you can display persistent instances in a tabular format.

See the Java document and the sample application for more details.

Customizing code templates

JdbcStore uses templates to generate classes. These templates are located in the directory: `com\lpc\jdbcstore\workbench\lpctemplate\default`.

The code generation relies on substitution of place holders to substitute appropriate values such as field and method names, etc.

```
// Attribute property descriptor
pd = new PropertyDescriptor("%ATTRIBUTE_NAME%", beanClass,
"%GETTER_NAME%", "%SETTER_NAME%");
pdVector.addElement(pd);
```

PropertyDescriptorTemplate.java

To customize a template, create a new directory and copy the contents of this directory into it (you can set the template directory in the workbench through the settings dialog).

Do not modify the place holder names (%XXXX%).

Delivering JdbcStore Applications

Java applications

When you deliver a JdbcStore application, you need to provide the following:

- a copy of your model (for applications, you can load using `loadFromFile`)
- the classes from `com.lpc.jdbcstore.run`
- the classes from `com.lpc.jdbcstore.swingUI` (required only if your application uses the JdbcStore swing components)
- any class required by your application

You cannot (and do not need to) release any classes from the workbench directory.

Java Applets

When you deliver a JdbcStore application, you need to provide the following:

- a link to your model (you will need to use `loadFromURL`)
- the classes from `com.lpc.jdbcstore.run` and `com.lpc.jdbcstore.swingUI` (required only if your application uses the JdbcStore swing components)
- any class required by your application

You may want to repackage the classes into JAR files. Some browsers currently do not support dependencies between JAR files. In this case you will need to package all the classes into one JAR file.

You cannot (and do not need to) release any classes from the workbench directory.

Working with the examples

Examples overview

JdbcStore includes a number of examples to illustrate the use of its class libraries. The Java files for the samples are located in the directory:
com\lpc\jdbcstore_source\examples.

The files included are:

AutoKeyGenerationDatabase

This example illustrates the use of keys generated by the database. It uses the sample table KEY_GEN_DB.

AutoKeyGenerationOrb

This example illustrates the use of keys generated by the broker. It uses the sample table KEY_GEN_ORB.

BidirectionalExample

This example illustrates the use of bi-directional relationships. It uses the tables CUTOMER and SALES_ORDER.

BrokerConnectExample

This example shows how to connect a broker using the connectionInfo class.

CacheExample

This example demonstrates setting the caching strategy at runtime.

DeleteExample

This example shows how to remove stored instances.

You should run the InsertCustomerExample before running this example.

DirectDriverConnectExample

This example shows how to connect a broker without using the connectionInfo class.

DriverExample

This example shows how to obtain information on registered JDBC drivers.

FetchAllExample

This example shows how to use the Factory fetchAll method.

FetchForKeyExample

This example demonstrates fetching instances by key.

FetchUsingAttributesValuesExample

This example demonstrates fetching instances using arbitrary attributes.

FetchUsingAttributesValuesOperatorsOrderByExample

This example demonstrates fetching instances using arbitrary attributes and operators (i.e. operators other than “=”).

FetchUsingAttributesValuesOrderByExample

This example demonstrates fetching instances using arbitrary attributes and sorting the instances returned.

InsertCustomer

This example shows how to create new persistent instances.

InsertJsCustomer

This example shows how to create new persistent instances that inherit from a superclass.

InsertJsEmployee

This example shows how to create new persistent instances that inherit from a superclass.

LoadModelFromFile

This example loads a model from a local file.

LoadModelFromUrl

This example loads a model using an URL.

LPCTableExample

This example uses the LPCTable panel and metadata from the model to display a type's instances in a JFC table panel in a Java application.

RemoveRelationshipExample

This example demonstrates deleting persistent instances and removing them from a relationship.

RowBroker

This example demonstrates how to use arbitrary SQL queries to fetch persistent objects and their related objects from the database.

TableExampleApplet

This example uses the LPCTable panel and metadata from the model to display a type's instances in a JFC table panel in a Java applet.

UpdateSalesOrderItems

This example demonstrates how to update persistent instances.

Customizing the examples

Except for the following examples:

- BrokerConnectExample
- DirectDriverConnectExample
- DriverExample
- LoadModelFromFile
- LoadModelFromUrl

that require individual customization (since they demonstrate connecting to a database or loading a model), the other examples implement a framework that enables you to customize them for your environment quickly.

Settings such as model file name, driver, URL, userid, password and other connection properties are maintained in a serialized file. A program is provided to serialize the settings.

Changing the example settings

To change the settings, edit the file CreateExampleSettings in the jdbcstore-source\examples sub-directory.

The following example is extracted from this program:

```
static public String url = "jdbc:dbaw://localhost:8889/Watcom/SQL  
Anywhere 5.0 Sample/SQL Anywhere 5.0 Sample";  
static public String user = "dba";  
static public String password = "sql";  
static public String driverName = "symantec.itools.db.jdbc.Driver";  
static public String modelFileName =  
"\\Development\\com\\lpc\\jdbcstore\\models\\SampleModel.ser";  
  
/* do not change this */  
static public String settingsFileName = "lpcExamplesSettings.ser";  
  
static public void main (String[] args) {  
  
    LPCCConnectionInfo connectionInfo = new LPCCConnectionInfo();  
  
    connectionInfo.url(url);  
    connectionInfo.property("user",user);  
    connectionInfo.property("password",password);
```

You can modify the URL, user, password, driverName or modelFileName variables. If you need to add new connection properties, add a line such as the following:

```
connectionInfo.property("MyProperty","MyPropertyValue");
```

After your changes are complete, compile and run the program. A serialized file of your settings named lpcExamplesSettings.ser will be created.

This file is used by most of the examples, since LPCEXAMPLESsystem.getConnectedModel() will load the model and connect to the specified URL.


```
try {
    model = LPCEXamplesSystem.getConnectedModel();
    broker = model.orb();
}
catch (Exception e) {
    e.printStackTrace();
    System.out.println("*** Error - " + e.getMessage());
    try { System.in.read(); } catch (IOException eio) { return; }
    return;
}
```

Creating the sample database

The examples provided by JdbcStore rely on a database containing the required tables.

Using SQL anywhere

A sample SQL anywhere database (jstore.db and jstore.log) is provided. It is located in the directory: sample_database\sql anywhere".

The database contains all of the tables and data to run the examples.

Using Access

A sample Access database (jstore.mdb) is provided. It is located in the directory: sample_database\Access.

The database contains all of the tables and data to run the examples.

Other databases

SQL is provided to define all the tables required for the examples. The SQL is located in the directory: sample_database\ddl\demo_ddl.sql.

You may have to modify the supplied SQL for your database

Sample data is also provided for the tables. It is provided in CSV (Comma Separated Values) format.

The following data files are provided:

- SALES_ORDER: 161.dat
- SALES_ORDER_ITEMS: 162.dat
- CONTACT: 163.dat
- CUSTOMER: 164.dat
- FIN_CODE: 165.dat
- FIN_DATA: 166.dat

- PRODUCT: 167.dat
- DEPARTMENT: 168.dat
- EMPLOYEE: 169.dat

Using the sample model

A model has been defined containing all of the types and relationships required by the examples.

Recompiling the examples

You can use the following command files to recompile the examples and samples:

- javac_examples_classes.bat: recompile the sample classes
- javac_examples.bat: recompiles the examples
- javac_swingUI.bat: recompiles the LPC JFC components
- javac_sampleApplication.bat: recompiles the sample application

You will need to modify the above command files to specify the directory where JdbcStore was installed.
