INTRODUCTION TO

CA Jasmine ODB

For Microsoft Windows 2000 Professional
Revised spring 2004

1 Introduction

Jasmine™ from Computer Associates (CA) is a large object-based eBusiness platform with numerous tools and services, ranging from business process identification, object-oriented database and application development, to dynamic report generation, support on business knowledge and multimedia management. This introduction focuses on the Jasmine Object Database (ODB), leaving much of the eBusiness aspects unmentioned.

The powerful ODB approach enables us to think about and model our domain in real-world terms without force-fitting data into traditional tables, rows, and columns as in traditional relational databases. Jasmine stores everything - entities, characters, numbers, pictures, video, and audio as objects. The benefits with this are numerous, inheritance, dependencies and querying is made easy through the use of fully object-oriented concepts and the *Object Database Query Language* (ODQL).

It's even possible to map your own database models directly into your programming language of choice (for example Java, C++, Visual Basic etc), meaning that the DBMS is able to use database models to generate application basics. Even the opposite is possible; you could just as easily turn your application data layer into a database model in Jasmine. Naturally this speed up the development process rapidly since the effort to transform conceptual models into database models is reduced. When the DBMS produces the foundation of the application code it also lessens the gap between the application and the database.

The purpose of this document is to familiarize and present common tasks and concepts found in the Jasmine ODB environment. The aim is not to provide a complete and in-depth description of the whole eBusiness enterprise but rather to point out key aspects that are helpful to students at the Department of Computer and System Sciences (DSV) at the University of Stockholm / Royal Institute of Technology (KTH) in Sweden when performing different course assignments. For a complete and detailed documentation of Jasmine, please access the CA ftp server, using anonymous login at: ftp.ca.com/pub/jasmine/docs/nt_202

1.1 Remarks about this tutorial and last minute changes

This tutorial's sole purpose is to make Jasmine as easy, fun and comprehendible as possible. With this in mind we also know that this introduction is far from complete and that it only targets parts of the whole DBMS environment. Vise from experience we know that errors are going to be introduced by this text, ranging all from typos to misunderstandings, so therefore it's recommended to keep an open mind to what is presented. ***Please do not try to skip through the text since that increases the risk of misinterpretation dramatically. Try to read as much as possible!***

1.2 Some Reading Guidelines…

Actions: This icon symbolizes operations you are expected to perform by yourself. This can for example be typed commands or mouse clicks on user interface buttons.

Recommendations: This icon represents additional tips that are not required but could help speed things up as you go along.

Warnings: These symbols point out things to avoid and well-known sources of errors.

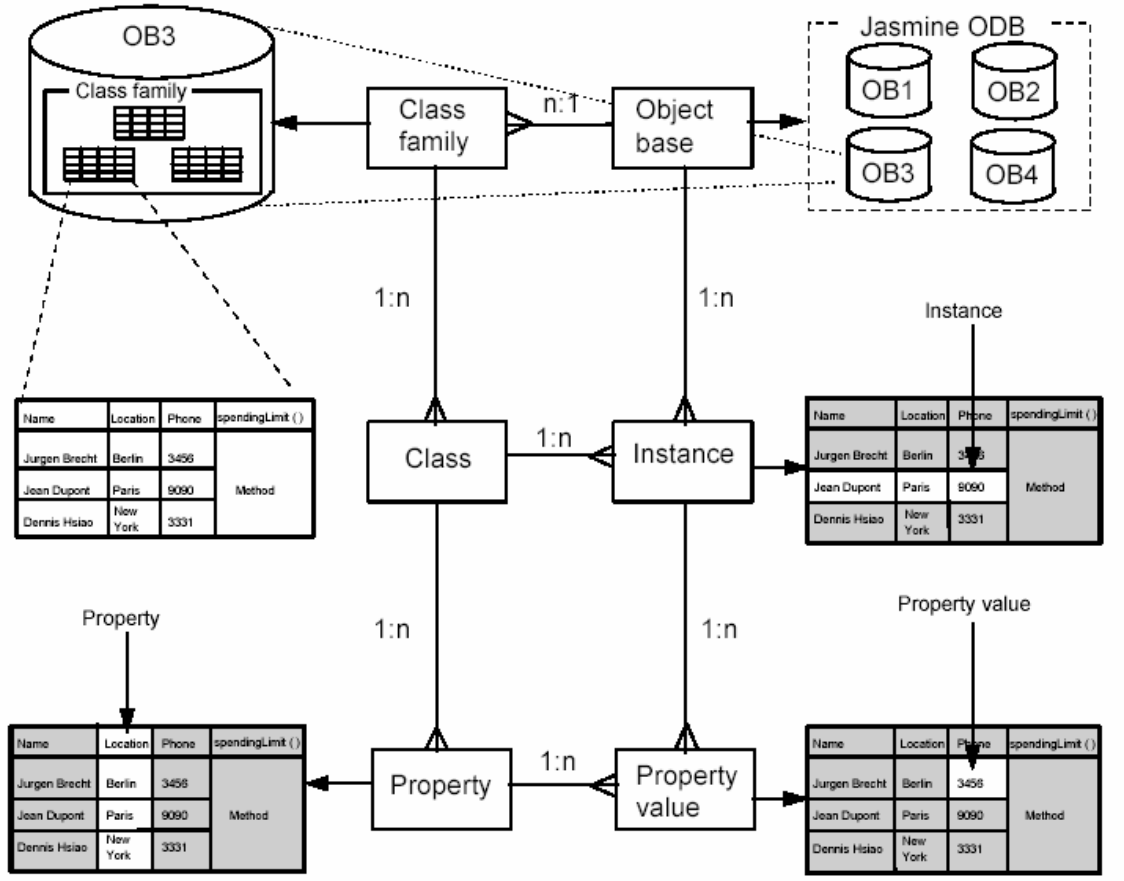
Important text: This format marks extra important parts in the text that we want to emphasis because we consider it to be vital to the understanding of the text and to the progress of your work.

`COMMANDS`: This denotes text that is used as input to the DBMS. The commands used in Jasmine are usually case sensitive.

Answers: Represents an expected feedback from the system given as a reply to user input.

2 Overview of the Jasmine ODB

This chapter presents vital concepts commonly encountered when working with Jasmine ODB. A short description of the data structures starts up this chapter and a discussion on *Jasmine Studio* and the ODQL interpreter follows.



2.1 Stores, Extents and Class families

Like most other database management systems (DBMSs), Jasmine ODB needs to specify the file space allocated on the physical drive where the data should be kept. In Jasmine ODB such a place is called a *Store*. A store can essentially be one of four different kinds where the two most notable are the *System Store* (that contains all system classes, third-party products and meta-information) and the *User Store* (which contains user defined class families that stores class definitions, method definitions and object instances). The system and all user stores together make up the database.

Stores are made up of *Extents* that are actually files on the physical drives. Thus by using multiple extents, a store can span several drives and be as large as your operating system permits. Optimization is achieved by placing stores for different applications on different physical drives.

Since Jasmine is object-oriented, classes and class instances (objects) are a vital concern of the DBMS. Related classes and their instances form *Class Families*. A store can hold several uniquely named class families. For more information refer to the *Jasmine ODB – Database Design and Implementation* manual found at: ftp.ca.com/pub/jasmine/docs/nt_202

2.2 Jasmine Studio versus the ODQL Interpreter

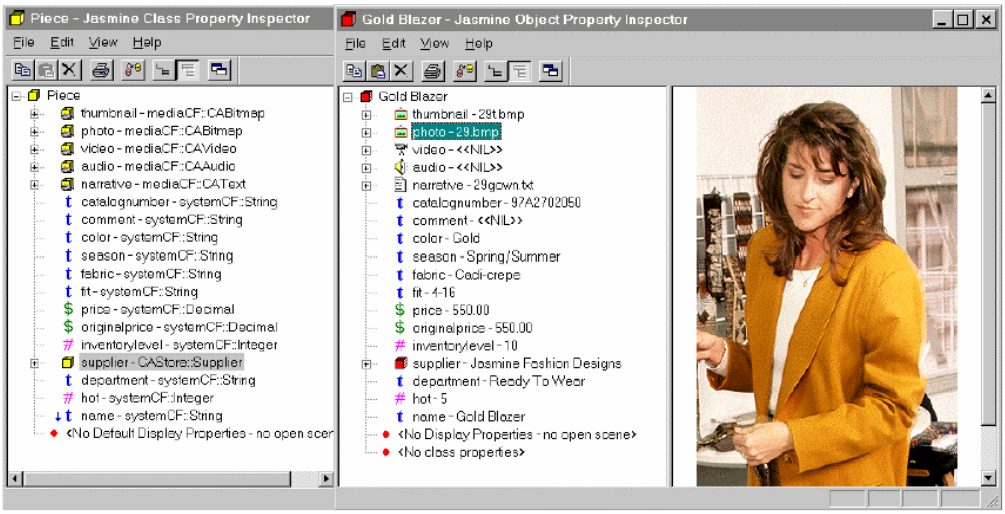
Jasmine commands and utilities do basically have two major purposes:

First the *Operating system commands* (programs) that act on the database's physical file storage: backup and restore, allocating file storage, defining class families, reading and defining configuration settings, starting and stopping the database engine, connecting to the database (from a client) etc. These commands are commonly used to install and configure the DBMS and are not often used for educational purposes.

Secondly, the *Database commands* and methods that act at the class and instance level, on items stored inside the database. They include the *Object Definition Language* (ODL), *Object Manipulation* (OML) and *Query Language* (OQL). These commands can only be executed within Jasmine methods, i.e. the *ODQL interpreter* or Jasmine client programs like the *Jasmine Studio*. In chapters 3.1 and 3.2 we'll be using operating system commands to set up the Jasmine DBMS environment, while in chapter 4 is dedicated for database commands.

If we want to go into the internals of our database, we have two main tools to use: *Jasmine Studio*, or the *ODQL interpreter*. Choosing the correct tool for each situation is very important, and impacts on your ability to administer, migrate, upgrade and support your database. The *Jasmine Studio* is a point and click graphical user interface tool for browsing the database schemas and their contents (the class instances) as well as for changing the database structure. The advantages of this tool are:

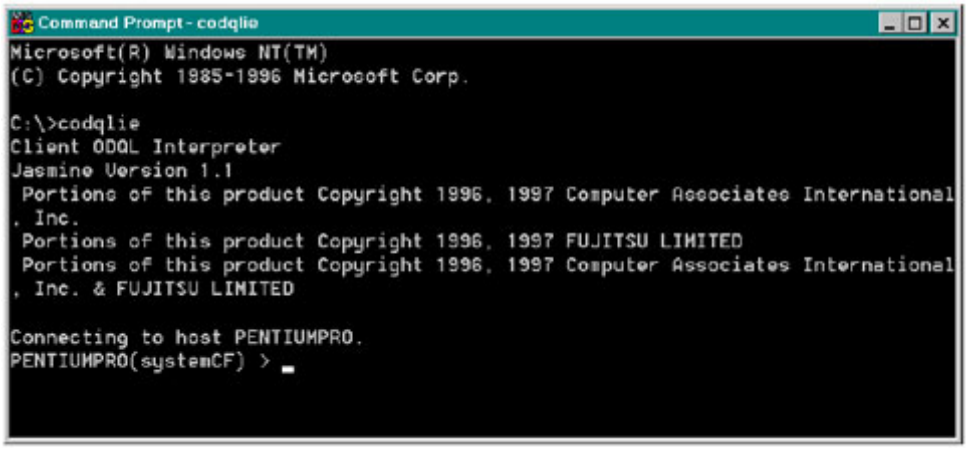
- It's very easy to learn and use.
- Only practical way to view multimedia data instances.
- Drag and Drop is convenient way of populating data, especially multimedia types.
- Building classes and compiling methods is a one-touch operation.
- It is an essential party to building applications with Jasmine.



The *ODQL interpreter* is the antithesis of the Studio. It is a command line environment which only accepts ODQL language input. While this makes the interface hostile to the casual user, it's a very powerful tool. Consider it for the following:

- It's the only way to run ODQL scripts. These could be testing code for methods, creating instances of data, or generally automating some task.
- It's the only medium which supports the bulk creation of test data (apart from using the LOAD utility, which assume you already have the database created on another server).
- It supports detailed inspection of all class attributes, instance data, etc.

- Classes defined here can select the type of collection properties they use (Array, List, set or Bag). The Database Administrator does not allow this for some reason.
- You can run ANY type of test query, including the *Group By* construct which the Studio does not support at all.
- Use the ODQL Class to help you build queries, and the Tuple structures required to hold the results. This can get quite messy, and the ODQL class is a great help when writing methods.
- You can run test scripts and capture the output to text files, thus offering a way of documenting/validating work done.



As you notice, the interpreter is used for two things: handling complex operations, and writing/running ODQL scripts. It should also be noted that the tool is unforgiving when it comes to errors. Minor errors in case can cause long chains of messages to appear and errors in methods can be awkward to track down. Complexity aside, those of you who are already familiar with other DBMSs will soon see the benefits of the ODQL interpreter.

Being able to record all database schema information, schema changes, compilation instructions and even data unloading and loading into scripts which can be executed, and re-executed at will is a vital part of database administration. If you lose a server and have to reconstruct the database from a particular point in time, you need full logs of all changes, and scripts to run to bring it up to par quickly. This is definitely not something you use a point-and-click tool for.

3 Setting up the Jasmine ODB environment

The following section addresses steps necessary to run Jasmine on the school environment at DSV in Kista. ***These steps are VERY important since the default Jasmine setting must be configured locally on each computer in order for the DBMS to run properly!*** Or put in other words – skip this chapter and you cannot continue at all, because things will not work!

You only need to do the changes made in sections 3.1 and 3.2 the first time you access the DBMS. Once the environment is properly set up, no further configuration of the Jasmine ODB should be needed.

3.1 Specifying the remote server

Log on to the computer with your database account to get database administrator privileges. Use the account information you received during account registration in the beginning of this course.

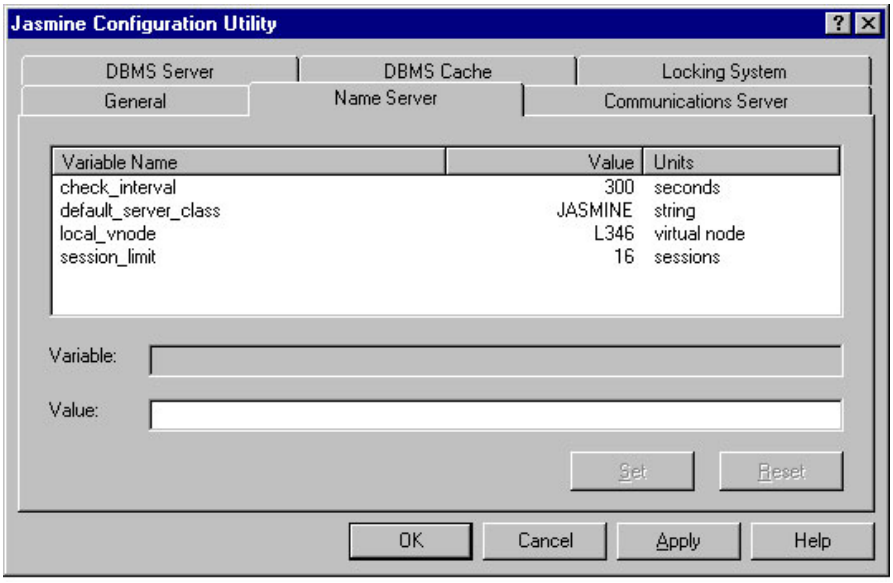
Open a *Windows Command Prompt* by issuing the command: **`cmd`** at the start-menu path: ***Start » Run…***

In the *Windows Command Prompt* that opens, type in the command: **`hostname`** to retrieve your computer name.

`l346` (this varies for every computer)

Start the *Jasmine Configuration* application found at the start-menu path: ***Start » Programs » DatabaseManagementSystems » Jasmine ii » Jasmine Configuration***

In the *Jasmine Configuration Utility* window, select the *Name Server* tab and change the parameter *local_vnode* to the name you received from the hostname command previously (in this example **`l346`**). Mark the *local_vnode* row and type in your hostname in the *Value* field. Then press *Set* to commit your changes and then *OK* to close the window.



Stop the Jasmine server by typing in the command: **`netStopJasmine`** in a *Windows Command Prompt*.

```
Net stop "Jasmine_Database"
The Jasmine service is stopping.
The Jasmine service was stopped successfully.
```

Or, if you don't have the Jasmine service started you get:

```
M:\>net stop "Jasmine_Database"
The Jasmine service is not started.
More help is available by typing NET HELPMSG 3521.
```

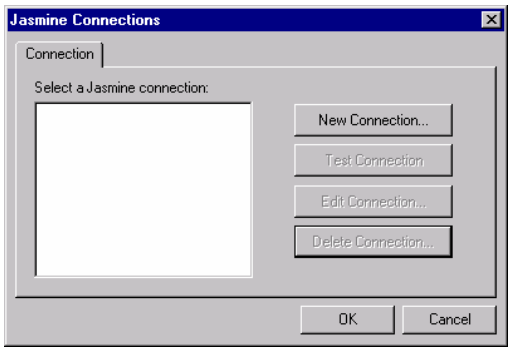
Restart Jasmine with the *Windows Command Prompt* command: **netStartJasmine**

```
Net start "Jasmine_Database"
The Jasmine service is starting. . . . . .
The Jasmine service was started successfully.
```

3.2 Connect to the local Jasmine installation

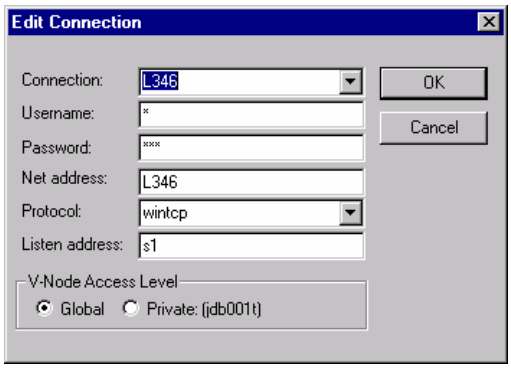
After the remote Jasmine server is set up on your local machine, start up *Jasmine Studio* by using the start-menu path: ***Start » Programs » DatabaseManagementSystems » Jasmine ii » Tools » Jasmine Studio***

In the *Jasmine Connections* window that appears, press the *New Connection…* button.



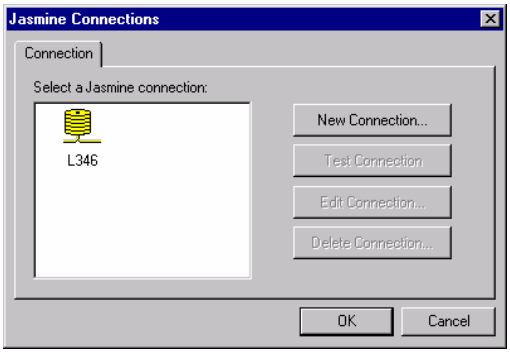
In the *New Connection* window, change the parameters as follows:

Connection: **l386** (the same hostname as you got earlier)
Username: *****
Password: **is4**
Net address: **l386** (the same hostname as you got earlier)
Protocol: **wintcp**
Listen address: **s1**
V-Node Access Level: **Global**



Press *OK* to create a new connection to your local Jasmine server and to terminate the *Add Connection* window.

From now on you can use the existing local connection icon in the *Jasmine Connections* window to access your databases. Double-click it to use it.



4 Getting started with the Jasmine Tools and the ODQL

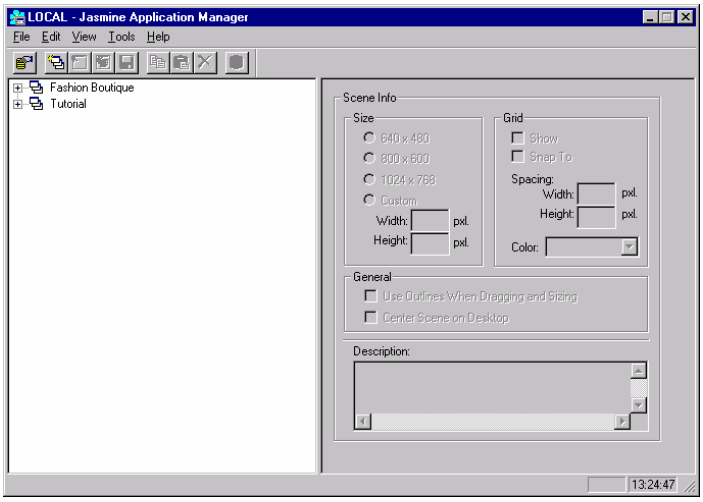
This chapter presents a short description on how to perform database commands through *Jasmine Studio* and the ODQL interpreter. It also gives a short introduction to the Jasmine ODB specific ODQL language.

4.1 Jasmine Studio

This section covers how to start up and work with the graphical user interfaces in Jasmine Studio, the *Jasmine Application Manager* and *Jasmine Class Browser*.

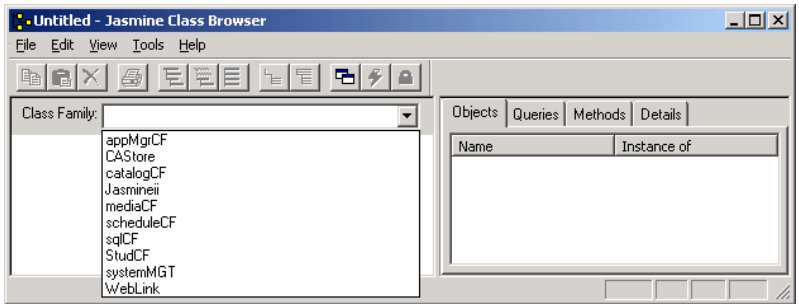
Start Jasmine Studio by following the start-menu path: ***Start » Programs » DatabaseManagementSystems » Jasmine ii » Tools » Jasmine Studio***

Double-click the local connection icon in the *Jasmine Connections* window to bring up the *Jasmine Application Manager*. This tool is used to create applications from within Jasmine trough the graphical user interface. As you can see there are two examples of applications available: *Fashion Boutique* and *Tutorial*. If you are interested, please feel free to get familiar to the examples and the functionality of the *Application Manager*; however this tutorial does not cover any of its functionalities.

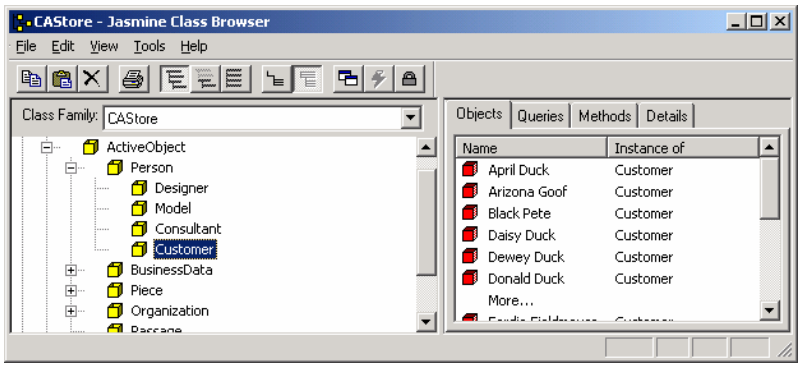


In the *Jasmine Application Manager*, choose: ***File » Database Administration…*** from the menu-bar to start the *Jasmine Class Browser*. The *Class Browser* is used to visualize and manipulate user data.

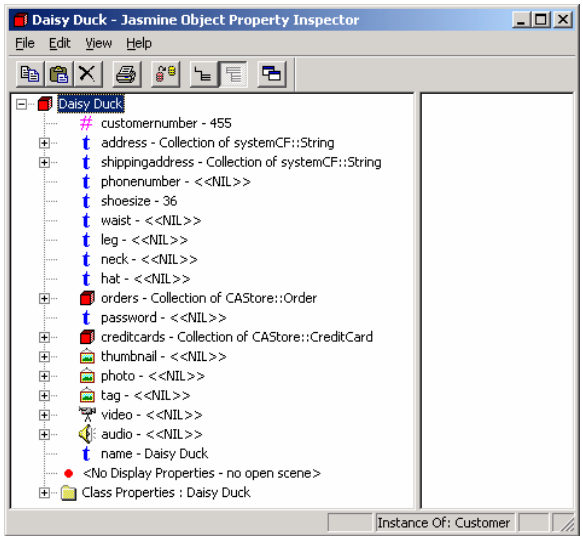
To access the data, select your *Class Family* from the drop down list and note that all the classes are shown in the hierarchy to the left as yellow boxes.



Select the class you want by clicking on its yellow icon and any instance of the class is shown on the right hand side as red boxes. Use the tabs on the right to specify queries and view methods of the class.



Double-clicking on a red boxed instance brings up the *Jasmine Object Property Inspector* where the values of the specific instance could be viewed and changed. Note that objects can in turn consist of other objects, both as collections and as singletons. These objects are also represented as red boxes within your selected object.



4.2 Using the ODQL interpreter (CODQLIE)

ODQL is the querying language used in Jasmine ODB. It is really more of a programming language that's fully capable of sequences, selections and loops. It can be used directly from a command line environment through the ODQL interpreter named *codqlie* or embedded in a programming language similar to SQL. Using embedded ODQL is outside of the scope of this introduction and the following steps just show how to start and stop *codqlie* and issue ODQL queries both interactively and as script through this tool.

The Jasmine server must be running in order for the codqlie tool to work. Use the netStartJasmine command in a Windows Command Prompt to check this.

To start codqlie in *interactive mode*, start a *Windows Command Prompt* and issue the command: **`codqlie`**

```
Client ODQL Interpreter
Jasmine ii ODB
Portions of this product Copyright 1996-2001 Computer
Associates International, Inc.
Portions of this product Copyright 1996-2001 FUJITSU LIMITED
Portions of this product Copyright 1996-2001 Computer
Associates International, Inc. & FUJITSU LIMITED
Connecting to host L398.
L398(systemCF) >
```

Issue an ODQL command by typing in your query row by row, ending each line with the character "**`;`**". Execute the row by pressing return. (User input in bold):

```
M:\>codqlie
Client ODQL Interpreter
Jasmine ii ODB
Portions of this product Copyright 1996-2001 Computer
Associates International, Inc.
Portions of this product Copyright 1996-2001 FUJITSU LIMITED
Portions of this product Copyright 1996-2001 Computer
Associates International, Inc. & FUJITSU LIMITED
Connecting to host L398.
L398(systemCF) > defaultCF CAStore;
L398(CAStore) > OrderItem set ois, oi;
L398(CAStore) > ois = select OrderItem from OrderItem;
L398(CAStore) > scan(ois,oi){oi.name.print();};

Apple Red Windbreaker
Beige Bag
Beige Crocheted Vest
Black Satin Sandal
Black Satin Sandal
Miniature Bag
White Pantsuit
Red Bag
```

… (and so on until end)

```
L398(CAStore) >
```

To quit the *codqlie* shell, type the command: **`end;`**

Another way of issuing commands is through scripts. Specify your whole ODQL query in a text file and place it on a drive. Process the script by giving the text file as a *–execFile* parameter to codqlie. Type in: *codqlie –execFile <drive:\directory\filename>* in a *Windows Command Prompt* to execute the script. In this example we execute the file *test.odql* placed at *m:\myQueries* by the command: **`codqlie -execFile m:\myQueries\test.oqdl`**

```
Client ODQL Interpreter
Jasmine ii ODB
Portions of this product Copyright 1996-2001 Computer
Associates International, Inc.
Portions of this product Copyright 1996-2001 FUJITSU LIMITED
Portions of this product Copyright 1996-2001 Computer
Associates International, Inc. & FUJITSU LIMITED
Connecting to host L398.
Apple Red Windbreaker
Beige Bag
Beige Crocheted Vest
Black Satin Sandal
```

… (and so on until end)

A good thing is to use batch-files in Windows to automate processes. A batch-file is just like a text file with one command on a single row. Create a new text document and type in your commands into it (for example just **`codqlie`**) and save it as *run.bat.* Next, give the command: **`run`** to execute every command you entered in the batch-file. (In this case it will just start codqlie). Use this technique to perform many commands at once but remember to issue the run command in the same location as you placed the batch-file.

The example below creates a folder named *myTest* on the *m:* drive, copies the file *myQuery.odql* to this location and executes *codqlie* with the file as a value given to the parameter *–execFile*. The results are piped out to the file *results.txt* which in turn is opened for editing through notepad.

```
@echo off
rem This text is treated as a remark
echo Creating folder...
md m:\myTest
echo Copying file...
copy c:\document\myQuery.odql m:\myTest
echo Executing codqlie...
codqlie –execFile m:\myTest\myQuery.odql > results.txt
echo Starting notepad...
notepad results.txt
echo End of script!
```

4.3 ODQL

This chapter presents an introduction to syntax and methods necessary to develop useful Jasmine ODQL queries.

The easiest way to run ODQL queries against Jasmine ODB is through the ODQL interpreter (codqlie) from a *Windows Command Prompt*, as mentioned in section 4.2. Codqlie takes a number of parameters where the most important ought to be *–execFile*. This parameter runs the file specified as a script against the database. Other useful parameters are easily displayed through the *–h* parameter which displays the full argument list.


All rows issued in codqlie have to have en ending semicolon "**;**".


It is possible to commentary text right in the middle of the ODQL text. Use the **/*** markup to initiate the start of a comment, and ***/** to terminate it.


All commands in ODQL are case sensitive, so check spelling and names carefully since Jasmine has a poor error feedback. Many of the reserved words are spelled in lower case, check the correct syntax in *Jasmine Database Developer's reference* found at: ftp.ca.com/pub/jasmine/docs/nt_202.


ODQL is not OQL! OQL is the foundation for ODQL but the syntax and functionality differ quite a bit. (For example ODQL cannot traverse through collections with the '.' qualifier. These need to be scanned through). In return you are able to use IF-THEN-ELSE statements etc. Do not expect every OQL syntax to work in Jasmine!

4.3.1 Writing an ODQL query

Every ODQL query acts within a class family and this is the first thing that needs to be specified when creating a query. (For a description of class families, refer to section 2.1). If you do not change the class family, the default family *systemCF* will be used instead, effectively disallowing you access to any user defined data. To change the class family, use the command *defaultCF <class_family_name>* to specify what class family that are of interest. In the following example we access the user defined class family *CAStore*:


Start a codqlie session by opening a *Windows Command Prompt* and issue the command: **codqlie**


Type in: **defaultCF CAStore;**
...

Next thing we need to do is to specify variables to be used during the processing of the query. This includes *Bags*, *Lists* and *Sets* as well as *Integers*, *Strings*, *Reals* and *Dates*. Bags (unordered collections allowing duplicates), Lists (ordered collections allowing duplicates) and Sets (ordered collections without duplicates) are *collection classes*, meaning that they could consist of any number of variables of a given type.

In contrast, singletons are single data items of specific types. This means for example that a bag could consist of a number of string variables but a singleton could never consist of anything else than a value. To make an example we define a select query, retrieving only one shoe size (***a string singleton NOT a collection***) and put the result in our specified string variable named *str*:

```
...
String str;
str =      select Customer.shoesize
from Customer
where Customer.name == "Donald Duck";
...
```

This only works because the query returns a single value and not a collection of values.

Finally, when we have the result, we need to present the information to the user. This can be done through the *print* method which is inherited to all system-defined variables. Specify the following command to print out the content of our *str* variable and end the query by specifying the *end* command:

```
...
str.print();
end;
```

```
42
```

4.3.2 Using collections

If you want to work with collections things get a little more complicated. The following example retrieves a collection of singletons. We begin by specifying the class family and then declare our collection by using the syntax *<data_type> set <collection_name>*. We also specify an order item variable named *order_item* that we are going to use later on. The first part *data_type* defines what kind of data the collection shall hold. The argument *collection_name* is the handle to this object and the keyword *set* is used to specify that the variable is a collection.

```
defaultCF CAStore;
OrderItem set order_item_collection;
OrderItem order_item;
...
```

It is possible to declare several variables of the same type on a single row. The following example expresses the same as the rows above:

```
defaultCF CAStore;
OrderItem set order_item_collection, order_item;
...
```

This creates an unordered collection of order items named *order_item_collection*. So far, this is an empty collection since we haven't allocated any order items to it yet. To do this we add the select statement:

```
...
order_item_collection = select OrderItem from OrderItem;
...
```

This will put all order item instances from the database and put it in our collection. To print the collection's content we can use the same *print* method as before but there is a better way. By using the ODQL method *scan(<collection>,<singleton-handle>){}* we make the printout look nicer. The scan method takes two arguments, a collection to scan and a singleton handle of the same data type as the content of the collection. This is why we needed to declare the singleton variable *order_item* earlier:

```
...
scan(order_item_collection,order_item){
      order_item.name.print();
};
end;
```

```
Apple Red Windbreaker, Beige Bag... (etc.)
```

This would go through the whole bag of order items and, for each loop, retrieve the order item attribute *name* and print this to the user. The singleton *order_item* acts as a handle for the active attribute in the *order_item_collection* during the scan loop.

Remember that stepping through data with the '.' qualifier only works on one-to-one relations. As soon as you go from a single object to a collection (one-to-many) you have to use some sort of iteration to get the data.

4.3.3 Additional useful methods

There are a couple of useful functions in ODQL that saves a lot of workarounds. The first one is the *hasElement* method. This only works on collections and scans trough it to find matches of the given argument. If the object is found, the method returns true. If not, it returns false. The following example searches for any accessory that is not colored blue. It's important only to search for object of the same data type as of the objects in the collection.

```
/* Show only non-blue accessories */
defaultCF CAStore;
String set strset;
strset = select Accessory.name
from Accessory
where not(Accessory.colors.hasElement("blue"));
strset.print();
```

```
Bag{ Red Umbrella, Beige Bag, Pigskin Bag, Red Bag, Large bag,
Red Sun Hat, Black Satin Sandal, Loafer, Sandal }
```

Other important functions are *union*, *differ* and *intersect* which basically works the same as their equivalents in SQL. Consider the following:

```
Integer set X, set Y, set result;
X = Bag{1,2,3};
Y = Bag{3,4,5};
result = X.union(Y);
result.print();
end;
```

`Bag{ 1,2,3,4,5 }` (all in X added to all in Y)

```
Integer set X, set Y, set result;
X = Bag{1,2,3};
Y = Bag{3,4,5};
result = X.differ(Y);
result.print();
end;
```

`Bag{ 1,2 }` (those in X minus those in Y)

```
Integer set X, set Y, set result;
X = Bag{1,2,3};
Y = Bag{3,4,5};
result = X.intersect(Y);
result.print();
end;
```

`Bag{ 3 }` (both in X and in Y)

It is useful to print additional text in your queries. This can be done by defining a string variable and adding text to this. Then use the *print* method to display the text:

```
…
String textOutput;
textOutput = “This is a message”;
textOutput.print();
…
```

Do not use undeclared variables or collections! If you fail to provide the instance, you cannot use the specified handle. Consider the following:

```
Integer set is1, set is2, set is3, set is4, set is5;
is1 = Bag{1,2,3,4};
is2 = Bag{1,3,5,7};
is3 = Bag{};
is4 = Bag{};
is3 = is1.union(is2);
is3.print();
is3 = is1.intersect(is2);
is3.print();
is4.directAdd(5);
is4.directAdd(81);
is4.print();
is4 = is4.add(73);
is4.directRemove(81);
is4.print();
is5.print();
is5.directAdd(5);
is5.print();
```

This would result in the following output:

```
Bag{ 4,2,1,3,5,7 }
Bag{ 1,3 }
Bag{ 5,81 }
Bag{ 5 }
NIL
NIL
```

This happens because the undeclared collection *is5* cannot be used since it haven’t been instantiated through the = *Bag{}* declaration. Collections or attributes that are NIL cannot be used! Also always use the method *directAdd* instead of the add method to put additional singletons in a collection.

Of course it is also possible to use ordinary program language constructs like *IF-THEN-ELSE* and *WHILE* in ODQL. The following example checks if any of the bags contains the integer value *2* and print out the result.

```
Integer set is1, set is2;
is1 = Set{1,2,3,4};
is2 = Set{1,3,5,7};
if (is1.hasElement(2)) {
    is1.print();
};
if (is2.hasElement(2)) {
    is2.print();
};
```

```
Set{ 1,2,3,4 }
```