



Konada.Db - User Guide

Accessing Oracle Databases with Ada

Frank Piron, KonAd GmbH *

July 2, 2007

Abstract

Konada.Db is an Ada library for easy Database Access with reasonable performance. Actually only Oracle databases are supported. Konada.Db for Oracle is built on top of Dmitry Anisimkov's Ada Oci library, an Ada Binding to the Oracle Call Interface.

Konada.Db for Oracle supports Bind Variables, Multiple Connections, Asynchronous Procedure Calls and an easy to use SQL interface to lob's - large objects. The management of Define and Bind Variables is done by the Library.

This Guide contains a tutorial introduction to Konada.Db. A basic knowledge of Oracle Databases and Ada programming is necessary to understand the material.

*In der Reis 5, D-79232 March-Buchheim, e-mail: frank.piron@konad.de,
<http://www.konad.de>

<i>CONTENTS</i>	2
-----------------	---

Contents

1 Preface	5
2 Installation Instructions	6
2.1 Prerequisites	6
2.2 Basic Installation	6
2.3 Test of the Installation	7
3 Tutorial Preparation	9
4 Connecting to the Database	10
4.1 Establishing a Connection	10
4.2 The Default Connection	12
4.3 The Utility Console Logon	12
4.4 The Source Code	13
5 Database Errors	14
6 Executing Sql Statements	15
6.1 Interaction with the Database System	15
6.2 A simple Example	16
6.3 Transaction Control	17
6.4 Handling Errors	19
6.5 Memory Management	20
6.6 The Source	20
7 The Select List	21
7.1 Sql Statement processing	21
7.2 Retrieving Simple Select List Information	22
7.3 The Konada.Db Typecat System	23
7.4 Retrieving detailed Select List Information	26

<i>CONTENTS</i>	3
8 Fetching and Extracting Data	29
8.1 A Simple Example	29
8.2 Date and Number Formats	30
8.3 Extracting to other Ada Datatypes	30
8.4 Handling Null Values	32
9 Using Bind Placeholders	33
9.1 Why Bind Placeholders?	33
9.2 Explicit Binds	37
9.3 Setting Values and implicit Binds	38
9.4 Data Extraction from Binds	41
10 Calling Stored Units	43
10.1 Calling Stored Units using Binds	43
10.2 Managing the SQL Instance	44
11 Persistent Types	46
11.1 An example	46
11.2 Setters and Getters	47
12 Asynchronous Execution	49
13 Locking	51
13.1 How to avoid Blocks	51
13.2 The Lock() Utility Procedure	52
14 Large Objects	53
14.1 Introduction	53
14.2 Lob Support in Konada.Db	53
14.3 Extracting Lobs	54
14.4 Modifying Lobs	54
14.5 Binary Get() and Set()	55

CONTENTS

4

15 List of Examples

56

1 Preface

To make Ada known to a broader developer community there is a need for some Standard Applications written in Ada. The strict type concept of Ada together with its OO capabilities make it a suitable language for developing database applications.

KonAd GmbH is currently developing a workflow engine which heavily uses Oracle Packages. To reuse developer skills in PL/SQL which is much like Ada83¹ and to take advantage of Ada's ability to manage big software systems we decided to use Ada to build our workflow client library.

Since it is not easy to get support for SQL*Module² and on the other hand the Oracle Call Interface³ gives greater flexibility, we decided to build an Ada database access library based on the OCI.

Some web search lead us to the Ada Oci binding written by Dmitriy Anisimkov. This binding to the OCI contains almost all of the OCI features including asynchronous procedure calls and basic Lob support⁴.

We built Konada.Db on top of Ada Oci to free the library user from the management of define and bind variables. In Konada.Db this is done via an opaque type `Sqltype`.

Binding variables is semi automatic and define is not necessary at all. Extraction functions for getting Data retrieved by a select statement are providing internal conversions to `Ada.Calendar.Time`, `Integer`, `Long_Float` or `String`. Konada.Db introduces a small set of typecodes which is Ada centered and hides nearly all database specific type constructs. Further the handling of lob's with SQL is implemented in a way that you may use lob's almost like other data.

But it should be said, that the heavy work was done by Dmitriy Anisimkov in writing Ada Oci. Thanks to him for the permission to include Ada Oci in this distribution and for his rapid bug fixing.

This Guide is intended to make the developer familiar with Konada.Db and to provide first advice for database programming with Ada.

¹At least in the basic language constructs

²An Oracle precompiler which permits embedded SQL in Ada

³OCI

⁴Lob = Large Object. You may save large binary or text objects into the database

2 Installation Instructions

2.1 Prerequisites

Before you can develop programs with Konada.Db you need the following components on your machine:

- One of the OS win98/nt/2000/xp, Gnu Linux⁵ or Sun Solaris⁶
- An Oracle Client installation with access to an Oracle database and an Oci version $\geq 8.1.6$. In your Database you should also have a “scott” schema installed⁷. You may join the Oracle technology network and download Oracle software for free. See www.otn.oracle.com.
- An Ada Compiler. Konada.Db is tested with the gnat compiler⁸ which is maintained by Ada Core Technologies. For free Download see <http://libre.adacore.com>.
- The Konada.Db distribution which contains the Ada Oci Binding of Dmitriy Anisimkov.

2.2 Basic Installation

First follow the instructions in the README file from the distribution top level directory. If you could not build `KONADA_ROOT\lib\dummy.exe` on windows then check if you have an environment variable `ORACLE_HOME` set to the root of the Oracle software tree before calling `make_konada`. This is the main cause for failure.

If you could not build the dummy executable on UNIX⁹ then most likely your UNIX account is missing an Oracle setup.

- Check that your `ORACLE_HOME` environment variable is set correctly.

⁵Konada.Db was tested successfully on Debian with gnat 3.15p and gnatgcc 3.4.2

⁶We tested successfully Solaris 8/9/10 on a Sparc 64 Bit Machine with gnat 3.15p, gnatgcc 3.4.2 and the pthread runtime.

⁷This is a very small database schema owned by user scott which the Oracle documentation refers to.

⁸As of June 2007 Konada.Db ist tested with gnatgcc 3.4.2, the gnat version which is bundled with gcc 3.4.2

⁹If i write “UNIX” i include Gnu/Linux of course

- Check that your PATH contains \$ORACLE_HOME/bin
- Check that your LD_LIBRARY_PATH contains \$ORACLE_HOME/lib

2.3 Test of the Installation

If not already done build demo1, demo2, demo3 in KONADA_ROOT\samples\simple according to the instructions in KONADA_ROOT\README. Now run demo1 from the command line (DOS Box on Windows) with:

```
> demo1 scott/tiger@<tnsname>
```

where <tnsname> is your database access descriptor. If you omit the tnsname then Konada.Db assumes local. If you call demo1 without arguments then the program asks for connection information. After logon enter an employee number and optionally change the name. Quit with entering 0 as an employee number.

Note on Windows Installations:

On windows there could pop up a message box saying that the procedure entry point for *OciEnvCreate* could not be found. In this case check your Installation. May be that you have more than one Oci.dll on your machine and there is one older version loaded at runtime which you not specified implicitly by setting ORACLE_HOME when building Konada.Db. Also if you have more than one Oracle software trees on your machine then use the Oracle Home Selector to set the Oracle Home to the one given when you built Konada.Db.

Demo2 is similar but creates a function emp_desc in the scott schema and then calls this function for an employee. Before termination the function is dropped.

Demo3 runs without interaction showing synchronous and asynchronous statement execution. Demo3 temporarily creates a procedure wait() in scott's schema.

If you have a recent installation of Gwindows on your machine you may also build KONADA_ROOT\samples\gui\gui_example but check the README first. Gui_Example provides a simple GUI frontend for SQL statement execution.

Finally you may build and run KONADA_ROOT\samples\lob\lobdemo. This

program manages images of employees in the scott schema. But follow carefully the instructions given in the README since some database administration activities are necessary before running lobdemo.¹⁰

¹⁰You have to create a table and load lob data.

3 Tutorial Preparation

In the following sections we will show the features of Konada.Db and give guidelines for development. Also some background information concerning the internals of Oci is presented.

To understand the content of this tutorial you should have basic knowledge of Oracle Databases e.g. you should know what is a TNS-Name, what is a Database Instance and how to write SQL statements. Also you should have learned the fundamentals of Ada programming.

The examples discussed in this tutorial are stored in `KONADA_ROOT\tutorial`. `Tut_4.2` is the second example of section no. 4¹¹ in the tutorial.

In some tutorial examples a TNS-Name "tut" is hardcoded. You should create such a TNS-Name in your Oracle environment pointing to a Database Instance with a *scott schema* installed to run these examples without modification.

The scott schema is necessary to run the examples. If you don't have one in your database then

- Create a user `scott` with password `tiger` in your database.
- Grant the roles `Connect` and `Resource` to him.
- Import the file `KONADA_ROOT\scott.dmp` to the user `scott` with the command:

```
>imp system/<password>@<tnsname> file=KONADA_ROOT\scott.dmp  
fromuser=scott touser=scott
```

The executable for `imp` may have a different name on your system like `imp80` or similar.

In the tutorial all pathnames are given in windows notation using "\"¹².

¹¹Connecting to the Database

¹²This definitely does *not* indicate any preference of the author for the windows operating system

4 Connecting to the Database

Before you can do anything with an Oracle database in your program you have to establish a *connection*, open a *session* and allocate a *server context*. In Konada.Db there is a *Connection_Type* and instances of this type contain and manage all these objects.

After a connection is established all subsequent actions may be performed in the context of this connection. You may establish more than one connection in your application program. In subsequent actions you may then specify with respect to which connection a particular action should be performed e.g. a select statement should be executed or a procedure should be called.

4.1 Establishing a Connection

And here is the most simple Konada.Db program. The *Hello World* of Konada.Db. `tut_4_1` creates a connection, outputs the server version and disconnects from the database. Below you see the line numbered source of `tut_4_1.adb`

Tut_4_1:

```
1  with Konada.Db.Connections; use Konada.Db.Connections;
2  with Text_IO; use Text_IO;
3  procedure Tut_4_1 is
4      Tut_Connection: Connection_Type;
5  begin
6      -- Logon
7      Logon(Conn => Tut_Connection,
8           Username => "scott",
9           Password => "tiger",
10          Database => "tut");
11      -- Output server version
12      Put_Line(Rdbms_Version(Tut_Connection));
13      -- And logoff. Not necessary here since done automatically
14      -- at end of program.
15      Logoff(Tut_Connection);
16  end Tut_4_1;
```

If Konada.Db is installed correctly and if `ADA_INCLUDE_PATH` and `ADA_OBJECTS_PATH` contain `KONADA_ROOT\lib` and `KONADA_ROOT\oci` then you may build `Tut_4_1`

with

```
> gnatmake -i tut_4_1 -largS -loci
```

on windows or

```
> gnatmake -i tut_4_1 -largS -lclntsh
```

on UNIX. Now if you run this little program the following may happen:

First Alternative is Success:

```
c:\konada\tutorial>tut_4_1
Oracle8i Enterprise Edition Release 8.1.7.0.0 - Production
With the Partitioning option
JServer Release 8.1.7.0.0 - Production
```

If you get an output similar to the above then all is fine.

Second Alternative is Database Error:

```
c:\konada\tutorial>tut_4_1
raised OCI.THICK.LIB_ERROR : ORA-12154: TNS: ... (language specific
error message)
```

The above error indicates, that you have no “tut” TNS-Name in your Oracle environment. The error is passed through by Oci.Thick to the main Unit. In the next section we will see how to handle Database errors. A different Database error would raise if you try to connect to an instance without a scott schema.

Third Alternative is OS Error:

On windows there could pop up a message box saying that the procedure entry point for *OciEnvCreate* could not be found. See the above Section *Test of the installation*[2.3](#).

The Logoff Statement in line 15 of the above listing is not really necessary since Logoff is called implicitly when Tut_Connection is destroyed.

4.2 The Default Connection

When your program does not need multiple connections then it is not necessary to mention connections at all in your code since Konada.Db manages a *Default Connection*. The Default Connection is always the first Connection established by the program and every function/procedure which has a *Conn* parameter takes this Default Connection if you don't supply a specific *Conn* value. Making use of this feature the Code of tut_4.1 simplifies to:

Tut_4.2:

```
1 with Konada.Db.Connections; use Konada.Db.Connections;
2 with Text_IO; use Text_IO;
3 procedure Tut_4_2 is
4 begin
5     -- Logon
6     Logon(Connect_String => "scott/tiger@tut");
7     -- Output server version
8     Put_Line(Rdbms_Version);
9 end Tut_4_2;
```

As you can see, no connection variable is declared. The default connection is used instead by the library. Also note the overloading of the logon procedure. You may supply an Oracle like connect string <user>/<password>@<tns_name> which contains the logon information. The default connection may also be useful in a multiple connection program since it may save keystrokes not to mention the mostly used connection everytime.

With the function/procedure `Get_Default_Connection()` and `Set_Default_Connection()` you can control the default connection in a multiple connection program.

4.3 The Utility Console Logon

In the package Konada.Db.Utills there is the overloaded Function `Console_Logon()` which permits a comfortable logon in console applications.

Tut_4.3:

```
1 with Konada.Db.Connections; use Konada.Db.Connections;
2 with Konada.Db.Utills; use Konada.Db.Utills;
```

```
3  with Text_Io; use Text_Io;
4  procedure Tut_4_3 is
5  begin
6      -- Logon, ask Information from user or get it from argv[1]
7      Console_Logon;
8      -- Output server version
9      Put_Line(Rdbms_Version);
10 end Tut_4_3;
```

Console_Logon() asks the logon information from the user. You may control the language of the prompted text by setting the variable Language in KONADA_ROOT\lib\Konada.ads. If you supply an Oracle connect string as the first command line parameter then Console_Logon uses this information and skips prompting.

```
c:\konada\tutorial>tut_4_3 scott/tiger@tut
Oracle8i Enterprise Edition Release 8.1.7.0.0 - Production
With the Partitioning option
JServer Release 8.1.7.0.0 - Production
```

In Konada.Db.Gui there is the procedure Gui_Logon which provides a little logon window. But you need Gwindows to build a Konada Gui program.

4.4 The Source Code

The spec for the konada connection package Konada.Db.Connections is KONADA_ROOT\lib\konada-db-connections.ads. Take a look at it. Note that the Session_Mode Parameter is not evaluated yet because holding multiple sessions on one connection is not implemented. Note also that you may test the availability of a connection with the function Connected().

Finally you will find the transaction control procedures Commit() and Rollback() which we will discuss later.

5 Database Errors

With *Database Error* i will denote an Error which occurs either in the Database Server or in the client's Oci Environment. If such an error occurs, an exception `Oci.Thick.Lib_Error` is raised inside the Ada Oci library which includes the Oracle error code and -message in its exception information. In most cases `Konada.Db` will pass through the exception to the application program. The other cases will be discussed in later sections.

You may call the functions `Sql_Error_Code` and `Sql_Error_Message` to retrieve the last Database Error Code as an Integer and the last Error Message as a string. These Functions are located in `Konada.Db.Sql`. Now look at the code snippet from `tut_5_1` which shows the block with the logon and an exception handler¹³.

Tut_5_1:

```
...
10 begin
11     Console_Logon;
12 exception
13     -- handle Oci Error
14     when Oci.Thick.Lib_Error =>
15         case Sql_Error_Code is
16             -- 12154=tnsname could not be resolved
17             when 12154 =>
18                 Put_Line("No Connection. Wrong tnsname!");
19             -- 01017=invalid username/password ...
20             when 1017 =>
21                 Put_Line("Authentication Failure. Check user/password!");
22             when others =>
23                 raise;
24             end case;
25             return;
26 end;
...
```

`Oci.Thick.Lib_Error` is caught and then dependent on `Sql_Error_Code` several actions may be performed. The Error Codes are listed in the Oracle Documentation¹⁴.

¹³Of course `Konada.Db.Sql` and `Oci.Thick` are withed in this example

¹⁴Oracle Database Error Messages

6 Executing Sql Statements

6.1 Interaction with the Database System

How does the interaction between an application program and a Database Server look like?

1. The application program establishes one or more connection(s) to the server.
2.
 - An Sql statement is send to the server or
 - A function/procedure is called on the server or
 - A transaction control statement¹⁵ is issued or
 - Errors are handled or
 - Data pieces retrieved form a `Select` statement are processed
3. The connection(s) are closed

In this section we will discuss Sql Statement execution and transaction control. Because the retrieval of data given by a `Select` statement is a more complex issue i will restrict the discussion in this section to DML statements¹⁶.

All data structures and procedures related to Sql statement execution in `Konada.Db` are packed into the type `Sqltype` declared in `Konada.Db.Sql`. The whole content of these data structures is called an *Sqlinstance*. When a variable of type `Sqltype` is declared it is initialized with an empty *Sqlinstance*. If you want to execute an Sql statement you proceed as follows:

1. Declare a variable of type `Sqltype`.
2. Establish a connection¹⁷.
3. Create an *Sqlinstance* providing a statement in the context of a connection.
4. Execute the *Sqlinstance*.
5. Commit or rollback the transaction

¹⁵*Commit or Rollback*

¹⁶These are *Insert*-, *Update*- and *Delete*-Statements

¹⁷and implicitly start a transaction

6.2 A simple Example

We start with a simple example. Suppose you want to change the name of the employee with number 7369 to "Adams". So you want to send the following Sql statement to the server:

```
update emp
  set ename='Adams'
 where empno=7369
```

The next example shows how to do it with Konada.Db:

Tut_6_1:

```
1  with Konada.Db.Connections; use Konada.Db.Connections;
2  with Konada.Db.Sql; use Konada.Db.Sql;
3  --
4  with Text_Io; use Text_Io;
5  -- for 'LF'=Linefeed
6  with Ada.Characters.Latin_1; use Ada.Characters.Latin_1;
7  --
8  procedure Tut_6_1 is
9      -- Declare variable
10     Change_Name: Sqltype;
11     Connection: Connection_Type;
12 begin
13     -- Logon
14     Logon(Conn => Connection,
15           Connect_String => "scott/tiger@tut");
16     -- Create an Sqlinstance in the context of "Connection"
17     Create(Newsql => Change_Name ,
18           Code    => "update emp          " &LF&
19                     " set ename='Adams'" &LF&
20                     " where empno=7369",
21           Conn    => Connection);
22     Execute(Change_Name);
23     -- Show how much rows were affected by the statement
24     Put_Line(Natural'Image(Rows_Processed(Change_Name)) &
25             " row(s) changed.");
26 end Tut_6_1;
```


In line 10 a variable of type `Sqltype` is declared which is initialized with an empty `Sqlinstance`. Then after logon with an explicit named connection the `Sqlinstance` is created in the context of the connection (lines 17-21). Of course we could have used no named connection and not supplied the *Conn* Parameter in the calls to *Logon* and *Create* thus forcing `Konada.Db` to use the default connection.

Now before you proceed you should inspect the table `emp` with `Sql*Plus` or your favorite query tool and ensure that the name of employee no 7369 has changed to "Adams". Note that no `commit` statement was necessary since `Oci` performs an implicit commit if you regularly log off. The normal completion of the program without exception thus includes a commit of all open transaction(s).

Look at line 24 of `tut_6_1` where the function `rows_processed()` is called to retrieve the number of rows affected by the dml statement. You may use this function also to give a warning to the user if the number of affected rows is big.

6.3 Transaction Control

With the opening of a new connection and with every *commit* or *rollback* statement a new transaction is started. Every transaction¹⁸ belongs to a connection/session and every dml statement belongs to a transaction. With the *commit* statement you end a transaction, make your database changes permanent and start a new transaction. With the *rollback* statement you make all changes caused by dml statements in the current transaction undone and start a new transaction.

If an `Oci` program terminates regularly all open transactions are committed. If it terminates caused by an exception all open transactions are rolled back.

The following example should clarify the control of transaction.

Tut_6_2:

```

1  with Konada.Db.Connections; use Konada.Db.Connections;
2  with Konada.Db.Sql; use Konada.Db.Sql;
3  --
4  with Text_Io; use Text_Io;
5  -- for 'LF'=Linefeed
6  with Ada.Characters.Latin_1; use Ada.Characters.Latin_1;
7  --
```

¹⁸in a non distributed environment

```
8  procedure Tut_6_2 is
9      -- Declare Sqlinstance variables
10     Change_Name1,
11     Change_Name2: Sqltype;
12     -- And Connection variables
13     Connection1,
14     Connection2: Connection_Type;
15     Choice: Character;
16 begin
17     -- Open two connections
18     Logon(Connection1, Connect_String => "scott/tiger@tut");
19     Logon(Connection2, Connect_String => "scott/tiger@tut");
20     -- Create an Sqlinstance in each connection
21     Create(Newsql => Change_Name1,
22           Code    => "update emp          " &LF&
23                     " set ename='Miller'" &LF&
24                     " where empno=7369",
25           Conn    => Connection1);
26     Create(Newsql => Change_Name2,
27           Code    => "update emp          " &LF&
28                     " set ename='Lopez'" &LF&
29                     " where empno=7499",
30           Conn    => Connection2);
31     -- Execute both
32     Execute(Change_Name1); Execute(Change_Name2);
33     -- ask user
34     Put("(1) commit only 1, (2) commit only 2, (3) commit both: ");
35     Get_Immediate(Choice);
36     case Choice is
37         when '1' =>
38             Commit(Connection1); Rollback(Connection2);
39         when '2' =>
40             Commit(Connection2); Rollback(Connection1);
41         when '3' =>
42             return;
43         when others =>
44             New_Line;
45             Put_Line("Illegal choice! All rolled back.");
46             Rollback(Connection1); Rollback(Connection2);
47     end case;
48 end Tut_6_2;
```

Again you may examine the data changes with SQL*plus and check whether all is like you expected.

Note that in the Example different rows are updated. Try to update the same row by setting "... empno=7369 ..." in the code of Change_Name2 in line 29. Terminate the program by sending a signal¹⁹ since it blocks itself by trying to update the same row from 2 different sessions. In section 13 when we will discuss *locking* we will see how to avoid such situations.

6.4 Handling Errors

As mentioned earlier most Database Errors are passed through by Konada.Db from Oci.Thick. However errors occurring during Execute() cause a Konada.Db.SQL_ERROR exception²⁰. The error message²¹ contains a pointer to a position in the statement code where the error cause most likely is located.

To get an example take tut_6_1.adb and change line 19 as follows:

```
19                                " set name='Adams'" &LF&
```

by changing ename to name. Build the executable and run it. You will get an error message:

```
c:\konada\tutorial>tut_6_1
```

```
raised KONADA.DB.SQL_ERROR : ORA-00904: invalid column name
```

```
update emp
  set  ->>> name='Adams'
  where empno=7369
```

Note that ->>> points to the wrong column name. Of course you may catch the exception and evaluate SQL_ERROR.CODE for individual error handling.

¹⁹E.g. by typing <CTRL>-C on your keyboard

²⁰Declared in KONADA_ROOT\lib\konada-db.ads

²¹Exception Information

6.5 Memory Management

Since variables of type `Sqltype` contain `Sqlinstances` with deep data structures you may wonder how memory is managed.

The type `Sqltype` is derived from `Limited_Controlled` and therefore memory is managed by the ada runtime support system. But if you want to release all dynamically allocated memory explicitly in your program you may call `Destroy()` for example `Destroy(Change_Name)`. The procedure `Destroy()` is defined in `Konada.Db.Sql`.

If you call `Create()` on a variable of type `Sqltype` which was already filled with an `Sqlinstance` by an earlier call of `Create()` then `Konada.Db` first finalizes the `Sqlinstance` before creating a new one. So you don't have to do this manually.

If you want to check whether a variable `var` contains an empty `Sqlinstance`²² you may call `Is_empty(var)` from `Konada.Db.Sql`. Later in 9 we will see examples in which this function is used to improve performance.

6.6 The Source

Take a look now at `KONADA_ROOT\lib\konada-db-sql.ads`, sections `INITIALIZATION` and `EXECUTION`.

Note that there is an additional boolean parameter `Async` in the parameter list of `Execute()` defaulting to `false`. We will see in 12 how to execute statements in a non blocking fashion. The procedures `Check_Finished()` and `Cancel()` are also related to this feature.

Finally note the function `Statement_Cat()` which returns an element of the enumeration type `Statement_Cat_Type=(dql, dml, other)`. You may use this function in your program when the statement code comes from an outer source²³. For example after detecting a `dml` statement you can display `Rows_Processed()` which does not make sense for a `ddl (=other)` statement.

²²That is: `Create(var)` was either never called or the `Sqlinstance` was finalized explicitly

²³Entered by the user for example

7 The Select List

7.1 Sql Statement processing

Before we go on to execute `select` statements and retrieve data, let's take a look on what the database does in the background when we send it an SQL statement. The steps below are specific to Oracle but every RDBMS has similar items:

1. Cursor Creation
2. Parsing
3. (`select`) Describe Results
4. (`select`) Define Output
5. Bind
6. Execute
7. (`select`) Fetch
8. Close Cursor

Each of the following paragraphs corresponds to a procedure in `Konada.Db.Sql` and summarizes the covered stages of SQL Statement processing.

Konada.Db.Sql.Create()

First a *Cursor* is created on the Server. The Cursor holds all server side data structures which are necessary to process the statement. In the second stage the Statement is *parsed*. This means that the statement is analyzed and translated into the RDBMS low level calls. An *Execution Plan*²⁴ is created. These two stages are covered by a call to `Create()` in `Konada.Db.Sql`.

Konada.Db.Sql.Execute()

If we have sent a query²⁵ then in the 3. stage the result is *described*. That is: the structure of the result is known now to the RDBMS and to the client library (Oci). In particular we may retrieve now the count of the columns of the result, the data types, the names and the lengths. The 4. stage is *define*. In the define stage the programmer has to provide storage locations - e.g. variables -

²⁴Strategy of the data retrieval

²⁵`select` Statement

into which the result values of a query are fetched²⁶. You would have to tell the RDBMS the datatypes of these variables and how many space in memory is given for each. The 5. stage, *bind* is discussed later in 9. In the 6. phase the statement is executed. In particular: a dml statement is submitted to the transaction and for a query the first data row is retrieved on the server. If the statement is an *update* then *locking*²⁷ is performed. All these stages - with the exception of *bind* - are performed by the `Execute()` procedure in `Konada.Db.Sql`. Especially you have not to provide *define* variables.

Konada.Db.Sql.Fetch()

Now for `select` statements the result rows may be fetched by the client program which is discussed in 8.

Konada.Db.Sql.Destroy()

Finally in the 8. stage the cursor is closed. This event takes place if you explicitly destroy an `Sqlinstance` using `Konada.Db.Sql.Destroy()` or if the ada runtime support system finalizes a variable of type `Sqltype`.

7.2 Retrieving Simple Select List Information

Every `Select Statement` returns a result which is structured like a table. The columns of the result table in their natural ordering constitute the *Select List* of the query. In this list every column has a unique position, a name, a datatype, a size and possibly a scale²⁸.

In this section i show how to determine the count of the columns, their names and their positions. Look at the following example:

Tut_7_1:

```

1  with Konada.Db.Connections; use Konada.Db.Connections;
2  with Konada.Db.Sql; use Konada.Db.Sql;
3  --
4  with Text_Io; use Text_Io;
5  -- for 'LF'=Linefeed
6  with Ada.Characters.Latin_1; use Ada.Characters.Latin_1;
7  --
8  procedure Tut_7_1 is
9      -- Declare Sqlinstance variable
```

²⁶The `Konada.Db` programmer is happy, because *define* is done completely by the library

²⁷Will be discussed later

²⁸for real numbers

```

10     Select_Emp: Sqltype;
11  begin
12     -- Logon using default connection
13     Logon("scott/tiger@tut");
14     -- Create an Sqlinstance
15     Create(Newsql => Select_Emp,
16           Code   => "Select * from emp");
17     -- Execute it implicitly describing the query and define
18     -- variables which will hold result values after fetch()
19     Execute(Select_Emp);
20     -- Output column names using functions Column_Count() and Column_N
21     for I in 1..Column_Count(Select_Emp) loop
22         Put_Line(Column_Name(Select_Emp,I));
23     end loop;
24  end Tut_7_1;

```

The interesting lines are 21 and 22 where the Functions `Column_Count()` and `Column_Name()` from `Konada.Db.Sql` are used. Remember from the last section that the query has to be *executed* before the select list information is available and these functions can be called. Otherwise `Column_Count()` would return 0 and `Column_Name()` the empty string.

Sometimes you know the column name and you want to determine the position of the column in the select list. For this task use `Konada.Db.Sql.Column_Position()` which returns the position of a column as a positive. Call it like this

```

...
pos:=Column_Position(Sqlcmd => Select_Emp,
                    Name    => "Empno");
...

```

`Column_Position()` does not require the name with exact case. If the name could not be found in the select list or if the statement is not already executed, `Konada.Db.Name_Error` is raised with the "name" argument mentioned in the exception information.

7.3 The Konada.Db Typecat System

Now before we go on to the more detailed select list information we have to discuss *Datatypes* in the circumstances of database programming. Every column

of a table in the database has a database specific type. In Oracle every column has one of the following datatypes:

- Character Datatypes
 - CHAR(*n*). This is for string columns with fixed length *n*. $n \leq 2000$ Byte.
 - VARCHAR2(*n*). For variable strings up to length *n*. $n \leq 4000$ Byte.
 - NVARCHAR2(*n*). Like VARCHAR2(*n*) but for unicode character sets.
 - LONG. For variable length character data up to 2GB. But this type is deprecated. Use CLOB instead (See 14).
- The NUMBER(*p*,*s*) datatype for integer and real numbers. Here *p* is the total number of digits and *s* the number of digits to the right of the decimal point. $p \leq 38$.
- The DATE datatype for points in time with a precision down to seconds.
- LOB = Large Object datatypes.
 - BLOB. For unstructured binary data up to 4GB.
 - CLOB. For character data up to 4GB. This type replaces the LONG datatype.
 - BFILE. In columns of this type you may store references to external OS-Files up to 4GB in size.
- RAW and LONG RAW. For unstructured binary data up to 2GB. This type is deprecated. Use BLOB or BFILE instead.
- ROWID. Internal datatype for storing the location of a table row on disk. This datatype cannot be given explicitly to a table's column. It is managed internally by the RDBMS.

If you write a program which retrieves data from a database server the column values must be stored in memory. Since you don't have the Oracle (or other RDBMS) datatypes available in your favourite programming language²⁹ there must be some intermediate software layer which is responsible for type conversion.

The Oracle Call interface (OCI) - a C library - provides Client datatypes into which the retrieved column values are converted. The Ada Oci Binding then

²⁹Unless you are using a database native developing tool like developer2000

provides more complex Ada types which hold the column values. And finally Konada.Db uses one `Data_Element_Type` - a discriminated variant datatype - which can hold any data value. Since Konada.Db is designed to be independent of a particular database system it uses it's own Typecat³⁰ schema.

The Typecat is no new datatype. You may imagine it as a *tag* on the `Data_Element_Type`. In fact it is the discriminant of the variant. The Typecat is an instance of `Konada.Db.Sql.Typecat_Type` which is an enumeration type:

```
type Typecat_Tpe is (StrB, StrUB, Int, Num,
                    Date, Blob, Clob, Bfile);
```

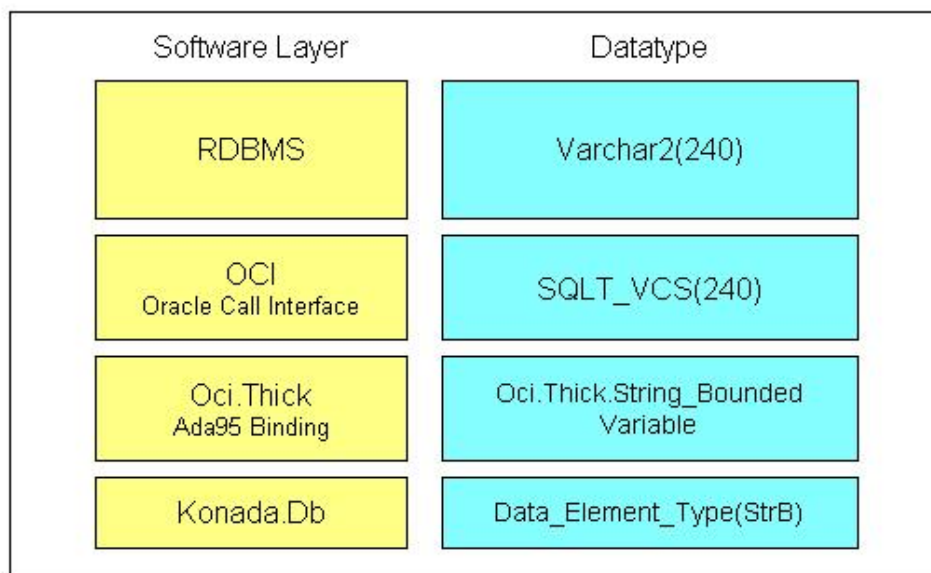


Figure 1: Varchar2() datatype from RDBMS to Konada.Db

After a column value is retrieved it is stored in a data element with a specific typecat. The typecat depends on the Oracle Datatype of course. The data elements are allocated in memory by Konada.Db if a `Select` statement is executed. This is done automatically by the library.

In the picture above you can see for the Oracle `Varchar2()` datatype how this type is represented in the Software layers from the RDBMS down to Konada.Db.

³⁰Short for *Type Category*

To summarize: All RDBMS specific datatypes are represented in Konada.Db by one discriminated `Data_Element_Type` which is a variant. Which datatype is actually stored in the variant is determined by the `Typecat` value of the `Data_Element_Type`.

Now after a select statement is executed and *define* has been performed you may retrieve the `Typecat` of each column with the function `Konada.Db.Sql.Col-type()`. `Tut_7_2` in `KONADA_ROOT\tutorial` shows how to use this function.

The following table shows how Oracle datatypes are mapped to Konada.Db `Typecat` values in the stage of *define*. Note that the `ROWID` datatype maps to an `StrUB` which is the `Typecat` value of an unbounded string. This is not very useful but there are conversion functions in Oracle Sql which you may call in the projection clause of your select statement.

Oracle Datatype	Konada.Db Typecat
CHAR(n) VARCHAR2(n) NVARCHAR2(n)	StrB
NUMBER(p,0) NUMBER(p,s)	Int Num
DATE	Date
BLOB CLOB BFILE	Blob Clob Bfile
LONG RAW LONG RAW ROWID	StrUB

Table 1: Oracle Datatype Mapping in Konada.Db

7.4 Retrieving detailed Select List Information

Now that you have learned about the `Typecat` values in Konada.Db, we go on and look how to retrieve more detailed information about the Select List.

`Tut_7_3` in `KONADA_ROOT\tutorial` shows how to get detailed information using the functions `Column_Name()`, `Column_Type()`, `Column_Size()` and `Column_Scale`. You may win some microseconds in performance if you use the procedure `Column_Info()`

which fills a `Column_Info_Type` record.

Tut_7_3

```

...
32      for I in 1..Column_Count(Select_Table) loop
33          Put(Column_Name(Select_Table,I));Set_Col(15);
34          -- determine coltype and print image
35          Put(Typecat_Type'Image(Column_Type(Select_Table, I)));Set_Col(2
36          -- colsize
37          Put(Natural'Image(Column_Size(Select_Table, I)));      Set_Col(2
38          -- colscale
39          if Column_Type(Select_Table,I) = Num then
40              Put_Line(Natural'Image(Column_Scale(Select_Table, I)));
41          else
42              New_Line;
43          end if;
44      end loop;
...

```

Play around with Tut_7_3. Connect to another Oracle Schema with more and bigger tables and look at the output. Tune the output formatting of the program. It is poor.

Note that

- For StrUB and lob columns (Blob, Clob, Bfile) the *Size* information is not available immediately after `Execute()` since the size for this kind of data is variable and is determined after `Fetch()` which is explained in the next section. Immediately after `Execute()` `Column_Size()` returns 0 for these columns.
- If a number column is specified simply as
`no_column number`
 instead of
`no_column number(n)`
 in the table creation statement (which is allowed in the Oracle DDL specification) then `Column_Size()` returns 0 for such a column.

You may wonder whether it is possible with Konada.Db to retrieve the original

Oracle datatype of a column. You have to query the Oracle Data Dictionary for this task.³¹

Finally in this section take a look at the spec of Konada.Db.Sql in KONADA-ROOT\lib especially the “Select List related Functions”.

³¹For example you may query the dictionary view `user_tab_columns`

8 Fetching and Extracting Data

In this section we will process data retrieved from the server. Suppose we have executed a `Select` statement. What is the next step? According to 7.1 we must *fetch* now the data rows from the server.

8.1 A Simple Example

After data is fetched and available on the client it may be *extracted* and processed. The next simple example shows how this is done with `Konada.Db`.

Tut.8.1:

```
1  with Konada.Db.Connections; use Konada.Db.Connections;
2  with Konada.Db.Sql; use Konada.Db.Sql;
3  --
4  with Text_Io; use Text_Io;
5  -- for 'LF'=Linefeed
6  with Ada.Characters.Latin_1; use Ada.Characters.Latin_1;
7  --
8  procedure Tut_8_1 is
9      -- Declare Sqlinstance variable
10     Select_Emp: Sqltype;
11     C: Character;
12 begin
13     -- Logon using default connection
14     Logon("scott/tiger@tut");
15     -- Create an Sqlinstance
16     Create(Newsql => Select_Emp,
17           Code    => "Select * from emp");
18     -- Execute it implicitly describing the query and define
19     -- variables which will hold result values after fetch()
20     Execute(Select_Emp);
21     -- now fetch iteratively from server as long as data available
22     while Fetch(Select_Emp) loop
23         for I in 1..Column_Count(Select_Emp) loop
24             Put(Column_Name(Select_Emp, I)); Set_Col(15);
25             -- extract to a string with get()
26             Put_Line(Get(Select_Emp,I));
27         end loop;
```

```
28         New_Line; Put("Press any key for next row ...");
29         Get_Immediate(C);
30         New_Line;
31     end loop;
32 end Tut_8_1;
```

You may stop the program by sending it a signal. In the outer while loop from line 22 up to line 31 data is iteratively fetched using `Konada.Db.Sql.Fetch()`. `Fetch()` returns true if data could be retrieved and false if no more data row is available.

In line 26 the data values are *extracted* using one of the many overloadings of the `Get()` Function. Here we are using the one which returns a string. In the above Example a position argument is used to determine the column in the select list but we could have used a column name as well. Position arguments are a little bit faster because the name has not to be resolved first.

8.2 Date and Number Formats

If you extract date or number formats you may control the formatting of the string output. For date columns you can change the default output format by changing `Konada.Db.Date.Format` which is by default set to `DD.MM.YYYY`.

For number values the default output format is computed when you execute a select statement and it cannot be changed.

But with providing an oracle format string for the `Format` parameter of the `Get()` function you may override the defaults.

8.3 Extracting to other Ada Datatypes

As said earlier the `Get()` function is heavily overloaded. You can get data as strings, integers, `Ada.Calendar.Time`, `Long.Float` and as `Files` or `Stream.Element_Arrays`. Every call to `Get()` may have one of the following three outcomes:

- **Success** because the typecat of the column is compatible with the datatype of the return value of the `Get()` overloading. For example if you extract an Integer value from a column with typecat **Int**.
- **Success** because `Konada.Db` did an implicit conversion. The above example program `Tut_8_1` contains such cases. E.g. the column values of the column `HIREDATE` are implicitly converted to a string.

- **Error** because `Konada.Db` did not convert and the typecat of the column is not compatible with the datatype of the return value of the `Get()` overloading. In this case `Konada.Db.Data_Error` is raised and the exception information shows which conversion could not be performed. This happens for example if you try to get an `Ada.Calendar.Time` from an integer.

The table below shows the behavior of `Get()` in detail.

Get() behavior	StrB	Int	Num	Date	Lob	StrUB
String	ok	ok	ok	ok	?	ok
Integer	!	ok	ok	!	!	!
Long_Float	!	ok	ok	!	!	!
Time	!	!	!	ok	!	!
File	!	!	!	!	ok	?
Binary	!	!	!	!	ok	!

Table 2: Behavior of data extracting. Horizontal: Konada Typecat. Vertical: Ada Datatype

Notes:

Lob is a shorthand for the three typecats **Blob**, **Clob** and **Bfile**. **Time** is for `Ada.Calendar.Time` and **Binary** for a `Stream_Element_Array`. The table entries have the following meanings:

- **ok** means that you may extract a column of the indicated typecat to the indicated Ada Datatype without problem.
- **?** means that extracting is possible but not recommended since implicit conversions with eventually unexpected results are done.
- **!** means that `Konada.Db.Data_Error` will be raised.

Now take a close look at Tut_8.2. There are used all overloadings of `Get()` except of File extraction and Binary extraction which will be discussed in 14. Play around with the example, produce some `Data_Error`-Exceptions and look at the messages.

Note that there is no reliable way to extract Oracle Long Raw columns³², which are deprecated. If you have the choice, migrate these columns to Blob.

³²They are mapped to the typecat `StrUB` by `Konada.Db`

8.4 Handling Null Values

If a table's column has no not-null constraint the value null is permitted for this column. This means that the value may be actually *unknown* or *undefined*. If a null value is used in an expression³³ then the value of the whole expression is null no matter what the other values are³⁴.

If you fetched a row from a query down to Konada.Db there may be a null value among the column values. What happens if you try to extract such a value? There are some easy rules depending on the return datatype of Get():

- **String** This overloading of Get() accepts a boolean parameter `raise_if_null` with default `false`. If you call Get() with this parameter set to `true` and you try to extract a null value then `Konada.Db.Null_Error` is raised. Otherwise `Konada.Db.Null_Descriptor` is returned which is set to the string "NULL".
- **Integer** or **Long_Float** In this case the behavior is the same as for the string overloading but the default return value is 0 resp. 0.0.
- **Other** In all other cases `Konada.Db.Null_Error` is raised.

If you want Get() not to raise an exception but returning a special substitute for null different from `Konada.Db.Null_Descriptor`³⁵ then use the `nvl()` single row function in your query like the example below shows:

```
select nvl(ename, 'Not Known')
from emp
where empno=7369
```

Here the String "Not Known" is extracted if the ename column is null for empno 7369.

As an exercise write a program which sets the ename column of employee no. 7369 to null. Then select this row and extract the ename column with `raise_if_null` set to `true`. Catch the exception and inform the user that no name is known for employee no. 7369. Get a name from the user and update the row with the new name.

³³E.g. a sum, product or even a logical expression

³⁴For example: the boolean expression `(null or true)` has value null!

³⁵resp. 0, 0.0

9 Using Bind Placeholders

Bind Placeholders are identifiers in an SQL-Statement preceeded by a colon as in

```
select *  
  from emp  
  where empno=:this_empno
```

In the above statement `this_empno` acts as a bind placeholder. The Oracle Oci Library expects this placeholder to be filled with an integer value before the statement is executed. This is the 5. stage in SQL statement-processing (see 7.1). In this stage memory has to be allocated for the placeholder³⁶. Then we have to tell Oci where this memory is located and which datatype must be used to interpret memory content. Finally we must fill in a value into the memory location.

This sounds complicated but Konada.Db provides an easy to use interface to perform binding. Now lets have a look on the reasons for using this feature of Oci.

9.1 Why Bind Placeholders?

Suppose you have to write an application for the human resources department of a company. Part of this application is a procedure which outputs data for an employee given by user input. With your knowledge of Konada.Db presented so far you will take the following approach in the design of this procedure:

1. Build an sql-statement string
Query_String:="select * from emp where empno=" & User_Input
based on the user's input Empno.
2. Create an sql-instance using the query string.
3. Execute it.
4. Fetch, extract and display data.

The result would look similar to Tut_9_1 below.

³⁶In most cases a variable is declared

Tut_9_1:

```
1  with Konada.Db.Connections; use Konada.Db.Connections;
2  with Konada.Db.Sql; use Konada.Db.Sql;
3  --
4  with Text_Io; use Text_Io;
5  -- for 'LF'=Linefeed
6  with Ada.Characters.Latin_1; use Ada.Characters.Latin_1;
7  --
8  procedure Tut_9_1 is
9      -- Declare Sqlinstance variable
10     Select_Emp: Sqltype;
11     -- For user Input
12     User_Input: String(1..10);
13     Last_Of_Input: Positive;
14 begin
15     -- Logon using default connection
16     Logon("scott/tiger@tut");
17     loop
18         Put("Give employee number(q for quit): ");
19         Get_Line(User_Input, Last_Of_Input);
20         if User_Input(1)='q' then
21             exit;
22         end if;
23         -- Create an Sqlinstance
24         Create(Newsql => Select_Emp,
25             Code    => "Select * from emp where empno=" &
26                 User_Input(1..Last_Of_Input));
27         -- Execute it implicitly describing the query and define
28         -- variables which will hold result values after fetch()
29         Execute(Select_Emp);
30         -- now fetch
31         if Fetch(Select_Emp) then
32             New_Line;
33             for I in 1..Column_Count(Select_Emp) loop
34                 Put(Column_Name(Select_Emp, I)); Set_Col(15);
35                 -- extract to a string with get()
36                 Put_Line(Get(Select_Emp,I));
37             end loop;
38         else
39             Put_Line("No employee with this number found!");
```

```

40         end if;
41     end loop;
42 end Tut_9_1;

```

In line 23-26 an sqlinstance is created again and again. Remember what that means:

1. A query string is build. That's easy here but maybe more complicated.
2. The statement is parsed, desribed and an execution plan is figured out by the RDBMS.
3. *Define* has to be performed.
4. Eventually a previously created sqlinstance has to be destroyed³⁷ before creating a new one in `Select_Emp`.

All this has to be done but for what? Only to change the `empno` value in the where clause of the statement! The same execution plan could have been reused, the define variables could have been reused, the whole sqlinstance could have been reused if we only could tell the library to supply another value for `Empno` and re-execute the query. But exactly this can be done with the `Bind()` and `Set()` procedures of `Konada.Db`.

The Code below shows how to implement the same functionality using the bind mechanism:

Tut_9_2:

```

1  with Konada.Db.Connections; use Konada.Db.Connections;
2  with Konada.Db.Sql; use Konada.Db.Sql;
3  --
4  with Text_Io; use Text_Io;
5  -- for 'LF'=Linefeed
6  with Ada.Characters.Latin_1; use Ada.Characters.Latin_1;
7  --
8  procedure Tut_9_2 is
9      -- Declare Sqlinstance variable
10     Select_Emp: Sqltype;
11     -- For user Input
12     User_Input: String(1..10);

```

³⁷This includes deallocating all memory related to the sqlinstance

```

13     Last_Of_Input: Positive;
14 begin
15     -- Logon using default connection
16     Logon("scott/tiger@tut");
17     -- Create an Sqlinstance, using a bind placeholder now\
        outside the loop
18     Create(Newsql => Select_Emp,
19           Code   => "Select * from emp where empno=:this_empno");
20     -- Bind it with typecat Int
21     Bind(Sqlcmd  => Select_Emp,
22         Name    => "this_empno",
23         Typecat => Int);
24     loop
25         Put("Give employee number(q for quit): ");
26         Get_Line(User_Input, Last_Of_Input);
27         if User_Input(1)='q' then
28             exit;
29         end if;
30         -- Set the new value, reusing the sqlinstance in Select_Emp
31         Set(Sqlcmd => Select_Emp,
32           Name    => "this_empno",
33           Value   => Integer'Value(User_Input(1..Last_Of_Input)));
34         -- Execute the sqlinstance
35         Execute(Select_Emp);
36         -- now fetch
37         if Fetch(Select_Emp) then
38             New_Line;
39             for I in 1..Column_Count(Select_Emp) loop
40                 Put(Column_Name(Select_Emp, I)); Set_Col(15);
41                 -- extract to a string with get()
42                 Put_Line(Get(Select_Emp,I));
43             end loop;
44         else
45             Put_Line("No employee with this number found!");
46         end if;
47     end loop;
48 end Tut_9_2;

```

In lines 21-23 `Bind()` is called to allocate a bind variable internally and the library is told which `Typecat` to use - in this case `Int`. In lines 31-33 `Set()` is used to supply a new value for the bind placeholder. Note the analogon to *define*

and `get`. The `sqlinstance` is created only once in `Select_Emp` and then reused in the loop as we have found to be more effective.

We have seen now that using bind placeholders is good for our resources and performance. But there are other reasons to use it:

- Bind placeholders are inevitable for calling stored procedures and functions (see 10).
- You *must* use bind placeholders for the modification of lob. See 14.
- Your code is leaner and reflects the logic of sql-processing since the `Create()` statement for the instance is separated from the value substitution e.g. the `Set()` statement.
- You may create the `sqlinstance` at an outer scope and supply the value inside a function. Later we will see examples for this method.

In the following section we will discuss the bind mechanism step by step.

9.2 Explicit Binds

We call a bind operation *explicit* if it is performed by a call to the `Bind()` procedure. Here are the steps for the use of an explicit bind:

1. Create an `sqlinstance` containing a bind placeholder.
2. (optional) Determine the position of your placeholder if you want to perform a positional bind. This is not necessary if you want to do a named bind or if you know already the position of your placeholder³⁸.
3. Call `Bind()` and beneath the `sqlinstance` supply
 - A position or a name.
 - A typecat. Default is `StrUB`.
 - A length. Only if the typecat is `StrB` you must provide a length since only for `StrB` it is unclear how many memory has to be taken.³⁹

³⁸E.g. because there is only one

³⁹The other cases are Lob's but for those the memory handling is completely different. See

4. Set a value.

Now look carefully at Tut_9_3 where all bind related functions and procedures are used. Note that there is no `Bindsizes()` function because there are no sized numbers on the client side in Oci. Inside the database we have sized numbers⁴⁰. So the sizes of bind placeholders are always known.

Play around with the example, produce same errors and look at the messages.

9.3 Setting Values and implicit Binds

After you have bound a placeholder explicitly you may set it to a specific value using one of the overloads of the `Set()` procedures. Now we encounter a situation similar to the extraction of data. We have bound with a specific typecat and now we may supply a value of a different datatype. For example

```
...  
Bind(Sqlcmd, 2, Num);  
Set(Sqlcmd, 2, "300.0");  
...
```

In the above example `Konada.Db` would use `Long_Float'Value()` to convert the given `String` to a `Num` data element. But this attempt may fail.

If you bound to a `Date` typecat and want to supply the value as a string you may provide an Oracle format mask as an additional parameter.

`Konada.Db` tries to convert your supported value. But best is to supply the value compatible to the typecat to which you have bound the placeholder.

The table below shows the behavior of `Set()` with respect to Typecat/Datatype combinations.

⁴⁰`number(23,2)` for example

Set() behavior	StrB	Int	Num	Date	Lob	StrUB
String	Size!	?	?	Format!	?	ok
Integer	?	ok	ok	!	!	?
Long_Float	?	?	ok	!	!	?
Time	!	!	!	ok	!	!
File	!	!	!	!	Bfile!	?
Binary	!	!	!	!	Bfile!	!

Table 3: Behavior of Set(). Horizontal: Bind Typecat. Vertical: Datatype of actually supplied Value

Notes:

- If you set a string value to an StrB bind placeholder and the length of the given string exceeds the length value used in the call to Bind() then `Constrained_Error` will be raised. Therefore the warning `Size!` is placed in the (String,StrB)-cell of the table above.
- If you supply a string value for a Date bind you have to provide an appropriate format mask. Otherwise the Oci Exception is passed through. See the warning `Format!` in the above table.
- There is no Set() overloading for File because the Filename is given as a string value. This would lead to a conflict with the string overloading. Instead of an overloading i used a new name `Set_File()`.
- Any attempt to set a Bfile placeholder will cause `Konada.Db.Data_Error`. See the `Bfile!` warnings in the table. So why use binds to a Bfile at all? This will be explained in [14⁴¹](#).
- In all cases with ? placed in a table cell there may occur errors or unexpected results. The cases with a ! will produce a `Konada.Db.Data_Error` exception.

You may have noticed that the call to Bind() is somewhat dispensable because the call to Set() should give us enough information to do the bind *implicitly*. Good news! `Konada.Db` does support implicit binds.

⁴¹You may use a Bfile bind placeholder to select a Bfile lob locator into a bind variable using a `Select ... into ...` statement

To get an example for implicit binds modify Tut_9_3. First move the line with the `Set()` Statements directly past the second call to `Bind()`⁴². Then comment out the `Bind()` Statements. Compile and run the program.

Note that the typecat of the first bind placeholder is `StrUB` now. This is because the string overloading of `Set()` does always an implicit bind to an `StrUB` data element which is a little bit slower than `StrB`. This is one reason to use explicit binds. Other reasons will be explained in later sections. But here are the rules for implicit binds inside `Konada.Db` depending on the overloading of `Set()`:

- **String** The string overloading does bind implicitly to an **StrUB** data element.
- **Integer** causes an implicit bind to an **Int** data element.
- **Long_Float** implicit bind to a **Num**.
- **Time** implicit bind to a **Date**.
- **File, Binary** implicit bind to a **Blob**.

Finally in this section note that it is possible to rebind a placeholder to another typecat in which case memory is freed and a new bind is created. E.g. the following is possible

```
...
1 Set(Change_Name,"new_ename","Muller");
2 Set(Change_Name,"empno_to_change", 7369);
3 Bind(Change_Name, "new_ename", StrB, 10);
4 Set(Change_Name,"new_ename","Muller");
...
```

In line 1 an implicit bind to an `StrUB` is created and in line 3 a rebind to a `StrB` is done. Note that you have to set the value again since it is has gone lost in line 3.

⁴²this is necessary for the typecat output

9.4 Data Extraction from Binds

Why should data extraction from a bind be done? You bound a placeholder, set a value. So why should you extract a value which is already known to the program?

It's true. There is no reason to extract data from a bind if we restrict our database commands to pure SQL. But the Oracle RDBMS⁴³ provides a procedural Language Extension to SQL called PL/SQL which looks in its earlier versions much like Ada83. In version 8 there are object oriented features built in the language but Oracle choosed an approach quite different to Ada. We will see Examples of PL/SQL procedures and functions in 10.

There is one PL/SQL construct which we discuss now: The *Anonymous Block*. This is a declare (optional) ... begin ... end; block like in Ada containing SQL statements, procedure or function calls and eventually other blocks. In such a block you may use a select ... into ... statement:

```
begin
  select ename into :this_ename
  from emp
  where empno=7369;
end;
```

This statement fetches the ename of employee no. 7369 into the bind variable :this_ename. If there is no employee matching the where-clause criteria the *Oracle Exception No_Data_Found* is raised. If more than one row satisfy the criteria the *Oracle Exception Too_Many_Rows* is raised. The Exceptions are passed through and cause a Konada.Db.Sql_Error Exception during Execute().

One possible use of data extraction from a bind should be evident now: you may extract the fetched value in a select ... into ... statement like the above. In Konada.Db for every overloading of Get() there is a corresponding overloading of Bget() which you may use for extraction from a bind. Look at example Tut_9_4 where the use of Bget() is shown. Note that you have to choose carefully the typecat of the bind according to the table at 30 ff. Otherwise you will eventually encounter a Konada.Db.Sql_Error caused by an Oracle value error at runtime. The extraction behavior of Bget() is exactly the same as that of Get() as listed in 8.3.

At this point you should examine KONADA_ROOT\samples\simple\demo1.

⁴³Like others e.g. Db2, PostgreSql, MS Sql-Server

There is no `Bget()` use in this demo but binding is shown and dml and dql statements both occur in this example.

10 Calling Stored Units

10.1 Calling Stored Units using Binds

As said in the previous section you may store PL/SQL procedures and functions in the database. These stored program units may be called by other units or they may be used to compose SQL statements. But how to call them from a client application program?

Take for example the function `emp_desc()` which you may create in your scott schema by executing the statement below with Sql*Plus. Another way to create it is running

KONADA_ROOT\samples\simple\demo2

But attention! If you quit the program, the function is dropped from the database!

Server Function Emp_Desc()⁴⁴

```
CREATE OR REPLACE function emp_desc(p_empno in number)
-- Short emp description
return varchar2
is
l_desc varchar2(100):='';
begin
    select 'Department: '||d.dname||' Name: '||e.ename
        into l_desc
        from emp e, dept d
        where e.empno=p_empno and
              e.deptno=d.deptno;

return l_desc;
exception
when NO_DATA_FOUND then
    l_desc:='No emp with number '||to_char(p_empno)||' found !';
    return l_desc;
when others then
    l_desc:='Error : '||substr(SQLERRM,1,91);
    return l_desc;
end;
```

⁴⁴Attention! This is PL/SQL Code, not Ada

The function `emp_desc()` takes a number parameter `p_empno` and returns a short description of the employee as a `varchar2`. There are two ways to call such a stored function from the client side:

1. Call it inside an SQL statement like

```
Select emp_desc(7369) from dual
```

2. Call it inside an anonymous block using a bind placeholder like

```
begin
  :description:=emp_desc(7369);
end;
```

Obviously the first method works only for function calls. And it works only for function calls if the function has no OUT parameters in its parameter list⁴⁵.

Look at `KONADA_ROOT\samples\simple\demo2` where the 2nd method to call a PL/SQL function from Konada.Db is demonstrated. In the example `Tut_10_1` in the tutorials directory i have rewritten `demo2` now calling a procedure with an OUT parameter.

10.2 Managing the SQL Instance

A situation which you may encounter quite frequently is that you want to wrap a server side PL/SQL program unit on the client side. Let us look again at `demo2` and our server function `emp_desc()`. We want to modify `demo2` so that we have an Ada function `Client_Emp_Desc()` which acts as a wrapper to the server function.

In `Tut_10_2` you see one possible approach to do it. Here is the code snippet with the definition of the wrapper function `Client_Emp_Desc()`:

```
...
function Client_Emp_Desc(Empno: in Integer)
  return String is
  Call_Emp_Desc: Sqltype;
begin
  Create(Call_Emp_Desc,
```

⁴⁵Yes: In PL/SQL functions may have OUT parameters!

```

        "begin                                " & LF &
        ":ret:=emp_desc(:emp_no);            " & LF &
        "end;                                ");
    Bind(Call_Emp_Desc,1, StrUB);
    Set(Call_Emp_Desc,2,Empno);
    Execute(Call_Emp_Desc);
    return Bget(Call_Emp_Desc,1);
end Client_Emp_Desc;
...

```

Note that in this approach the sqlinstance is local in the function. From the software engineering point of view this is o.k. but every time the function is called, the sqlinstance is created, binding and Set() is done and finally the sqlinstance is freed. This is not very reasonable with respect to performance.

I think that there are two main techniques for solving this problem:

The first idea⁴⁶ is to move the sqlinstance to an outer scope and test inside the wrapper function whether the instance is ready to use or must be set up⁴⁷. Tut_10_3 shows the implementation.

The 2nd approach to sqlinstance-management for functions and procedures is to build a persistent type e.g. Employee and make the function/procedure (e.g. Emp_Desc()) a method of it. Objects of this type may contain sqlinstances used for the persistence layer and in addition for Emp_Desc() computation. This solution is shown in the next section.

⁴⁶Not very Ada like but sometimes a sufficient solution

⁴⁷Use Is_Empty() to test whether an sqltype variable contains an Sqlinstance

11 Persistent Types

Generally spoken a persistent type provides methods to

- load an object (instance) of the type from a persistent media⁴⁸
- modify the objects content
- create a new object
- delete an object
- save the object back to the persistent media
- (optional) call some database procedures/functions.

11.1 An example

Note that the above definition is a very simple one. We will examine an example now, an implementation of the persistent type `Employee`. The code for this example is located in `KONADA_ROOT\samples\persist`. Build and run the main unit `Employee`. The persistent type `Employee_Type`⁴⁹ is derived from `Limited_Controlled` and its data members are completely `privat`. Moreover it is discriminated and in a declaration you may supply an employee no. for the discriminant.

Implementation Note: I declared the member `Emp_Desc_Cmd` to be an *access* to an `Sqltype` - why? We want to wrap the server function `Emp_Desc()` in a client *function*. This client function has to call `Konada.Db.Sql.Execute()` with an `OUT` parameter. Since the client function takes an `Employee_Type` instance as an `IN` parameter i use indirection to provide `Emp.Emp_Desc_Cmd.all` as an `OUT` parameter to `Execute()`.

The code of the overwritten procedure `Initialize()` loads the database data into the instance of `Employee_Type` according to the given employee no. Of course the main program has to connect to the database for this initialization to succeed. If you supply a zero value for the discriminant an empty instance of `Employee_Type` is initialized and you may load the data later with a call to the method `Load()`.

⁴⁸In our context from the Oracle database

⁴⁹Declared in `Employee_Pkg.ads`

```

...
Console_Logon;
...
declare
  Emp: Employee_Type(7844);
begin
  <Call Methods on emp here>
end;
...

```

Play around with the main unit and look carefully at the code in `Employee_Pkg.adb`. Especially note how the *status* member of an employee instance is managed. The methods `New_Emp()` and `Delete_Emp()` are not yet written. Try it!

Some suggestions to improve the persistent type:

- You may call `Save()` in `Finalize()` depending on a flag

```
Save_In_Finalize: Boolean:={true|false};
```

declared at library level.

- You may control the set of loaded employee objects e.g. to ensure that no employee individual is loaded twice⁵⁰. For this you have to keep a list of loaded employee numbers at library level and check in `Initialize()` whether an employee is already on the list.

11.2 Setters and Getters

If the data members of a type are private you have to use special procedures and functions to modify or read them. These units are usually called *Setters* and *Getters*. In our example there is only one such pair implemented⁵¹. Add other pairs as an exercise.

As an alternative to setters and getters you may take a different approach using partial views.

⁵⁰This is a standard approach when maintaining an *object cache* of a persistent type

⁵¹The pair to modify/read the `Ename` member

- First declare a tagged type with members which you want to change directly. Declare this type in the public part of your package. Let us call it the *Public Type*
- Derive the complete type from the public type with a private extension to hold Sqlinstances and other hidden members.
- Now you may change the public members directly like this:
`Emp.Ename:="Jones"`

12 Asynchronous Execution

Normally in so called *OLTP*⁵² Database Applications the databases response time of an application call is expected to be in the range of some seconds. So the application can wait until the database server has completed execution.

If this is not possible because a call may need time in the range of some minutes or even more you could of course use ada tasking to deal with such a situation. The task could establish its own connection to the database and then issue the *long running call*. The main unit could read a variable shared with the task to determine if the call had already completed.

Konada.Db gives you another option. You may call `Execute()` with the parameter `Asnyc` set to `true`. Control is returned immediately then. Later you may determine using the `Check_Finished()` procedure wether the call has been completed. But attention! There is a subtle restriction associated with asynchronous calls: *The connection used by the asynchronous call may not be used by other calls until the asynchronous call has completed*. This is no severe restriction since you may open a dedicated connection for the asynchronous call.

You may cancel an asynchronous call with the procedure `Cancel()`. `Cancel()` instantly terminates the asynchronous call and resets the associated `Connection`. You may alternatively call the procedure `Reset()` which has the same effect but additionally for dql-statements reexecutes the query. This process of cancelling and resetting may take some seconds of time⁵³.

Note that if you don't call `Check_Finished()` until the asynchronous call has completed and additionally you don't cancel or reset the call then every subsequent call on the same connection will raise an exception.

Look at `KONADA_ROOT\samples\simple\demo3.adb` now to see an example of an asynchronous call. The demo also shows `Reset()` and `Cancel()`.

Demo3(Snippet):

...

```
70      --
71      -- Now call wait asynchronously
72      --
73      Execute(Sqlcmd => Waitproc, Async => True);
```

⁵²OnLine Transaction Processing

⁵³3-6 seconds

```
74      -- the following is ok since in the default connection
75      Put_Line("Executing Emp2"); New_Line;
76      Create(Emp2,"select * from emp order by 1"); Execute(Emp2);
77      -- Cancel(Waitproc);
78      -- But uncommenting the line below gives an error since also in
79      -- Conn_For_Async
80      -- Create(Emp,"select empno, ename from emp order by 1",Conn_For_Async);
      Execute(Emp);
81      -- return;

...

```

If you uncomment line 80 you will get an `Konada.Db.Sql` exception reporting an Oracle 24338 error because the program tries to call on a connection⁵⁴ which is already occupied by a running asynchronous call. If you additionally uncomment line 77 (and line 81 because executing the following code does not make any sense now) the error disappears because the connection is freed from the asynchronous call by `Cancel()`.

⁵⁴`Conn_For_Async`

13 Locking

13.1 How to avoid Blocks

In this section we will learn how to avoid blocking situations like the one shown in [18](#)(Tut_6.2) where two update statements on the same row, executed from different sessions block the application.

In Oracle SQL you may use the `Select ... for update ...` statement to acquire a row-level lock. If the row is already locked and you try to lock it with the `NOWAIT` option you will get an ORA-0054 Error. Open an sqlplus session to the scott schema and type

```
SQL> update
  2  emp
  3  set  ename='Brown'
  4  where empno=7369;
```

Don't commit and don't quit the session. Now open another sqlplus session to scott and type

```
SQL> select empno
  2  from
  3  emp
  4  where empno=7369
  5  for update NOWAIT;
emp
*
ERROR in Line 3:
ORA-00054: resource busy and acquire with NOWAIT specified
```

Now it's easy e.g. to add a function method `Lock()` to our `Employee Type`^{[55](#)} which tries to lock an employee instance and returns true if the lock could have been placed and false otherwise. The function has simply to call a `Select ... for update ...` Statement with the `NOWAIT` option. The function then catches the `Konada.Db.Sql` exception and checks whether `Sql_Error_Code=54` in which case it returns false otherwise or if no exception occurred it returns true. Try it as an exercise!

⁵⁵See [11](#)

13.2 The Lock() Utility Procedure

Konada.Db provides a utility procedure Lock() for row-level locking. Look at the specification in KONADA_ROOT\lib\konada-db-utils.ads and read the comments. In Tut_13_1 the use of the Lock() procedure is shown. You should lock the 7369-emp-row before starting Tut_13_1 by opening an sqlplus session and typing

```
SQL> update
  2  emp
  3  set  ename='Brown'
  4  where empno=7369;
```

Then you may experiment by changing the call to the Lock() procedure and by quitting or committing the sqlplus session while Tut_13_1 is waiting.

Note that the blocker information is only a good guess because in general there could be more than one lock - especially in multithreaded applications - which the program is waiting for. In such a situation the Lock() procedure reports the first blocker which blocks the session supplied by the Conn parameter to Lock().

Note also that the dba (most likely you) has to grant the select privilege on the dynamic performance views V\$SESSION and V\$LOCK to the database user who is given as the username parameter to the Konada.Db applications Logon()-call. Otherwise you will not get any blocker information from Lock(). It would suffice to grant the SELECT_CATALOG_ROLE role to the user.

14 Large Objects

14.1 Introduction

In Oracle Databases you may store (or refer in Bfile columns) large data chunks called *large Objects* or *Lobs*. As outlined in 7.3 you may store binary or character data inline that is: directly in a tables column of a *Blob*⁵⁶ or a *Clob*⁵⁷ type. Further you may store in a *Bfile*⁵⁸ column a reference to an external operating system file.

Of course in (relational) databases without lob support you could have a table column *Filepath* which would store the location of a file in a file system tree. You could write an application then to manage the database vs. file system relation. Users would search a file using the databases query techniques and then retrieve the file to a client with a file transport mechanism⁵⁹.

But if you store large data *inline* in a database table then all the operations on this data e.g. *insert*, *update*, *delete* are involved in transactions. Because of this you will never have a mismatch between the structured data like *Filename*, *Keywords*, ... and the corresponding large data if your application is well written. You can use the databases mechanisms to ensure integrity between structured and unstructured data. This is a great feature especially in creating content management systems or archives.

14.2 Lob Support in Konada.Db

Konada.Db supports operations on lob columns through an easy to use interface which is consistent with the already introduced procedures/functions for the structured datatypes. For example you may select lobs, fetch and extract them using *Get()* or *Get_File()*. You may bind a lob placeholder and set it to a file's content or load it with stream data using *Set()* or *Set_File()*⁶⁰.

Although Konada.Db's lob support includes Bfile columns i think that they are not so important because they

- lack the *Involved in Transactions*-Feature discussed above and

⁵⁶Binary Large Object

⁵⁷Character Large Object

⁵⁸Binary File

⁵⁹FTP, SMB or a self written transport layer

⁶⁰This is not possible for the Bfile type

- cannot be modified or created by database operations. Only the *reference* to the data chunk can be manipulated by the databases operations.

So i will focus the examples and discussions on *inline* lobs⁶¹ from now on.

14.3 Extracting Lobs

Before we can run our examples we have to create a table with a lob column in scott's schema. Moreover this table should contain some data. If you have already built the lobdemo located in KONADA_ROOT\samples\lob you are prepared. If not then look at the README file in the lobdemo directory and execute at least step 1) and 2).

Now you should have a table Portrait in your scott schema. This table has a column Image of type Blob. If you successfully loaded the images (step 2) of the README) you can start with the first example now.

Build and execute Tut_14_1 and then display image.jpg in the tutorial directory e.g. with a web browser. The interesting lines in the code of Tut_14.1 are

...

```
25      Bytes_Retrieved:=Get_File(Sqlcmd      => Select_Image,
26                                     Name      => "image",
27                                     File_Name => File_Name);
```

...

where the image file is extracted from the sqlinstance using the Get_File() function. You may supply an additional parameter Buffer_Size which defaults to 131072 (=128K). Supply a larger value to improve performance when extracting a big lob content⁶². In Tut_14_2 the same functionality is implemented but now using a select ... into ... statement with a bind variable. Data extraction is done with Bget_File() in this case.

14.4 Modifying Lobs

You may insert or update lobs using bind variables with a few lines of code. Look at Tut_14_3. Before you run Tut_14_3 you should copy some jpeg file to KONA-

⁶¹Blobs and Clobs

⁶²Big >= 5M

DA_ROOT\tutorial and name it image.jpg. Now run Tut_14_3. After the run the portrait of employee 7369 should have changed to the file you copied. The most easy way to check whether the update was successful is to use lobdemo since you have a menu item (s)how there. Alternatively you may delete image.jpg, run Tut_14_1 and then check the new image.jpg with your web browser.

The next example shows how lob operations are involved in transactions. Play around with Tut_14_4 and verify that lob modifications are involved in transactions like ordinary data changes.

Finally you should examine the lobdemo program in KONADA_ROOT\samples\lob. Especially have a look at the handling of null values. You may set an image to null with the menu item (e)rase.

14.5 Binary Get() and Set()

In KONADA_ROOT\lib\konada-db-sql.ads you will find overloadings of the Get() and Set() procedures with a Stream_Element_Array parameter. These overloadings may only be used for lob. I don't have a good example for their use. But possibly in a three tier environment with a web server, an Oracle Database and a client machine it would be useful to read and write directly to network sockets instead of saving the lob content to a temporary file.

15 List of Examples

The following list contains all examples given in the tutorial together with a short description and a reference to their appearance in the text. Demo programs are not listed.

- **Tut_4_1.** The “Hello World” of Konada.Db. Shows how to establish a connection, the use of `Rdbms_Version()` and `Logoff()`. => [4.1](#)
- **Tut_4_2.** Use of the Default Connection. => [4.2](#)
- **Tut_4_3.** `Console_Logon` Utility. => [4.3](#)
- **Tut_5_1.** Handling of Database Errors. => [13](#)
- **Tut_6_1.** Simple statement execution and `Rows_Processed()` function. => [6.2](#)
- **Tut_6_2.** Transaction Control. => [18](#)
- **Tut_7_1.** Shows use of `Column_Count()` and `Column_Name()` for select lists. => [28](#)
- **Tut_7_2.** How to retrieve Typecat Information with `Column_Type()`. => [30](#)
- **Tut_7_3.** Shows retrieval of detailed select list information. => [7.4](#)
- **Tut_8_1.** Simple data extraction. => [8.1](#)
- **Tut_8_2.** Data extraction using overloadings of `Get()`. => [8.3](#)
- **Tut_9_1.** Creating an `sqlinstance` based on user input without using bind placeholders. => [9.1](#)
- **Tut_9_2.** Same as `Tut_9_1` but now using bind placeholders and `Set()`. => [9.1](#)
- **Tut_9_3.** Shows more procedures/functions related to binding. => [9.2](#)
- **Tut_9_4.** Data extraction from a bind placeholder using `Select ... into ...`. => [43](#)
- **Tut_10_1.** Calling a stored function. => [45](#)
- **Tut_10_2.** Wrapping a stored function (poor idea). => [10.2](#)

- **Tut_10_3.** Wrapping a stored function (better idea) . => [47](#)
- **Tut_13_1.** Shows the use of the Lock() utility procedure. => [13.2](#)
- **Tut_14_1.** Extracting a Blob to a file using Get_File(). => [14.3](#)
- **Tut_14_2.** Same as Tut_14_1 but now using a bind placeholder and Select ... into => [62](#)
- **Tut_14_3.** Modifying a Blob using Set_File(). => [14.4](#)
- **Tut_14_4.** Shows that operations on lobs are covered by transaction control. => [14.5](#)