

Direct Oracle Access 4.0 User's Guide



allroundautomations

Contents

Contents	3
Direct Oracle Access	5
Installation	7
TOracleSession component	9
TOracleSession reference	11
TOracleLogon component	44
TOracleLogon reference	45
TOracleQuery component	49
TOracleQuery reference	60
TOracleDataSet component	89
TOracleDataSet reference	92
TQBDefinition object	131
TQBDefinition reference	132
TQBField object	135
TQBField reference	136
TOracleNavigator component	138
TOraclePackage component	139
TOraclePackage reference	141
TOracleEvent component	146
TOracleEvent reference	147
TLOBLocator	152
TLOBLocator reference	157
TOracleObject object	164
TOracleObject reference	166
TOracleReference object	180
TOracleReference reference	181
TOracleScript component	184
TOracleScript reference	187
TOracleCommands object	195
TOracleCommands reference	196
TOracleCommand object	198
TOracleCommand reference	199
TOracleDirectPathLoader component	203
TOracleDirectPathLoader reference	206
TDirectPathColumns object	211
TDirectPathColumns reference	212
TDirectPathColumn object	213
TDirectPathColumn reference	214
TOracleQueue component	216
TOracleSessionPool component	229
TOracleTimestamp object	235
TOracleTimestamp reference	236
TXMLType object	240
TXMLType reference	241
The Package Wizard	242
TOracleCustomPackage component	251
TOracleCustomPackage reference	252
TPLSQLRecord object	254
TPLSQLRecord reference	255
TPLSQLTable object	256
TPLSQLTable reference	257

TOracleProvider component	260
TOracleProvider reference.....	261
Unit reference.....	262
Direct Oracle Access Preferences	266
Direct Oracle Access Designtime Property Defaults.....	268
Multi-threaded applications	269
Dynamic Link Libraries	270
Translating standard messages from English.....	272
Oracle Net compatibility issues	274
Personal Oracle Lite compatibility issues	276
Index	277

Direct Oracle Access

VERSION 4.0, AUGUST 2003

If you use Delphi or C++Builder to access an Oracle database, this component set can make your life a lot easier. With the Direct Oracle Access components and objects you access an Oracle database directly, skipping the Borland Database Engine, only using SQL*Net. This gives you the following advantages:

- No distributing, installing and configuring the BDE. You can use any 32 bits Delphi or C++Builder version to develop Client/Server applications
- No BDE overhead or tradeoffs: your queries will run up to five times faster
- Automatic Master/Detail configuration
- Automatically enforce server-constraints on the client
- Make use of server generated values (defaults, trigger modified columns)
- Query By Example mode for high performance queryable forms without any programming
- Use PL/SQL blocks for server logic in your application
- Increase batch performance with Array DML or Direct Path Loading
- Easily execute SQL scripts similar to SQL*Plus through the TOracleScript component
- Monitor database access information with the Oracle Monitor utility
- Access your stored packages easily through the TOraclePackage component or the Package Wizard
- Encapsulation of standard Oracle packages (dbms_alert, dbms_job, utl_file, ...)
- Many Oracle specific features supported
- Compatible with SQL*Net 1 thru Oracle Net 9, and with Personal Oracle Lite thru Oracle9i

Component overview

Direct Oracle Access consists of the following components:

- You use the TOracleSession component to connect to an Oracle database and to control transactions. You can use many sessions simultaneously, accessing different databases.
- The TOracleLogon component allows you to let a user specify a username, password and database for a TOracleSession through a standard logon dialog.
- You can use a TOracleQuery to execute any SQL statement or PL/SQL block in a session. This is a very low-level component that works directly on top of SQL*Net without any overhead. It should therefore always be used when you don't need data-aware components on the results of a query.
- The TOraclePackage provides a convenient interface to functions, procedures, variables and constants in a stored package.
- The TOracleEvent component allows your application to react to dbms_alert signals and dbms_pipe messages in a background execution thread.

- The `TOracleDataSet` is the source for all your data-aware components. It internally uses a `TOracleQuery` to retrieve and update the database.
- The `TOracleDirectPathLoader` allows you to load data at the highest possible speed by using the Oracle Direct Load Engine.
- The `TOracleQueue` allows you to easily enqueue and dequeue messages through an Oracle Advanced Queue .
- The `TOracleSessionPool` provides a session pooling mechanism for server applications.
- The `TOracleScript` component provides a convenient way to run SQL scripts.
- The `TOracleNavigator` is a component very similar to the standard `TDBNavigator`. It provides additional buttons to support the QBE mode (Query By Example) and record-level refreshing of a `TOracleDataSet`.
- The `TOracleProvider` is a component similar to the standard `TProvider` and can be used to create multi-tiered applications that use the Direct Oracle Access components. This component is obsolete for Delphi & C++Builder 5 and later.

To support Oracle8 and Oracle9 complex data types, Direct Oracle Access provides objects to encapsulate the LOB Locator (`TLOBLocator`), Timestamp (`TOracleTimestamp`), Object (`TOracleObject`), Reference (`TOracleReference`) and XMLType (`TXMLType`). The last three objects are only available in the Object version of Direct Oracle Access.

Unit overview

- Oracle Unit containing `TOracleSession`, `TOracleLogon`, `TOracleQuery`, `TOraclePackage`, `TOracleEvent`, `TLOBLocator`, `TOracleTimestamp`, `TOracleObject`, `TOracleReference`, `TXMLType`, `TOracleDirectPathLoader`, `TOracleQueue`, `TOracleSessionPool` and `TOracleScript`
- OracleData Unit containing the `TOracleDataSet` component
- OracleNavigator Unit containing the `TOracleNavigator` component
- OracleProvider Unit containing the `TOracleProvider` component (Requires Client/Server or Enterprise Edition)
- OracleCI The Oracle Call Interface
- OracleMonitor The interface to the Oracle Monitor utility

License terms for the registered version

See the License.rtf file in the Direct Oracle Access installation directory.

Contacting Allround Automations

If you have any technical questions or suggestions, email us at support@allroundautomations.com. For sales information you can contact sales@allroundautomations.com.

You can also visit our website at: <http://www.allroundautomations.com>.

Installation

To install the Direct Oracle Access components into your development environment, run the included setup.exe. This will install the design time package, units and on-line help file. Depending on the programming language you are using, some additional steps may be required:

Installing the TOraclewwDataSet component for Delphi

If you are using Woll2Woll's InfoPower 3.01 or later, you can install the TOraclewwDataSet component to allow the InfoPower controls to use Direct Oracle Access. From the Delphi Component menu select Install component. From the Unit file name you can browse to the OraclewwData.pas file in Delphi's lib directory. You need to install this component into the Direct Oracle Access package (doa.dpk), which is located in the same directory.

Installing the TOracleProvider component for Delphi 4 or later

If you are using the Client/Server or Enterprise edition of Delphi 4 or later, you can additionally install the TOracleProvider component if you wish to create multi-tiered applications with Direct Oracle Access. From the Delphi Component menu select Install component. From the Unit file name you can browse to the OracleProviderReg.pas file in Delphi's lib directory. You need to install this component into the Direct Oracle Access package (doa.dpk), which is located in the same directory.

Note that in Delphi 5 and later the TOracleProvider is only supported for backward compatibility. For new projects you should use the standard TDataSetProvider component instead. After upgrading a multi-tiered application to Delphi 5 or later, you should consider replacing the TOracleProvider components with TDataSetProvider components.

Installing the TOraclewwDataSet component for C++Builder

If you are using Woll2Woll's InfoPower 3.01 or later, you can install the TOraclewwDataSet component to allow the InfoPower controls to use Direct Oracle Access. From the C++Builder Component menu select Install component. From the Unit file name you can browse to the OraclewwData.cpp file in C++Builder's lib directory.

Installing the TOracleProvider component for C++Builder 4 or later

If you are using the Enterprise edition of C++Builder 4 or later, you can additionally install the TOracleProvider component if you wish to create multi-tiered applications with Direct Oracle Access. From the C++Builder Component menu select Install component. From the Unit file name you can browse to the OracleProviderReg.cpp file in C++Builder's lib directory.

Demos

In the Demos directory, you find the following subdirectories with demo projects:

QueryGrid	Allows the user to enter any SQL statement and view the results in a grid.
DeptGrid	Queries and updates the dept table in a grid.
LongRaw	Reads and writes long raw columns.
ThreadDemo	A multi-threaded application.
ObjectGrid	Manipulates persistent objects. Oracle8 only.

DeptEmp	An example of a master/detail update-form using data-aware components.
PictureDemo	An example using Blob fields.
3Tier	Client/Server only. A 3 Tier application.
PkgApply	An example using a stored package to select records and apply changes for a dataset.
ExtProc	An example of an external procedure in a DLL called from within SQL.
QueueDemo	A demo that enqueues and dequeues messages through the TOracleQueue component.

TOracleSession component

Unit

Oracle

Description

This component is the link to the Oracle database. After setting the Logon properties (LogonUsername, LogonPassword and LogonDatabase), you can establish a connection to the database by calling the LogOn method or setting the Connected property to True. If you want the end-user to specify these logon properties, you can simply use the TOracleLogon component to allow him or her to make a connection.

The TOracleSession is equivalent to an Oracle database session, so it also needs to control the transactions. The Commit and Rollback methods can be used for basic transaction control, but you can also use the Savepoint and RollbackToSavepoint methods for more refined transaction management. The SetTransaction method and IsolationLevel property affect which database changes from other sessions are visible to a session.

You can set many properties at the session level that affect the default behavior of the other database access components that are linked to it. The NullValue property dictates how null values are returned, and the Preferences property contains many other settings that you may want to set for each project.

Special considerations need to be taken into account when creating Multi-threaded applications or working with Dynamic Link Libraries, as described in the respective sections.

Support for standard packages

The TOracleSession component provides a convenient interface to most of the standard packages provided by Oracle: dbms_alert, dbms_application_info, dbms_job, dbms_output, dbms_pipe and utl_file. You can call the functions and procedures within these packages by using the properties that encapsulate them.

Microsoft Transaction Server support

The TOracleSession component provides support for the Oracle Services for MTS. This service allows you to make use of 2 important concepts of MTS: resource pooling and transaction control. By setting the Pooling property to spMTS, physical database sessions will be pooled so that they can be shared between multiple MTS clients. This minimizes the number of concurrent sessions, and prevents that each client request performs a new logon. Furthermore, transactions are now controlled at a higher level by MTS, which even allows distributed transactions in a heterogeneous environment.

External Procedure support

The Oracle8 Server has an External Procedure Service that allows you to call your Object Pascal or C++ functions from within SQL or PL/SQL. You need to make these functions available through a DLL on the database server, which obviously should be an Intel / Windows machine. This can be useful in the following situations:

- You have some application logic that needs to be executed both on the client and on the server.
- You need to implement an algorithm that would be too slow in PL/SQL.

- You need to access services on the server that are not accessible from within PL/SQL, e.g. directory information, disk information, or other devices.

The `TOracleSession` has an `ExtProcShare` procedure that allows you to access the database within the same session and transaction as the caller of the external procedure. This also allows you to raise Oracle exceptions from within external procedures through the `ExtProcRaise` procedure. Furthermore the `TOracleSession` provides functions to convert the low-level parameter data types that are passed to or returned by the external procedure.

Example - Logging on

To log on to the database, you need to set the `LogonUsername`, `LogonPassword` and (optionally) the `LogonDatabase` properties of the session. After this, you can call the `LogOn` method. The following code is an example of this:

```
with MySession do
try
    LogonUsername := 'scott';
    LogonPassword := 'tiger';
    LogonDatabase := 'chicago';
    LogOn;
    ShowMessage('Logged on successfully.');
```

```
except
    on E:EOOracleError do ShowMessage(E.Message);
end;
```

Instead of using `TOracleSession.LogOn`, you can use the `TOracleLogon.Execute` function. This will handle the logon dialog, allowing the user to enter another username, password or database. The return value of this function is `False` if the user only pressed the cancel button. After using this method, you can test the `Connected` property of the session to check if the logon was successful. You will typically call this method in an `OnCreate` event of the main form of your application:

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
    MainLogon.Execute;
    if not MainSession.Connected then Halt;
end;
```

At the first logon an attempt is made to find, load and initialize the Oracle Interface DLL (something like `ora73.dll`, `ora804.dll` or `oci.dll`). If `SQL*Net` or `Net8` is not properly installed, you can expect an exception that describes the error situation in some detail.

TOracleSession reference

This chapter describes all properties, methods and events of the TOracleSession component.

TOracleSession.AfterLogOn

Declaration

```
type TOracleSessionEvent = procedure(Sender: TOracleSession) of
    Object;
property AfterLogOn: TOracleSessionEvent;
```

Description

This event gets called at the end of the LogOn call.

See also

BeforeLogOn

TOracleSession.ApplyUpdates

Declaration

```
procedure ApplyUpdates(const DataSets: array of TOracleDataSet;
    Commit: Boolean);
```

Description

Applies the cached updates of the datasets to the database. If a dataset is a master in a master/detail relation, the updates in all detail datasets will be applied as well. Each dataset must have its CachedUpdates property set to True.

If the Commit parameter is True, the changes will also be committed. If the Commit parameter is False, the updates will not be committed so that you can perform some additional actions before calling CommitUpdates or Rollback:

```
procedure TDeptForm.ApplyButtonClick(Sender: TObject);
begin
    // Apply the changes and do not yet commit
    Session.ApplyUpdates([DeptDataSet], False);
    // Perform some checks and commit or rollback accordingly
    if DeptEmpValid then
        Session.CommitUpdates([DeptDataSet])
    else
        Session.Rollback;
end;
```

See also

CancelUpdates

CommitUpdates

TOracleSession.AutoCommit

Declaration

```
property AutoCommit: Boolean;
```

Description

When true, each update, insert or delete statement is automatically committed by Oracle. Updates in stored procedures and PL/SQL blocks are not committed automatically.

WARNING

This property is obsolete and will no longer be available in version 3.4!

See also

Commit

Rollback

Savepoint

RollbackToSavepoint

TOracleSession.BeforeLogOn

Declaration

```
type TOracleSessionEvent = procedure(Sender: TOracleSession) of  
    object;  
property BeforeLogOn: TOracleSessionEvent;
```

Description

This event gets called at the beginning of the LogOn call. If you want to show a "logging on..." dialog, you can use this event to trigger it.

See also

AfterLogOn

TOracleSession.BreakExecution

Declaration

```
procedure BreakExecution;
```

Description

Breaks the execution of the currently running query in this session, resulting in error "ORA-01013, user requested cancel of current operation". It can be used in multi-threaded applications to abort long running queries on the server.

TOracleSession.BytesPerCharacter

Declaration

```
type TBytesPerCharacterOption = (bc1Byte, bc2Bytes, bc3Bytes,  
    bc4Bytes, bcAutoDetect);  
property BytesPerCharacter: TBytesPerCharacterOption;
```

Description

This property indicates to the session how many bytes is used to store 1 character. This depends on the the character set that is used when creating the database. Setting this value to bcAutoDetect will cause the session to automatically query this information when necessary.

Warning: the default value of this property is bc1Byte, indicating that 1 character takes up 1 byte. For multi-byte character sets you must set this property to the correct value, or to bcAutoDetect. Otherwise, you may expect the following error:

ORA-01026: multiple buffers of size greater than 2000 in the bind list

TOracleSession.CancelUpdates

Declaration

```
procedure CancelUpdates(const DataSets: array of TOracleDataSet);
```

Description

Cancels the cached updates of the datasets by clearing the local change logs. If a dataset is a master in a master/detail relation, the updates in all detail datasets will be cancelled as well. Each dataset must have its CachedUpdates property set to True.

See also

ApplyUpdates

CommitUpdates

TOracleSession.CheckConnection

Declaration

```
function CheckConnection(Reconnect: Boolean): TCheckConnectionResult;  
type TCheckConnectionResult = (ccOK, ccError, ccReconnected);
```

Description

When the server machine has gone down, the database has been shutdown, the network has had a problem, or the database session has been killed, you may end up with a TOracleSession that does not have a working connection with the database anymore. To check if this is the case, and to possibly reconnect the session, you can use the CheckConnection function. The result can have one of the following values:

ccOK	The connection is still okay.
ccError	There is an error with the connection.
ccReconnected	There was an error with the connection, but it has been successfully re-established.

Use the Reconnect parameter to indicate if the function should attempt to re-establish the connection in case of an error. Note that all TOracleDataSet and TOracleQuery components will be closed after re-establishing a connection, so some additional actions will probably be required to handle this situation.

TOracleSession.Commit

Declaration

```
procedure Commit;
```

Description

Commits the current transaction.

See also

Rollback

Savepoint

RollbackToSavepoint

AutoCommit

TOracleSession.CommitUpdates

Declaration

```
procedure CommitUpdates(const DataSets: array of TOracleDataSet);
```

Description

Commits the cached updates of the datasets after calling ApplyUpdates with the Commit parameter set to False. The current database transaction will be committed and the change logs of the datasets will be cleared. If a dataset is a master in a master/detail relation, the updates in all detail datasets will be committed as well. Each dataset must have its CachedUpdates property set to True.

See also

ApplyUpdates

CancelUpdates

TOracleSession.ConnectAs

Declaration

```
type TConnectAsOption = (caNormal, caSYSDBA, caSYSOPER);  
property ConnectAs: TConnectAsOption;
```

Description

Indicates if the user should be logged on with SYSDBA, SYSOPER or a normally privileged connection. The caSYSDBA and caSYSOPER options are only possible if you have been granted the corresponding system privilege and if you know the corresponding password.

SYSOPER permits you to perform alter database open/mount, alter database backup, archive

log, and recover, and includes the restricted session privilege.

SYSDBA contains all system privileges with admin option, and the sysoper system privilege; permits create database and time-based recovery.

In both situations the default schema is now SYS instead of the schema of the username of the session.

The TOracleLogon dialog supports this feature through the IdConnectAs option.

Note

This property will only work on Net8.

TOracleSession.Connected

Declaration

property Connected: Boolean;

Description

Indicates if the session is connected to a database. Setting this property is equivalent to calling LogOn or LogOff, but allows you to establish a connection from within Delphi's IDE.

See also

LogOn

LogOff

RollbackOnDiconnect

LogonUsername

LogonPassword

LogonDatabase

TOracleLogon

TOracleSession.Cursor

Declaration

property Cursor: TCursor;

Description

Allows you to determine the shape of the mouse cursor during SQL*Net initialization, logon, and when session properties are set that require database access.

TOracleSession.DBMS_Alert

Declaration

```
TDBMS_Alert = class(TDBMSPackage)
procedure Set_Defaults(const Sensitivity: Integer);
procedure Register(const Name: string);
procedure Remove(const Name: string);
procedure RemoveAll;
procedure WaitAny(out Name: string; out Message: string; out Status:
    Integer; const TimeOut: Integer);
procedure WaitOne(const Name: string; out Message: string; out
    Status: Integer; const TimeOut: Integer);
procedure Signal(const Name: string; const Message: string);
end;
property DBMS_Alert: TDBMS_Alert;
```

Description

The dbms_alert package provides support for the asynchronous notification of database events. To wait for events in a background thread of your application, you can use the TOracleEvent component instead. Detailed information about this package can be found in the *Server Application Developer's Guide*.

The following constants have been defined for the dbms_alert package:

wsAlert	Wait status: Alert occurred
wsTimeOut	Wait status: TimeOut occurred

The following example signals alert 'SP' with message 'EXIT' and commits it:

```
with MainSession do
begin
    DBMS_Alert.Signal('SP', 'EXIT');
    Commit;
end;
```

TOracleSession.DBMS_Application_Info

Declaration

```
TDBMS_Application_Info = class(TDBMSPackage)
procedure Set_Module(const Module_Name: string; const Action_Name:
    string);
procedure Set_Action(const Action_Name: string);
procedure Read_Module(out Module_Name: string; out Action_Name:
    string);
procedure Set_Client_Info(const Client_Info: string);
procedure Read_Client_Info(out Client_Info: string);
end;
property DBMS_Application_Info: TDBMS_Application_Info;
```

Description

The dbms_application_info package can be used with Oracle Trace and the SQL trace facility to record the name of the executing module or transaction in the database for use later when tracking the performance of various modules. Detailed information about this package can be found in the *Server Tuning Guide*.

The following example registers the title of the application through the `Set_Client_Info` procedure after connecting the session in the `OnCreate` event handler of the `main` form. The application title will be visible in the `v$session` table for this session.

```
procedure TMainForm.FormCreate(Sender: TObject);
begin
    with MainSession do
        begin
            Connected := True;
            DBMS_Application_Info.Set_Client_Info(Application.Title);
        end;
    end;
```

TOracleSession.DBMS_Job (Oracle 7.2 or higher)

Declaration

```
TDBMS_Job = class(TDBMSPackage)
procedure Submit(out Job: Integer; const what: string; const
    Next_Date: TDateTime; const Interval: string; const No_Parse:
    Boolean);
procedure Remove(const Job: Integer);
procedure Change(const Job: Integer; const What: string; const
    Next_Date: TDateTime; const Interval: string);
procedure What(const Job: Integer; const What: String);
procedure Next_Date(const Job: Integer; const Next_Date: TDateTime);
procedure Interval(const Job: Integer; const Interval: string);
procedure Broken(const Job: Integer; const Broken: Boolean; const
    Next_Date: TDateTime);
procedure Run(const Job: Integer);
end;
property DBMS_Job: TDBMS_Job;
```

Description

You can use the procedures in the `dbms_job` package to schedule and manage jobs in the job queue. Detailed information about this package can be found in the *Server Administrator's Guide*.

The following example schedules a PL/SQL procedure `Clean_Up('ALL')` to be called every other day at 4:00, starting tomorrow:

```

procedure TMainForm.ButtonClick(Sender: TObject);
var Job: Integer;
begin
    MainSession.DBMS_Job.Submit(Job, 'Clean_Up(''ALL'');', Date + 1 +
        StrToTime('04:00'), 'sysdate + 2', True);
    ShowMessage('Job ' + IntToStr(Job) + ' scheduled');
end;

```

TOracleSession.DBMS_Output

Declaration

```

TDBMS_Output = class(TDBMSPackage)
procedure Enable(const Buffer_Size: Integer);
procedure Disable;
procedure Put(const a: Variant);
procedure Put_Line(const a: Variant);
procedure New_Line;
procedure Get_Line(out Line: string; out Status: integer);
procedure Get_Lines(out Lines: string; var NumLines: Integer);
end;
property DBMS_Output: TDBMS_Output;

```

Description

The dbms_output package can be used to send messages to or from stored procedures, packages, and triggers. The Put and Put_Line procedures in this package allow you to place information in a buffer that can later be read by the Get_Line or Get_Lines procedure within the same session. Detailed information about this package can be found in the *Server Application Developer's Guide*.

The following constants have been defined for the dbms_output package:

glSuccess	Get line: Success
glNoMoreLines	Get line: No more lines

The following example enables output, calls a stored procedure, and displays the output that this stored procedure has produced in a Memo:

```

procedure TMainForm.ButtonClick(Sender: TObject);
var Status: Integer;
    Line: string;
begin
    // Enable output for a maximum of 100,000 bytes
    MainSession.DBMS_Output.Enable(100000);
    // Call the procedure
    MyProcedure.Execute;
    // Retrieve all lines and display them in a memo
    Memo.Clear;
    repeat
        MainSession.DBMS_Output.Get_Line(Line, Status);
        if Status <> glSuccess then Break;
        Memo.Lines.Add(Line);
    until False;
end;

```

TOracleSession.DBMS_Pipe

Declaration

```

TDBMS_Pipe = class(TDBMSPackage)
procedure Pack_Message(const Item: Variant);
procedure Pack_Message_Raw(const Item: string);
procedure Pack_Message_Rowid(const Item: string);
procedure Unpack_Message(out Item: Variant);
procedure Unpack_Message_Raw(out Item: string);
procedure Unpack_Message_Rowid(out Item: string);
function Next_Item_Type: Integer;
function Create_Pipe(const PipeName: string; const MaxPipeSize:
    Integer; const Private: Boolean): Integer;
function Remove_Pipe(const PipeName: string): Integer;
function Send_Message(const PipeName: string; const TimeOut:
    Integer; const MaxPipeSize: Integer): Integer;
function Receive_Message(const PipeName: string; const TimeOut:
    Integer): Integer;
procedure Reset_Buffer;
procedure Purge(const PipeName: string);
function Unique_Session_Name: string;
end;
property DBMS_Pipe: TDBMS_Pipe;

```

Description

The dbms_pipe package allows two or more sessions in the same instance to communicate. Oracle pipes are similar in concept to the pipes used in UNIX, but Oracle pipes are not implemented using the operating system pipe mechanisms. Information sent through Oracle pipes is buffered in the systemglobal area (SGA). Detailed information about this package can be found in the *Server Application Developer's Guide*.

The following constants have been defined for the dbms_pipe package:

cpSuccess	Create pipe: Success
rpSuccess	Remove pipe: Success
smSuccess	Send message: Success

smTimedOut	Send message: Timed out
smInterrupted	Send message: Interrupted
rmSuccess	Receive message: Success
rmTimedOut	Receive message: Timed out
rmRecordTooBig	Receive message: Record too big
rmInterrupted	Receive message: Interrupted
niNoMoreItems	Next item: No more items
niVarchar2	Next item: Varchar2
niNumber	Next item: Number
niRowid	Next item: Rowid
niDate	Next item: Date
niRaw	Next item: Raw

The following example sends a message to pipe 'MYPIPE' with the text 'Button pressed' and the current date as information:

```
procedure TMainForm.ButtonClick(Sender: TObject);
begin
    with MainSession.DBMS_Pipe do
        begin
            Pack_Message('Button pressed');
            Pack_Message(Now);
            if Send_Message('MYPIPE', 60, 8192) = smSuccess then
                ShowMessage('Message sent');
        end;
    end;
```

TOracleSession.DesignConnection

Declaration

```
property DesignConnection: Boolean;
```

Description

A connection made at design-time during development is usually not very suitable when you distribute your application to an end-user. By setting the DesignConnection property to True, the design-time values of the following properties will be cleared at run-time:

- ♦ Connected
- ♦ LogonUsername
- ♦ LogonPassword
- ♦ LogonDatabase

This way you can distribute your applications unchanged, and always have a connection at design-time during development.

TOracleSession.ErrorMessage

Declaration

```
function ErrorMessage(ErrorCode: Integer): string;
```

Description

Returns the message associated with the ErrorCode. The message corresponds to the messages you can find in the *"Oracle Server Messages Guide"*.

See also

ReturnCode

TOracleSession.ExpirationMessage

Declaration

```
property ExpirationMessage: string;
```

Description

Runtime property that contains an error message after an attempt was made to logon with an account that has expired.

See also

SetPassword

TOracleSession.ExternalAUT

Declaration

```
property ExternalAUT: Pointer;
```

Description

Runtime read-only property to obtain the Net8 authentication handle (also known as a user handle) of the session. This property is only valid if the session is connected, and if it is in OCI8 mode.

TOracleSession.ExternalENV

Declaration

```
property ExternalENV: Pointer;
```

Description

Runtime read-only property to obtain the Net8 environment handle for the session. This property is only valid if the session is connected, and if it is in OCI8 mode.

TOracleSession.ExternalLDA

Declaration

```
property ExternalLDA: Pointer;
```

Description

Runtime property to attach the TOracleSession to an LDA (Logon Data Area) of another host program, or to access the LDA of the session. When you set this property, it is equivalent to calling the LogOn method. Setting ExternalLDA to nil is equivalent to calling LogOff.

This property may be useful when working with Dynamic Link Libraries, though it is usually better and easier to use the Share procedure.

Warning

This property is not available if you are using Net8 in OCI8 mode. If you use ExternalLDA in your application, it may be a good idea to set the UseOCI7 property to True. This way, your application will function properly on SQL*Net and Net8 clients. Net8 specific features will not be available however.

TOracleSession.ExternalSRV

Declaration

```
property ExternalSRV: Pointer;
```

Description

Runtime read-only property to obtain the Net8 server handle for the session. This property is only valid if the session is connected, and if it is in OCI8 mode.

TOracleSession.ExternalSVC

Declaration

```
property ExternalSVC: Pointer;
```

Description

Runtime property to attach the TOracleSession to a Net8 service context handle of another host program, or to access the service context handle of the session. When you set this property, it is equivalent to calling the LogOn method. Setting ExternalSVC to nil is equivalent to calling LogOff.

This property may be useful when working with Dynamic Link Libraries, though it is usually better and easier to use the Share procedure.

TOracleSession.ExtProcRaise

Declaration

```
procedure ExtProcRaise(ErrorNumber: Integer; const ErrorMessage:
    string);
```

Description

If the session is shared from the context of an external procedure call, you can use this procedure to raise an Oracle exception when the procedure returns to the caller. The Oracle exception will have the given ErrorNumber and ErrorMessage. For example:

```
Session.ExtProcRaise(20000, 'Department number does not exist');
```

If you are trapping an EOracleError exception, you can propagate it to the caller like this:

```
try
    Query.Execute;
except
    on E: EOracleError do
        Session.ExtProcRaise(E.ErrorCode, E.Message);
end;
```

Note that processing continues after a call to ExtProcRaise, it merely sets a status for the call that is translated to an Oracle exception later. Any output parameter or return value you may have defined will be ignored though.

TOracleSession.ExtProcShare

Declaration

```
procedure ExtProcShare(Context: Pointer);
```

Description

If you want to call Object Pascal or C++ functions from within SQL or PL/SQL, you can make use of the Oracle8 External Procedure Service. This service can map PL/SQL calls to an external procedure call in a DLL. The DLL must be defined through an Oracle library, and each procedure within the DLL must be defined through a PL/SQL external procedure definition.

As an example we assume that we have a DLL called dept.dll with a function EmpCount that returns the number of employees in a department. Implementing this function in a DLL does not make much sense in real life, but it serves quite well as a simple and complete demonstration of an external procedure.

Create an Oracle library

First we need to create an Oracle library that encapsulates our DLL:

```
create or replace library DeptLib AS 'C:\ExtProc\dept.dll';
```

This library will be referenced in the external procedure definitions.

Create an external procedure definition

Next we need to create an external procedure definition for the EmpCount function in the library:

```
create or replace
  function EmpCount(p_DeptNo in dept.deptno%type)
    return number
  as external language c
    name "EmpCount"
    library DeptLib
    with context;
```

Regardless whether the DLL is created in Delphi or C++Builder, we must always use the "language c" option. The "calling standard pascal" option that Oracle provides seems incompatible with Delphi functions, so we need to use the cdecl directive for our Delphi functions if we want to get the right parameter mapping. The "with context" option adds an implicit parameter to the external procedure that allows you to reuse the database session of the caller. The "with context" option is required if you want to use the ExtProcShare procedure!

Note that you can also place these external function definitions in a package specification, so that you can encapsulate related DLL functions in a single package if desired.

For additional information about external procedure definitions, see the *Application Developer's Guide*.

Create a DLL with one or more external functions

This simple example DLL contains just one external function, EmpCount:


```

library dept;

uses
    SysUtils,
    Oracle,
    DeptDataModuleUnit in 'DeptDataModuleUnit.pas';

var
    // The saved exit procedure
    SaveExit: Pointer;
    // The data module with a session and a query
    DataModule: TDeptDataModule = nil;
    // The OCI number result for the EmpCount function
    EmpCountResult: TOCINumber = nil;

// Count the number of employees in the given department
function EmpCount(Context: Pointer; p_DeptNo: TOCINumber):
    TOCINumber; cdecl;

var
    DeptNo: Integer;
begin
    Result := nil;
    with DataModule do
        try
            // Share the session of the caller with the TOracleSession
            Session.ExtProcShare(Context);
            // Convert the OCI number to a plain integer and set the variable
            DeptNo := Session.OCINumberToInt(p_DeptNo);
            EmpCountQuery.SetVariable('deptno', DeptNo);
            // Execute the query that counts the employees
            EmpCountQuery.Execute;
            // Allocate an OCI number for the result if necessary
            if EmpCountResult = nil then
                EmpCountResult := Session.OCINumberCreate;
            Result := EmpCountResult;
            // Get the count from the result set
            Session.OCINumberFromInt(Result,
            EmpCountQuery.Field('empcount'));
        except
            // Translate all Delphi exceptions to Oracle exceptions
            on E: EOracleError do
                Session.ExtProcRaise(E.ErrorCode, E.Message);
            on E: Exception do
                Session.ExtProcRaise(20000, E.Message);
        end;
    end;

// Free all preserved resources when the DLL is unloaded
procedure LibExit;
begin
    if EmpCountResult <> nil then
        DataModule.Session.OCINumberFree(EmpCountResult);
    DataModule.Free;
    // Restore the exit procedure

```

```

    ExitProc := SaveExit;
end;

// Export the EmpCount function
exports
    EmpCount;

begin
    // Create the data module when the DLL is loaded
    DataModule := TDeptDataModule.Create(nil);
    // Save and override the exit procedure
    SaveExit := ExitProc;
    ExitProc := @LibExit;
end.

```

This example can serve as a model for external procedure DLL's. It creates a data module during initialization, which is destroyed when the DLL is unloaded. Each function first calls ExtProcShare with the context of the session of the caller. The first call will suffer from an initial delay as the DLL is loaded, Net8 initialization is performed, and the context session is shared. Subsequent calls will have an immediate response, because the DLL will remain loaded for the duration of the session and the data module will also remain instantiated.

Parameter mapping

The parameter and return value of the external procedure are of type TOCINumber. This is an encapsulation of an OCI (Oracle Call Interface) number, but is in fact not more than a simple pointer to an opaque data structure. The OCINumberToInt and OCINumberFromInt procedures that are used in the function convert between an integer and an OCI number. The OCINumberCreate function creates an OCI number for the return value. Similar functions exist for float parameters (OCINumberToFloat and OCINumberFromFloat) and date parameters (OCIDateToDateTime and OCIDateFromDateTime). String parameters (varchar2, char) are mapped to zero-terminated strings (PChar), and binary integer parameters (pls_integer and binary_integer) are mapped to standard 32 bit integers. See the *Application Developer's Guide* for more details about default and explicit parameter type mapping.

For parameters that require that you allocate memory within the call (e.g. strings or OCI return values), you must make sure that the data remains valid when the call returns. You must also make sure that you do not introduce memory leaks for each call. In the example above we reuse the same return value after it has been allocated once, and free it in the finalization section. For OCI number or OCI date output parameters, there will already be an allocated OCI number or OCI date instance.

Exception handling

If an external procedure raises an exception that propagates to the External Procedure Service, the process will stop and an *Unknown Software Exception* will occur. You should make sure that this never happens. You can use the ExtProcRaise procedure to raise a proper Oracle exception for the caller. In the example above we handle all Delphi exceptions and propagate them as an appropriate Oracle error to the caller.

TOracleSession.FlushObjects

Declaration

```
procedure FlushObjects;
```

Description

Besides flushing each individual modified object instance in the cache, you can also flush all modified instances at once in one single network roundtrip, thereby reducing network traffic. You can do so by calling the Commit method of the session, in which case your transaction will be ended. You can also call the FlushObjects method of the session to flush all objects without committing the current transaction.

TOracleSession.InTransaction

Declaration

```
function InTransaction: Boolean;
```

Description

Indicates if the session has started a transaction.

TOracleSession.IsolationLevel

Declaration

```
type TIsolationLevelOption = (ilUnchanged, ilReadCommitted,  
    ilSerializable);  
property IsolationLevel: TIsolationLevelOption;
```

Description

Determines how Oracle controls data concurrency for the session. Setting the IsolationLevel to ilReadCommitted causes your transaction to see all records committed at any time in other sessions. Setting it to ilSerializable causes you to see only those records committed before your transaction started.

When IsolationLevel is set to stUnchanged, the initialization parameter ISOLATION_LEVEL of the Oracle instance determines data concurrency control.

See also

SetTransaction

TOracleSession.LogOff

Declaration

```
procedure LogOff;
```

Description

This procedure logs off from the database. LogOff is automatically called when a session is destroyed (by calling the Free method or by destroying the component that owns the session).

See also

LogOn

Connected

RollbackOnDisconnect

TOracleSession.LogOn

Declaration

```
procedure LogOn;
```

Description

Logs on to the database. If the session is already logged on, it is first logged off.

If an error occurs, an EOracleError exception is raised.

See also

LogOff

Connected

LogonUsername

LogonPassword

LogonDatabase

TOracleLogon

TOracleSession.LogonDatabase

Declaration

```
property LogonDatabase: string;
```

Description

Together with LogonUsername and LogonPassword this property is used by the LogOn procedure to construct the connect-string Username/Password@Database.

See also

LogOn

LogOff

Connected

LogonUsername

LogonPassword

TOracleLogon

TOracleSession.LogonPassword

Declaration

```
property LogonPassword: string;
```

Description

Together with LogonUsername and LogonDatabase this property is used by the LogOn procedure to construct the connect-string Username/Password@Database.

See also

LogOn

LogOff

Connected

LogonUsername

LogonDatabase

TOracleLogon

TOracleSession.LogonUsername

Declaration

```
property LogonUsername: string;
```

Description

Together with LogonPassword and LogonDatabase this property is used by the LogOn procedure to construct the connect-string Username/Password@Database.

You can also assign a complete connect string as above to LogonUsername. After connecting to the database, the three properties will reflect the actual values.

See also

LogOn

LogOff

Connected

LogonPassword

LogonDatabase

TOracleLogon

TOracleSession.MessageTable

Declaration

```
property MessageTable: string;
```

Description

The TOracleDataSet component can translate error messages raised by primary key, unique key, foreign key and check constraints through a message table. The name of this message table is defined at the session level, and should be defined like this:

```
create table my_messages
(
  Constraint_Name varchar2(30) not null,
  Actions          varchar2(3)  not null,
  Parent_Child     varchar2(1)  not null,
  Error_Message    varchar2(2000)
);
```

Constraint_Name	the name of the constraint that is violated.
Actions	Indicates if the message is to be displayed for inserts ('I'), updates ('U') and deletes ('D'). You can combine the letters to indicate multiple actions, e.g. 'IU' for insert and update. You can use '*' to indicate all actions.
Parent_Child	Indicates if the message is to be displayed when the constraint is violated at the parent ('P') or child ('C') side. This column is only useful for foreign key constraints, use a '*' for all other constraints.
Error_Message	The actual message that is displayed.

Example

Table employees has a primary key emp_pk on column empno, and a foreign key emp_dept_fk on the deptno column to the departments table. The following messages could be defined:

Constr_Name	Actions	PC	Message
EMP_PK	*	*	Employee number already exists
EMP_DEPT_FK	*	C	Department does not exist
EMP_DEPT_FK	D	P	Cannot delete department while employees exist
EMP_DEPT_FK	U	P	Cannot change department number while employees exist

See also

TOracleDataSet.EnforceConstraints

TOracleDataSet.UseMessageTable

TOracleSession.MonitorMessage

Declaration

```
procedure MonitorMessage(const Msg: string);
```

Description

Sends the given string to the Oracle Monitor, where it will be displayed as an activity with the message as description. The message activity will have a different color than other activities, so that they can easily be recognized.

This function is only useful if you have included the OracleMonitor unit in your project.

TOracleSession.MTSOptions

Declaration

```
type TMTSOptions = set of (moImplicit, moUniqueServer);
property MTSOptions: TMTSOptions;
```

Description

If the Pooling property is set to spMTS, you can use this property to control how the database session is obtained from the pool:

moImplicit	When the TOracleSession is logged on, it is implicitly enlisted in any MTS transaction. If you disable this option, you will only make use of the session pooling mechanism of the Oracle Service for MTS.
moUniqueServer	Not yet used.

When a session is enlisted in an MTS transaction, you should never explicitly commit or rollback your updates. The MTS Server will implicitly commit the Oracle transaction when the MTS application commits the MTS transaction. The Oracle transaction will be rolled back when the MTS transaction is aborted.

If you call TOracleSession.Commit or Rollback when the session is enlisted in an MTS transaction, an exception will be raised. Component methods that would normally commit or rollback a transaction (TOracleDataSet.Post, TOracleSession.ApplyUpdates), will not do so when the session is enlisted in an MTS transaction but will rely on MTS transaction control instead.

TOracleSession.NullValue

Declaration

```
type TNullValueOption = (nvNull, nvUnAssigned);
property NullValue: TNullValueOption;
```

Description

When you retrieve a null field or variable, the NullValue property of the session determines the actual value you receive.

If NullValue = nvNull, you receive a Null variant, which behaves as follows:

- When Null is assigned to a non-variant variable, an EVariantError is raised
- To test for Null, use `if VarIsNull(v) then`
- If you use relational operators on Null, it will be treated as 'less' than other values
- If you use non-relational operators on Null, the result will be Null

If NullValue = nvUnAssigned, you receive an UnAssigned variant, which behaves as follows:

- When UnAssigned is assigned to a non-variant variable, it converts to 0, "" or zero date

- To test for UnAssigned, use `if VarIsEmpty(v) then`
- If you use relational operators on UnAssigned, an `EvariantError` is raised
- If you use non-relational operators on UnAssigned, an `EvariantError` is raised

TOracleSession.OCIDateCreate

Declaration

```
function OCIDateCreate: TOCIDate;
```

Description

This support function for external procedures allocates an OCI date instance. This is only useful if you need to return an OCI date from a function. To free the OCI date instance call `OCIDateFree`.

TOracleSession.OCIDateFree

Declaration

```
procedure OCIDateFree(OCIDate: TOCIDate);
```

Description

This support function for external procedures frees an OCI date instance. This is only useful if you have previously created an OCI date through `OCIDateCreate` and need to free it to deallocate the memory.

TOracleSession.OCIDateFromDateTime

Declaration

```
procedure OCIDateFromDateTime(OCIDate: TOCIDate; DateValue:  
    TDateTime);
```

Description

This support function for external procedures converts a `TDateTime` to an OCI date. You can use `OCIDateToDateTime` for the opposite conversion.

TOracleSession.OCIDateToDateTime

Declaration

```
function OCIDateToDateTime(OCIDate: TOCIDate): TDateTime;
```

Description

This support function for external procedures converts an OCI date to a `TDateTime`. You can use `OCIDateFromDateTime` for the opposite conversion.

TOracleSession.OCINumberCreate

Declaration

```
function OCINumberCreate: TOCINumber;
```

Description

This support function for external procedures allocates an OCI number instance. This is only useful if you need to return an OCI number from a function. To free the OCI number instance call OCINumberFree.

TOracleSession.OCINumberFree

Declaration

```
procedure OCINumberFree(OCINumber: TOCINumber);
```

Description

This support function for external procedures frees an OCI number instance. This is only useful if you have previously created an OCI number through OCINumberCreate and need to free it to deallocate the memory.

TOracleSession.OCINumberFromFloat

Declaration

```
procedure OCINumberFromFloat(OCINumber: TOCINumber; FloatValue:  
    Double);
```

Description

This support function for external procedures converts a Double to an OCI number. You can use OCINumberToFloat for the opposite conversion.

TOracleSession.OCINumberFromInt

Declaration

```
procedure OCINumberFromInt(OCINumber: TOCINumber; IntValue: Integer);
```

Description

This support function for external procedures converts an Integer to an OCI number. You can use OCINumberToInt for the opposite conversion.

TOracleSession.OCINumberToFloat

Declaration

```
function OCINumberToFloat(OCINumber: TOCINumber): Double;
```

Description

This support function for external procedures converts an OCI number to a Double. You can use OCINumberFromFloat for the opposite conversion.

TOracleSession.OCINumberToInt

Declaration

```
function OCINumberToInt(OCINumber: TOCINumber): Integer;
```

Description

This support function for external procedures converts an OCI number to an Integer. You can use OCINumberFromInt for the opposite conversion.

TOracleSession.OnChange

Declaration

```
type TOracleSessionEvent = procedure(Sender: TOracleSession) of  
    Object;  
property OnChange: TOracleSessionEvent;
```

Description

Called whenever the Session gets connected or disconnected. Could be used to enable menu-items.

TOracleSession.OptimizerGoal

Declaration

```
type TOptimizerGoalOption = (ogUnchanged, ogChoose, ogFirstRows,  
    ogAllRows, ogRule)  
property OptimizerGoal: TOptimizerGoalOption;
```

Description

Determines how the Oracle optimizer approaches to optimize a SQL statement. For more information, see the *"Oracle 7 Server Concepts"* manual.

TOracleSession.POLite

Declaration

```
function POLite: Boolean;
```

Description

Returns true if connected to a Personal Oracle Lite database. Only valid if connected.

TOracleSession.Pool

Declaration

property Pool: TOracleSessionPool;

Description

Determines from which pool the session will obtain a connection when LogOn is called or Connected is set to True. The connection is released into the pool when the session is disconnected.

If this property is nil and the Pooling property is set to spInternal, the global SessionPool will be used instead.

TOracleSession.Pooling

Declaration

type TOracleSessionPooling = (spNone, spInternal, spMTS);

property Pooling: TOracleSessionPooling;

Description

This property indicates that a session should be obtained from a session pool. The session pool will either be managed internally in the application, or it will be managed by the Oracle Service for MTS. This can be useful if you need to frequently create and free TOracleSession instances, but can't afford to make new database connections every time because of performance consequences.

This option is most useful in an MTS (Microsoft Transaction Server) environment, because a TMTSDDataModule and TMTSAutoObject instances are continuously created and destroyed by the MTS Server between client requests, and any TOracleSession that is owned by this instance will also be created and destroyed. This would imply that each request would cause a database logon and logoff. To prevent this, you can set the Pooling property to one of the following values:

- | | |
|------------|--|
| spInternal | The database sessions will be pooled internally by Direct Oracle Access. The Pool property determines which pool will be used. |
| spMTS | The database session will be pooled by the Oracle Service for MTS. This service is only available for Oracle8i and later, and is a requirement if you want to use spMTS. |

Each time a TOracleSession instance is logged on, it will attempt to reuse a database session from the pool. If there are no sessions available, a new session will be created. When the TOracleSession instance is logged off, the database session will be released into the pool. Make sure that you commit or rollback your updates before logging off the session, unless it is enlisted in an MTS transaction!

TOracleSession.Preferences

Declaration

```
property Preferences: TSessionPreferences;
```

Description

The Preferences property can be used to affect the behavior of other database access components linked to the session. The following properties can be defined:

```
property FloatPrecision: Integer;
```

Maximum precision to be represented as a floating point field (Double). Setting this property to 0 will cause all non-integer numbers to be represented as a floating point field, even though a Double has a maximum precision of 15 digits. Setting it to a non-zero value will cause higher-precision numbers to be represented as a string field. The number will be converted to a string on the server, using the current NLS_LANG settings. This preference affects fields in TOracleQuery and TOracleDataSet components.

```
property IntegerPrecision: Integer;
```

Maximum precision to be represented as an integer field. Setting this property to 0 will cause all integer numbers with a precision of 9 digits or less to be represented as an integer, and all other numbers as a floating point value, depending on the FloatPrecision preference. This preference affects fields in TOracleQuery and TOracleDataSet components.

```
property SmallIntPrecision: Integer;
```

Maximum precision to be represented as a small integer field (SmallInt). Setting this property to 0 will cause all integer numbers with a precision of 4 digits or less to be represented as a small integer. This preference only affects fields in the TOracleDataSet component, since the TOracleQuery component does not have small integer fields. The default value of this property is -1, so that small integer fields will never occur.

```
property MaxStringFieldSize: Integer;
```

Maximum string size to be represented as a string field. Setting this property will cause all larger string fields to be represented as a TMemoField. This preference only affects fields in the TOracleDataSet component, since the TOracleQuery component does not have memo fields. The default value of this property is 0, so that strings will never be represented by memo fields.

Standard BDE DataSets such as TTable and TQuery always treat strings longer than 255 characters as memo fields. Compatibility with BDE datasets may be one reason to set this property. Another reason may be the fact that DBRichEdit controls only support formatted text on TMemoFields. For TOracleDataSet components in a remote datamodule in a multi-tiered application in Delphi 5, it is required that this property is set to 255. This happens automatically.

```
property UseOCI7: Boolean;
```

Use OCI7 on Net8 for this session. Setting this property to True will cause the session to use the old, but more stable SQL*Net 2.3 functions in Net8. This can be helpful in case of Net8 problems, but can only be used if your application is not using any Net8 specific objects (TLOBLocator, TOracleObject and TOracleReference).

```
property ConvertCRLF: Boolean;
```

Convert between CRLF pairs (Client) and LF (Server). In an Oracle Database, multiple lines of text are separated by just a linefeed (#10) character. On the Windows Operating System, the carriage return / linefeed (#13 / #10) combination is used. Direct Oracle Access will automatically convert between these two conventions. Setting this preference to False will

disable this conversion. Use this preference with care.

property TrimStringFields: Boolean;

Remove trailing spaces from TOracleDataSet fields when fetching varchar2 and char columns from the database. These trailing spaces usually just get in the way when modifying string fields, so this property is True by default.

property ZeroDateIsNull: Boolean;

Determines if a TDateTime value of 0.0 is interpreted as Null. If you set this property to False, TDateTime values cannot be Null as 0.0 would correspond to 30/12/1899. To use date variables that can be Null you must use Variant values, which can of course be Null or Unassigned.

property NullLOBIsEmpty: Boolean;

Determines if a null BLOB or CLOB posted through a TOracleDataSet should be treated as an empty LOB (empty_blob() or empty_clob()) instead. It can be useful to prevent null LOB columns, because you can now assume that a valid LOB Locator is present in each record.

property TemporaryLOB: TTemporaryLOBOption;

type TTemporaryLOBOption = (tlNone, tlCache, tlNoCache);

Determines if a BLOB or CLOB posted through a TOracleDataSet should be passed as a normal LOB (tlNone), a cached temporary LOB (tlCache), or an uncached temporary LOB (tlNoCache). For more information about temporary LOB's, see the TLOBLocator.CreateTemporary section. This property is ignored on Oracle Net 8.0 clients, which do not support temporary LOB's.

property NullObjectIsEmpty: Boolean;

Determines if a null object posted through a TOracleDataSet should be treated as an empty object (the default constructor with nulls for all non-object attributes) or as atomically null. It can be useful to prevent null object columns, because you can now assume that a valid object is present in each record.

TOracleSession.ReturnCode

Declaration

function ReturnCode: Integer;

Description

The result of the last executed session procedure. This number corresponds to the codes you can find in the *"Oracle Server Messages Guide"*.

See also

ErrorMessage

TOracleSession.Rollback

Declaration

procedure Rollback;

Description

Rolls back the current transaction.

See also

Commit

Savepoint

RollbackToSavepoint

AutoCommit

TOracleSession.RollbackOnDisconnect

Declaration

```
property RollbackOnDisconnect: Boolean;
```

Description

This property indicates if the session should rollback the current transaction if it is disconnected. This applies to the LogOff procedure, the Connected property and to freeing a TOracleSession component. If an application crashes without freeing its TOracleSession components, the Oracle Server will always rollback the current transaction.

TOracleSession.RollbackToSavepoint

Declaration

```
procedure RollbackToSavepoint(const ASavepoint: string);
```

Description

Rolls back the current transaction to a previously identified savepoint.

See also

Commit

Rollback

Savepoint

AutoCommit

TOracleSession.Savepoint

Declaration

```
procedure Savepoint(const ASavepoint: string);
```

Description

Identifies a point in a transaction to which you can later roll back. In the parameter, you can pass any name to identify the savepoint.

See also

Commit

Rollback

RollbackToSavepoint

AutoCommit

TOracleSession.ServerVersion

Declaration

```
function ServerVersion: string;
```

Description

Returns a string containing the version of the database that the session is connected to (something like 'Personal Oracle7 Release 7.2.2.3.1').

TOracleSession.SetPassword

Declaration

```
procedure SetPassword(const NewPassword: string);
```

Description

Changes the password of the current user. For Oracle7, the session must already be connected. For Oracle8, the password can be set without being connected to the server.

Password expiration

In an Oracle8 server you can enable password expiration as part of the user authentication process of your database. You can set the lifetime of a password, the grace period (the period that a user only gets a warning that the password has expired), and a password history (to make sure that a user does not reuse the same password for a specific amount of time).

The TOracleLogon component handles password expiration automatically. It issues a warning during the grace period that the password will expire within a number of days and asks the user if he or she wants to change the password now. After the grace period, the TOracleLogon component will force the user to change the password, or the logon will fail.

To support this mechanism in your own application without using the TOracleLogon component, the TOracleSession has an ExpirationMessage property and a SetPassword method.

After a successful call to LogOn during the grace period, the ExpirationMessage property contains the Oracle warning message that specifies the number of days left before the password expires.

After an unsuccessful call to LogOn after the grace period, you need to change the password without actually being logged on. The SetPassword method supports this. You need to set the LogonUsername, LogonPassword (the old password) and the LogonDatabase properties and call the SetPassword method with the new password. If the method completes successfully, the password is changed but the session is not connected. You still need to call the LogOn method afterwards.

The following example logs on to the database, issues a warning during the grace period, and asks for a new password when it has expired:

```

with MySession do
try
  LogonUsername := UsernameEdit.Text;
  LogonPassword := PasswordEdit.Text;
  LogonDatabase := DatabaseEdit.Text;
  Logon;
  if ExpirationMessage <> '' then
    ShowMessage(ExpirationMessage)
  else
    ShowMessage('Logged on successfully.');
```

```

except
  on E:EOraclError do
  begin
    if E.ErrorCode <> 28001 then
      ShowMessage(E.Message);
    else try // ORA-28001, The account has expired
      SetPassword(GetNewPassword);
      Logon;
    except
      on E:EOraclError do ShowMessage(E.Message);
    end;
  end;
end;
end;
```

You can also call `SetPassword` when the `TOraclSession` is already connected. After the call to `SetPassword`, the session remains connected. For SQL*Net 1.x or 2.x, you can only use `SetPassword` when the session is connected. Otherwise you will receive a "Not logged on" exception.

TOraclSession.SetTransaction

Declaration

```

type TTransactionMode = (tmReadOnly, tmReadWrite, tmReadCommitted,
  tmSerializable);
procedure SetTransaction(const ATransactionMode: TTransactionMode);
```

Description

Sets the transaction mode to `tmReadOnly`, `tmReadWrite`, `tmReadCommitted` or `tmSerializable`. You can use this method to override the `IsolationLevel` property of the session for an individual transaction. After the transaction is ended by a `Commit` or `Rollback`, the mode of the next transaction will again be derived from the `IsolationLevel` of the session.

See also

`IsolationLevel`

TOracleSession.Share

Declaration

```
procedure Share(ToSession: TOracleSession);
```

Description

Shares the physical database connection of the session with another session. The TOracleSession instance for which this procedure is called must be connected. This procedure is useful if you are using dynamic link libraries and want to share a session between a host application and a DLL, if both are using Direct Oracle Access. For other scenarios you need to use the ExternalLDA or ExternalSVC properties.

Warning: You must log off the session to which the host session is shared, before the host session itself is logged off.

TOracleSession.SQLTrace

Declaration

```
type TSQLTraceOption = (stUnchanged, stTrue, stFalse);  
property SQLTrace: TSQLTraceOption;
```

Description

This property can enable or disable the SQL trace option for the session. When set to stTrue, you can analyze all executed SQL statements with Oracle's tkprof utility. It gives information about CPU time, elapsed time, disk I/O, etc of each statement. See the *"Oracle 7 Server Tuning"* manual for more information about tkprof.

When SQLTrace is set to stUnchanged, the initialization parameter SQL_TRACE of the Oracle instance determines if SQL trace is enabled or disabled.

TOracleSession.StatementCache

Declaration

```
property StatementCache: Boolean;
```

Description

This property indicates whether or not the session will use a client side statement cache. It requires Oracle Net 9.2 or later, and will be ignored for older Oracle versions. When enabled, this feature provides and manages a cache of statements for the session. On the server cursors are ready to be used without the need to parse the statement again, even if these cursors are closed by the application on the client.

The StatementCacheSize property determines the maximum number of cached statements. The least recently used statements will be removed from the cache when this maximum is reached. This ensures that only the most frequently used statements remain in the cache, and also ensures that the database will have a minimal number of open cursors.

If the StatementCache is enabled, the TOracleQuery.Optimize and TOracleDataSet.Optimize settings will be overruled.

TOracleSession.StatementCacheSize

Declaration

```
property StatementCacheSize: Integer;
```

Description

The StatementCacheSize property determines the maximum number of cached statements when the StatementCache is enabled for the session.

TOracleSession.ThreadSafe

Declaration

```
property ThreadSafe: Boolean;
```

Description

When true, multiple threads can execute queries and datasets in this session. This is achieved by serializing all thread access to the session. You can use threads to access different sessions without setting this property to True.

For more information, read the Multi-threaded applications section.

TOracleSession.UTL_File (Oracle 7.3 or higher)

Declaration

```
TUTL_File = class(TDBMSPackage)
  function FOpen(const Location: string; const Filename: string;
    const Open_Mode: string): TUTL_File_Type;
  function Is_Open(const AFile: TUTL_File_Type): Boolean;
  procedure FClose(var AFile: TUTL_File_Type);
  procedure FClose_All;
  procedure Get_Line(const AFile: TUTL_File_Type; out Buffer: string);
  procedure Put(const AFile: TUTL_File_Type; const Buffer: string);
  procedure New_Line(const AFile: TUTL_File_Type; const Lines:
    Cardinal);
  procedure Put_Line(const AFile: TUTL_File_Type; const Buffer:
    string);
  procedure Putf(const AFile: TUTL_File_Type; const Format: string;
    const Args: array of string);
  procedure FFlush(const AFile: TUTL_File_Type);
end;
property UTL_File: TUTL_File;
```

Description

The utl_file package has been provided since version 7.3 of the Oracle Server. It adds file input/output capabilities to PL/SQL for files that are located on the database server. Detailed information about this package can be found in the *Server Application Developer's Guide*.

Most of the functions and procedures in the utl_file package make use of a file handle, which is of type TUTL_File_Type. You can simply declare a variable of this type in your application to identify a file. The following example creates a file on the database server in UNIX directory

/transfer/output and writes the contents of a memo to it:

```
procedure TMainForm.ButtonClick(Sender: TObject);
var Handle: TUTL_File_type;
begin
    with MainSession do
        begin
            Handle := UTL_File.FOpen('/transfer/output', 'info.txt', 'w');
            for i := 0 to Memo.Lines.Count - 1 do
                UTL_File.Put_Line(Handle, Memo.Lines[i]);
            UTL_File.FClose(Handle);
        end;
    end;
```

Unlike other standard packages, utl_file raises a user-defined exception in case of error situations. These user-defined exceptions are translated to a specific exception type: EUTL_File_Error. This exception contains an Error property that can have the following values:

ufInvalidPath	File location or filename is invalid
ufInvalidMode	Open_Mode parameter in FOpen is invalid
ufInvalidFilehandle	File handle is invalid
ufInvalidOperation	File cannot be opened or operated on as requested
ufReadError	OS error occurred during read operation
ufWriteError	OS error occurred during write operation
ufInternalError	Unspecified error in PL/SQL

By creating an exception block you can simply test for specific error situations:

```
procedure TMainForm.ButtonClick(Sender: TObject);
var Handle: TUTL_File_type;
begin
    with MainSession do
        try
            Handle := UTL_File.FOpen('/transfer/output', 'info.txt', 'w');
            for i := 0 to Memo.Lines.Count - 1 do
                UTL_File.Put_Line(Handle, Memo.Lines[i]);
            UTL_File.FClose(Handle);
        except
            on E: EUTL_File_Error do
                begin
                    if E.Error = ufInvalidPath then
                        ShowMessage('/transfer/output is not a valid file location');
                    else
                        ShowMessage(E.Message);
                    end;
                end;
        end;
    end;
```

TOracleLogon component

Unit

Oracle

Description

This component provides a standard logon dialog and a set password dialog. When calling the Execute method, the Logon properties of the Session are copied into the dialog as default. The SetPassword method can be used to allow a user to set his or her password.

You can control the behavior and appearance of the logon dialog through the Options property, such as the presence of a database list and a logon history.

The logon dialog automatically handles Oracle8's password expiration feature if Net8 is installed. On SQL*Net 2 an expired password cannot be changed.

You can translate the texts of the logon dialog by setting the logon text string constants.

TOracleLogon reference

This chapter describes all properties, methods and events of the TOracleLogon component.

TOracleLogon.AliasDropDownCount

Declaration

```
property AliasDropDownCount: Integer;
```

Description

Determines how many items are visible in the combo box list with database names. This is only useful if the IdDatabaseList option is enabled.

TOracleLogon.Caption

Declaration

```
property Caption: string;
```

Description

The caption of the logon dialog window. If you leave this property empty, the default caption will be 'Oracle Logon', unless you have changed the default messages.

TOracleLogon.Execute

Declaration

```
function Execute: Boolean;
```

Description

Executes the logon dialog, returns False if the user only pressed the cancel button. You can examine the Connected property of the Session to check if the logon was successful.

TOracleLogon.HistoryIniFile

Declaration

```
property HistoryIniFile: string;
```

Description

Determines the name of the ini-file where the logon information is stored when the IdLogonHistory option is enabled. If you have also defined the HistoryRegSection property, this will take precedence.

TOracleLogon.HistoryRegSection

Declaration

```
property HistoryRegSection: string;
```

Description

Determines the registry section in the hkey_current_user registry where the logon information is stored when the IdLogonHistory option is enabled. A typical value for this property would be Software\MyCompany\MyProduct\LogonHistory.

TOracleLogon.HistorySize

Declaration

```
property HistorySize: Integer;
```

Description

Determines how many entries will be saved for the logon history when the IdLogonHistory option is enabled. The least recently used entry will be removed when this limit is reached.

TOracleLogon.HistoryWithPassword

Declaration

```
property HistoryWithPassword: Boolean;
```

Description

Determines if the password is stored for a logon history entry in the registry when the IdLogonHistory option is enabled.

If you set this property to False, the password must be entered when a logon entry is selected by an end-user. Only the username and database will be recalled.

If you set this property to True, an encrypted password will be stored in the registry for a logon history entry. The username, password and database will be recalled, and the end-user merely needs to press the OK button.

TOracleLogon.Options

Declaration

```
type TLogonOption = (ldAuto, ldDatabase, ldDatabaseList,  
    ldLogonHistory);  
type TLogonOptions = set of TLogonOption;  
property Options: TLogonOptions;
```

Description

The logon options allow you to control the following behavior of the logon dialog:

- ♦ IdAuto Causes the execute procedure to log on without showing a dialog if both username and password are specified for the Session.

- ♦ **IdConnectAs** Displays a dropdown list that allows you to connect as SYSDBA or SYSOPER. The list will only be visible when using Net8.
- ♦ **IdDatabase** Allows the user to enter a database name in the logon dialog.
- ♦ **IdDatabaseList** Displays a combo box list with database names. These database names are extracted from the SQL*Net / Net8 administration file tnsnames.ora, and can programmatically be accessed through the OracleAliasList. The size of the list is controlled by the AliasDropDownCount property.
- ♦ **IdLogonHistory** Displays a dropdown list with previously entered logon information, so that an end-user can quickly connect to a database. This logon history is stored in the hkey_current_user registry or in an ini-file. This option can further be controlled by the HistoryRegSection, HistoryIniFile, HistorySize and HistoryWithPassword properties.

TOracleLogon.Picture

Declaration

```
property Picture: TPicture;
```

Description

An optional picture that will be displayed in the logon dialog window.

TOracleLogon.Retries

Declaration

```
property Retries: word;
```

Description

Determines how many times that a user can retry to specify a username and password.

TOracleLogon.Session

Declaration

```
property Session: TOracleSession;
```

Description

The session which will be logged on.

TOracleLogon.SetPassword

Declaration

```
function SetPassword: Boolean;
```

Description

Executes a change-password dialog and changes the password of the user in the database of the session. This function returns true if the password is changed successfully.

TOracleQuery component

Unit

Oracle

Description

You use the TOracleQuery component to execute SQL statements or PL/SQL blocks. Set the Session property to define the TOracleSession in which the query will execute. The text of the query can be set in the SQL property and executed with Execute. If the query is a select statement, you can access the fields of a record with the Field method, and call Next to retrieve additional records until Eof indicates that no more records are available. Whenever an error occurs during the execution of a query, an EOracleError exception is raised.

The Variables property allows you to declare input/output variables for the query at design or run time. To assign a value to a variable at run time, use SetVariable. To use PL/SQL Tables or perform array DML, you can pass an array of values to this method. To get the value of an output variable, use GetVariable.

TOracleQuery can execute any SQL statement:

- ♦ Data Manipulation Language (select, update, insert, etc.)
- ♦ Data Definition Language (create table, create procedure, grant role, etc.)
- ♦ Transaction Control (commit, rollback, savepoint, etc.)
- ♦ Session Control (alter session, set role)
- ♦ System Control (alter system)

Besides that, it allows you to execute PL/SQL blocks on the server, enabling you to partition your client/server application in the best possible way. A single PL/SQL block can do a lot of work on the server in only one network roundtrip, which could otherwise generate a lot of network traffic. A PL/SQL Block is also the basis for stored procedure calls.

A TOracleQuery can also be defined as a cursor variable. In this case, the SQL property can be empty as the cursor on the server defines the actual select statement.

Example - Selecting data

This example selects all departments from the dept table. It uses a TOracleQuery with a simple select statement. After calling the Execute method, the first row is automatically fetched and accessible by using the Field methods. By calling the Next method until Eof, you can retrieve all records. To fill a TStringGrid, the following loop could be used:

```
// SelectQuery.SQL = select * from dept
//                      order by deptno
with SelectQuery do
try
  Execute;
  Row := 1;
  // Fill the grid
  while not Eof do
  begin
    DeptGrid.Cells[0, Row] := Field('DEPTNO');
    DeptGrid.Cells[1, Row] := Field('DNAME');
    DeptGrid.Cells[2, Row] := Field('LOC');
    Inc(Row);
    Next;
  end;
except
  on E:EOracleError do ShowMessage(E.Message);
end;
```

Remember that the Field function returns a variant of the type that resembles the type in the database:

Varchar	String
Number(s, p)	Integer if scale and precision allow it, otherwise double
Date	TDateTime
Char	String
Rowid	String
Raw	String, converted to hex format
Long	String
Long raw	Variant array of bytes

The beauty of variants is that Delphi will try to convert these to the datatype needed in your program. Therefore you don't need to typecast the integer expression Field('DEPTNO') to a string. For more information on variants read the appropriate section in the Object Pascal Language Guide.

Example - Using variables

The following example uses a PL/SQL block with input and output variables. When an employee is inserted into the emp table, the name and salary are passed to the query as input variables by using SetVariable. The query returns the number of the employee (empno) that was generated by an Oracle sequence, which is retrieved by calling GetVariable.

```

// InsertQuery.SQL =
// begin
//   select empsequence.nextval into :empno from dual;
//   insert into emp (empno, ename, sal)
//   values (:empno, :ename, :sal);
//   commit;
// end;
try
  with InsertQuery do
    begin
      SetVariable('ENAME', Emp.ENAME);
      SetVariable('SAL', Emp.Sal);
      Execute;
      Emp.EmpNo := GetVariable('EMPNO');
    end;
  except
    on E:EOracleError do ShowMessage(E.Message);
  end;

```

The easiest way to use variables is to declare them at design time through the Variables property of the TOracleQuery component. It is also possible to declare them at run time by using DeleteVariables to dispose of existing variables and DeclareVariable to declare new ones:

```

with InsertQuery do
  begin
    DeleteVariables;
    DeclareVariable('ENAME', otString);
    DeclareVariable('SAL', otFloat);
    DeclareVariable('EMPNO', otInteger);
    ...
  end;

```

Example - Long & Long Raw

These two Oracle data types can get really big (up to 2GB). Therefore, a few restrictions have been made regarding long values:

- Unlike other query-fields, long values are not buffered. Every time the Field function of a long value is called, the value is fetched from the database. On the other hand, as long as you do not call the field function, the value is not fetched.
- Whenever a query detects a long column in the select-list, only 1 record is buffered, regardless of the query's ReadBuffer property.
- Output variables of a long data type are not really supported. Use a select statement whenever possible. If you must use a long output variable, the maximum size it can contain is equal to the size it has on input, so it is in fact an input/output variable. If a longer value is assigned to the variable, you will get an "ORA-06502: PL/SQL: numeric or value error", so you'll have to be sure that the allocated size is enough.

Long Raws are handled as strings or zero based variant arrays of bytes. To access the byte-array with the highest possible performance, make use of the VarArrayLock and VarArrayUnlock functions. To check for the size of the value, make use of

VarArrayHighBound. The next example fetches a picture, determines the size, and saves the binary data to disk:

```
// SelectEmpPictureQuery.SQL =
// select picture from emp
// where empno = :empno
try
  with SelectEmpPictureQuery do
    begin
      SetVariable('EMPNO', Emp.EmpNo);
      Execute;
      Picture := Field('PICTURE');
      Size    := VarArrayHighBound(Picture, 1) + 1;
      Ptr     := VarArrayLock(Picture);
      WriteFile('Employee.gif', Ptr^, Size);
      VarArrayUnlock(Picture);
    end;
except
  on E:EOraclError do ShowMessage(E.Message);
end;
```

Two low-level methods are provided to get easier and faster access to Long and Long Raw data. The Field and SetVariable methods provide a consistent access to all data types, but can lead to some overhead in memory usage, memory movement, and database access.

If you know the internal structure of a Long or Long Raw, you may also exactly know its size and can fetch exactly what you need with a minimum of network roundtrips. The GetLongField method can help you with this:

```
GetLongField(FieldId: Integer; Buffer: Pointer; Offset, Length:
  Integer): Integer
```

If for example you know that a Long Raw column is a 16-color bitmap, you know the width and height is stored at positions 18 and 22. If you first fetch these two integers, you can determine the size of the bitmap and fetch the rest:

```
var wh: TPoint;
    Size: Integer;
    Bitmap: Pointer;
begin
  Query.GetLongField(BmpField, @wh, 18, SizeOf(wh));
  Size := ((wh.x * wh.y) div 2) + 70;
  GetMem(Bitmap, Size);
  Query.GetLongField(BmpField, Bitmap, 0, Size);
  ...
end;
```

If you use the SetVariable method to set the value of a Long or Long Raw variable, it is necessary to pass a string (Long or Long Raw) or variant array of bytes (Long Raw) for the value. It is probably more efficient to pass a pointer to the data that is to be used, instead of converting and copying large amounts of memory. The SetLongVariable method allows you to do just that:

```
SetLongVariable(Name: String; Buffer: Pointer; Length: Integer)
```

The memory pointed to by the Buffer pointer is not copied, so it must remain valid until the query is executed. The following example sets a variable to some recorded wav file in memory and inserts a new record:

```
var WavBuffer: Pointer;  
    WavLength: Integer;  
begin  
    RecordWavFile(WavBuffer, WavLength);  
    Query.SetVariable('wavname', WavName.Text);  
    Query.SetLongVariable('wavfile', WavBuffer, WavLength);  
    Query.Execute;  
    FreeMem(WavBuffer, WavLength);  
end;
```

Note: Due to the low-level nature of the GetLongField and SetLongVariable methods, they do not convert between LF and CR/LF combinations for Long values, unlike the Field and SetVariable methods. For Long Raw values this is not an issue.

Example - Array DML

Array DML is an Oracle feature that allows an application to insert, update or delete multiple records with one single statement in one single network roundtrip. This can increase performance significantly for batch processing. In a WAN configuration, dramatic performance gains can be achieved.

To make use of array DML, you need to supply arrays of variant values to the SetVariable method of a TOracleQuery. All arrays should be of equal length. The following example inserts the four well-known departments into the dept table:

```

var deptno, dname, loc: variant;
begin
  deptno := VarArrayCreate([0, 3], varVariant);
  dname  := VarArrayCreate([0, 3], varVariant);
  loc    := VarArrayCreate([0, 3], varVariant);
  deptno[0] := 10;
  dname[0]  := 'Accounting';
  loc[0]    := 'New York';
  deptno[1] := 20;
  dname[1]  := 'Research';
  loc[1]    := 'Dallas';
  deptno[2] := 30;
  dname[2]  := 'Sales';
  loc[2]    := 'Chicago';
  deptno[3] := 40;
  dname[3]  := 'Operations';
  loc[3]    := 'Boston';
  // insert into dept (deptno, dname, loc) values (:deptno, :dname,
  :loc)
  with InsDeptQuery do
  begin
    SetVariable('deptno', deptno);
    SetVariable('dname', dname);
    SetVariable('loc', loc);
    Execute;
  end;
end;

```

Error handling may require some special attention when using array DML. If an exception occurs, you will usually need to know which record caused it. The `OnArrayError` event can be used to handle the errors of each individual record:

```

procedure TForm1.InsDeptQueryOnArrayError(Sender: TOracleQuery;
  Index: Integer; ErrorCode: Integer; const ErrorMessage: string;
  var Continue: Boolean);
var dname: Variant;
begin
  dname := Sender.GetVariable('dname');
  ShowMessage('Error inserting department ' + dname[Index] + #13#10 +
    ErrorMessage);
end;

```

Use the zero-based `Index` parameter to access the record in the array that caused the error. You can use the `Continue` parameter to cancel or continue (default) the processing of the remainder of the array.

The `Execute` method will automatically execute all records in the declared array. If you need to process only a certain part of the array, you can use the `ExecuteArray` method:

```

function TOracleQuery.ExecuteArray(Index, Count: Integer): Integer;

```

`Index` is the zero-based starting position in the array, and `Count` is the number of records to process. The result indicates the number of records that were successfully processed.

Example - Cursor variables

Since Version 7.2 Oracle supports cursor variables. These variables are a reference to a cursor for a select statement that is defined and opened on the server. They offer the following advantages:

- Cursors are defined and maintained on a central point on the server
- An end-user needs only execute privileges on the procedure, not on the underlying objects of the cursors

Using a cursor variable as a TOracleQuery

Because a cursor variable is equivalent to a TOracleQuery with a select statement, DOA implements the cursor variable type as a TOracleQuery. To use a cursor variable, you need at least two TOracleQuery components: one with a PL/SQL block to call the procedure that opens the cursor, and one for the cursor itself:

```
begin
  with Query1 do
    begin
      Clear;
      SQL.Add('begin');
      SQL.Add('  employee.opencursor(:p_empcursor, :p_order)');
      SQL.Add('end;');
      DeclareVariable('p_empcursor', otCursor);
      DeclareVariable('p_order', otString);
      SetComplexVariable('p_empcursor', CursorQuery);
      SetVariable('p_order', 'ename');
      Execute;
    end;
  with CursorQuery do
    begin
      Execute;
      while not Eof do
        begin
          Memo.Items.Add(Field('ename'));
          Next;
        end;
      end;
    end;
  end;
```

The packaged procedure `employee.opencursor` might look like this:

```

type t_empcursor is ref cursor return emp%rowtype;

procedure getcursor(p_empcursor in out t_empcursor, p_order in
    varchar2) is
begin
    if p_order = 'ename' then
        open p_empcursor for select * from emp order by ename;
    elsif p_order = 'empno'
        open p_empcursor for select * from emp order by empno;
    else
        open p_empcursor for select * from emp;
    end if;
end;

```

In this example, Query1 calls the packaged function `employee.opencursor` to open a cursor to select all employees in a certain order. The `CursorQuery` is assigned to the `p_empcursor` variable. You need to use the `SetComplexVariable` method for this. Next, all rows are fetched and the employee names are displayed in a memo.

Using a cursor variable in a TOracleDataSet

To create a `TOracleDataSet` based on a cursor variable, enter a PL/SQL block with a call to the procedure that opens a cursor in the SQL property, for example:

```

begin
    employee.opencursor(:p_empcursor, :p_order);
end;

```

If you declare the `:p_empcursor` variable as `otCursor`, the dataset will detect the cursor variable, and will retrieve the records from the cursor after executing the PL/SQL block when the dataset is opened or refreshed.

To make the results of a cursor variable in a dataset updateable, you must include the `rowid` in the cursor, just like with the select statement of a normal dataset. To do this, you can define a cursor type and procedure in a package as follows:

```

cursor empcursor is select emp.*, rowid from emp;
type t_empcursor is ref cursor return empcursor%rowtype;
procedure opencursor(p_empcursor in out t_empcursor, p_order in
    varchar2);

```

Now that the `rowid` is included, the dataset can use it to lock, refresh, update or delete records. You also need to specify the updating table in the `UpdatingTable` property of the dataset, in this case: `emp`. Normally the dataset would determine the updating table by inspecting the select statement in the SQL property, but this is invisible in case of a cursor variable.

PL/SQL Tables

PL/SQL Tables are array-like variables that can be used in PL/SQL. They can be declared like this:


```
type <type_name> is table of <data_type> index by binary_integer;
```

PL/SQL Tables can be used in PL/SQL blocks where they can be passed as parameters to stored functions or procedures. This allows you to transfer large amounts of information in a single call. In a client/server situation, this can reduce network traffic significantly. PL/SQL Tables can be input, output and input/output variables, and can grow and shrink dynamically as needed.

Declaring a PL/SQL Table

To declare a variable as a PL/SQL Table in a TOracleQuery or TOracleDataSet, check the PL/SQL Table checkbox in the variables property editor. This checkbox is only enabled if the data type of the variable is string, integer, float or date. PL/SQL Tables of other data types are currently not supported.

After this, you can define the maximum size of the table. Even though a PL/SQL Table can grow unlimited on the server, on the client it has a predefined maximum size beyond which it cannot grow. If the variable is a PL/SQL Table of strings, you must also define the maximum string size.

The amount of memory that a PL/SQL Table can occupy is limited to 32512 bytes for Oracle7. For Oracle8 there is no practical limit. The variables property editor shows the memory that the PL/SQL Table occupies. When it passes the Oracle7 limit, it will be displayed in red.

You can also declare a PL/SQL Table variable at runtime, which requires two steps. First you must call `DeclareVariable` as usual, followed by a call to `DimPLSQLTable`.

For PL/SQL Tables that are used as parameters in a call to a procedure or function in a package generated through the Package Wizard, you can use the `TPLSQLTable` object instead.

Using a PL/SQL Table

Assigning values to a PL/SQL Table is very similar to array DML. You need to declare an array of variants, and use this array with the `SetVariable` method. The following example assigns a value to a PL/SQL Table named 'Numbers':

```
Table := VarArrayCreate([0, 2], varVariant);
Table[0] := 1.5;
Table[1] := 3.0;
Table[2] := 4.0;
MyQuery.SetVariable('Numbers', Table);
```

When a PL/SQL Table is an input/output or output variable, you need to retrieve the modified table after executing a PL/SQL block. You can do so by using the `GetVariable` method. It returns an array of variants with the same low bound that it had on input. When the PL/SQL Table is an output variable (in other words, it has not been assigned a value before the PL/SQL block was executed), the array will be zero-based.

PL/SQL Table example

The following example shows how to use a stored procedure that deletes employee records based on a PL/SQL Table with employee numbers. As an output PL/SQL Table, it returns each error message that occurred when deleting a record. It also returns the number of successfully deleted records.

Stored procedure definition

Package Employee defines two types for the PL/SQL Tables with employee numbers and error messages. The DeleteEmployees procedure loops through the employee numbers and attempts to delete these records. In case of an error, the error message table is updated.

```

type t_EmpNoTable is table of emp.empno%type index by binary_integer;
type t_ErrorTable is table of varchar2(200) index by binary_integer;

procedure DeleteEmployees(p_EmpNoTable in t_EmpNoTable,
                          p_ErrorTable out t_ErrorTable,
                          p_Deleted out integer) is
    l_Deleted integer;
    i integer;
begin
    i := 0;
    l_Deleted := 0;
    loop
        begin
            i := i + 1;
            delete emp where empno = p_EmpNoTable(i);
            p_ErrorTable(i) := 'Okay';
            l_Deleted := l_Deleted + 1;
        exception
            -- End of PL/SQL Table
            when no_data_found then exit;
            -- Error deleting record
            when others then p_ErrorTable(i) := sqlerrm;
        end;
    end loop;
    p_Deleted := l_Deleted;
end;
```

PL/SQL block to execute the procedure

The PL/SQL block that we need for a TOracleQuery merely contains a call to the DeleteEmployees packaged procedure. The p_EmpNoTable variable is declared as a PL/SQL Table of integers, p_ErrorTable is a PL/SQL Table of strings, and p_Deleted is an integer.

```

begin
    Employee.DeleteEmployees(:p_EmpNoTable, :p_ErrorTable, :p_Deleted);
end;
```

Delphi code to delete employees

The following Delphi code tries to delete employees 7839, 7369 and 7788. After executing the procedure, it shows how many records were successfully deleted and what errors occurred for each employee.

```
with DeleteEmployeesQuery do
begin
    EmpNoTable := VarArrayCreate([0, 2], varVariant);
    EmpNoTable[0] := 7839;
    EmpNoTable[1] := 7369;
    EmpNoTable[2] := 7788;
    SetVariable('p_EmpNoTable', EmpNoTable);
    Execute;
    Session.Commit;
    ErrorTable := GetVariable('p_ErrorTable');
    s := string(GetVariable('p_Deleted')) + ' record(s) deleted'#13#10;
    for i := 0 to 2 do
        s := s + string(EmpNoTable[i]) + ' - ' + string(ErrorTable[i]) +
            #13#10;
    ShowMessage(s);
end;
```

TOracleQuery reference

This chapter describes all properties, methods and events of the TOracleQuery component.

TOracleQuery.AfterQuery

Declaration

```
type TOracleQueryEvent = procedure(Sender: TOracleQuery) of Object;  
property AfterQuery: TOracleQueryEvent;
```

Description

Triggered when the query ends. For a select statement, this is when Eof is reached. For all other statements, this is immediately after Execute ends.

See also

BeforeQuery

TOracleQuery.BeforeQuery

Declaration

```
type TOracleQueryEvent = procedure(Sender: TOracleQuery) of Object;  
property BeforeQuery: TOracleQueryEvent;
```

Description

Triggered when the query begins, immediately before Execute starts.

See also

AfterQuery

TOracleQuery.BreakThread

Declaration

```
procedure BreakThread;
```

Description

Breaks the current operation in Threaded mode. This can either lead to an "ORA-01013: user requested cancel of current operation" if the query is running on the server at the time, or can simply lead to just an OnThreadFinished event.

See also

ThreadIsRunning

State

TOracleQuery.Cancel

Declaration

```
procedure Cancel;
```

Description

You can use this procedure to cancel a select statement. Eof will become True and no more records can be fetched. Note that this procedure does not break execution of a running query, which can be accomplished by calling TOracleSession.BreakExecution.

TOracleQuery.Clear

Declaration

```
procedure Clear;
```

Description

Clears the SQL statement and deletes all declared Variables.

TOracleQuery.ClearVariables

Declaration

```
procedure ClearVariables;
```

Description

Assigns nulls to all declared variables.

See also

Variables

DeleteVariables

SetVariable

TOracleQuery.Close

Declaration

```
procedure Close;
```

Description

Closes the query, freeing resources on the server. If you don't have many queries, you don't need to call close. Leaving them open increases performance.

TOracleQuery.Cursor

Declaration

property Cursor: TCursor;

Description

Determines the shape of the mouse cursor while executing the query. Only crDefault, crHourGlass and crSQLWait are useful here.

TOracleQuery.Debug

Declaration

property Debug: Boolean;

Description

When set to true the SQL statement and all variable values will be displayed when executed.

TOracleQuery.DeclareAndSet

Declaration

procedure DeclareAndSet(Name: **string**; Type: Integer; Value: Variant);

Description

Declares a variable of a certain name and type, and sets its initial value. This procedure is particularly handy if you have to declare and set many variables at runtime.

TOracleQuery.DeclareVariable

Declaration

procedure DeclareVariable(Name: **string**; Type: Integer);

Description

Declares a variable of a certain name and type.

See also

Variables

DeleteVariable

DeleteVariables

TOracleQuery.DeleteVariable

Declaration

```
procedure DeleteVariable(AName: string);
```

Description

Deletes the specified variable.

See also

Variables

DeclareVariable

TOracleQuery.DeleteVariables

Declaration

```
procedure DeleteVariables;
```

Description

Deletes all declared variables.

See also

Variables

ClearVariables

DeclareVariable

TOracleQuery.Describe

Declaration

```
procedure Describe;
```

Description

Describe enables you to get the field descriptions of a query without actually executing it.

TOracleQuery.DimPLSQLTable

Declaration

```
procedure DimPLSQLTable(Name: string; TableSize, StringSize:  
    Integer);
```

Description

Defines a previously declared variable as a PL/SQL Table. The StringSize parameter must be 0 if the data type of the variable is anything other than string. The following example declares a PL/SQL Table of floats, with a maximum size of 100:

```
with MyQuery do
begin
  DeclareVariable('Numbers', otFloat);
  DimPLSQLTable('Numbers', 100, 0);
end;
```

For more information about PL/SQL Tables, see the PL/SQL Tables section.

TOracleQuery.Eof

Declaration

```
function Eof: Boolean;
```

Description

Indicates if there are any rows left to fetch. Eof will become True under the following conditions:

- ♦ After calling Next when positioned on the last row of the result set.
- ♦ After calling Execute for a select statement, if no records were selected.
- ♦ After calling Execute, for all non-select statements.
- ♦ After calling Prior when positioned on the first row of the result set.
- ♦ After calling MoveTo or MoveBy for a row beyond the first or last row in the result set.

TOracleQuery.ErrorLine

Declaration

```
function ErrorLine: Integer;
```

Description

In case of a parse error, ErrorLine indicates the line where the error occurred (one-based).

Warning

This property is not available in Net8 8.0.x (on 8.1 it is present again). If your application requires it, it may be a good idea to set the UseOCI7 preference of the session to True. This way, your application will function properly on SQL*Net and Net8 clients. Oracle8 features will not be available however.

See also

ErrorPosition

TOracleQuery.ErrorPosition

Declaration

```
function ErrorPosition: Integer;
```

Description

In case of a parse error, ErrorPosition indicates the character position on the ErrorLine where the error occurred (one-based).

Warning

This property is not available in Net8 8.0.x (on 8.1 it is present again). If your application requires it, it may be a good idea to set the UseOCI7 preference of the session to True. This way, your application will function properly on SQL*Net and Net8 clients. Oracle8 features will not be available however.

See also

ErrorLine

TOracleQuery.Execute

Declaration

```
procedure Execute;
```

Description

Executes the query. If the query is a select statement the first record is immediately fetched and can be accessed through the Field methods.

See also

SQL

Next

Eof

TOracleQuery.ExecuteArray

Declaration

```
function ExecuteArray(Index, Count: Integer): Integer;
```

Description

In case of an array DML statement, you can use this method if you need to process only a certain part of the array. Index is the zero-based starting position in the array, and Count is the number of records to process. The result indicates the number of records that were successfully processed.

See also

Execute

OnArrayError

TOracleQuery.Field

Declaration

```
function Field(FieldId: Variant): Variant;
```

Description

Returns the field value as a variant for simple data types. The FieldId can be specified as the name or the index (zero based) of the field. The second is slightly faster, and can be used if many records are retrieved with many columns. If the field is null, the NullValue property of the session determines if it will be returned as a Null or Unassigned variant.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

If the field is a Long or Long Raw, some restrictions apply.

For Oracle8 complex fields other than LOB's (otCursor, otObject or otReference), you cannot use the Field method. Instead, you must use a specific method to obtain the corresponding object:

Field Type	Method	Object Type
Cursor	GetCursor	TOracleQuery
LOB	LOBField	TLOBLocator
Object	ObjField	TOracleObject
Reference	RefField	TOracleReference

See also

- FieldName
- FieldIndex
- FieldSize
- FieldType
- FieldIsNull
- FieldOptional
- FieldCount

TOracleQuery.FieldAsDate

Declaration

```
function FieldAsDate(FieldId: Integer): TDateTime;
```

Description

Returns a field value as a date (converted if necessary). If you don't want to use variants, use the FieldAs... functions instead of the Field function.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

See also

- FieldAsString

FieldAsInteger

FieldAsFloat

TOracleQuery.FieldAsFloat

Declaration

```
function FieldAsFloat(FieldId: Integer): Double;
```

Description

Returns a field value as a float (converted if necessary). If you don't want to use variants, use the FieldAs... functions instead of the Field function.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

See also

FieldAsString

FieldAsInteger

FieldAsDate

TOracleQuery.FieldAsInteger

Declaration

```
function FieldAsInteger(FieldId: Integer): Integer;
```

Description

Returns a field value as an integer (converted if necessary). If you don't want to use variants, use the FieldAs... functions instead of the Field function.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

See also

FieldAsString

FieldAsFloat

FieldAsDate

TOracleQuery.FieldAsString

Declaration

```
function FieldAsString(FieldId: Integer): string;
```

Description

Returns a field value as a string (converted if necessary). This function raises an exception when called for object fields, except when the object is a SYS.XMLTYPE, in which case the XML text is returned.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

See also

FieldAsInteger

FieldAsFloat

FieldAsDate

TOracleQuery.FieldCount

Declaration

```
function FieldCount: Integer;
```

Description

Returns the number of fields in the select list of the query.

See also

Field

FieldName

TOracleQuery.FieldIndex

Declaration

```
function FieldIndex(FieldId: string): Integer;
```

Description

Returns the index of the specified field.

See also

Field

FieldName

TOracleQuery.FieldsIsNull

Declaration

```
function FieldsIsNull(FieldId: Integer): Boolean;
```

Description

Indicates if the specified field contains a null value. You can also examine the value of the Field function, but FieldsIsNull is slightly faster and is independent of the setting of the NullValue property of the Session.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

See also

Field

FieldIndex

FieldOptional

TOracleQuery.FieldName

Declaration

```
function FieldName(FieldId: Integer): string;
```

Description

Returns the name of the specified field.

See also

Field

FieldIndex

TOracleQuery.FieldOptional

Declaration

```
function FieldOptional(FieldId: Integer): Boolean;
```

Description

Indicates if the specified field can contain null values.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

See also

Field

FieldIndex

FieldIsNull

TOracleQuery.FieldPrecision

Declaration

```
function FieldPrecision(FieldId: Integer): Integer;
```

Description

Returns the precision of the specified field. Only useful for number fields.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

See also

Field

FieldIndex

FieldType

FieldScale

TOracleQuery.FieldScale

Declaration

```
function FieldScale(FieldId: Integer): Integer;
```

Description

Returns the scale of the specified field. Only useful for number fields.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

See also

Field

FieldIndex

FieldType

FieldPrecision

TOracleQuery.FieldSize

Declaration

```
function FieldSize(FieldId: Integer): Integer;
```

Description

Returns the size of the specified field. Only useful for varchar fields.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

See also

Field

FieldIndex

FieldType

FieldScale

FieldPrecision

TOracleQuery.FieldType

Declaration

```
function FieldType(FieldId: Integer): Integer;
```

Description

Returns the type of the specified field (otInteger, otString, and so on).

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

See also

Field

FieldIndex

TOracleQuery.First

Declaration

```
procedure First;
```

Description

Closes the query, freeing

TOracleQuery.FunctionType

Declaration

```
function FunctionType: Integer;
```

Description

The function type of the last Execute. This number corresponds to the numbers you can find in the *"Programmer's Guide to the Oracle Call Interface"*

TOracleQuery.GetComplexVariable

Declaration

```
function GetComplexVariable(Name: string): TObject;
```

Description

Returns the object previously associated with the variable by using the SetComplexVariable method.

TOracleQuery.GetCursor

Declaration

```
function GetCursor(const FieldId: Variant): TOracleQuery;
```

Description

For cursor fields (for sub-queries such as nested tables), this function creates and returns a TOracleQuery object. After executing the returned sub-query you can select rows from it only once. You cannot re-execute it a second time, or use the GetCursor function to obtain the sub-query again.

You are responsible for freeing the returned TOracleQuery instance when it is no longer needed.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

TOracleQuery.GetLongField

Declaration

```
function GetLongField(FieldId: Integer; Buffer: Pointer; Offset,  
    Length: Integer): Integer;
```

Description

Low level function to retrieve <Length> bytes of the specified long or long raw field into a <Buffer>, starting at position <Offset>. The function returns the number of bytes that were actually retrieved. No LF => CR/LF conversion is performed for long fields.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

See also

SetLongVariable

TOracleQuery.GetVariable

Declaration

```
function GetVariable(Name: string): Variant;
```

Description

Retrieves the value of the specified variable after the query has been executed. In Delphi 4 and later you can also specify the variable by its zero-based index. If the variable is a Long or Long Raw, some restrictions apply.

See also

Variables

DeclareVariable

SetVariable

TOracleQuery.Last

Declaration

```
procedure Last;
```

Description

Fetches the last row for a select statement. This function can only be called for a Scrollable query.

TOracleQuery.LOBField

Declaration

```
function LOBField(const FieldId: Variant): TLOBLocator;
```

Description

For LOB fields, this function returns a TLOBLocator object that you can subsequently use to

access the LOB data.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

TOracleQuery.MoveBy

Declaration

```
procedure MoveBy(Distance: Integer);
```

Description

Fetches a row relative to the current ScrollPosition for a select statement. The Distance parameter indicates the number of records that should be moved. A negative number will move the ScrollPosition backwards. Moving beyond the first or last row will set Eof to True. This function can only be called for a Scrollable query.

TOracleQuery.MoveTo

Declaration

```
procedure MoveTo(Position: Integer);
```

Description

Moves to a specific row in the result set of a select statement. The Position parameter is the 1-based number of the row. Moving beyond the last row will set Eof to True. This function can only be called for a Scrollable query.

TOracleQuery.Next

Declaration

```
procedure Next;
```

Description

Fetches the next row for a select statement.

See also

Execute

Eof

TOracleQuery.ObjField

Declaration

```
function ObjField(const FieldId: Variant): TOracleObject;
```

Description

For object fields, this function returns a TOracleObject that you can subsequently use to access the object.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

TOracleQuery.OnArrayError

Declaration

```
type TOnArrayErrorEvent = procedure(Sender: TOracleQuery; Index:
    Integer; ErrorCode: Integer; const ErrorMessage: string; var
    Continue: Boolean) of Object;
property OnArrayError: TOnArrayErrorEvent;
```

Description

Triggered when an error occurs in an array DML statement. Index is the zero-based position of the record in the array that caused the error. The Continue parameter can be used to cancel or continue the processing of the remaining records.

See also

ExecuteArray

TOracleQuery.OnThreadError

Declaration

```
type TThreadErrorEvent = procedure(Sender: TOracleQuery; ErrorCode:
    Integer; const ErrorMessage: string) of Object;
property OnThreadError: TThreadErrorEvent;
```

Description

Fires when an exception occurs in Threaded mode. It has 2 parameters:

ErrorCode	Indicates the number of any Oracle error that might have occurred. If another exception occurred, ErrorCode will be 0.
ErrorMessage	The message of the exception.

See also

OnThreadFinished

TOracleQuery.OnThreadExecuted

Declaration

```
type TOracleQueryEvent = procedure(Sender: TOracleQuery) of Object;
property OnThreadExecuted: TOracleQueryEvent;
```

Description

If a query was executed in Threaded mode, this event will be fired when initial execution has finished, but before any records are fetched.

See also

OnThreadRecord

OnThreadFinished

OnThreadError

TOracleQuery.OnThreadFinished**Declaration**

```
type TOracleQueryEvent = procedure(Sender: TOracleQuery) of Object;  
property OnThreadFinished: TOracleQueryEvent;
```

Description

If a query has been executed in Threaded mode, this event will be fired after the query is executed and, if applicable, all records have been fetched. If an exception occurs this event is still called, so this is a good place to perform any clean-up code.

See also

OnThreadExecuted

OnThreadRecord

OnThreadError

TOracleQuery.OnThreadRecord**Declaration**

```
type TOracleQueryEvent = procedure(Sender: TOracleQuery) of Object;  
property OnThreadRecord: TOracleQueryEvent;
```

Description

If a query has been executed in Threaded mode, this event will be fired after each record that is fetched. After the last record has been fetched, the OnThreadFinished event will be fired.

See also

OnThreadExecuted

OnThreadFinished

OnThreadError

TOracleQuery.Optimize**Declaration**

```
property Optimize: Boolean;
```

Description

When set to True, queries will not unnecessarily re-parse SQL when repeatedly executed. Additional parse operations will only occur if you change the SQL text, add or remove variables, or change the types of existing variables between calls. This setting when increase performance when repeatedly executing the same SQL with different variable values.

If this property is set to False, the corresponding database cursor will be closed immediately after executing a non-select statement, or when Eof is reached for select statements. This can be useful if you want to minimize the number of open database cursors.

This property is overruled if you are using Oracle Net 9.2 and have enabled the StatementCache of the corresponding session. In this situation the cursor will be released into the cache after execution or after reaching Eof, regardless of the Optimize setting.

TOracleQuery.Prior

Declaration

```
procedure Prior;
```

Description

Fetches the prior row for a select statement. Calling Prior when positioned on the first row will set Eof to True. This function can only be called for a Scrollable query.

TOracleQuery.ReadBuffer

Declaration

```
property ReadBuffer: Integer;
```

Description

Number of rows that will be transferred across the network at once for select statements. This property can have a great impact on performance.

If the query contains a Long or Long Raw field, a ReadBuffer of 1 will be used.

TOracleQuery.RefField

Declaration

```
function RefField(const FieldId: Variant): TOracleReference;
```

Description

For reference fields, this function returns a TOracleReference that you can subsequently use to access the reference.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

TOracleQuery.ReturnCode

Declaration

```
function ReturnCode: Integer;
```

Description

The result of the last Execute. This number corresponds to the codes you can find in the "Oracle Server Messages Guide".

TOracleQuery.RowCount

Declaration

```
function RowCount: Integer;
```

Description

Returns the number of rows that have been fetched by the application for the query. Immediately after executing the query, RowCount will return 1 if any records are fetched or will return 0 if no records are fetched.

See also

RowsProcessed

TOracleQuery.RowId

Declaration

```
function RowId: string;
```

Description

Returns the hexadecimal notation (or 64-based on Oracle8) of the rowid of the last insert, update, delete or select for update statement. Note that RowId returns the value of the last affected row. In case of a select for update, the last affected row is the last row in the ReadBuffer.

TOracleQuery.RowsProcessed

Declaration

```
function RowsProcessed: Integer;
```

Description

Returns the number of rows processed by the query. Useful for select, update and delete statements. In case of a select statement, RowsProcessed increments by ReadBuffer, and indicates how many records have been transferred from the server.

See also

RowCount

TOracleQuery.Scrollable

Declaration

```
property Scrollable: Boolean;
```

Description

Oracle9 introduced the concept of scrollable queries. For a normal query you can only fetch the Next record, but a scrollable query allows you to navigate to any absolute (First, Last, MoveTo) or relative (Prior, MoveBy) position in the result set. This does come at a performance penalty, so you should preferably use a normal query if you do not need the

additional navigation functionality.

Note

This feature does not yet work properly on Oracle 9.0 and 9.2 due to Oracle Bugs 2286367 and 2478181, which may cause ORA-00600 exceptions.

TOracleQuery.ScrollPosition

Declaration

property ScrollPosition: Boolean;

Description

The 1-based scroll position within the result set for a Scrollable query.

TOracleQuery.Session

Declaration

property Session: TOracleSession;

Description

The session in which the query will execute.

TOracleQuery.SetComplexVariable

Declaration

procedure SetComplexVariable(Name: **string**; **const** Value: TObject);

Description

Complex variables (variables of type otCursor, otCLOB, otBLOB, otBFile, otTimestamp, otTimestampTZ, otTimestampLTZ, otObject or otReference) cannot be treated as simple string, numeric, date or binary values. So, rather than setting the value of a variable with SetVariable, you must use SetComplexVariable to associate a corresponding object instance with it:

Variable type	Object type
Cursor	TOracleQuery
LOB	TLOBLocator
Timestamp	TOracleTimestamp
Object	TOracleObject
Reference	TOracleReference

If the variable is an OUT or IN/OUT variable, the changes are immediately visible in the associated object after execution of the query. Therefore, you do not need to call GetComplexVariable for these variables after execution of the query. The following example creates a TLOBLocator object, associates it with variable 'lobvar', executes the query, and indicates if it returned a null LOB:

```

var
  LOB: TLOBLocator;
begin
  LOB := TLOBLocator.Create(Session, otBLOB);
  Query.SetComplexVariable('lobvar', LOB);
  Query.Execute;
  if LOB.IsNull then ShowMessage('Statement returned a null LOB');
  LOB.Free;
end;

```

Note

Timestamp variables can be accessed through `GetVariable` and `SetVariable` instead of the `SetComplexVariable` procedure. The values will be of the `TDateTime` data type though, so you are restricted to a maximum precision of a millisecond.

TOracleQuery.SetLongVariable

Declaration

```

procedure SetLongVariable(Name: string; Buffer: Pointer; Length:
  Integer);

```

Description

Low level procedure to set the value of the specified long or long raw variable to the address pointed to by <Buffer> with length <Length>. Memory is not copied to minimize overhead, so the address must remain valid until the query is executed. No CR/LF => LF conversion is performed for long variables.

See also

`GetLongField`

TOracleQuery.SetVariable

Declaration

```

procedure SetVariable(Name: string; Value: Variant);

```

Description

Sets the value of the specified variable. In Delphi 4 and later you can also specify the variable by its zero-based index. If the variable is a Long or Long Raw, some restrictions apply. The value can be an array of variants to perform array DML.

For complex variables (`otCursor`, `otCLOB`, `otBLOB`, `otBFile`, `otObject` or `otReference`), you must use `SetComplexVariable` instead of `SetVariable`. For Timestamp variables you can access the value as a `TDateTime` (maximum precision = 1 millisecond) or use the `SetComplexVariable` method to access it as a `TOracleTimestamp` (maximum precision = 1 nanosecond).

See also

`Variables`

`ClearVariables`

DeclareVariable

GetVariable

TOracleQuery.SQL

Declaration

```
property SQL: TString;
```

Description

The SQL text of the query. You can enter any valid SQL statement or PL/SQL block. When using variable names in this text, precede them with a colon and declare them in the Variables property.

When calling stored program units, you can place the call in a PL/SQL Block. For example, to call the 'get_parameter_value' function in the 'dbms_utility' package, you can use the following SQL:

```
begin
    :res := dbms_utility.get_parameter_value(parnam => :parnam,
                                             intval => :intval,
                                             strval => :strval);
end;
```

As you can see, parameters and return values are normal variables. You can pass parameters by name or by position, use constants instead of variables, or omit them in case they have an appropriate default value. In short, a stored program unit can be called in the same way as you would call it from PL/SQL on the server.

TOracleQuery.State

Declaration

```
type TQueryState = (qsIdle, qsExecuting, qsFetching);
function State: TQueryState;
```

Description

Indicates if the query is idle (qsIdle), executing the statement (qsExecuting) or fetching records (qsFetching).

See also

Threaded

TOracleQuery.StringFieldsOnly

Declaration

```
property StringFieldsOnly: Boolean;
```

Description

When defining fields for a select statement, the TOracleQuery component will use the appropriate data types. A varchar2 field is defined as string, a number field is defined as

integer or double, and a date is defined as `TDateTime`. When the `StringFieldsOnly` property is set to `True`, all fields will be defined as strings. Numbers and dates will be converted to strings on the server, as defined by the `NLS_LANG` settings of the current session.

TOracleQuery.SubstitutedSQL

Declaration

```
function SubstitutedSQL: string;
```

Description

If you are using substitution variables, you can retrieve the SQL text with substituted variables by calling the `SubstitutedSQL` function.

See also

Variables

TOracleQuery.Threaded

Declaration

```
property Threaded: Boolean;
```

Description

When the `Threaded` property of a `TOracleQuery` is set to `True`, all processing will be performed in a background thread, allowing your application to continue while the query is running on the database server. Special considerations need to be taken into account when creating multi-threaded applications.

When `Threaded` is set to `True`, you can execute the query and forget about it in the main thread of your application. Executing the statement and fetching records is performed in a background thread. When execution is finished, the `OnThreadExecuted` event is fired. Next, for select statements, the `OnThreadRecord` event is fired for each record that is fetched. Finally the `OnThreadFinished` event is fired. In case of an exception, the `OnThreadError` event will be fired.

If `ThreadSynchronized` is set to `True`, the `OnThread` events will be synchronized with the main thread of your application. In this case you don't need to be aware of any issues of multi-threaded application programming. When `ThreadSynchronized` is set to `False`, the main thread of your application and the thread events of the `TOracleQuery` run asynchronous.

In threaded mode the flow of your application is different than in non-threaded mode. Normally you would execute a query, fetch results until `Eof` is reached, and handle any exception in an exception handler. The following example fills a `TListBox` component with employee names in a background thread and enables the listbox when all employees are fetched:

```

procedure TMyForm.EmpButtonClick(Sender: TObject);
begin
    // Clear the list and disable it
    EmpList.Clear;
    EmpList.Enabled := False;
    // Select the employee names in a background thread
    EmpQuery.SQL.Text := 'select ename from emp order by ename';
    EmpQuery.Threaded := True;
    EmpQuery.Execute;
    // We're done, let the events handle it from here
end;

procedure TMyForm.EmpQueryThreadRecord(Sender: TOracleQuery);
begin
    // Add the fetched employee name to the list
    EmpList.Items.Add(Sender.Field('ename'));
end;

procedure TMyForm.EmpQueryThreadFinished(Sender: TOracleQuery);
begin
    // After fetching all employee names, enable the list
    EmpList.Enabled := True;
end;

```

See also

TOracleSession.ThreadSafe
 BreakThread
 ThreadIsRunning
 State

TOracleQuery.ThreadIsRunning**Declaration**

```
function ThreadIsRunning: Boolean;
```

Description

Indicates if the thread to which the query is associated is running. If this is the case you cannot execute another query yet.

See also

Threaded

TOracleQuery.ThreadSynchronized**Declaration**

```
property ThreadSynchronized: Boolean;
```

Description

If this property is set to True, all events associated with a TOracleQuery's Threaded mode will

be synchronized with the main thread of your application. If it is set to `False`, these events and the main thread of the application will run asynchronous.

See also

`OnThreadExecuted`

`OnThreadRecord`

`OnThreadFinished`

`OnThreadError`

TOracleQuery.TimestampField (Oracle8i only)

Declaration

```
function TimestampField(const FieldId: Variant): TOracleTimestamp;
```

Description

For Timestamp fields, this function returns a `TOracleTimestamp` object that you can subsequently use to access the individual timestamp properties.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

TOracleQuery.VariableCount

Declaration

```
function VariableCount: Integer;
```

Description

Returns the number of variables that are declared. Useful for iterating variables.

See also

`VariableName`

`VariableType`

`VariableIndex`

TOracleQuery.VariableIndex

Declaration

```
function VariableIndex(const AName: string): Integer;
```

Description

Use this function to determine the index of a variable by its name. The result is the zero based index of the variable. If the variable does not exist, -1 is returned.

See also

`VariableCount`

`VariableName`

VariableType

TOracleQuery.VariableName

Declaration

```
function VariableName(Index: Integer): string;
```

Description

Returns the name of the specified variable. Index is zero-based.

See also

VariableCount

VariableType

VariableIndex

TOracleQuery.Variables

Declaration

```
property Variables: TVariables;
```

Description

The variables property is only accessed at design time. It will invoke a property editor, which enables you to declare variables manually, or scan for variables in the SQL text. At run time, you access variables through the SetVariable, SetComplexVariable, GetVariable, GetComplexVariable, DeclareVariable and DimPLSQLTable methods.

Whenever you want to use a SQL statement that needs some input or output, you can use variables. In the SQL statement a variable is preceded by a colon. For example:

```
update emp
set sal = :new_sal
where empno = :empno_to_update
```

This update statement has two variables: new_sal and empno_to_update. One is needed to identify the employee, the other supplies the new value for sal. When you execute this query, you will get an error:

```
ORA-01008: not all variables bound
```

This is because Oracle has recognized the two variables in the SQL text, but finds no variables declared in the query that supply a value. To declare the variables, use the Variables property editor. New_sal should be declared as a Float, empno_to_update must be declared as an Integer. By declaring them at design time, you can assign a value to them at run time by using SetVariable:

```
Query.SetVariable('empno_to_update', 7894);
Query.SetVariable('new_sal', 3500.00);
Query.Execute;
```

If you declare variables that are not included in the SQL text (as :variable), you will get the following error:

```
ORA-01036: illegal variable name/number
```

The names of the variables and the number of variables in the SQL text must exactly correspond to the variables in the Variables property editor, otherwise you get ORA-01008 or ORA-01036.

Bind variables vs substitution variables

All variables except substitution variables are so called bind variables: the statement is sent to the server unchanged, accompanied by a binary representation of the bind variables. Bind variables have their limitations though, they can only be used where you could normally use a variable in PL/SQL: in where clauses, select lists, update statements, insert statements, PL/SQL statements, and so on. You cannot use a bind variable where its value could change the meaning of the statement (eg. to identify a table or column).

Substitution variables remove this limitation, as they are substituted by their values in the text of the statement before it is sent to the server. The following statement is only valid if the :table_name and :order_by_clause variables are substitution variables:

```
select * from :table_name
order by :order_by_clause
```

If the variables would be declared as bind variables of type string, the statement would be invalid, as you cannot use bind variables to identify table or column names.

If the value of the :table_name variable is 'emp' and the value of the :order_by_clause variable is 'ename', the actual statement sent to the server would be:

```
select * from emp
order by ename
```

Why not always use substitution variables? There are 2 very good reasons:

- Because the SQL text changes each time you change the value of a substitution variable, the server needs to parse the statement every time. Changing the value of a bind variable does not require a new parse operation and is therefore faster.
- For substitution variables you need take care of any conversion issues. Just think about quotes in strings, decimal points in numbers, date and time formats, and so on.

In short: always use bind variables, unless their limitations don't allow it.

Variable types

The following variable types can be declared. Between parenthesis you see the type identifiers you can use for the DeclareVariable procedure:

- Integer (otInteger)

A 32 bits integer, equivalent to a number(1) through number(9) value in Oracle

- Float (otFloat)

Floating point values, equivalent to any number value in Oracle. Note however that a

Delphi floating point number (double) has a precision of 15 digits, whereas an Oracle number can have a precision of up to 38 digits.

- ◆ String (otString)

Strings limited to 2000 (Oracle7) or 4000 (Oracle8) bytes. This variable type is equivalent to Oracle's varchar2 SQL variable.

- ◆ Date (otDate)

Date and date/time values. Note that a TDateTime in Delphi is more accurate than an Oracle date, as its time fraction include tenths of a second.

- ◆ Long (otLong)

A string of arbitrary size (2GB). Note that if you use a PL/SQL block in a TOracleQuery, all your variables become PL/SQL variables that are limited to 32KB! Some restrictions apply to using the Long and Long Raw data type, read the Long & Long Raw example for details.

To set the value of a Long variable, you can either use the SetVariable procedure or the SetLongVariable procedure. The last one gives you better performance.

- ◆ Long Raw (otLongRaw)

This binary data type can also be up to 2GB, just like the Long data type. Unlike a Long variable, Long Raw data is not subject to character set interpretation by SQL*Net or Net8. Data is passed to or from the server as it is.

To set the value of a Long Raw variable, you can pass a string or a variant array of bytes to SetVariable procedure. Use the SetLongVariable for optimal performance. Read the Long & Long Raw example for more details.

- ◆ Cursor (otCursor)

A cursor variable is a representation of a PL/SQL cursor variable. Instead of passing data to a cursor variable, you must associate a TOracleQuery instance to it by using the SetComplexVariable procedure. This TOracleQuery instance can be used to fetch records from the cursor after it has been opened in a stored procedure. Read the Cursor variables example for more details.

- ◆ CLOB (otCLOB)

A CLOB variable represents the Oracle8 CLOB data type. It can only be used when connected to Oracle8 through Net8. Instead of passing data to a CLOB variable, you must associate a TLOBLocator instance to it. After executing a statement that initializes the TLOBLocator, you can subsequently use it to read or write data to it.

- ◆ BLOB (otBLOB)

Similar to the CLOB variable type.

- ◆ BFile (otBFile)

Similar to the CLOB variable type.

- ◆ Reference (otReference)

A Reference variable represents a reference to an Oracle8 Object instance (or REF). It can only be used when connected to Oracle8 through Net8. You can associate a TOracleReference instance with the variable by using the SetComplexVariable procedure.

- Object (otObject)

An Object variable represents a an Oracle8 Object instance (or ADT). It can only be used when connected to Oracle8 through Net8. You can associate a TOracleObject instance with the variable by using the SetComplexVariable procedure.

- PL/SQL String (otPLSQLString)

The string variable mentioned above is a representation of the SQL varchar2 variable in Oracle: it is limited to 2000 (Oracle7) or 4000 (Oracle8) bytes. In PL/SQL you can use varchar2 variables of up to 32KB though. If you are using large PL/SQL strings you can use a PLSQL String variable which has a maximum size of 32512 bytes.

- Fixed length char (otChar)

When a varchar2 variable value is compared with a char column in a where clause, trailing spaces are significant. This can sometimes lead to unexpected results. To compare a char column you can use a variable of type char, which uses blank-padded comparison semantics.

- Substitution (otSubst)

Unlike all other variables, the substitution variable is replaced by its value in the SQL text before the statement is sent to the server. This way you can use the variable at any position in the statement, but you must take care of any conversion issues for strings, numbers and dates. The SubstitutedSQL function returns the SQL with all substitution variables replaced by their values.

- Timestamp (otTimestamp)

A Timestamp variable. You can access a timestamp as a TDateTime value (maximum precision = 1 millisecond) or use the SetComplexVariable method to access it as a TOracleTimestamp value (maximum precision = 1 nanosecond).

- Timestamp with time zone (otTimestampTZ)

A Timestamp with time zone variable (see above).

- Timestamp with local time zone (otTimestampLTZ)

A Timestamp with local time zone variable (see above).

PL/SQL Tables

For string, integer, float and date data types you can declare PL/SQL Tables. These are array like structures that you can pass to stored procedures, and can be used to transfer much information in one network roundtrip. Read the separate section about PL/SQL Tables for more details.

See also

DeleteVariables

VariableCount

VariableIndex

VariableName

VariableType

TOracleQuery.VariableType

Declaration

```
function VariableType(Index: Integer): Integer;
```

Description

Returns the type (otString, otInteger, ...) of the specified variable. Index is zero-based.

See also

VariableCount

VariableName

VariableIndex

TOracleQuery.WarningFlags

Declaration

```
function WarningFlags: Integer;
```

Description

Warnings returned by the last Execute. The following bit values are defined:

- | | |
|----|--|
| 1 | There is a warning |
| 2 | Data was truncated during fetch |
| 4 | NULL encountered during aggregate function |
| 16 | UPDATE or DELETE without WHERE |
| 32 | PL/SQL with compilation errors |

Note

For Net8, only the 'PL/SQL with compilation errors' flag will work.

TOracleQuery.XMLField

Declaration

```
function XMLField(const FieldId: Variant): TXMLType;
```

Description

For SYS.XMLTYPE fields, this function returns a TXMLType that you can subsequently use to access the the object. If you access the Field or FieldAsString functions for a SYS.XMLTYPE field, the XML text will bve returned.

In Delphi 4 or later this function is overloaded and accepts the index or name of the field.

TOracleDataSet component

Unit

OracleData

Description

Use the TOracleDataSet component if you want to make use of data-aware components (DBEdit, DBGrid, etc). TOracleDataSet is derived from the standard TDataSet component. It inherits most of its properties, methods and events from it. In addition to this, you will find properties and methods derived from the TOracleQuery component.

To use a TOracleDataSet, set the Session property to connect to a database. Enter a select statement for the dataset in the SQL property. Any input variables in the select statement should be declared in the Variables property and should be assigned a value at run-time through the SetVariable method. If the Session is logged on, you can set Active to True in order to open the dataset. Invoke the Fields editor of the dataset (by right clicking on the dataset icon) to set individual field properties. After linking a standard datasource to the dataset you are ready to use data-aware components.

Making a dataset updateable

To be able to update records, a dataset needs to know the 'rowid' of every retrieved record. A rowid is an internal Oracle structure that uniquely identifies a record in the database. As long as you do not include it in the query, the CanModify property of the dataset remains False. To make an updateable dept dataset, the SQL property would be: `select dept.*, dept.rowid from dept`

Furthermore, you should not use column aliases for the updateable table in the select statement of an updateable dataset. These alias names would be used in subsequent insert or updates to the updating table, and would therefore lead to an error.

If for some reason it is not possible to supply a rowid in the query, you can use the OnApplyRecord event to make the dataset updateable.

Selecting derived fields in a dataset

Sometimes you need to select fields in a dataset that are not columns of one table. You can do some calculations in the select statement, call stored functions, or join columns from other tables.

Some rules apply to creating derived fields in a dataset:

- You should use an alias for a column that is a calculation or the result of a stored function. Otherwise the name of the field is unpredictable.
- If the dataset is updateable, joined tables must be placed after the updating table in the select statement. The first table in the from clause is assumed the updating table. You can also set the UpdatingTable property to indicate which table in a join statement can be updated.
- If the dataset is updateable, the FieldKind property of a derived field must be `fkInternalCalc`. This indicates to the dataset that it should never be included in an insert or update statement.
- The ReadOnly property of a derived field should normally be set to True.

The following example selects two derived fields for the emp table:

```
select emp.*, emp.rowid, dept.dname, emp_max_sal(empno) max_sal
from emp, dept
where dept.deptno = emp.deptno
```

The department name (dname) is joined via the dept table, and a stored function emp_max_sal is called. The dept table is placed after the updating table emp, and a column alias max_sal is given to the result of the stored function.

Validating input

There are various ways to validate the input of a user in a dataset, like the Range properties of a field, the OnValidate event of a field or the BeforePost event of a dataset.

However, one of the strengths of Oracle is its capability to assure data integrity within the database. You can use primary key, unique key, foreign key and check constraints for declarative integrity checking. If any procedural logic is required, you can use database triggers. This way your data is secure, no matter how it is updated.

The easiest way for your Delphi application to validate user input is to rely on the integrity constraints present inside the database. Why specify them redundantly in your applications code? The EnforceConstraints property, MessageTable property and OnTranslateMessage event enable you to fully exploit the integrity constraints of your Oracle database.

Making use of the Oracle Dictionary

The Oracle Dictionary contains, among other things, information about tables, columns, and constraints. The previous section about validation is an example of how the TOracleDataSet can use this information to provide an application that automatically picks up this information. The OracleDictionary property allows you to specify which information in the Oracle Dictionary should be used by the TOracleDataSet.

Record locking

The Oracle server is able to lock individual records in the database. You can make use of this by setting the LockingMode property. You can control if records are locked when a user starts to edit them, when a record is posted to the database, or not at all.

Regardless of the locking mode, the dataset will always check if the record in the dataset still matches the record in the database. This prevents a user from making changes to an old image of a record.

Refreshing records

Records in the database can sometimes differ from the records in the dataset. Other users may have changed these records, or maybe there is some server side logic that modifies records as they are posted to the database. To make sure that users always work with the actual image of a database record, you can set the RefreshOptions property to a value that is appropriate to your application.

Cached updates

By default the TOracleDataSet will immediately apply and commit each posted record to the database. Sometimes this is not acceptable, in which case you can make use of CachedUpdates. Records are initially posted to a local change log, and can later be applied to the database by calling TOracleSession.ApplyUpdates.

Cursor variables

To create a dataset that is based on a cursor variable, simply create a PL/SQL block in the SQL property to call the procedure that opens the cursor. For more information about cursor variables, see the example that explains this feature.

Updating complex query results

Sometimes your queries are too complex to allow them to be simply updated through a single table. The TOracleDataSet has an OnApplyRecord event that allows you to program your own lock, insert, update, delete and refresh actions. Several functions and procedures are provided that support this feature.

TOracleDataSet reference

This chapter describes all properties, methods and events of the TOracleDataSet component.

TOracleDataSet.AfterFetchRecord

Declaration

```
type TAfterFetchRecordEvent = procedure(Sender: TOracleDataSet;  
    FilterAccept: Boolean; var Action: TAfterFetchRecordAction) of  
    Object;  
type TAfterFetchRecordAction = (afContinue, afPause, afStop,  
    afCancel);  
property AfterFetchRecord = TAfterFetchRecordEvent;
```

Description

This event is called after fetching a record from the database. It can be used to display progress information or limit the number of records that a dataset can contain.

The FilterAccept parameter indicates if the current record is accepted by the filter criteria as defined by the OnFilterRecord event or the Filter expression. The Action parameter can be used to indicate to the dataset what action is to be performed after fetching the record:

afContinue	Continue fetching subsequent records for the dataset. This is the default action.
afPause	Don't fetch any additional records right now, but continue fetching when the application requests more records.
afStop	Stop fetching records, resulting in a truncated result set.
afCancel	Stop fetching records and clear the result set. This results in an empty dataset.

The following example asks the end-user if he or she wants to continue the query after 1000 records have been fetched:

```

procedure TMyForm.EmpDataSetAfterFetchRecord(Sender: TOracleDataSet;
  FilterAccept: Boolean; var Action: TAfterFetchRecordAction);
var Msg: string;
    Btn: Integer;
begin
  if Sender.RecordCount = 1000 then
  begin
    Msg := Format('%d Records fetched', [Sender.RecordCount]);
    Btn := MessageBox(Handle,
      PChar(Msg + #13#10 + 'Continue?'),
      'Question',
      mb_IconQuestion + mb_YesNoCancel);

    case Btn of
      id_Yes: Action := afContinue;
      id_No: Action := afStop;
      id_Cancel: Action := afCancel;
    end;
  end;
end;

```

Note that you cannot access the field values in this event, because the record is not yet available. If you need to know the field values of the currently fetched record, use the `NewValue` of the fields instead.

TOracleDataSet.AfterQBE

Declaration

```

type TOracleDataSetEvent = procedure(Sender: TOracleDataSet) of
  Object;
property AfterQBE: TOracleDataSetEvent;

```

Description

Fires when the dataset leaves QBE mode, but before the query is executed. You can perform checks on the QBE values, modify QBE values, restore the appearance of queryable controls, and so on. When an exception is raised in this event, the dataset will remain in QBE mode.

See also

BeforeQBE
OnCancelQBE

TOracleDataSet.AfterQuery

Declaration

```

type TOracleDataSetEvent = procedure(Sender: TOracleDataSet) of
  Object;
property AfterQuery: TOracleDataSetEvent;

```

Description

Triggered when the last record of the dataset is retrieved. Therefore, `BeforeQuery` is only useful if `QueryAllRecords` is set to `True`.

See also

BeforeQuery

TOracleDataSet.AfterRefreshRecord

Declaration

```
type TOracleDataSetEvent = procedure(Sender: TOracleDataSet) of
    Object;
property AfterRefreshRecord: TOracleDataSetEvent;
```

Description

Triggered when a record is refreshed by calling the RefreshRecord method or after a record is edited or posted to the database and the RefreshOptions include roAfterInsert, roAfterUpdate or roBeforeEdit.

TOracleDataSet.BeforeQBE

Declaration

```
type TOracleDataSetEvent = procedure(Sender: TOracleDataSet) of
    Object;
property BeforeQBE: TOracleDataSetEvent;
```

Description

Fires when the dataset goes in QBE mode. You can use this event to preset query values or to change the appearance of queryable controls on the form. Raising an exception in this event has no effect.

See also

AfterQBE

OnCancelQBE

TOracleDataSet.BeforeQuery

Declaration

```
type TOracleDataSetEvent = procedure(Sender: TOracleDataSet) of
    Object;
property BeforeQuery: TOracleDataSetEvent;
```

Description

Triggered when the select statement begins.

See also

AfterQuery

TOracleDataSet.CachedUpdates

Declaration

property CachedUpdates: Boolean;

Description

Cached updates is a mechanism that causes all inserts, updates and deletes to be cached in a local change log. When the ApplyUpdates method of the related TOracleSession is called, all these changes will be applied to the database in one single transaction. When cached updates are not used, each record that is posted in the dataset is immediately applied to the database and committed.

There are two advantages to using cached updates:

- 1 You have more control over integrity rules that apply to multiple records and tables.
- 2 There is less network traffic involved in one call to ApplyUpdates than when many small transactions are generated for each posted record.

The disadvantages are that other users will see the cached updates later, and that if you are using the ImLockImmediate LockingMode, records will be locked longer. Also note that you can only use cached updates if the query of the dataset includes a rowid.

See also

ApplyUpdates

CancelUpdates

CommitUpdates

UpdatesPending

UpdateStatus

TOracleDataSet.ClearQBE

Declaration

procedure ClearQBE;

Description

If a dataset is modified for QBE, then ClearQBE will remove these query criteria from the SQL and will refresh the dataset with the complete result set.

TOracleDataSet.ClearVariables

Declaration

procedure ClearVariables;

Description

Assigns nulls to all declared variables.

See also

Variables

DeleteVariables

TOracleDataSet.CloseAll

Declaration

```
procedure CloseAll;
```

Description

Closes the dataset, including all associated cursors. By default, the cursor of the select statement of the dataset will remain open when you close the dataset, so that it does not need to be parsed again when it is reopened.

TOracleDataSet.CommitOnPost

Declaration

```
property CommitOnPost: Boolean;
```

Description

Determines if posting a record is automatically followed by a commit. When set to False, the application is responsible for transaction management.

Note

This property is overruled if CachedUpdates is set to True.

TOracleDataSet.CompareQueryVariables

Declaration

```
function CompareQueryVariables(AQuery: TOracleQuery): Boolean;
```

Description

This procedure can be used in an OnApplyRecord event handler to compare the current dataset record with the values in the database. All old field values (the OldValue) in the dataset will be compared to the corresponding variable values in the query. Variables and fields are matched by their names. Fields that do not match any query variable will not be compared.

If any of the fields do not match the corresponding variable value, the function result will be False.

See also

OnApplyRecord

DeclareQueryVariables

SetQueryVariables

GetQueryVariables

TOracleDataSet.CompressBLOBs

Declaration

property CompressBLOBs: Boolean;

Description

When True, BLOB fields (Long Raw columns or BLOB columns) will be stored compressed in the database. When a record is fetched, the data is decompressed, and when it is posted as a result of an Insert or Edit, the data will be compressed. This can significantly reduce database storage size and network traffic, but increases memory and cpu usage when fetching and posting BLOB field data on the client. The CompressionFactor property indicates the compression factor of posted BLOB data.

There are 2 compression methods available. A built-in proprietary LZH compression method (defined in the OracleCompress unit) will be used if you do not specify the OnCompressBLOB and OnDecompressBLOB event handlers. This method achieves a reasonable compression factor, and compresses with a speed of approximately 1MB/second and decompresses with a speed of 10MB/second on a 1.5GHz CPU. Furthermore it recognizes whether or not a BLOB field was not previously compressed, so that you can mix compressed and uncompressed BLOB data in the same dataset. Since this is a proprietary method, you cannot use the BLOB data with any other tools than a Direct Oracle Access application.

You can alternatively use your own compression method by implementing the OnCompressBLOB and OnDecompressBLOB event handlers. The advantage of this method may be that you can use non-proprietary method (e.g. zip), so that data is accessible from other tools, or that you can use a compression method that is most suited for the type of BLOB data that is stored in a specific column.

TOracleDataSet.CompressionFactor

Declaration

property CompressionFactor: Double;

Description

This property indicates the compression factor (uncompressed size / compressed size) of posted BLOB data when CompressBLOBs is True. When a BLOB field is inserted or updated, this compression factor will be updated. The initial value of this property is 0, which indicates that no compressed BLOB data has been written. The factor will be reset to 0 when the dataset is closed.

TOracleDataSet.CountAllRecords

Declaration

property CountAllRecords: Boolean;

Description

When this property is set to True, the dataset will first count the number of records that the query will return, before fetching the records. This is useful when you do not want to immediately fetch all records when the dataset is opened, but still want the RecordCount to

reflect the total number of records. The scroll bar in a TDBGrid will display the correct position if `CountAllRecords` is `True`, or if `QueryAllRecords` is set to `True`. Setting both properties to `True` is not useful.

Note that the records will be counted by calling the `CountQueryHits` function, which executes a `select count(*)` SQL statement. When the count query cannot be efficiently executed, setting `CountAllRecords` is not very useful as it will merely cause a long delay before the dataset is opened. In this case you might consider setting `QueryAllRecords` to `True`.

TOracleDataSet.CountQueryHits

Declaration

```
function CountQueryHits: Integer;
```

Description

To determine how many records a dataset will retrieve before opening the dataset, you can use the `CountQueryHits` function. This will execute a `"select count(*)"` for the select statement of the dataset. It can also be used in an `AfterQBE` event handler to determine if the query criteria do not return too many rows.

TOracleDataSet.CreateObject

Declaration

```
function CreateObject(const FieldName: string): TOracleObject;
```

Description

Creates a `TOracleObject` instance for the specified field name. The field name must correspond to an object field in the result set. The application is responsible for freeing the created object instance. Consider the following SQL statement:

```
select id, period from contracts
```

If *period* is an object field with 2 attributes *start_date* and *end_date*, the result set will display these 2 individual attributes. The `CreateObject` function allows you to reassemble the object, and call its methods or perform other actions with it. For example:

```

procedure TMainForm.ContractDataSetCalcFields(DataSet: TDataSet);
var Period: TOracleObject;
begin
    Period := ContractDataSet.CreateObject('period');
    try
        Duration.Value := Period.CallMethod('duration', parNone);
    finally
        Period.Free;
    end;
end;

```

TOracleDataSet.Cursor

Declaration

```
property Cursor: TCursor;
```

Description

Determines the shape of the mouse cursor while executing SQL statements. Only crDefault, crHourGlass and crSQLWait are useful here.

TOracleDataSet.Debug

Declaration

```
property Debug: Boolean;
```

Description

When set to true the select statement and all variable values will be displayed when executed.

Besides the select statement, Debug will also cause update, insert, delete and lock statements to be shown, as well as statements executed to enforce constraints when EnforceConstraints is set to True.

TOracleDataSet.DeclareAndSet

Declaration

```
procedure DeclareAndSet(Name: string; Type: Integer; Value: Variant);
```

Description

Declares a variable of a certain name and type, and sets its initial value. This procedure is particularly handy if you have to declare and set many variables at runtime.

TOracleDataSet.DeclareQueryVariables

Declaration

```
procedure DeclareQueryVariables(AQuery: TOracleQuery);;
```

Description

This procedure can be used in an OnApplyRecord event handler to declare all variables in a

query with a data type that corresponds to the fields in the dataset. Variables and fields are matched by their names. You can also prefix a variable name with 'old_', in which case it will also be declared. Variables that do not match any dataset field will not be declared.

See also

OnApplyRecord

SetQueryVariables

GetQueryVariables

CompareQueryVariables

TOracleDataSet.DeclareVariable

Declaration

```
procedure DeclareVariable(Name: string; Type: Integer);
```

Description

Declares a variable of a certain name and type.

See also

Variables

DeleteVariable

DeleteVariables

TOracleDataSet.DeleteVariable

Declaration

```
procedure DeleteVariable(AName: string);
```

Description

Deletes the specified variable.

See also

Variables

DeclareVariable

TOracleDataSet.DeleteVariables

Declaration

```
procedure DeleteVariables;
```

Description

Deletes all declared variables

See also

Variables

ClearVariables
DeclareVariable

TOracleDataSet.DesignActivation

Declaration

```
property DesignActivation: Boolean;
```

Description

It can be very useful to always have datasets active at design time, but opening and closing a dataset often occurs under control of the application. By setting the DesignActivation property to True, the value of the Active property at design-time will be ignored at run time, and prevents that your datasets are active by mistake when you run your application.

TOracleDataSet.Detachable

Declaration

```
property Detachable: Boolean;
```

Description

When a session logs off, all datasets that are linked to that session and that are not Detachable will be closed. When the Detachable property is set to True, the dataset will remain open in this situation. You will no longer be able to perform any operation that requires database access though, such as refreshing, inserting, updating, deleting, and so on.

Note

If the session logs on again, you will have to refresh a previously detached dataset before you can perform any database access operations again.

TOracleDataSet.DetailFields

Declaration

```
property DetailFields: string;
```

Description

If the dataset is a detail in a master/detail relation, this property names the fields of the detail that relate to the MasterFields. If more than one field is used, separate them with a semicolon. The order of these fields must correspond to the order of the MasterFields. The query of the dataset must contain variables in the where clause with exactly the same names as the DetailFields. If you leave this property empty, the query of the dataset must contain variables in the where clause with the same names as the MasterFields. For more information, see the Master property.

See also

Master
MasterFields

TOracleDataSet.ErrorLine

Declaration

```
function ErrorLine: Integer;
```

Description

In case of a parse error, ErrorLine indicates the line where the error occurred (one-based).

See also

ErrorPosition

TOracleDataSet.ErrorPosition

Declaration

```
function ErrorPosition: Integer;
```

Description

In case of a parse error, ErrorPosition indicates the character position on the ErrorLine where the error occurred (one-based).

See also

ErrorLine

TOracleDataSet.ExecSQL

Declaration

```
procedure ExecSQL;
```

Description

You can use the ExecSQL procedure to execute other statements than the select statement, for example insert, update, or delete statements, Data Definition Language statements (create table, create procedure, grant role, etc.), Session Control statements (alter session, set role), System Control statements (alter system) and PL/SQL Blocks.

For select statements you should use the Open procedure (or set the Active property to True), so that the result set will be retrieved.

TOracleDataSet.ExecuteQBE

Declaration

```
procedure ExecuteQBE;
```

Description

When the dataset is in QBE mode, the query can be executed by calling the ExecuteQBE method. After this, QBEMode will be set to False.

TOracleDataSet.ExternalCursor

Declaration

```
property ExternalCursor: TOracleQuery;
```

Description

If you want to use the result set of a TOracleQuery instance for data-aware controls or other components that require a TDataSet as input, you can use the ExternalCursor of a TOracleDataSet. This can be especially useful if the TOracleQuery instance is used as a cursor parameter obtained through a call to a package generated by the Package Wizard.

Note

If the dataset is connected with an ExternalCursor, it cannot implicitly be refreshed. Use the BeforeRefresh event handler of the dataset to reopen the cursor, or prevent refresh operations.

TOracleDataSet.Filter

Declaration

```
property Filter: string;
```

Description

Filtering in a TOracleDataSet basically behaves the same as described with the standard TDataSet. Simply assign a string to the Filter property to define the filter conditions and set the Filtered property to True to reduce the result set to the corresponding selection of records. Note that all records selected by the SQL statement are transferred to the client, and filtering is performed locally. This is a possible disadvantage of filtering, it might cause more network traffic (and time) than needed to display a number of records. You should always consider if a where clause in your SQL statement wouldn't do the same job more efficiently. However, since all transferred records are buffered on the client, a change in the Filter property will very quickly give you a result because the database will not be accessed again, all work is now done locally.

If you want to display employees from department 20, you could use a filter like this:

```
deptno = 20
```

Or, if you want to display employees from departments 20 to 40:

```
(deptno >= 20) and (deptno <= 40)
```

Or, if you want to display all employees with a name starting with 'S', hired since 1-1-1980:

```
(ename = 'S*') and (hiredate >= '1-1-1980')
```

Some general rules about the filtering syntax

- Don't place quotes around field names, quotes are only used for string and date literals.
- You may use square brackets around field names.

- You can use the * and ? wildcards in string comparisons.
- Always put quotes around a date literal. If you don't, it will be interpreted as a numeric calculation (20-01-1967 = -1948).
- The format of a date literal depends on your local settings.
- You can perform calculations like 12 * sal + comm > 50000.

See also

Filtered

FilterOptions

TOracleDataSet.Filtered

Declaration

```
property Filtered: Boolean;
```

Description

To apply filter conditions specified in the Filter property or the OnFilterRecord event handler, set Filtered to True.

See also

Filter

FilterOptions

TOracleDataSet.FilterOptions

Declaration

```
type TFilterOption = (foCaseInsensitive, foNoPartialCompare);  
type TFilterOptions = set of TFilterOption;  
property FilterOptions: TFilterOptions;
```

Description

Set FilterOptions to specify whether or not filtering is case insensitive when filtering on string or character fields, and whether or not partial comparisons for matching filter conditions is allowed.

By default, FilterOptions is set to an empty set. For filters based on string fields, set FilterOptions to foCaseInsensitive to catch all variations on a string regardless of capitalization.

For filter conditions based on multiple conditions or fields, set FilterOptions to foNoPartialCompare to force exact matches only on comparison.

See also

Filter

Filtered

TOracleDataSet.GetComplexVariable

Declaration

```
function GetComplexVariable(Name: string): TObject;
```

Description

Returns the object previously associated with the variable by using the SetComplexVariable method.

This procedure is only useful if you are using the ExecSQL procedure.

TOracleDataSet.GetQueryVariables

Declaration

```
procedure GetQueryVariables(AQuery: TOracleQuery);;
```

Description

This procedure can be used in an OnApplyRecord event handler to refresh the current dataset record. All field values in the dataset will be set to the corresponding variable values in the query. Variables and fields are matched by their names. Fields that do not match any query variable will not be refreshed.

See also

OnApplyRecord

DeclareQueryVariables

SetQueryVariables

CompareQueryVariables

TOracleDataSet.GetUpdatingTable

Declaration

```
function GetUpdatingTable: string;
```

Description

Returns the name of the table that is going to be updated by the dataset, which is determined by the UpdatingTable property or the first table name after the from-clause of the SQL text.

TOracleDataSet.GetVariable

Declaration

```
function GetVariable(Name: string): Variant;
```

Description

Retrieves the value of the specified variable. In Delphi 4 and later you can also specify the variable by its zero-based index.

See also

Variables

DeclareVariable

SetVariable

TOracleDataSet.LockingMode

Declaration

```
type TLockingModeOption = (lmLockImmediate, lmCheckImmediate,
    lmLockDelayed, lmNone);
```

```
property LockingMode: TLockingModeOption;
```

Description

If you are creating a multi-user application, the locking strategy needs some attention. TOracleDataSet has a LockingMode property that lets you define the locking behavior. When a record is locked, it is important that the record in the database still contains the same values as the record in the dataset. After all, since the dataset has queried these records, they might have been changed by another user. The LockingMode property can be one of the following values:

lmLockImmediate	When the user starts editing a record, it is locked and a check is performed to see if it has been changed. The lock remains until the user posts or cancels the changes.
lmCheckImmediate	When the user starts editing a record, a check is performed to see if it has been changed, but the record is not locked. Therefore, when the user posts the record, it is locked and checked again.
lmLockDelayed	When the user posts an edited record, it is locked and a check is performed to see if it has been changed. After this, the lock is released.
lmNone	No locking or checks are performed. This should only be used in single user applications.

The default value for the LockingMode is lmCheckImmediate. This has the advantage that there are no locks during editing, but the user is still notified immediately if the record has been changed by another user. If you want to use lmLockImmediate, which guarantees that a user locks a record during editing, you need to make sure that locks are not released by commits and rollbacks in other parts of the application using the same session.

TOracleDataSet.Master

Declaration

```
property Master: TOracleDataSet;
```

Description

If the dataset is a detail in a master/detail relation, this property defines the master dataset.

To create a master/detail form, you need two datasets that preferably are related through a foreign key constraint. You can link the detail dataset to the master dataset through the Master

property of the detail dataset. When you set this property, you are asked if you wish to modify the query to add variables to the where clause. If you accept this, a where clause is added to the SQL property, variables are created for the foreign key column(s) and the MasterFields and DetailFields are set. The value of the variables is controlled by the master dataset.

Some basic rules need to be followed when creating a master/detail form:

- ♦ The where clause of the detail dataset must always contain a comparison between the foreign key columns and a variable with the same name (where a = :a and b = :b). The dataset requires this to restrict the records to the context of the master.
- ♦ The foreign key columns must be included in the field list of the detail dataset if new records are to be inserted. The dataset copies the values of the MasterFields to the DetailFields when a new record is created.
- ♦ Automatic Master/Detail configuration is invoked when setting the Master property of a detail dataset, but only works when:
 - ♦ The SQL property of the master and detail are set correctly
 - ♦ The Session of either the master or detail is connected
 - ♦ One foreign key constraint exists from the detail to the master

Non-referential details

If a detail dataset does not have a foreign key column to the master table, then you can omit the DetailFields. Assume that a Master dataset contains a start_date and end_date column, and you want to show all details with a creation_date column within the start_date and end_date of the master. The where clause detail query could look like this:

```
where creation_date >= :start_date and creation_date <= :end_date
```

In this situation you do not have a 1 on 1 relation between master and detail fields. You can leave the DetailFields property empty, and use variable names in the detail query that match the names of the MasterFields (as shown in the example above).

Nested tables and varrays

If the table of the master dataset contains a collection column or attribute (nested table or varray), you can create a detail dataset for this collection. At design time you do not need to define the SQL for the detail dataset. Simply set the master property and select a collection column or attribute from the selection screen. After doing so, the SQL of the detail dataset will be something like this:

```
select d.*
from table(select m.lines
            from   invoices m
            where  m.id = :id) d
```

In this case the master table is *invoices*, the collection column is *lines*, and the primary key column of the master table is *id*. The basic SQL that is generated by the Automatic Master/Detail configuration can of course be refined by specifying specific columns instead of *d.**, by adding an order by clause, and so on.

Note

You can only use TOracleDataSet components in a Master/Detail relation. You cannot mix

these with TOracleQuery, TQuery, TTable or other components.

See also

MasterFields

DetailFields

TOracleDataSet.MasterFields

Declaration

```
property MasterFields: string;
```

Description

If the dataset is a detail in a master/detail relation, this property names the fields of a master that identify a master record. If more than one field is used, separate them with a semicolon. If you leave the DetailFields property empty, the query of the dataset must contain variables in the where clause with the same names as the MasterFields. For more information, see the Master property.

See also

Master

DetailFields

TOracleDataSet.OnApplyRecord

Declaration

```
type TApplyRecordEvent = procedure(Sender: TOracleDataSet; Action:
    Char; var Applied: Boolean; var NewRowId: string) of object;
property OnApplyRecord: TApplyRecordEvent;
```

Description

The OnApplyRecord event provides a convenient way to overrule the default behavior of a TOracleDataSet component when checking, locking, inserting, updating, deleting and refreshing records. Within this event, you can use TOracleQuery, TOraclePackage or custom package components to perform the required actions. This is useful in one of the following situations:

- ♦ You wish to access the table through stored procedures rather than through SQL. This is often referred to as a 'table API', where the end-user does not need privileges on the table, but all access is under control of stored procedures.
- ♦ The results of the dataset's query are not directly updateable, for example when you query a non-updateable view or any other case where a rowid and updating table are not available.

The Action parameter indicates which action should be performed:

'C' Check if the record in the dataset's buffer is still the same as in the database. You do this by comparing the database values with the OldValue of the fields. This action will only occur if the LockingMode property of the dataset is lmCheckImmediate. With all other LockingModes, the check can be combined with the lock action.

- 'L' Lock the record. You can also use this action to check if another user has changed the record in the database.
- 'I' Insert the record. If you are using a rowid in the dataset, return the rowid of the newly inserted record in the NewRowId parameter.
- 'U' Update the record.
- 'D' Delete the record.
- 'R' Refresh the record. Note that you don't actually need this action to refresh a record after an insert or update, because you can simply update the value of the fields during the Insert and Delete action.

For all actions but Insert, it is required that you identify the record in the database. You can do so by referring to the OldValue of the fields or the RowId function of the dataset. The examples below demonstrate this.

With the Applied parameter you indicate if the event handler has applied this action. This way you can choose to overrule only some specific actions, and let other actions be handled by the TOracleDataSet. If you overrule only some actions, you must set the NewRowId parameter during inserts, because a rowid is required for the default dataset actions. Only when all actions are overruled, the rowid can be skipped completely.

The following is an example of an OnApplyRecord event handler that overrules the delete action by setting a deleted indicator column of the record to 'Y', rather than actually deleting it. The dataset provides a rowid to identify the record. All other actions are performed by the default processing of the dataset.

```
procedure TMainForm.DeptDataSetApplyRecord(Sender: TOracleDataSet;
  Action: Char;
  var Applied: Boolean; var NewRowId: string);
begin
  if Action = 'D' then
    begin
      // DeptDelQuery.SQL = update dept set deleted = 'Y' where rowid =
      :dept_rowid
      with DeptDelQuery do
        begin
          SetVariable('dept_rowid', Sender.RowId);
          Execute;
        end;
      // We handled the delete, but all other actions must be handled
      by the dataset
      Applied := True;
    end;
  end;
```

Most of the time things will be a little bit more complicated than the example mentioned above. You will usually need to declare a lot of variables, set the value of each individual variable to the corresponding dataset field value, and execute the query. To make this easier you can declare the variables and set their values with a single call. Additionally you can also compare query variables with the dataset fields to check if a record has changed, and set field values to query variable values to refresh a dataset record.

Declaring variables

To declare the variables in a TOracleQuery component, you can use the DeclareQueryVariables procedure:

```
procedure DeclareQueryVariables(AQuery: TOracleQuery);
```

All variables in the SQL property of the query will be declared for which a corresponding field exists in the dataset. Variables and fields are matched by their names. The data type of the declared variables will correspond to the data type of the fields. If the variable name is 'doa__rowid', it will be declared as a string. You can also prefix a variable name with 'old_', in which case it will also be declared. This feature is useful when you need to pass the old value of a field to the query, which will be explained below. Variables that do not match any dataset field will not be declared. Consider a dataset with the following select statement:

```
select deptno, dname, loc
  from dept
 order by deptno
```

To update a record, we want to call the DeptAPI package, which has an UpdateRecord procedure:

```
begin
  DeptAPI.UpdateRecord(:old_deptno, :deptno, :dname, :loc);
end;
```

To declare the variables of this PL/SQL block, we can simply call DeclareQueryVariables and pass the query with this SQL text. After that, the old_deptno and deptno variables will be declared as integers, and dname and loc will be declared as strings.

Setting variable values

To set the values of the variables in a TOracleQuery component, use the SetQueryVariables procedure:

```
procedure SetQueryVariables(AQuery: TOracleQuery);
```

The values of the fields will be copied to the variables in the query. If the variable name is prefixed with 'old_', the OldValue of the field will be copied. If the variable name is 'doa__rowid', it will be set to the rowid of the current record. After setting the variable values you can execute the query. The following code implements the update example from the previous section:

```
DeptDataSet.DeclareQueryVariables(UpdateQuery);
DeptDataSet.SetQueryVariables(UpdateQuery);
UpdateQuery.Execute;
```

The DeptDataSet contains the select statement, and the UpdateQuery contains the call to DeptAPI.UpdateRecord. This code can run inside an OnApplyRecord event handler to overrule the default update action.

Refreshing a dataset record

To refresh a dataset record you can use the GetQueryVariables procedure:

```
procedure GetQueryVariables(AQuery: TOracleQuery);
```

This procedure copies the variable values of the query to the fields of the dataset. Let's assume that the DeptAPI package contains a FetchRecord procedure that can be called like this:

```
begin
    DeptAPI.FetchRecord(:old_deptno, :dname, :loc)';
end;
```

The department number identifies the dept record, and the dname and loc column values are returned. To call this procedure you can use the following code inside an OnApplyRecord event handler to refresh the current record:

```
DeptDataSet.DeclareQueryVariables(FetchQuery);
DeptDataSet.SetQueryVariables(FetchQuery);
FetchQuery.Execute;
DeptDataSet.GetQueryVariables(FetchQuery);
```

The variables are declared, the value of the old_deptno variable is set, the procedure is called, and the field values are updated with the current values in the database.

Checking a dataset record

To check if a record in the database has changed since it was fetched into the dataset, you can use the CompareQueryVariables function:

```
function CompareQueryVariables(AQuery: TOracleQuery): Boolean;
```

This function compares the variable values of the query with the current values of the fields. It can be called in a very similar way to GetQueryVariables:

```
DeptDataSet.DeclareQueryVariables(FetchQuery);
DeptDataSet.SetQueryVariables(FetchQuery);
FetchQuery.Execute;
if not DeptDataSet.CompareQueryVariables(FetchQuery) then
    raise Exception.Create('Department is changed by another user');
```

The variables are declared, the value of the old_deptno variable is set, the procedure is called, and the field values are compared with the current values in the database.

Exception handling

If an exception occurs during an OnApplyRecord event, the TOracleDataSet will handle it correctly. All previous changes will be rolled back (to a savepoint if necessary), and the error will be translated by the OnTranslateMessage event handler or through the MessageTable. There is no need to handle exceptions within the OnApplyRecord event, unless you want to specifically react to it.

Restrictions

Two restrictions apply to what you can do in an OnApplyRecord event:

- 1 Don't commit or rollback. If you must rollback some of the actions you have performed, set your own savepoint at the beginning of the event and rollback to this savepoint.
- 2 Don't navigate to other records in the same TOracleDataSet or in its master dataset.

TOracleDataSet.OnCancelQBE

Declaration

```
type TCancelQBEEvent = procedure(Sender: TOracleDataSet; var Requery: Boolean) of Object;
property OnCancelQBE: TCancelQBEEvent;
```

Description

Fires when QBE mode is cancelled. You can restore the appearance of queryable controls in this event. When an exception is raised, the dataset will remain in QBE mode.

The Requery boolean parameter can be used to specify whether the last result will be requeried or that no requery will take place and the dataset remains empty. By default, the dataset will requery.

See also

AfterQBE

BeforeQBE

TOracleDataSet.OnCompressBLOB

Declaration

```
type TCompressionEvent = procedure(Sender: TOracleDataSet; Field: TBlobField; InStream, OutStream: TMemoryStream) of Object;
property OnCompressBLOB: TCompressionEvent;
```

Description

Fires when CompressBLOBs = True and a Long Raw or BLOB field is posted to the database as a result of an Insert or Edit. The event handler must read all uncompressed bytes from the InStream and write the compressed bytes to the OutStream. If the Size of the OutStream is 0 after this event, the dataset will post the original uncompressed data.

Example

The following example uses Delphi's ZLib functionality to compress a BLOB field:

```
procedure TMainForm.DataSetCompressBLOB(Sender: TOracleDataSet;
  Field: TBlobField; InStream, OutStream: TMemoryStream);
var OutBuf: Pointer;
    OutSize: Integer;
begin
  // Compress the input stream into a buffer through ZLib
  CompressBuf(InStream.Memory, InStream.Size, OutBuf, OutSize);
  // Write the buffer into the output stream
  OutStream.Write(OutBuf^, OutSize);
  // Free the ZLib buffer
  FreeMem(OutBuf, OutSize);
end;
```

See also

OnDecompressBLOB

TOracleDataSet.OnDecompressBLOB

Declaration

```
type TCompressionEvent = procedure(Sender: TOracleDataSet; Field:
    TBLOBField; InStream, OutStream: TMemoryStream) of Object;
property OnDecompressBLOB: TCompressionEvent;
```

Description

Fires when CompressBLOBs = True and a Long Raw or BLOB field is fetched from the database. The event handler must read all compressed bytes from the InStream and write the uncompressed bytes to the OutStream. If the Size of the OutStream is 0 after this event, the dataset will assume that the data was not compressed.

Example

The following example uses Delphi's ZLib functionality to decompress a BLOB field:

```
procedure TMainForm.DataSetDecompressBLOB(Sender: TOracleDataSet;
    Field: TBlobField; InStream, OutStream: TMemoryStream);
var OutBuf: Pointer;
    OutSize: Integer;
begin
    // Decompress the input stream into a buffer through ZLib
    DecompressBuf(InStream.Memory, InStream.Size, 2 * InStream.Size,
        OutBuf, OutSize);
    // Write the buffer into the output stream
    OutStream.Write(OutBuf^, OutSize);
    // Free the ZLib buffer
    FreeMem(OutBuf, OutSize);
end;
```

See also

OnCompressBLOB

TOracleDataSet.OnTranslateMessage

Declaration

```
type TTranslateMessageEvent = procedure(Sender: TOracleDataSet;
    ErrorCode: Integer; ConstraintName: string; Action: Char; var Msg:
    string) of Object;
property OnTranslateMessage: TTranslateMessageEvent;
```

Description

When an Oracle error occurs during an insert, update, delete or lock issued by the dataset, this event is triggered. If EnforceConstraints is set to True, it is also triggered when a constraint is checked and violated after a user has changed the corresponding field.

OnTranslateMessage enables you to translate a technical Oracle message into a user-friendly message. The meaning of the parameters is as follows:

Sender	The dataset that caused the error
ErrorCode	The Oracle error number

ConstraintName	The name of the primary key, unique key, foreign key or checkconstraint that was violated (if applicable).
Action	A character indicating that an insert ('I'), update ('U'), delete ('D') or lock ('L') has taken place.
Msg	The original message that you can modify.

See also

EnforceConstraints

TOracleDataSet.Optimize

Declaration**property** Optimize: Boolean;**Description**

When set to True, queries will not unnecessarily re-parse SQL when repeatedly executed. Additional parse operations will only occur if you change the SQL text, add or remove variables, or change the types of existing variables between calls. This setting when increase performance when repeatedly executing the same SQL with different variable values.

If this property is set to False, the corresponding database cursor will be closed immediately after executing a non-select statement, or when Eof is reached for select statements. This can be useful if you want to minimize the number of open database cursors.

This property is overruled if you are using Oracle Net 9.2 and have enabled the StatementCache of the corresponding session. In this situation the cursor will be released into the cache after execution or after reaching Eof, regardless of the Optimize setting.

TOracleDataSet.OracleDictionary

Declaration**property** OracleDictionary: TOracleDictionary;**Description**

The OracleDictionary property allows you to control which information in the Oracle dictionary will automatically be used by the TOracleDataSet. Enabling OracleDictionary options allows your application to pick up database changes dynamically and automatically, without any modifications.

The OracleDictionary contains the following properties:

property EnforceConstraints: Boolean;

If your application relies on the constraints in the Oracle database, these constraints are checked when the database is updated. You might still want single-column constraints to be enforced immediately after the user has changed the corresponding field. After all, it could be quite irritating to a user if he or she is notified that an employee already exists in the database after entering a complete employee record. This could have been prevented by checking the primary key constraint immediately.

Similarly, if `CachedUpdates` are used, all multi-column constraints should also be checked when the record is posted to the local change log. This prevents a series of error messages when a user attempts to apply the updates from the local change log to the database.

The dataset will enforce all constraints automatically if you set the `EnforceConstraints` property to `True`. A primary key, unique key, foreign key or check constraint is enforced if:

- ♦ It is enabled
- ♦ It is a named constraint. Otherwise there is no way to identify the constraint, and it can therefore not lead to a user-friendly message. Unnamed constraints can be created when they are part of a 'create table' statement. This should be avoided.

To translate the standard constraint violation messages generated by the server to a more user-friendly and understandable messages, you can use the `OnTranslateMessage` event or the `UseMessageTable` property.

Remember that extra database accesses are required when enforcing constraints. The constraints need to be read from the Oracle dictionary when the dataset is first opened, and need to be checked when fields or records are validated. This might lead to slight performance degradation.

property `DisabledConstraints: TDisabledConstraints;`

This is a list of constraints that will not be enforced by the dataset, but are left to the server instead. When you double-click this property, a list of constraints defined for the updating table is displayed, and you can select constraints that should not be enforced. You can manipulate this property at run-time as a normal `TStringList`.

property `UseMessageTable: Boolean;`

Use the `MessageTable` defined at the session level to translate error messages caused by constraints. This way you don't need to write an `OnTranslateMessage` event handler, and are able to add constraints to the database without the need to modify your application. Adding a record to the message table is all that is required for your application to pick up the new constraint.

property `FieldKinds: Boolean;`

Automatically check which fields belong to the updating table. These fields will become data fields (`FieldKind = fkData`), and all other fields will become read-only `InternalCalc` fields (`FieldKind = fkInternalCalc`).

property `DefaultValues: Boolean;`

Apply the default values defined for the columns of the updating table to the field values of newly created records. These default values will be applied when the record is created locally in the dataset.

property `DynamicDefaults: Boolean;`

Determine default values every time a new record is created. Normally the dataset will determine these default values just once, but in case these defaults can change (e.g. an expression that includes the `sysdate` function), you can let it determine the values every time.

property `DisplayFormats: Boolean;`

Determine the `DisplayFormat` for `TFloatFields`. The `DisplayFormat` and `EditFormat` will correspond to the precision and scale of the column. If for example a column is defined as `number(7,2)`, the field's `DisplayFormat` will be `##,##0.00` and the `EditFormat` will be `####0.00` (so that the user is not bothered with the group separator when he or she modifies the value).

property RangeValues: Boolean;

Determine the MinValue and MaxValue for TIntegerFields and TFloatFields. If for example a column is defined as number(7,2), the field's MinValue will be -99999.99 and the MaxValue will be +99999.99.

property RequiredFields: Boolean;

When True, fields that are reported as not null by Oracle will have their TField.Required property set to True. If you set RequiredFields to False, all TField.Required properties will be set to False.

A note about performance

Querying the dictionary for this information can obviously degrade performance. This depends on the efficiency of your Oracle dictionary views, and on the speed of the network. All queried dictionary information is cached at the session level though, so an end-user should only experience a delay when this information is needed the first time during the lifetime of the session. When the session is disconnected, the cached dictionary information is removed.

TOracleDataSet.ProviderOptions

Declaration

```
type TOracleProviderOption = (opNoKeyFields, opNoIndexDefs,
    opNoDefaultOrder, opNoCommit);
type TOracleProviderOptions = set of TOracleProviderOption;
property ProviderOptions: TOracleProviderOptions;
```

Description

The ProviderOptions property controls the behavior of a TOracleDataSet when connected to a TDataSetProvider:

opNoKeyFields	The dataset will not attempt to determine the key fields.
opNoIndexDefs	The dataset will not attempt to determine the index definitions.
opNoDefaultOrder	The dataset will not determine the order by clause of the SQL statement.
opNoCommit	The dataset will not commit transactions. You will have to implement your own transaction management in the server application.

TOracleDataSet.QBEDefinition

Declaration

```
property QBEDefinition: TQBEDefinition;
```

Description

To define the behavior of a TOracleDataSet during QBE mode, you can modify the QBEDefinition property. This will bring up a property editor with the following fields:

- ◆ Save QBE Values

When a query is executed, the QBE values are saved and restored the next time that the dataset goes in QBE mode.

- ♦ Allow File Wildcards

Besides the SQL wildcard characters (%) and (_), the familiar file wildcard characters (*) and (?) are also accepted. These characters are converted to SQL wildcard characters when the query is executed.

- ♦ Allow Operators

When enabled, the user can enter operators in the QBE fields. These operators are

- ♦ not *value* (equivalent to <> *value* or != *value*)
- ♦ > *value*
- ♦ < *value*
- ♦ = *value*
- ♦ (not) like *value*
- ♦ (not) between *value-1* and *value-2*
- ♦ (not) in (*value-1*, *value-2*, ..., *value-n*)
- ♦ (not) *value-1* or *value-2* or ... or *value-n*

These operators can be used for string fields, number fields and date fields. As a result, you can not access the values of the fields in QBE mode, because the user input does not translate to a value of the actual type of the field. To access the data as a string, use the QBEField.Value property instead, which is a string value that represents the actual expression that the user has entered.

- ♦ QBE Font color

Specifies the font color of data-aware controls linked to this dataset when it is in QBE mode. Specify clNone to leave the font color unchanged. The dropdown list shows the standard colors, and the button to the right of the list allows you to select custom colors.

- ♦ QBE Background color

Specifies the background color of data-aware controls linked to this dataset when it is in QBE mode. Specify clNone to leave the background color unchanged. The dropdown list shows the standard colors, and the button to the right of the list allows you to select custom colors.

- ♦ Queryable

Defines if the selected field is queryable. If not, the field will be read-only in QBE mode. BLOB fields cannot be queryable, and all options will be disabled for these fields.

- ♦ Automatic Contains

Defines if the selected field only needs to contain the QBE value. If for example the user enters 'mi', both Smith and Jamison are selected. This option is only enabled for string fields.

- ♦ Automatic Partial Match

Defines if the selected field only needs to partially match the QBE value. If for example the user enters 'Jo', both Jones and Johnson are selected. This option is only enabled for string fields.

- ♦ Case Insensitive

Defines that the selected field will be compared case insensitive. When this option is enabled, query performance may decrease if the field is indexed, because this index can only partially be used. This option is only enabled for string fields.

- ♦ Ignore Time

Defines that for the selected field any time fraction in the database will be ignored. This is particularly useful when querying timestamp columns where the user does not know the exact time, but does know the date. If the user specifies a time fraction in the query field, time fractions will not be ignored and the values must exactly match. This option is only enabled for date fields.

Besides these options you will find a test button on this property editor which lets you test the QBE Definition by parsing a query for each queryable field. When your query passes this test, you can safely assume that all queryable fields can indeed be queried. When an error occurs, the field, error message, and possible solution are displayed.

TOracleDataSet.QBEMode

Declaration

```
property QBEMode: Boolean;
```

Description

The TOracleDataSet supports a Query By Example (QBE) mechanism. When a TOracleDataSet is in QBE mode, query values can be entered in the data-aware controls of that dataset. After that, the query can be executed and the dataset will leave QBE mode, allowing the user to view or modify the result set.

To easily support QBE in your application, you can use the TOracleNavigator component. It is derived from the standard TDBNavigator, and adds two buttons to it: Enter Query and Execute Query. The Enter Query button will stay down when pressed, indicating that the dataset is in QBE mode. When the Enter Query button is pressed again, QBE mode will be cancelled.

How does QBE work?

When QBEMode is set to True, the dataset is cleared and one query-record is created. When the query is executed, the field values of this query-record are translated to an additional expression for the where clause of the original SQL statement of the dataset. The SQL statement is modified and new query variables will be added as needed. As a result, the SQL statement and variables will dynamically change at run-time when QBE mode is being used.

Consider a dataset with the following SQL statement:

```
select empno, ename, deptno from emp
order by ename
```

When the user enters 10 in the deptno field during QBE mode, the SQL statement will be modified as follows when it is executed:

```
select empno, ename, deptno from emp
where deptno = 10
order by ename
```

QBE and complex queries

For complex queries it may be necessary to make some field modifications to get QBE to work properly. As described in the previous section, the SQL statement is modified for QBE and this can lead to errors in some situations. The test button on the QBE Definition property editor can help you test your query to find and solve these errors.

Ambiguous column name when using joins

If a where clause is generated for a join select statement, this may result in ambiguous column names. Consider the following statement:

```
select empno, ename, deptno, dname
from emp, dept
where dept.deptno = emp.deptno
order by ename
```

When the user enters a value in the deptno field during QBE mode, the SQL statement will be modified as follows when it is executed:

```
select empno, ename, deptno from emp
where (dept.deptno = emp.deptno)
and deptno = 10
order by ename
```

The generated deptno column name in the where clause is ambiguous, as it is not clear if it belongs to the emp or dept table (both contain a deptno column). This will lead to an "ORA-00914: column ambiguously defined" error message. To resolve this, you can set the Origin property of the deptno field to 'emp.deptno'. This Origin property will be used during QBE to name the column in the where clause. To set the Origin field property at design-time, you must make the fields persistent. You can also set the Origin property at run-time to avoid persistent fields.

Invalid column name when using aliases

When a column has an alias in a select statement, the fieldname will be based on that alias. When this field is used in QBE mode, the following statement might be generated:

```
select empno employee_number, ename employee_name from emp
where employee_number = 7385
order by ename
```

The employee_number alias is used in the where clause, which unfortunately is not accepted by Oracle and will result in an "ORA-00904: invalid column name" error message. To resolve this situation, you must use the Origin property of the field as described in the previous section.

It can also be that you are using an alias to name the result of an arithmetic expression or a function call:

```
select empno, ename, salgrade(empno) salgrade from emp
order by ename
```

In this case the salgrade stored function is called to return the salary grade of an employee. If the salgrade field is to be queryable, you must specify the complete expression 'salgrade(empno)' in the Origin property of the field instead of specifying a table and column name.

QBE Limitations

The following limitations apply to QBE.

- ♦ In normal mode, wildcard characters are only allowed in string fields. This is a limitation of the TField and data-aware components, which require that the text in the control must be converted to the internal representation of the field. If AllowOperators is enabled, wildcards can be used for all field types.
- ♦ When cached updates are enabled for a dataset and there are pending updates, the dataset cannot go in QBE mode. Various queries across cached updates can lead to confusing situations for an end-user.
- ♦ QBE mode cannot be enabled on datasets that are based on a cursor variable.
- ♦ Select statements with set operations (union, intersect, minus) are currently not supported. You must use a view instead.

TOracleDataSet.QBEModified

Declaration

```
property QBEModified: Boolean;
```

Description

Indicates that the SQL text of the dataset is modified for QBE. This will be True if the user or the application has entered one or more QBE values in QBE Mode and has executed the query. It will be false if no QBE values were entered, or if ClearQBE was called.

TOracleDataSet.QueryAllRecords

Declaration

```
property QueryAllRecords: Boolean;
```

Description

When set to True, all records will be retrieved from the database when the dataset is opened. When set to False, records are retrieved when a data-aware component or a program requests it. If a query can return many records, set this property to False if initial response time is important.

TOracleDataSet.ReadBuffer

Declaration

```
property ReadBuffer: Integer;
```

Description

Number of rows that will be transferred across the network at once for select statements. This property can have a great impact on performance.

If the query contains a Long or Long Raw field, a ReadBuffer of 1 will be used.

TOracleDataSet.ReadOnly

Declaration

```
property ReadOnly: Boolean;
```

Description

Specifies if the data can be modified.

TOracleDataSet.RefreshOptions

Declaration

```
type TRefreshOption = (roBeforeEdit, roAfterInsert, roAfterUpdate,  
    roAllFields);  
type TRefreshOptions = set of TRefreshOption;  
property RefreshOptions: TRefreshOptions;
```

Description

The RefreshOptions property controls how and when the TOracleDataSet refreshes individual records in the dataset. This way you can make sure that the dataset always reflects the current status of the database when records are modified.

Refreshing server generated values

In a Delphi/Oracle application, values in the dataset fields may be changed on the server during a post in three ways:

1. A column has a default value in the table definition, and the corresponding field is left blank during insert. The dataset will not include this column in the insert statement, and the server will apply the default value of the table definition.
2. A row level before update or insert trigger on the table modifies some columns. This is frequently used to generate primary key values through a sequence, to set the values of audit columns, or to apply complex default values.
3. The dataset field has a DefaultValue property. When the field is blank during an insert, the dataset will send this string as part of the insert statement, which will be evaluated on the server. The DefaultValue is not evaluated on the client, because you can enter any valid SQL function here: `add_months(trunc(sysdate), 3)` is a valid DefaultValue.

You probably do not want to keep these server-generated values a secret to your application user. To make a dataset reflect the actual values in the database after a post, set the following

options:

<code>roAfterInsert</code>	The record is automatically re-fetched after the dataset has inserted a record. Default values and columns modified in a row level before insert trigger are immediately visible in the dataset.
<code>roAfterUpdate</code>	The record is automatically re-fetched after the dataset has updated a record. This is only useful if a row level before update trigger can modify column values.

These options will generate one extra network roundtrip for each post. If there are no default values for a table and there are no triggers that modify columns, you should not enable these options.

Refreshing outdated records

When a user starts to edit a record in the dataset, the corresponding database record may have been changed by another users since the record was fetched. If the `LockingMode` is set to `ImCheckImmediate` or `ImLockImmediate`, the dataset will give an error message that the record has been changed by another user. To prevent this, you can enable the `roBeforeEdit` option. When the user starts to edit a record, it will be refreshed with the current values in the database.

When this option is enabled, the `LockingModes` `ImCheckImmediate` and `ImLockDelayed` are equivalent, as there is no need for any immediate check.

Refreshing derived fields

When a record is refreshed, by default only the fields belonging to the updating table will be fetched from the database. Derived fields that result from joins, function calls, calculations, and so on will therefore not be refreshed. If you want to refresh derived fields as well, enable the `roAllFields` option.

This is accomplished by re-executing the SQL statement for just the current record. This requires that the where clause is extended with the `rowid` of the current record. Let's assume the following SQL statement:

```
select emp.*, dept.dname, dept.loc
  from emp, dept
 where dept.deptno = emp.deptno
```

Before editing or after inserting or updating a record, the following statement will be executed to refresh all fields:

```
select emp.*, dept.dname, dept.loc
  from emp, dept
 where dept.deptno = emp.deptno
       and emp.rowid = :doa__rowid
```

If however the statement is a little bit more complicated, problems can occur:

```
select emp.*, dept.dname, dept.loc
  from emp, dept
 where dept.deptno = emp.deptno
       and emp.sal < 5000
```

If a record is inserted or updated that does not meet the criteria, it would not be fetched

anymore. Let's assume you update a record and set the 'sal' field to 6000. The following statement would no longer retrieve a record because `emp.sal < 5000` is no longer true:

```
select emp.*, dept.dname, dept.loc
  from emp, dept
 where dept.deptno = emp.deptno
        and emp.sal < 5000
        and emp.rowid = :doa__rowid
```

To prevent this problem, you can add an `/* END_REFRESH */` hint in the SQL statement that tells the dataset which part of the where clause to exclude from the refresh statement:

```
select emp.*, dept.dname, dept.loc
  from emp, dept
 where dept.deptno = emp.deptno
        /* END_REFRESH */
        and emp.sal < 5000
```

Note that the hint must be literally `/* END_REFRESH */`, and may not differ in any way.

TOracleDataSet.RefreshRecord

Declaration

```
procedure RefreshRecord;
```

Description

Refreshes the current record in the dataset by fetching it from the database.

See also

AfterRefreshRecord

OnApplyRecord

RefreshOptions

TOracleDataSet.RowId

Declaration

```
function RowId: string;
```

Description

Returns the hexadecimal notation (or 64-based on Oracle8) of the rowid of the current record. The RowId is used internally, and is therefore not available as a field.

TOracleDataSet.SearchRecord

Declaration

```
type TSearchRecordOption = (srForward, srBackward, srFromCurrent,
    srFromBeginning, srFromEnd, srIgnoreCase, srIgnoreTime,
    srPartialMatch);
type TSearchRecordOptions = set of TSearchRecordOption;
function SearchRecord(const FieldNames: string; const FieldValues:
    Variant; Options: TSearchRecordOptions): Boolean;
```

Description

To search for a record in an active dataset you can use the SearchRecord function. The parameters of this function have the following meaning:

FieldNames	The names of the fields on which to search. If you want to search on more than one field, separate the names with a semi-colon.																		
FieldValues	A variant array containing the values to match with the fields. If one field is specified you can simply pass the single value to match. If more than one field is specified, use the VarArrayOf constructor to specify the values, e.g. VarArrayOf(['Smith', 20]).																		
Options	A set of options that control various aspects of the search: <table><tr><td>srForward</td><td>Search in forward direction (Default)</td></tr><tr><td>srBackward</td><td>Search in backward direction</td></tr><tr><td>srFromCurrent</td><td>Start searching from the current record (Default)</td></tr><tr><td>srFromBeginning</td><td>Start searching from the first record</td></tr><tr><td>srFromEnd</td><td>Start searching from the last record. srBackward is default if you do not specify a direction.</td></tr><tr><td>srIgnoreCase</td><td>Ignore case when comparing string fields</td></tr><tr><td>srIgnoreTime</td><td>Ignore the time fraction when comparing date fields</td></tr><tr><td>srWildcards</td><td>Interpret wildcard characters * and ? when comparing strings</td></tr><tr><td>srPartialMatch</td><td>Search values only need to partially match the field values (e.g. field value 'SMITH' matches search value 'SM')</td></tr></table>	srForward	Search in forward direction (Default)	srBackward	Search in backward direction	srFromCurrent	Start searching from the current record (Default)	srFromBeginning	Start searching from the first record	srFromEnd	Start searching from the last record. srBackward is default if you do not specify a direction.	srIgnoreCase	Ignore case when comparing string fields	srIgnoreTime	Ignore the time fraction when comparing date fields	srWildcards	Interpret wildcard characters * and ? when comparing strings	srPartialMatch	Search values only need to partially match the field values (e.g. field value 'SMITH' matches search value 'SM')
srForward	Search in forward direction (Default)																		
srBackward	Search in backward direction																		
srFromCurrent	Start searching from the current record (Default)																		
srFromBeginning	Start searching from the first record																		
srFromEnd	Start searching from the last record. srBackward is default if you do not specify a direction.																		
srIgnoreCase	Ignore case when comparing string fields																		
srIgnoreTime	Ignore the time fraction when comparing date fields																		
srWildcards	Interpret wildcard characters * and ? when comparing strings																		
srPartialMatch	Search values only need to partially match the field values (e.g. field value 'SMITH' matches search value 'SM')																		
Result	True if a matching record was found, otherwise False. If a record is found, it will become the current record in the dataset.																		

The following example searches for the last occurrence of an employee whose name starts with 'A' in department 20:

```
Found := SearchRecord('ename;deptno',
    VarArrayOf(['A*', 20]),
    [srFromEnd, srWildCards]);
```

TOracleDataSet.SequenceField

Declaration

```
property SequenceField: TSequenceField;
```

Description

The SequenceField property can be used to assign an Oracle sequence to a field in the dataset. When you double-click this property at design time, it brings up a property editor where you can select the sequence, the field, and the moment when the sequence's next value will be applied to the field:

- ♦ On New Record When the user creates a new empty record in the dataset.
- ♦ On Post When a new record is posted to the dataset.
- ♦ On Server The <sequence>.nextval expression will be part of the insert statement that is executed on the server. This is the most efficient moment if the RefreshOptions of the dataset includes roAfterInsert, as it will require one less network roundtrip than the other two moments.

When the field is a master field, you cannot apply a sequence 'On Server', because the detail dataset needs to know the master fields before that.

Note

In most cases a sequence is used for a primary key column, which by definition results in a required field. If the ApplyMoment is 'On Post' or 'On Server', you must set the required property of the field to False, because the user does not need to supply a value for the field before it is posted.

TOracleDataSet.Session

Declaration

```
property Session: TOracleSession;
```

Description

The session in which the dataset will execute.

TOracleDataSet.SetComplexVariable

Declaration

```
procedure SetComplexVariable(Name: string; const Value: TObject);
```

Description

Assign an object instance to a complex variable. See TOracleQuery.SetComplexVariable for details.

This procedure is only useful if you are using the ExecSQL procedure.

TOracleDataSet.SetLongVariable

Declaration

```
procedure SetLongVariable(Name: string; Buffer: Pointer; Length: Integer);
```

Description

Low level procedure to set the value of the specified long or long raw variable to the address pointed to by <Buffer> with length <Length>. Memory is not copied to minimize overhead, so the address must remain valid until the query is executed. No CR/LF => LF conversion is performed for long variables.

This procedure is only useful if you are using the ExecSQL procedure.

See also

TOracleQuery.SetLongVariable

TOracleDataSet.SetQueryVariables

Declaration

```
procedure SetQueryVariables(AQuery: TOracleQuery);
```

Description

This procedure can be used in an OnApplyRecord event handler to set all variable values in a query to the corresponding fields values in the dataset. Variables and fields are matched by their names. You can also prefix a variable name with 'old_', in which case the field's OldValue will be used. Variables that do not match any dataset field will not be set.

See also

OnApplyRecord

DeclareQueryVariables

GetQueryVariables

CompareQueryVariables

TOracleDataSet.SetVariable

Declaration

```
procedure SetVariable(Name: string; Value: Variant);
```

Description

Sets the value of the specified variable. In Delphi 4 and later you can also specify the variable by its zero-based index.

See also

Variables

DeclareVariable

GetVariable

TOracleDataSet.SQL

Declaration

property SQL: TStrings;

Description

The SQL-text for the select statement. You can also specify a PL/SQL block to call a procedure that opens a cursor variable.

See TOracleQuery.SQL for details.

TOracleDataSet.StringFieldsOnly

Declaration

property StringFieldsOnly: Boolean;

Description

When defining fields for a select statement, the TOracleDataSet component will use the appropriate field data types. A varchar2 field is defined as TStringField, a number field is defined as TIntegerField or TFloatField, and a date is defined as TDateField. When the StringFieldsOnly property is set to True, all fields will be defined as TStringFields. Numbers and dates will be converted to strings on the server, as defined by the NLS_LANG settings of the current session.

TOracleDataSet.SubstitutedSQL

Declaration

function SubstitutedSQL: string;

Description

If you are using substitution variables, you can retrieve the SQL text with substituted variables by calling the SubstitutedSQL function.

See also

Variables

TOracleDataSet.UniDirectional

Declaration

property UniDirectional: Boolean;

Description

When True, the TOracleDataSet will no longer buffer previously fetched records. As a result there will not be any memory overhead when processing large numbers of records, which can be useful for batch processing or reporting functionality.

Note

A UniDirectional dataset can not be used for functions that require navigation to prior records

or to the first record, such as a TDBGrid or a TDBNavigator. When these navigation functions are used, an exception will be raised.

TOracleDataSet.UniqueFields

Declaration

```
property UniqueFields: string;
```

Description

If you want to use an updateable dataset for a view with an "instead of" triggers, you must use the UniqueFields property. The problem with "instead of" trigger is that the Oracle Server does not return the RowId of a newly created record, and that there is no way to determine the fields that identify a record of the underlying tables. Specifying the names of these fields in the UniqueFields property allows the dataset to determine the RowId of the record. If more than one field identifies a record in the view, separate their names with a semi-colon.

TOracleDataSet.UpdatesPending

Declaration

```
property UpdatesPending: Boolean;
```

Description

Indicates if there are any cached updates that are not yet applied to the database.

TOracleDataSet.UpdateStatus

Declaration

```
type TUpdateStatus = (usUnmodified, usInserted, usModified,  
    usDeleted)  
property UpdateStatus: TUpdateStatus;
```

Description

Returns the status of the current record in the dataset.

TOracleDataSet.UpdatingTable

Declaration

```
property UpdatingTable: string;
```

Description

Specifies the name of the updating table when the dataset is based on a cursor variable.

In case of a select statement that uses multiple tables, you can use this property to specify which table will be updated. If left blank, the first table after the from clause will be used.

TOracleDataSet.VariableCount

Declaration

```
function VariableCount: Integer;
```

Description

Returns the number of variables that are declared. Useful for iterating variables.

See also

VariableName

VariableType

VariableIndex

TOracleDataSet.VariableIndex

Declaration

```
function VariableIndex(const AName: string): Integer;
```

Description

Use this function to determine the index of a variable by its name. The result is the zero based index of the variable. If the variable does not exists, -1 is returned.

See also

VariableCount

VariableName

VariableType

TOracleDataSet.VariableName

Declaration

```
function VariableName(Index: Integer): string;
```

Description

Returns the name of the specified variable. Index is zero-based.

See also

VariableCount

VariableType

VariableIndex

TOracleDataSet.Variables

Declaration

```
property Variables: TVariables;
```

Description

The variables property is only accessed at design time. It will invoke a property editor, which enables you to declare variables manually, or scan for variables in the SQL text. At run time, you access variables through the SetVariable, GetVariable and DeclareVariable methods.

See TOracleQuery.Variables for details.

See also

DeclareVariable

SetVariable

GetVariable

DeleteVariable

DeleteVariables

ClearVariables

TOracleDataSet.VariableType

Declaration

```
function VariableType(Index: Integer): Integer;
```

Description

Returns the type (otString, otInteger, ...) of the specified variable. Index is zero-based.

See also

VariableCount

VariableName

VariableIndex

TQBEDefinition object

The TQBEDefinition object controls the behavior of a TOracleDataSet during QBE mode. At design time, you can use the QBEDefinition property editor to set the various properties. At run time you can also use the QBEDefinition property to access the dataset and field level QBE properties. The TQBEDefinition exposes all the necessary properties and methods.

The AllowFileWildCards, AllowOperators, and SaveQBESValues control QBE behavior at the dataset level, whereas you can use the Fields property or FieldByName method to control field-level QBE behavior.

The BackgroundColor and FontColor properties provide an easy way to present a visual clue for the end-user that a dataset is in QBE mode.

TQBEDefinition reference

This chapter describes all properties and methods of the TQBEDefinition object.

TQBEDefinition.AllowFileWildCards

Declaration

property AllowFileWildCards: Boolean;

Description

When True, the familiar file wildcard characters (* and ?) are accepted besides the SQL wildcard characters (%) and (_). These characters are converted to SQL wildcard characters when the query is executed.

TQBEDefinition.AllowOperators

Declaration

property AllowOperators: Boolean;

Description

When enabled, the user can enter operators in the QBE fields. These operators are

- ♦ *not value* (equivalent to *<> value* or *!= value*)
- ♦ *> value*
- ♦ *< value*
- ♦ *= value*
- ♦ (not) *like value*
- ♦ (not) *between value-1 and value-2*
- ♦ (not) *in (value-1, value-2, ..., value-n)*
- ♦ (not) *value-1 or value-2 or ... or value-n*

These operators can be used for string fields, number fields and date fields. As a result, you can not access the values of the fields in QBE mode, because the user input does not translate to a value of the actual type of the field. To access the data as a string, use the QBEField.Value property instead, which is a string value that represents the actual expression that the user has entered.

TQBEDefinition.BackgroundColor

Declaration

property BackgroundColor: TColor;

Description

When the dataset enters QBEMode, the background color of all data-aware controls that are

linked will change to the color specified in this property. The following 2 conditions must be met before the background color of a control is changed:

- ♦ The BackgroundColor property must not be clNone (the default value).
- ♦ The current background color of a control must be clWindow.

Providing a visual clue that a dataset is in QBE Mode is important, because it prevents the common mistake that you think you are inserting a new record, but are in fact entering QBE criteria, or vice versa.

TQBEDefinition.DataSet

Declaration

```
property DataSet: TOracleDataSet;
```

Description

The dataset that the TQBEDefinition instance belongs to.

TQBEDefinition.FieldByName

Declaration

```
function FieldByName(const AName: string): TQBField;
```

Description

Returns the QBE Field for the given field name. This allows you to control the QBE behavior for individual fields. Note that the QBE Fields are instantiated when the FieldDefs of the dataset are created. Therefore the dataset must be active, or you must have called FieldDefs.Update.

If a field of the given name does not exist in the dataset, FieldByName will return nil.

TQBEDefinition.FieldCount

Declaration

```
property FieldCount: Integer;
```

Description

The number of QBE Fields.

TQBEDefinition.Fields

Declaration

```
property Fields[Index: Integer]: TQBField;
```

Description

The zero-based array of QBE Fields. This allows you to control the QBE behavior for

individual fields. Note that the QBE Fields are instantiated when the FieldDefs of the dataset are created. Therefore the dataset must be active, or you must have called FieldDefs.Update. Use the FieldCount property to determine the size of the Fields array.

The following example makes all fields of a dataset queryable:

```
for i := 0 to EmpDataSet.QBEDefinition.FieldCount - 1 do  
    EmpDataSet.QBEDefinition.Fields[i].Queryable := True;
```

To modify a specific QBE Field, you can use the FieldByName method.

TQBEDefinition.FontColor

Declaration

```
property FontColor: TColor;
```

Description

When the dataset enters QBEMode, the font color of all data-aware controls that are linked will change to the color specified in this property. The following 2 conditions must be met before the font color of a control is changed:

- The FontColor property must not be clNone (the default value).
- The current font color of a control must be clWindowText.

Providing a visual clue that a dataset is in QBE Mode is important, because it prevents the common mistake that you think you are inserting a new record, but are in fact entering QBE criteria, or vice versa.

TQBEDefinition.SaveQBEValues

Declaration

```
property SaveQBEValues: Boolean;
```

Description

When a query is executed, the QBE values are saved and restored the next time that the dataset goes in QBE mode. If this property is False, the fields will be empty after entering QBE mode.

TQBField object

The TQBField object allows you to control QBE behavior at the field level. For each field in the dataset, there will be a corresponding TQBField instance with the same FieldName. You can access the TQBField instances of a TOracleDataSet through the Fields property or FieldByName method of the QBEDefinition property.

TQBEField reference

This chapter describes all properties and methods of the TQBEField object.

TQBEField.AutoContains

Declaration

```
property AutoContains: Boolean;
```

Description

Defines if the database field only needs to contain the QBE Value. If for example the user enters 'mi', both Smith and Jamison are selected. This option is only useful for string fields.

TQBEField.AutoPartialMatch

Declaration

```
property AutoPartialMatch: Boolean;
```

Description

Defines if the database field only needs to partially match the QBE value. If for example the user enters 'Jo', both Jones and Johnson are selected. This option is only useful for string fields.

TQBEField.CaseInsensitive

Declaration

```
property CaseInsensitive: Boolean;
```

Description

Defines that the field will be compared case insensitive. When this option is enabled, query performance may decrease if the field is indexed, because this index can only partially be used. This option is only useful for string fields.

TQBEField.FieldName

Declaration

```
property FieldName: string;
```

Description

The name of the dataset field that this QBE Field corresponds to.

TQBField.IgnoreTime

Declaration

```
property IgnoreTime: Boolean;
```

Description

Defines that for the database field any time fraction will be ignored. This is particularly useful when querying timestamp columns where the user does not know the exact time, but does know the date. If the user specifies a time fraction in the query field, time fractions will not be ignored and the values must exactly match. This option is only useful for date fields.

TQBField.LastValue

Declaration

```
property LastValue: Variant;
```

Description

Indicates the last QBE value that was used. If you have enabled the SaveQBValues property of the QBEDefinition, you can also set this value to control the values that are initially presented when the dataset enters QBE Mode.

TQBField.Queryable

Declaration

```
property Queryable: Boolean;
```

Description

Defines if the field is queryable. If not, the field will be read-only in QBE mode. BLOB fields cannot be queryable, and this option will be ignored for these fields.

TQBField.Value

Declaration

```
property Value: Variant;
```

Description

Contains the QBE value that has been entered by the user. Use this property to examine a field value when AllowOperators is enabled, because the value of the fields will be empty in this situation.

TOracleNavigator component

Unit

OracleNavigator

Description

The TOracleNavigator component (a database navigator) is used to move through the data in a TOracleDataSet and perform operations on the data, such as inserting a blank record or posting a record. It is derived from the standard TDBNavigator, and adds three buttons to support the QBE mode (Query By Example) of the TOracleDataSet component:



From left to right:

- | | |
|----------------|---|
| Enter Query | The TOracleDataSet that is linked to the related DataSource will go in QBE mode. The button will stay down to indicate that query values can be entered in the data-aware controls that are related to this DataSource. When the button is pressed again, QBE mode will be cancelled. |
| Execute Query | A query will be executed that returns those records that meet the QBE query criteria. These records can subsequently be viewed or modified. |
| Refresh Record | Reresh the current record by calling RefreshRecord. |

Note that these three buttons are only enabled when the DataSource of the TOracleNavigator is linked to a TOracleDataSet component. The standard refresh button can be used to switch between an empty query record and a record with the previously used query values.

All properties, methods and events are exactly the same as with the standard TDBNavigator. See the Delphi or C++Builder documentation for detailed information. The 3 additional QBE buttons are named nbEnterQBE, nbExecuteQBE and nbRefreshRecord.

TOraclePackage component

Unit

Oracle

Description

The package is probably the most valuable concept of the procedural extension of the Oracle database. It allows a developer to encapsulate functions, procedures, type definitions, variables and constants into a single program unit. Moreover, a package separates the interface (specification) from the implementation (body), which allows for private and public definitions and can prevent dependency problems. All of this is very appealing to Delphi developers, who are used to the same concepts with Delphi's units.

To make access to functions, procedures, variables and constants in a package as easy as possible, Direct Oracle Access offers the TOraclePackage component. You only need to set the Session property, specify the name of the package in the database, and you are ready to use it. No additional definitions are required for the objects inside the package. To call procedures or functions, you can simply use the CallProcedure and Call...Function methods. To access variables or constants, you can use the SetVariable and Get...Variable methods.

A TOraclePackage allows you to make use of boolean functions, parameters, constants and variables. SQL*Net does not support this datatype, but booleans are automatically converted to integers by the TOraclePackage component when passing them across the network.

As a result of this simplified interface, complex parameters such as LOB Locators, Objects and Cursors cannot be used. You can use a TOracleQuery with an appropriate PL/SQL Block and appropriate variables instead, or you can use the Package Wizard.

Note

If you make a lot of use of packages in your application, you should consider using the Package Wizard instead of the TOraclePackage component. The effort to generate custom package classes will easily be made up by the advantages.

Example - Using a package

To demonstrate the TOraclePackage component, this example uses the standard package sys.dbms_pipe. This application reads a message pipe and displays all messages in a memo.

Declaring the package

To use the dbms_pipe package, we create a TOraclePackage component, set the Session property to the appropriate session, set the PackageName property to 'dbms_pipe' (or 'sys.dbms_pipe' if a public synonym does not exist), and set the Name property to DbmsPipe. In this example we want to use named parameters, so we set the ParameterMode property to pmNamed.

Calling a function

To receive a message in pipe 'demo_pipe', we need to use the Call...Function method to call dbms_pipe.receive_message and pass the name of the pipe and a timeout value of 60 seconds. The result of the function indicates if we received a message, a timeout, or some error:

```

with DbmsPipe do
try
  Status := CallIntegerFunction('receive_message', ['pipename',
    'demo_pipe', 'timeout', 60]);
  case Status of
    0: AddMessageToMemo;
    1: ShowMessage('Timeout');
    2: ShowMessage('Record in pipe too big for buffer');
    3: ShowMessage('Interrupted');
  end;
except
  on E:EOraclError do ShowMessage(E.Message);
end;

```

The parameter names ('pipename' and 'timeout') and values ('demo_pipe' and 60) are passed directly as an open array constructor. The type of the value implicitly declares the type of the parameter. If a function has no parameters, you need to use the constant parNone, because an empty open array constructor [] does not exist in Delphi:

```
s := DbmsPipe.CallStringFunction('unique_session_name', parNone);
```

Output parameters

Because the parameters are passed as an open array constructor, it might seem that out or in/out parameters are not supported. Output parameters can be declared during the function or procedure call by passing a parString, parInteger, parFloat, parDate or parBoolean constant instead of a value. Values of output parameters can be retrieved after a function or procedure call with the GetParameter method. In the example we need to call the dbms_pipe.unpack_message procedure to retrieve a string item from the message:

```

with DbmsPipe do
try
  CallProcedure('unpack_message', ['item', parString]);
  Memo.Items.Add(GetParameter(0));
except
  on E:EOraclError do ShowMessage(E.Message);
end;

```

For an in/out parameter, the type of the value that is assigned to it on input implicitly declares the type of the parameter. If you want to pass a null to an in/out parameter (which is untyped), you must use a par... constant instead of Null.

Packaged variables and constants

To retrieve the value of a variable or constant, you can use the Get...Variable method:

```
MaxWait := DbmsPipe.GetIntegerVariable('maxwait');
```

To set the value of a variable, you can use the SetVariable method:

```
MyPackage.SetVariable('factor', 12.55);
```

TOraclePackage reference

This chapter describes all properties, methods and events of the TOraclePackage component.

TOraclePackage.CallBooleanFunction

Declaration

```
function CallBooleanFunction(const FunctionName: string; const
    Parameters: array of Variant): Variant;
```

Description

Calls the specified function and passes the parameters to it. The result of the function will be converted to a boolean value on the server.

See also

ParameterMode

TOraclePackage.CallDateFunction

Declaration

```
function CallDateFunction(const FunctionName: string; const
    Parameters: array of Variant): Variant;
```

Description

Calls the specified function and passes the parameters to it. The result of the function will be converted to a date value on the server.

See also

ParameterMode

TOraclePackage.CallFloatFunction

Declaration

```
function CallFloatFunction(const FunctionName: string; const
    Parameters: array of Variant): Variant;
```

Description

Calls the specified function and passes the parameters to it. The result of the function will be converted to a floating-point value on the server.

See also

ParameterMode

TOraclePackage.CallIntegerFunction

Declaration

```
function CallIntegerFunction(const FunctionName: string; const
    Parameters: array of Variant): Variant;
```

Description

Calls the specified function and passes the parameters to it. The result of the function will be converted to an integer value on the server.

See also

ParameterMode

TOraclePackage.CallProcedure

Declaration

```
procedure CallProcedure(const ProcedureName: string; const
    Parameters: array of Variant);
```

Description

Calls the specified procedure and passes the parameters to it.

See also

GetParameter

ParameterMode

TOraclePackage.CallStringFunction

Declaration

```
function CallStringFunction(const FunctionName: string; const
    Parameters: array of Variant): Variant;
```

Description

Calls the specified function and passes the parameters to it. The result of the function will be converted to a string value on the server.

See also

ParameterMode

TOraclePackage.Cursor

Declaration

```
property Cursor: TCursor;
```

Description

Determines the shape of the mouse cursor when the package is accessed.

TOraclePackage.Debug

Declaration

```
property Debug: Boolean;
```

Description

When set to true the PL/SQL block and all variable values will be displayed when the package is accessed.

TOraclePackage.GetBooleanVariable

Declaration

```
function GetBooleanVariable(const VariableName: string): Variant
```

Description

Retrieves the value of the specified variable, which will be converted to a boolean value on the server.

TOraclePackage.GetDateVariable

Declaration

```
function GetDateVariable(const VariableName: string): Variant
```

Description

Retrieves the value of the specified variable, which will be converted to a date value on the server.

TOraclePackage.GetFloatVariable

Declaration

```
function GetFloatVariable(const VariableName: string): Variant
```

Description

Retrieves the value of the specified variable, which will be converted to a floating-point value on the server.

TOraclePackage.GetIntegerVariable

Declaration

```
function GetIntegerVariable(const VariableName: string): Variant
```

Description

Retrieves the value of the specified variable, which will be converted to an integer value on the server.

TOraclePackage.GetParameter

Declaration

```
function GetParameter(const ParameterId: Variant): Variant;
```

Description

Retrieves the value of an out or in/out parameter after a procedure or function call. The index is the zero-based position of the parameter. If you are using a named parameter mode, you can also specify the name of the parameter.

In Delphi 4 or later this function is overloaded and accepts the index or name of the parameter.

TOraclePackage.GetStringVariable

Declaration

```
function GetStringVariable(const VariableName: string): Variant
```

Description

Retrieves the value of the specified variable, which will be converted to a string value on the server.

TOraclePackage.Optimize

Declaration

```
property Optimize: Boolean;
```

Description

When this property is set to True, multiple calls to the same procedure or function will only cause a single parse on the server.

TOraclePackage.PackageName

Declaration

```
property PackageName: string;
```

Description

The name of the package in the database.

TOraclePackage.ParameterMode

Declaration

```
type TParameterModeOption = (pmNamed, pmPositional);  
property ParameterMode: TParameterModeOption;
```

Description

The method to pass parameters to functions and procedures in the package: named or positional. When this property is set to pmPositional, a string that specifies the name must

precede each parameter that is passed to a function or procedure.

ParameterMode = pmNamed

```
IsDBA := dbms_session.CallBooleanFunction('is_role_enabled', ['role',  
  'DBA']);
```

ParameterMode = pmPositional

```
IsDBA := dbms_session.CallBooleanFunction('is_role_enabled',  
  ['DBA']);
```

TOraclePackage.Session

Declaration

```
property Session: TOracleSession;
```

Description

The session in which the package will be accessed.

TOraclePackage.SetVariable

Declaration

```
procedure SetVariable(const VariableName: string; const Value:  
  Variant)
```

Description

Sets the packaged variable to the specified value. The type of the value variant will be converted to the type of the variable on the server.

TOracleEvent component

Unit

Oracle

Description

The TOracleEvent component can be used in an application that needs to react to dbms_alert signals or dbms_pipe messages. These signals and messages are typically generated in database triggers or server processes to pass information to other database sessions. The TOracleEvent component works in a separate execution thread in the background, without interfering with the normal program flow of your application. When an event occurs, the OnEvent event handler is called, which is synchronized with the main thread of the application.

Declaring the TOracleEvent

After creating a TOracleEvent in your application, link it to a TOracleSession component by setting the Session property. Note that this TOracleSession will be duplicated and will not be used otherwise and that the original session does not even have to be connected. This way, no interference will occur with other database access in your application. To wait for dbms_pipe messages, set the ObjectType property to otPipe, and specify the name of the pipe in the ObjectNames property. To wait for dbms_alert signals, set the ObjectType property to otAlert, and specify one or more signal names in the ObjectNames property separated by semicolons. Write an OnEvent event handler that reacts to the events. If you want to know that no event has occurred in a certain amount of time, set the TimeOut property to a non-zero value, and write an OnTimeOut event handler.

Using the TOracleEvent

After calling the Start method of the TOracleEvent, it will create a new execution thread, log on with the duplicated session, and will start to wait for the event. Each time an event occurs, the OnEvent event handler will be called. To stop a TOracleEvent, call the Stop method. When a TOracleEvent component is destroyed (e.g. when a form or data module is destroyed), the Stop method is automatically called. The stop method will also logoff the duplicated session.

Pipes or Alerts?

If you need to decide between using dbms_pipe messages and dbms_alert signals, you might consider the following aspects:

- ♦ Signals are part of a transaction, messages are not. Therefore, a signal is visible in the TOracleEvent component when the transaction is committed, whereas a message is immediately visible and cannot be rolled back.
- ♦ A single signal can be received by multiple sessions, a single message will only be received by one session. Therefore, it usually only makes sense to have only one session receiving messages of a certain pipe.
- ♦ A message pipe can become full when no session is receiving messages, eventually blocking the session that sends messages in this pipe.

TOracleEvent reference

This chapter describes all properties, methods and events of the TOracleEvent component.

TOracleEvent.InternalSession

Declaration

```
property InternalSession: TOracleSession;
```

Description

When the TOracleEvent is started, it will create a new internal session that will be used to wait for the database events. This run time property provides access to this internal session, which can be used within the various event handlers.

TOracleEvent.KeepConnection

Declaration

```
property KeepConnection: Boolean;
```

Description

When true, the internal session of the TOracleEvent component will remain connected after calling the Stop procedure. This can be useful if you frequently call Start and Stop, because this will prevent frequent logon/logoff operations. To explicitly logoff the internal session, call LogOff. Freeing the TOracleEvent will also implicitly logoff the internal session.

TOracleEvent.LogOff

Declaration

```
procedure LogOff;
```

Description

Calling this procedure will logoff the internal session. This is only useful if the KeepConnection property is set to True. Freeing the TOracleEvent will also implicitly logoff the internal session.

TOracleEvent.ObjectNames

Declaration

```
property ObjectNames: string;
```

Description

The meaning of this property depends on the value of the ObjectType property:

- otPipe The ObjectNames property defines the name of the pipe.
- otAlert The ObjectNames property defines the name(s) of the signal(s). In case of multiple signals, separate the names with semicolons. When one of multiple

signals occurs, the OnEvent handler will provide the actual signal name.

TOracleEvent.ObjectType

Declaration

```
type TEventObjectType = (otPipe, otAlert);;
property ObjectType: TEventObjectType;
```

Description

Determines the object type of the event:

- | | |
|---------|---|
| otPipe | The object is a dbms_pipe, and the ObjectNames property contains the name of a pipe. |
| otAlert | The object is a dbms_alert, and the ObjectNames property contains the name of one or more signals, separated by semicolons. |

TOracleEvent.OnError

Declaration

```
type TOnErrorEvent = procedure(Sender: TOracleEvent; const
    Error: Exception) of object;
property OnError: TOnErrorEvent;
```

Description

Triggered when an exception occurs during event handling.

TOracleEvent.OnEvent

Declaration

```
type TOnEventEvent = procedure(Sender: TOracleEvent; const
    ObjectName: string; const Info: Variant) of object;
property OnEvent: TOnEventEvent;
```

Description

Triggered when the specified event occurs. When Synchronized is True, this event handler will be synchronized with the main thread of the application. When Synchronized is False, this event handler must be thread-safe.

The ObjectName parameter contains the name of the pipe or signal. This information is only relevant if you have entered multiple signal names in the ObjectNames property.

The Info parameter is a zero-based array of variants. In case of a dbms_pipe message it contains all the items that were packed into the message when dbms_pipe.send_message was called. You may expect strings, doubles and TDateTime elements in the array. If no items were packed into the message, the Info parameter will be Null. In case of a dbms_alert signal, the Info array always contains one string element that is the message that was passed with the dbms_alert.signal call.

The following is an example of a general event handler that will display all event info from any TOracleEvent component in a memo:

```
procedure TMyForm.OnEvent(Sender: TOracleEvent; const ObjectName:
    string; const Info: Variant);
var i: Integer;
begin
    Memo.Lines.Add('Event from ' + Sender.Name + ' on ' + ObjectName);
    if VarIsArray(Info) then
        for i := 0 to VarArrayHighBound(Info, 1) do
            Memo.Lines.Add(Info[i]);
end;
```

TOracleEvent.OnStart

Declaration

```
type TEventEvent = procedure(Sender: TOracleEvent) of object;
property OnStart: TEventEvent;
```

Description

Triggered when the TOracleEvent instance starts.

TOracleEvent.OnStop

Declaration

```
type TEventEvent = procedure(Sender: TOracleEvent) of object;
property OnStop: TEventEvent;
```

Description

Triggered when the TOracleEvent instance stops, either because the Stop procedure is called, the TOracleEvent instance is freed, or if some exception has occurred that stops event handling.

TOracleEvent.OnTimeOut

Declaration

```
type TOnTimeOutEvent = procedure(Sender: TOracleEvent; var Continue:
    Boolean) of Object;
property OnTimeOut: TOnTimeOutEvent;
```

Description

Triggered when the number of seconds specified by the TimeOut property have passed since the last event occurred. When Synchronized is True, this event handler will be synchronized with the main thread of the application. When Synchronized is False, this event handler must be thread-safe.

The Continue boolean can be set to False by the event handler to stop the TOracleEvent component.

TOracleEvent.Session

Declaration

```
property Session: TOracleSession;
```

Description

The session that will be used to wait for the event to occur. Note that this session will be duplicated to avoid any interference with database access in the rest of the application, and that the original session does not have to be connected. The duplicated internal session can be accessed in the event handlers through the InternalSession property.

TOracleEvent.Start

Declaration

```
procedure Start;
```

Description

Starts the TOracleEvent component. The Session will be duplicated and logged on, and the component will start to wait for the specified event to occur.

TOracleEvent.Started

Declaration

```
property Started: Boolean;
```

Description

Indicates if the TOracleEvent component is currently waiting for an event.

TOracleEvent.Stop

Declaration

```
procedure Stop;
```

Description

Stops the TOracleEvent component. This method is automatically called when the component is destroyed (e.g. when its parent form or module is destroyed). The internal session will be logged off, unless you have set the KeepConnection property to True.

TOracleEvent.Synchronized

Declaration

```
property Synchronized: Boolean;
```

Description

Determines if the OnEvent and OnTimeOut event handlers are synchronized with the main thread of the application.

When the event handlers are synchronized, you don't need to worry about the fact if the code in these event handlers is thread-safe. They will execute as if they were a normal part of the message loop of your application.

When the event handlers are not synchronized, they will be executed immediately when the event occurs, parallel with the main thread of the application. The event handlers must be thread-safe, so you need to pay special attention to code that accesses global variables and code that performs screen output.

TOracleEvent.TimeOut

Declaration

property TimeOut: Integer;

Description

Determines how many seconds can pass without the occurrence of an event before the OnTimeOut event handler is called. If the value is 0, no OnTimeOut event will occur.

TLOBLocator

Unit

Oracle

Description

Oracle8 has introduced a new and more flexible long datatype: the LOB (Large Object). Currently, there are 3 different kinds of LOB's:

- ♦ CLOB Character LOB, analogous to a Long
- ♦ BLOB Binary LOB, analogous to a Long Raw
- ♦ BFile Binary File, to access files on the database server

You rarely access the data of a LOB column directly. Instead, you will use a LOB Locator to read or write the data of such a column. A LOB Locator is stored in the record that contains a LOB column and points to the actual data. When you include a LOB column in a select list, you will actually just fetch the LOB Locator, not the data. The LOB Locator is encapsulated in the TLOBLocator object, which gives you full access to all the powerful features of LOB's.

When a LOB column contains a null value, the record does not contain a LOB Locator. Therefore, when selecting a null LOB, you cannot access the LOB data. When updating a null LOB, you first need to update the column to contain an empty LOB (which is something completely different than a null LOB) and use this newly created LOB Locator to write the actual data. When inserting a new record with LOB columns, the same mechanism applies. Temporary LOB's, created through the CreateTemporary constructor, do not have this limitation. You can create the LOB, write data to it, and subsequently use it for Inserts, Updates and PL/SQL calls.

You can obtain a TLOBlocator object in three ways:

- ♦ By creating one with TLOBLocator.Create or TLOBLocator.CreateTemporary
- ♦ By selecting a LOB column and using the TOracleQuery.LOBField method
- ♦ By accessing a LOB attribute of an object through the TOracleObject.LOBAttr method

Because the TLOBLocator is a descendant of the TStream object, you can use the familiar Read, Write and Seek methods to access the data of the LOB, as well as the Size property.

Besides using the TLOBLocator object, Direct Oracle Access allows you to access the data of a LOB in two more ways:

- ♦ Use the TOracleQuery.Field method after a select statement. This works identical to Long and Long Raws. These methods use the LOB Locator that is fetched in the select statement to retrieve the LOB data.
- ♦ The TOracleDataSet will transparently handle a LOB as a Delphi BLOB field. The TemporaryLOB Preference defines at the session level if and how temporary LOB's will be used when posting dataset records with LOB fields.

For additional information about LOB's, you can read Oracle's *"Server Application Developer's Guide"*.

Example - Selecting a LOB Locator

After executing a select statement with one or more LOB columns, you can use the `Field` method to access the data. CLOB's will be returned as a string, BLOB's and BFiles will be returned as a zero based variant array of bytes. You can access a specific piece of the LOB data by using the `GetLongField` method. All of this is exactly the same as accessing a Long or Long Raw column. Note that the actual data of the LOB is only fetched from the server at the moment that you access it. The data is not pre-fetched or buffered on the client.

You can also obtain the LOB Locator of the selected LOB column. To do so, use the `LOBField` method of the `TOracleQuery`. This method takes the name or index of the LOB field as a parameter, and returns a `TLOBLocator` instance that you can subsequently use to access the LOB. You can test if the LOB column is null by using the `IsNull` method of the `TLOBLocator`. If it is null, you cannot access the LOB data.

The `TLOBLocator` object is a `TStream` descendant, so you can use the familiar `Seek`, `Read`, and `Size` methods to retrieve the data of the LOB. The following is an example of how to retrieve the last 100 bytes of a LOB:

```
var LOB: TLOBLocator;
    Buffer: array[0..99] of Byte;
begin
    // select lobcolumn from lobtable where id = 1
    with LOBQuery do
        begin
            Execute;
            LOB := LOBField('lobcolumn');
            if not LOB.IsNull then
                begin
                    LOB.Seek(-100, soFromEnd);
                    LOB.Read(Buffer, 100);
                end;
            end;
        end;
end;
```

Example - Updating LOB data

To update LOB data, you must obtain the `TLOBLocator` and use it to write the data. This is not much different than selecting a LOB, except that you must now lock the record with the LOB. You can do so by using a 'select ... for update'. After writing the data, you can use the `Size` property or the `Trim` method to remove any remaining data that the LOB contained before the update:

```

var LOB: TLOBLocator;
    Buffer: array[0..99] of Byte;
begin
    // select id, lobcolumn from lobtable for update
    with LOBQuery do
        begin
            Execute;
            LOB := LOBField('LOBCOLUMN');
            LOB.Write(Buffer, 100);
            LOB.Trim; // Set the size to the current position, which is 100
        end;
    end;
end;

```

The TLOBLocator has 3 additional update methods:

- ♦ Copy(Source: TLOBLocator; Length: Integer)
Copies <Length> bytes of the data from the current position of <Source> to the current position of the TLOBLocator.
- ♦ Append(Source: TLOBLocator)
Appends all the data of <Source> to the end of the TLOBLocator.
- ♦ Erase(Length: Integer)
Fills <Length> bytes at the current position of the TLOBLocator with zeroes (BLOB) or spaces (CLOB).

No LOB data is passed over the network for these methods, all work is done on the server.

Example - Inserting LOB data

You need to insert new LOB data into a table in two situations:

1. You are inserting a new record.
2. You are updating a record where the old value of the LOB column is null. In this case, the record does not contain a LOB Locator that you can use to write data to.

If you are not using a temporary LOB, you must first insert/update an empty LOB Locator into the record. To do so, you can use the SQL function `empty_blob()` or `empty_clob()` in an insert or update statement. On the server, the LOB Locator will be initialized. The initialized LOB Locator can then be returned to the client in a variable by using the new Oracle8 returning clause. After this, you can start writing data to the LOB column. Note that you must use `SetComplexVariable` to set a LOB variable.

The following is an example of an insert:

```

var LOB: TLOBLocator;
    Buffer: array[0..99] of Byte;
begin
    // insert into lobtable (id, lobcolumn) values (:id, empty_blob())
    // returning lobcolumn into :lobcolumn
    with LOBQuery do
        begin
            SetVariable('id', 1);
            // Create a new BLOB (initially Null)
            LOB := TLOBLocator.Create(Session, otBLOB);
            // Assign it to the returning variable
            SetComplexVariable('lobcolumn', LOB);
            Execute;
            // After the insert, use the LOB Locator to write the data
            LOB.Write(Buffer, 100);
            LOB.Free;
        end;
    end;
end;

```

If you are using temporary LOB's, you can write the LOB data before executing the query, and directly use this LOB data for an insert or update without a returning clause:

```

var LOB: TLOBLocator;
    Buffer: array[0..99] of Byte;
begin
    // insert into lobtable (id, lobcolumn) values (:id, :lobcolumn)
    with LOBQuery do
        begin
            SetVariable('id', 1);
            // Create a new temporary BLOB and write the data
            LOB := TLOBLocator.CreateTemporary(Session, otBLOB, True);
            LOB.Write(Buffer, 100);
            // Assign it to the returning variable
            SetComplexVariable('lobcolumn', LOB);
            // Insert it
            Execute;
            LOB.Free;
        end;
    end;
end;

```

Example - Using a BFile LOB Locator

The BFile LOB can be used to read files on the database server. The BFile LOB has a directory and a filename property that specify the file on the server. Apart from selecting the data, you can create a new TLOBLocator instance, assign a directory and filename to it, and access the file without any SQL:

```

var LOB: TLOBLocator;
    Buffer: array[0..99] of Byte;
begin
    // Create a new BFile
    LOB := TLOBLocator.Create(MainSession, otBFile);
    // Assign a directory alias and filename
    LOB.Directory := 'DATA_DIR';
    LOB.Filename := 'X100.DAT';
    // Read the first 100 bytes
    if LOB.FileExists then LOB.Read(Buffer, 100);
    LOB.Free;
end;

```

The directory is an alias (case sensitive!) that must first be created on the server by using a 'create directory' SQL statement. The FileExists method indicates if the file exists in the directory on the server. If the directory alias does not exist, an exception will be raised.

To set the directory or filename of a BFile column in the database, you can use the BFilename SQL function, for example:

```
update filetable set filecolumn = bfilename(:directory, :filename)
```

You can also create a BFile LOB Locator, set the directory and filename property, associate it with a variable in an update or insert statement and execute it:

```

var LOB: TLOBLocator;
begin
    // Create a new BFile
    LOB := TLOBLocator.Create(MainSession, otBFile);
    // Assign a directory alias and filename
    LOB.Directory := 'DATA_DIR';
    LOB.Filename := 'X100.DAT';
    // Set the LOB variable
    LOBQuery.SetComplexVariable('filecolumn', LOB);
    // update filetable set filecolumn = :filecolumn
    LOBQuery.Execute;
    LOB.Free;
end;

```

TLOBLocator reference

This chapter describes all properties and methods of the TLOBLocator object.

TLOBLocator.Append

Declaration

```
procedure Append(Source: TLOBLocator);
```

Description

Appends all the data of the source LOB Locator to the end of the TLOBLocator. All processing is done on the server, so that no LOB data is transferred across the network. You cannot use this function if Buffering is enabled.

TLOBLocator.Assign

Declaration

```
procedure Assign(Source: TLOBLocator);
```

Description

Assigns the Source LOB Locator to this LOB Locator, so that they point to the same LOB data.

TLOBLocator.AsString

Declaration

```
property AsString: string;
```

Description

Use this property to read the complete contents of a LOB column into a string variable. You can also set this property to write a string to a LOB column.

TLOBLocator.Buffering

Declaration

```
property Buffering: Boolean;
```

Description

When your application performs many small reads or writes on CLOB's or BLOB's, you can use buffering to increase performance. When you set the Buffering property of a TLOBLocator to True, Net8 will buffer these reads and writes, thereby reducing the number of network roundtrips.

When you use buffering for writes, the buffer will be flushed to the server in three situations:

1. You set the Buffering property back to False
2. You Free the TLOBLocator instance

3. You call the FlushBuffer method

Note that before you commit a transaction with buffered LOB writes, you must make sure that the buffers are flushed. Otherwise these writes will not be part of the committed transaction.

Also note that you cannot use other update methods than write when buffering is enabled. This includes Trim, Erase, Copy, Append and setting the Size property.

TLOBLocator.Clear

Declaration

```
procedure Clear;
```

Description

Sets the LOB Locator to null.

See also

IsNull

TLOBLocator.Copy

Declaration

```
procedure Copy(Source: TLOBLocator; Length: Integer);
```

Description

Copies <Length> bytes of the data from the current position of the Source LOB Locator to the current position of this LOB Locator. All processing is done on the server, so that no LOB data is transferred across the network. You cannot use this function if Buffering is enabled.

TLOBLocator.Create

Declaration

```
constructor Create(ASession: TOracleSession; ALOBType: Integer);
```

Description

This constructor creates a TLOBLocator instance for the specified session and of the specified type (otCLOB, otBLOB or otBFile). After you have created a LOB Locator, it will be null until you assign a Directory and Filename (BFile), set it to empty with SetEmpty (CLOB and BLOB), or receive an initialized LOB Locator from the server in a LOB variable.

TLOBLocator.CreateTemporary

Declaration

```
constructor CreateTemporary(ASession: TOracleSession; ALOBType: Integer; Cache: Boolean);
```

Description

This constructor creates a temporary TLOBLocator instance for the specified session and of the specified type (otCLOB or otBLOB). The Cache parameter indicates whether the LOB data should be cached on the client (True) or not (False). Uncached LOB data will be stored on the server when written.

TLOBLocator.Directory

Declaration

```
property Directory: string;
```

Description

For BFile LOB Locators, this property defines the directory of the file on the server. This is an alias (case sensitive!) that must first be created on the server by using a 'create directory' SQL statement.

Together with the Filename property, it defines the full file specification of the BFile.

TLOBLocator.Erase

Declaration

```
function Erase(Length: Integer): Integer;
```

Description

Fills <Length> bytes at the current position of the LOB Locator with zeroes (BLOB) or spaces (CLOB). All processing is done on the server, so that no LOB data is transferred across the network. You cannot use this function if Buffering is enabled.

TLOBLocator.FileExists

Declaration

```
function FileExists: Boolean;
```

Description

Determines if the file indicated by the Filename property exists in the directory indicated by the Directory property. If the directory alias does not exist in the database, an exception is raised.

TLOBLocator.Filename

Declaration

```
property Filename: string;
```

Description

For BFile LOB Locators, this property defines the filename of the file on the server. You can use the FileExists method to determine if the file exists in the directory on the server.

Together with the Directory property, it defines the full file specification of the BFile.

TLOBLocator.FlushBuffer

Declaration

```
procedure FlushBuffer;
```

Description

If Buffering is enabled for the LOB Locator, you can use this function to flush the buffer to the server.

TLOBLocator.IsNull

Declaration

```
function IsNull: Boolean;
```

Description

Determines if the LOB Locator is null.

See also

Clear

TLOBLocator.LoadFromFile

Declaration

```
procedure LoadFromFile(const FileName: string);
```

Description

You can directly write the contents of a file to a CLOB or BLOB by using this procedure.

See also

SaveToFile

TLOBLocator.Name

Declaration

```
property Name: string;
```

Description

Use this property to provide a logical name for the LOB Locator. This property will be used by the Oracle Monitor in the object tree. If you do not provide a name, a default name will be generated when the instance is sent to the monitor. This name will be 'LOBLocator' followed by a monitor id number.

TLOBLocator.Position

Declaration

```
property Position: LongInt;
```

Description

Use this property to determine the current zero-based byte position of the LOB Locator as used by the Read, Write, Seek, Copy, Erase and Trim methods. Position is a read-only property, and can be modified by using the Seek method.

TLOBLocator.Read

Declaration

```
function Read(var Buffer; Count: Longint): Longint;
```

Description

The read method reads <Count> bytes from the current position of the LOB Locator into the buffer variable. The result indicates the number of bytes that were actually read. The current position can be set by using the Seek method.

TLOBLocator.SaveToFile

Declaration

```
procedure SaveToFile(const FileName: string);
```

Description

You can directly write the contents of a CLOB, BLOB or BFile to a file by using this procedure.

See also

LoadFromFile

TLOBLocator.Seek

Declaration

```
function Seek(Offset: Longint; Origin: Word): Longint;
```

Description

The seek method can be used to set the current position of the LOB Locator. This position can then be used by subsequent calls to Read, Write, Trim, Copy or Erase. The Origin determines how the seek is performed and how the Offset is interpreted:

Value	Meaning
soFromBeginning	Offset is from the beginning. Seek moves to the position Offset. Offset must be ≥ 0 .
soFromCurrent	Offset is from the current position. Seek moves to Position + Offset.
soFromEnd	Offset is from the end. Offset must be ≤ 0 to indicate a number of bytes before the end of the TLOBLocator.

TLOBLocator.SetEmpty

Declaration

```
procedure SetEmpty;
```

Description

Sets the LOB Locator to empty. Only valid for CLOB's and BLOB's. An empty LOB Locator can be initialized on the server and can subsequently be used to write data to.

TLOBLocator.Size

Declaration

```
property Size: LongInt;
```

Description

Use Size to get or set the size of the LOB. When you set Size to a larger value, the LOB will be padded with zeroes (BLOB) or spaces (CLOB). You cannot set the Size property if Buffering is enabled.

Note

In Delphi 2 this is a read-only property. Use the Trim method to set the size in Delphi 2.

TLOBLocator.Temporary

Declaration

```
property Temporary: Boolean;
```

Description

This property is True if the TLOBLocator instance is for a Temporary LOB (created through CreateTemporary).

TLOBLocator.Trim

Declaration

```
procedure Trim;
```

Description

Removes all data after the current position of the LOB Locator. All processing is done on the server, so that no LOB data is transferred across the network. You cannot use this function if Buffering is enabled.

TLOBLocator.Write

Declaration

```
function Write(const Buffer; Count: Longint): Longint;
```

Description

The write method writes Count bytes at the current position of the TLOBLocator from the buffer variable. The result indicates the number of bytes that were actually written. The current position can be set by using the Seek method.

TOracleObject object

Unit

Oracle

Description

The most important new feature of Oracle8 for application development is the Object Extension. It allows an application to access the database in an object-oriented way by introducing object attributes, methods, persistence, and so on. The client-side cache of Net8 allows a programmer to create very efficient client/server applications, without compromising the application with all kinds of optimization logic.

An Oracle8 object is encapsulated in the Delphi object TOracleObject. You can obtain a TOracleObject instance by creating one with the Create constructor, by selecting an object in a TOracleQuery and using the ObjField method, by pinning (also called dereferencing) a TOracleReference, or by accessing an object's embedded object through TOracleObject.ObjAttr.

An object can be atomically null, which means that the whole object is null rather than all its attributes. To test for atomic nullness, use the IsNull method. To make an object atomically null, use the Clear method. An object will become "not null" when one of its attributes is set.

Simple object attributes are manipulated by using the GetAttr and SetAttr methods. Complex object attributes are accessed by using the LOBAttr (for LOB Locator attributes), ObjAttr (for embedded object attributes) or RefAttr (for reference attributes) methods. Object methods (member functions and procedures) can be called by using the CallMethod method.

Persistent standalone objects (those objects that are either created with an associated table or pinned from a reference) can be flushed to the database server by using the Flush method. Flushing is required after a persistent standalone object has been modified, or after it has been deleted by using the Delete method. Furthermore, persistent standalone objects can be locked by using the Lock method, and can be refreshed from the database by using the Refresh method.

Objects can also be varray or nested table collections. A collection object does not have attributes. Instead, it has elements that can be of a simple (string, integer, float, date) or complex (reference, object) data type. To access an element of a collection, TOracleObject provides 3 zero-based array properties: Elements (for simple element types), ObjElements (for object element types) and RefElements (for reference element types). Use the ElementCount method to determine the number of elements in the collection. You can delete an element by using the DeleteElement method, which results in a "gap" in the collection. Use the ElementExists method to determine if an element exists at a certain position in the collection.

Note

This object is only available in the Object version of Direct Oracle Access.

Example - Selecting an object

The following example selects an address object attribute from the Persons object table, tests if it is null, and displays the City attribute of the address.

```
var Address: TOracleObject;
begin
  Query.SQL.Text := 'select Name, Address from Persons';
  Query.Execute;
  while not Query.Eof do
  begin
    Address := Query.ObjField('Address');
    if not Address.IsNull then
      ShowMessage(Query.Field('Name') + ' lives in ' +
        Address.GetAttr('City'));
    Query.Next;
  end;
end;
```

Example - Updating an object

The following example selects a person object by reference, pins and locks the reference, converts the address attributes to uppercase and flushes this modification to the database.

```
var Person: TOracleObject;
    Street, City: string;
begin
  Query.SQL.Text := 'select ref(P) Person from Persons P';
  Query.Execute;
  while not Query.Eof do
  begin
    Person := Query.RefField('Person').Pin(poLatest, plExclusive);
    Street := Person.GetAttr('Address.Street');
    if Street <> UpperCase(Street) then
      Person.SetAttr('Address.Street', UpperCase(Street));
    City := Person.GetAttr('Address.City');
    if City <> UpperCase(City) then
      Person.SetAttr('Address.City', UpperCase(City));
    if Person.Modified then
      Person.Flush;
    Person.Free;
    Query.Next;
  end;
end;
```

TOracleObject reference

This chapter describes all properties and methods of the TOracleObject object.

TOracleObject.Assign

Declaration

```
procedure Assign(Source: TOracleObject);
```

Description

Assigns the Source object to this object by making a "deep copy". Both objects must be of the same type.

TOracleObject.AttrIsNull

Declaration

```
function AttrIsNull(const AName: string): Boolean;
```

Description

Determine if an attribute is null. If the attribute is part of an embedded object, use the dot-notation (e.g. 'Person.Address.Zip') to specify its name.

See also

ClearAttr

TOracleObject.CallComplexMethod

Declaration

```
procedure CallComplexMethod(const AMethodName: string; const  
Parameters: array of Variant; Result: TObject);
```

Description

When the return value of a method is of a complex data type, you must use CallComplexMethod instead of CallMethod. This method has one extra TObject parameter that is used to receive the return value. To call the method Parent, which returns a TPerson reference, you could use the following code:

```
// Create a new reference
RefParent := TOracleReference.Create(Session, 'TPerson');
// Get the reference to a parent
Person.CallComplexMethod('Parent', parNone, RefParent);
```

TOracleObject.CallMethod

Declaration

```
function CallMethod(const AMethodName: string; const Parameters:
    array of Variant): Variant;
```

Description

You can call an object method by using CallMethod. The method is executed on the server. Therefore, each method call results in a single network roundtrip. To determine the age of a TPerson object, you could use the following statement:

```
Age := Person.CallMethod('Age', ['p_AtDate', Now]);
```

The first parameter ('Age') specifies the name of the method. The second parameter is an array of variants that specify the parameters of the method. The return value of CallMethod is a variant that contains the return value of the executed method. For methods that do not return a value ('member procedures'), a Null is returned.

You can use a named or positional notation for the parameters. In case of a named notation, each parameter value must be preceded by the parameter name. The following statement uses a positional notation:

```
Age := Person.CallMethod('Age', [Now]);
```

Because the 'p_AtDate' parameter has a default value, it can also be omitted. Because Delphi does not allow empty open array constructors, you can pass the parNone constant:

```
Age := Person.CallMethod('Age', parNone);
```

You can also use GetAttr for methods that do not need parameters, thereby making it transparent to the application if something is a method or an attribute (much like you are used to in Delphi).

When a method has out or in/out parameters, you can retrieve their value with GetParameter after calling the method. Out parameters do not need to be specified in the parameter list, as they do not have a value on input. You can also omit in/out parameters, in which case they will be null on input.

When a parameter is of a complex data type (LOB Locator, Reference or Object), you need to pass an instance of the appropriate object type to the method. Because the parameters are specified as variants, you must typecast the object instance to a LongInt. To call the method IsParent, which takes a TPerson reference as parameter, you could use the following code:

```

if Person.CallMethod('IsParent', ['p_Parent',
  LongInt(AnotherPerson.Reference)]) then
  ShowMessage(AnotherPerson.GetAttr('Name') + ' is a parent of ' +
  Person.GetAttr('Name'));

```

A parameter of a complex data type can never be omitted. Even if it is an out parameter, you must still specify the object instance in which you wish to receive the value.

When the return value of a method is of a complex data type, you must use `CallComplexMethod` instead of `CallMethod`.

TOracleObject.Clear

Declaration

```
procedure Clear;
```

Description

Makes an object atomically null, and clears all its attributes.

See also

IsNull

TOracleObject.ClearAttr

Declaration

```
procedure ClearAttr(const AName: string);
```

Description

Sets an attribute to null. Can be used for simple and complex data types. If the attribute is part of an embedded object, use the dot-notation (e.g. 'Person.Address.Zip') to specify its name.

See also

AttrIsNull

TOracleObject.Create

Declaration

```
constructor Create(ASession: TOracleSession; const ATypeName: string;
  const ATable: string);
```

Description

When you create a new `TOracleObject` instance, you must specify the database session and the name of the type in the Oracle database. When creating a persistent object, you must also specify the table name. The following example creates a persistent `TPerson` object instance that will be stored in the `Persons` table:


```
Person := TOracleObject.Create(Session, 'TPerson', 'Persons');
```

If the object type or table is defined in a different schema and no synonyms are defined, you should prefix them with the owner:

```
Person := TOracleObject.Create(Session, 'Scott.TPerson',  
    'Scott.Persons');
```

After creating a new object instance, all attributes will be null and the object itself will be atomically null.

TOracleObject.Delete

Declaration

```
procedure Delete;
```

Description

To delete a persistent standalone object in the database, you can call the Delete method and flush this modification to the database:

```
Father := Person.RefAttr('Father').Pin(poAny, plNone);  
Person.ClearAttr('Father');  
Person.Flush;  
Father.Delete;  
Father.Flush;  
Father.Free;
```

If you access the object after it has been deleted, an exception will be raised that it has been deleted or is marked for delete. Transient objects cannot be deleted.

TOracleObject.DeleteElement

Declaration

```
procedure DeleteElement(Index: Integer);
```

Description

To delete an element in a collection, you can use the DeleteElement method. DeleteElement takes a zero-based index of the element that is to be deleted. The elements after this index will not be moved by one position. Instead, the collection will have a non-existent element. You can test for the existence of an element at a given index by using the ElementExists method. The following example removes the last job from the Jobs collection of a person:

```
Jobs := Person.ObjAttr('Jobs');  
if Jobs.ElementCount > 0 then  
    Jobs.DeleteElement(Jobs.ElementCount);  
Person.Flush;
```

Instead of DeleteElement you can also use TrimElements to remove elements from the end of a collection.

TOracleObject.ElementCount

Declaration

```
function ElementCount: Integer;
```

Description

Returns the number of elements of a collection object, including any deleted elements.

TOracleObject.ElementExists

Declaration

```
function ElementExists(Index: Integer): Boolean;
```

Description

Determines if an element exists at a certain Index in a collection object.

See also

DeleteElement

TOracleObject.Elements

Declaration

```
property Elements: array of Variant;
```

Description

For collection objects with elements of a simple type (string, integer, float or date), the Elements array gives access to the variant values of the individual elements:

```
// Get the PhoneList collection: varray of varchar2(12)
PhoneList := Person.ObjAttr('PhoneList');
// Put each phone number element in a memo
for Index := 0 to PhoneList.ElementCount - 1 do
begin
    if PhoneList.ElementExists(Index) then
        Memo.Lines.Add(PhoneList.Elements[Index]);
end;
```

You can also use the Elements array to set the value of an individual element. If you assign a value to the element after the last one, the collection will automatically expand, which will be reflected by ElementCount.

TOracleObject.Flush

Declaration

```
procedure Flush;
```

Description

Flushes the changes of a standalone persistent object to the database. The object must be

Modified. Use this method after creating, modifying or deleting a persistent standalone object.

TOracleObject.GetAttr

Declaration

```
function GetAttr(const AName: string): Variant;
```

Description

Returns the variant value of an attribute of a simple data type (string, integer, float or date). If the attribute is part of an embedded object, use the dot-notation (e.g. 'Person.Address.Zip') to specify its name. If the attribute is null, the NullValue property of the session determines if it will be returned as a Null or Unassigned variant.

You can also access LOB attributes with GetAttr. CLOB's will be handled as strings, and BLOB's and BFiles will be handled as zero-based variant arrays of bytes. You use GetAttr in exactly the same way that you use the Field method for LOB fields in a TOracleQuery. Embedded object attributes and reference attributes cannot be accessed through the GetAttr method.

You can also use GetAttr for methods that do not need parameters, thereby making it transparent to the application if something is a method or an attribute (much like you are used to in Delphi).

See also

AttrsNull

CallMethod

LOBAttr

ObjAttr

RefAttr

TOracleObject.GetParameter

Declaration

```
function GetParameter(const ParameterId: Variant): Variant;
```

Description

When a method has out or in/out parameters, you can retrieve their value with GetParameter after calling the method with CallMethod:

```
Person.CallMethod('DisplayInfo', parNone);  
Name := Person.GetParameter('p_Name');  
Addr := Person.GetParameter('p_Addr');
```

The parameter can be specified by its name or by its zero-based position.

TOracleObject.IsArray

Declaration

```
function IsArray: Boolean;
```

Description

Indicates if the object is an array collection.

See also

IsCollection

IsTable

TOracleObject.IsCollection

Declaration

```
function IsCollection: Boolean;
```

Description

Indicates if the object is a collection.

See also

IsTable

IsArray

TOracleObject.IsLocked

Declaration

```
function IsLocked: Boolean;
```

Description

Indicates if the object is locked. Only valid for standalone persistent objects.

See also

Lock

TOracleObject.IsNull

Declaration

```
function IsNull: Boolean;
```

Description

Indicates if the object is atomically null.

See also

Clear

TOracleObject.IsTable

Declaration

```
function IsTable: Boolean;
```

Description

Indicates if the object is a table collection.

See also

IsCollection

IsArray

TOracleObject.LOBAttr

Declaration

```
function LOBAttr(const AName: string): TLOBLocator;
```

Description

You can access a LOB Locator attribute by using the LOBAttr method, which returns a TLOBLocator instance. When you modify the LOB Locator with the Clear, SetEmpty, Directory or Filename methods and properties, this will affect the object instance, which will become Modified:

```
with PersonRef.Pin(poAny, plNone) do
try
  LOBAttr('Picture').Directory := 'BMP_DIR';
  LOBAttr('Picture').Filename  := 'scott.bmp';
  Flush;
finally
  Free;
end;
```

If you use an existing LOB Locator attribute to modify LOB data of a CLOB or BLOB, the object instance will not become Modified. Instead, the data is written directly to the LOB location.

If the LOB attribute is currently null, you must first set it to empty, flush the instance to the server, and refresh it to retrieve an initialized LOB Locator.

Note that the object instance must be locked before you can modify its LOB data. This can be achieved by using the Lock method, or by specifying a plExclusive PinLockOption when calling the TOracleReference.Pin method:

```

Person := PersonRef.Pin(poAny, plNone);
// Get the notes LOB Locator attribute
Notes := Person.LOBAttr('Notes');
if Notes.IsNull then
begin
    // Notes is null, set it to empty, flush the instance, and
    refresh it
    Notes.SetEmpty;
    // Flushing it will also lock it
    Person.Flush;
    // Retrieve the initialized LOB Locator
    Person.Refresh;
else begin
    Person.Lock;
end;
Notes.Write(Memo.Text, Length(Memo.Text));
Notes.Trim;
Person.Free;

```

See also

GetAttr

SetAttr

TOracleObject.Lock**Declaration**

```
procedure Lock;
```

Description

Locks the object in the database. Only valid for persistent standalone objects.

See also

IsLocked

TOracleObject.MaxElements**Declaration**

```
function MaxElements: Integer;
```

Description

Returns the maximum number of elements for an array collection object.

TOracleObject.Modified

Declaration

```
property Modified: Boolean;
```

Description

Indicates if the object is modified. Only valid for persistent standalone objects.

A persistent standalone object will become modified when:

- ♦ One of its attributes is changed
- ♦ The Delete method is called
- ♦ Modified is explicitly set to True

Modified will become False when:

- ♦ The object is flushed
- ♦ The FlushObjects method of the session is called
- ♦ The Commit method of the session is called
- ♦ It is explicitly set to False

TOracleObject.Name

Declaration

```
property Name: string;
```

Description

Use this property to provide a logical name for the Object. This property will be used by the Oracle Monitor in the object tree. If you do not provide a name, a default name will be generated when the instance is sent to the monitor. This name will be 'OracleObject' followed by a monitor id number.

TOracleObject.ObjAttr

Declaration

```
function ObjAttr(const AName: string): TOracleObject;
```

Description

If you wish to access an embedded object attribute as a TOracleObject, you can use the ObjAttr method. A TOracleObject instance is returned that accesses the same instance data as the embedding object:

```
Address := Person.ObjAttr('Address');
Address.SetAttr('Street', 'Church road 78');
// Will display 'Church road 78'
ShowMessage(Person.GetAttr('Address.Street'));
```

You can set an embedded object attribute to null by using the ClearAttr method of the embedding object, or by using the Clear method of the embedded object. Similarly, you can test if an embedded object is Null with the AttrIsNull method of the embedding object, or by

using the `IsNull` method of the embedded object.

TOracleObject.ObjElements

Declaration

```
property ObjElements: array of TOracleObject; default;
```

Description

For collection objects with elements of an object type, the `ObjElements` array gives access to the `TOracleObject` instances of the individual elements. Because the `ObjElements` array is the default array property of a `TOracleObject`, you can omit the `ObjElements` property name:

```
// Get the Jobs collection: table of TJob
Jobs := Person.ObjAttr('Jobs');
// Put the company attribute of each TJob element in a memo
for Index := 0 to Jobs.ElementCount - 1 do
begin
  if Jobs.ElementExists(Index) then
    Memo.Lines.Add(Jobs[Index].GetAttr('Company'));
end;
```

You can also use the `ObjElements` array to set the value of an individual element. If you assign a value to the element after the last one, the collection will automatically expand, which will be reflected by `ElementCount`. Note that the source object is copied to the collection, no reference is made to the original `TOracleObject` instance:

```
// Create a new transient TJob instance
Jobs := Person.ObjAttr('Jobs');
Job := TOracleObject.Create(Session, 'TJob', '');
Job.SetAttr('Company', 'Borland International');
Job.SetAttr('StartDate', Date);
// Assign it to the element after the last one
Jobs[Jobs.ElementCount] := Job;
// The TJob is copied, so it can safely be freed
Job.Free;
// Flush the embedding object
Person.Flush;
```

TOracleObject.RefAttr

Declaration

```
function RefAttr(const AName: string): TOracleReference;
```

Description

Reference attributes can be accessed by using the `RefAttr` method. You can for example pin a reference attribute in the same way as pinning a reference field in a query: by using the `Pin` method of the `TOracleReference` object that is returned by the `RefAttr` method:


```

Mother := Person.RefAttr('Mother').Pin(poAny, plNone);
ShowMessage(Mother.GetAttr('Name'));
Mother.Free;

```

You can set a reference attribute by using the `TOracleReference.Assign` method:

```

RefMother := Brother1.RefAttr('Mother');
Brother2.RefAttr('Mother').Assign(RefMother);

```

TOracleObject.RefElements

Declaration

property RefElements: **array of** TOracleReference;

Description

For collection objects with elements of a reference type, the `RefElements` array gives access to the `TOracleReference` instances of the individual elements:

```

// Get the Jobs collection: table of ref TJob
Jobs := Person.ObjAttr('Jobs');
// Put the company attribute of each ref TJob element in a memo
for Index := 0 to Jobs.ElementCount - 1 do
begin
  if Jobs.ElementExists(Index) then
  begin
    Job := Jobs.RefElements[Index].Pin(poAny, plNone);
    Memo.Lines.Add(Job.GetAttr('Company'));
    Job.Free;
  end;
end;

```

You can also use the `RefElements` array to set the value of an individual element. If you assign a value to the element after the last one, the collection will automatically expand, which will be reflected by `ElementCount`. Note that the source reference is copied to the collection, no reference is made to the original `TOracleReference` instance.

TOracleObject.Reference

Declaration

function Reference: TOracleReference;

Description

Returns the reference of this object. Only valid for persistent standalone objects.

TOracleObject.Refresh

Declaration

```
procedure Refresh;
```

Description

Refreshes an object instance from the database. Only valid for persistent standalone objects.

TOracleObject.SetAttr

Declaration

```
procedure SetAttr(const AName: string; const Value: Variant);
```

Description

Sets the variant value of an attribute of a simple data type (string, integer, float or date). If the attribute is part of an embedded object, use the dot-notation (e.g. 'Person.Address.Zip') to specify its name. If the object is a standalone persistent object, it will be become modified, and can subsequently be flushed to the database.

You can also use the SetAttr method to write data to a LOB Locator attribute:

```
Person := PersonRef.Pin(poAny, plNone);  
Person.SetAttr('Notes', Memo.Text);
```

If the LOB attribute is currently null, the SetAttr method will automatically set it to empty, flush the instance to the server, and refresh it to retrieve an initialized LOB Locator. After this, the data will be written to the LOB location. SetAttr will also automatically lock the instance if necessary.

Embedded object attributes and reference attributes cannot be accessed through the SetAttr method.

See also

ClearAttr

LOBAttr

ObjAttr

RefAttr

TOracleObject.TimestampAttr

Declaration

```
function TimestampAttr(const AName: string): TOracleTimestamp;
```

Description

You can access a Timestamp attribute by using the TimestampAttr method, which returns a TOracleTimestamp instance. When you modify the Timestamp through any of its methods or properties, this will affect the object instance, which will become Modified.

TOracleObject.TrimElements

Declaration

```
procedure TrimElements(Count: Integer);
```

Description

You can use the TrimElements method to remove elements from the end of the collection. The following example removes the last element from the Jobs collection attribute of a TPerson object:

```
Jobs := Person.ObjAttr('Jobs');  
if Jobs.ElementCount > 0 then Jobs.TrimElements(Jobs.ElementCount -  
    1);  
Person.Flush;
```

TOracleObject.XMLAttr

Declaration

```
function XMLAttr(const AName: string): TXMLType;
```

Description

This function returns a TXMLType instance for a SYS.XMLTYPE attribute.

TOracleReference object

Unit

Oracle

Description

The most important new feature of Oracle8 for application development is the Object Extension. It allows an application to access the database in an object-oriented way by introducing object attributes, methods, persistence, and so on. The client-side cache of Net8 allows a programmer to create very efficient client/server applications, without compromising the application with all kinds of optimization logic.

An Oracle8 object reference plays an important role in an object application, and is encapsulated in the TOracleReference object. You can obtain a TOracleReference instance by creating one with the Create constructor, by selecting a reference in a TOracleQuery and using the RefField method, or by accessing an object's RefAttr or Reference methods.

A reference can be pinned (also called dereferenced) by using the Pin method. The referenced object is fetched from the client-side object cache or from the database server, resulting in a new TOracleObject instance.

Note

This object is only available in the Object version of Direct Oracle Access.

Example - Selecting a reference

The following example selects a reference column ('Mother') from the Persons object table, tests if the reference is null, Pins the reference, and displays the name of the referenced TPerson object:

```
var Mother: TOracleObject;
    RefMother: TOracleReference;
begin
    Query.SQL.Text := 'select Name, Mother from Persons';
    Query.Execute;
    while not Query.Eof do
    begin
        RefMother := Query.RefField('Mother');
        if not RefMother.IsNull then
        begin
            Mother := RefMother.Pin(poAny, plNone);
            ShowMessage('The mother of ' + Query.Field('Name') + ' = ' +
                Mother.GetAttr('Name'));
            Mother.Free;
        end;
        Query.Next;
    end;
end;
```

TOracleReference reference

This chapter describes all properties and methods of the TOracleReference object.

TOracleReference.Assign

Declaration

```
procedure Assign(Source: TOracleReference);
```

Description

Assigns the Source reference to this reference.

TOracleReference.Clear

Declaration

```
procedure Clear;
```

Description

Sets the reference to null.

See also

IsNull

TOracleReference.Create

Declaration

```
constructor Create(ASession: TOracleSession; const ATypeName:  
    string);
```

Description

Creates a new reference for the specified session and of the specified type. The typename can be prefixed with the schema name if it resides in another schema and no synonyms are defined.

TOracleReference.Hex

Declaration

```
property Hex: string;
```

Description

Property to get or set the hexadecimal representation of a reference.

TOracleReference.IsNull

Declaration

```
function IsNull: Boolean;
```

Description

Indicates if the reference is null.

See also

Clear

TOracleReference.Name

Declaration

```
property Name: string;
```

Description

Use this property to provide a logical name for the Reference. This property will be used by the Oracle Monitor in the object tree. If you do not provide a name, a default name will be generated when the instance is sent to the monitor. This name will be 'OracleReference' followed by a monitor id number.

TOracleReference.Pin

Declaration

```
type TPinOption = (poAny, poRecent, poLatest);
type TPinLockOption = (plNone, plExclusive);
function Pin(PinOption: TPinOption; PinLockOption: TPinLockOption):
    TOracleObject;
```

Description

The Pin method (also known as 'dereference') can be used to fetch an object instance. A new TOracleObject instance is created for the referenced object. If the reference is null, the Pin method returns nil. The application is responsible for freeing the pinned TOracleObject instance.

The PinOption parameter specifies which copy of the instance is required by the application and thereby determines the consistency level:

Value	Meaning
poAny	Get any copy that may be available in the cache, otherwise fetch it from the database
poRecent	Get a copy from the cache if it was fetched in the current transaction, otherwise fetch it
poLatest	Get a copy from the cache if it is locked, otherwise fetch it

The PinLockOption specifies how to lock the instance in the database:

Value	Meaning
pINone	Don't place a lock on the instance
pIExclusive	Place an exclusive lock on the instance

TOracleScript component

Unit

Oracle

Description

The TOracleScript component allows you to define and execute a SQL script with multiple SQL statements. This can be useful if you need to execute multiple statements that cannot be used in a PL/SQL Block. This is very often the case for installation scripts, which typically contain a lot of DDL (Data Definition Language) statements for the creation of tables, sequences, view, program units, and so on.

Defining the script

The Lines property of the TOracleScript contains the plain text of the SQL script. This follows the basic syntax rules of SQL*Plus. A semi-colon or a slash separates SQL statements, unless they contain a PL/SQL section, in which case they need to be terminated by a slash (the PL/SQL itself will contain semi-colons). The following example would create a table and a procedure:

```
-- drop the dept table if it already exists
drop table dept;

-- create the dept table
create table dept
(
    deptno number(4)    not null,
    dname  varchar2(14) not null,
    loc    varchar2(13)
);

-- create the deptcount function
create or replace function deptcount return integer as
    result integer;
begin
    select count(*) into result from dept;
    return(result);
end;
/
```

The easiest way to define a script is to use the component editor at design time. Just double-click on the TOracleScript instance and type the text. You can immediately run it within the editor to test it, and view the output (if applicable). You can switch between a script and a commands page by selecting the corresponding tabs at the top of the editor. On the commands page your view is limited to a single command from the script, and pressing the execute button will execute just that command. On this page you can also add and delete commands, and navigate through the commands. Making a change on the commands page will be reflected on the script page, and vice versa.

Non-SQL commands

Besides all SQL commands the following non-SQL commands are supported:

CONNECT *Username/Password@Database*


```

DEFINE Variable=Value
EXIT
PAUSE Message
PROMPT Message
QUIT
REMARK Comment
SET Option ON/OFF (Option = ECHO | EXITONERROR | FEEDBACK | SCAN | TERMOUT)
SET COLWIDTH Value
UNDEFINE Variable

```

These commands perform the same function as in SQL*Plus or set an equivalent property of the TOracleScript instance. In the script they do not need to be separated by semi-colons. The following script connects as scott and drops the dept table:

```

connect scott/tiger
pause About to drop table dept...
drop table dept;
prompt Table DEPT has been dropped

```

Output

The TOracleScript has an Output property, which is a TStrings containing the output lines that were generated by the executed commands. The OutputOptions property lets you control exactly what information will be written to the Output. This applies to the command text itself, the feedback, errors, and the result data. You can also explicitly capture the output through the OnOutput event.

Substitution variables

Just like in SQL*Plus you can use substitution variables to make your script customizable. You can use the SetVariable procedure to set the value of the variable, and use it in the script by preceding the variable name with an ampersand. For example, if you allow the end user to specify the initial size of a table, then the script could look like this:

```

-- create the dept table
create table dept
(
    deptno number(4)    not null,
    dname  varchar2(14) not null,
    loc    varchar2(13)
)
storage(initial &initial_size M);

```

The initial_size variable can be set through the SetVariable procedure:

```
MyScript.SetVariable('initial_size', '10');
```

Error handling

Individual SQL statements can lead to an error. You can control if such an error should stop further execution of the script through the ExitOnError property.

An SQL error will also lead to an OnError event, which you can specifically handle. After execution you can also check the ErrorCode and ErrorMessage property of each command.

Extending the script functionality

If the standard functionality of the TOracleScript component is not sufficient for your application, you can use the OnCommand, AfterCommand and OnData events to perform your own processing for all or specific commands.

TOracleScript reference

This chapter describes all properties, methods and events of the TOracleScript component.

TOracleScript.AddOutput

Declaration

```
procedure AddOutput(const S: string);
```

Description

Adds the given string to the Output.

TOracleScript.AfterCommand

Declaration

```
type TOracleScriptCommandEvent = procedure(Sender: TOracleScript; var  
    Handled: Boolean) of object;  
property AfterCommand: TOracleScriptCommandEvent;
```

Description

This event is fired after execution of a command. You can use the CurrentCommand property to access the properties of the command that was just executed. The Handled parameter indicates if the command was handled internally or by the OnCommand event. If it was not handled, this parameter will be False.

TOracleScript.AutoCommit

Declaration

```
property AutoCommit: Boolean;
```

Description

Determines if update, insert and delete statements are immediately committed after execution.

TOracleScript.ColWidth

Declaration

```
property ColWidth: Integer;
```

Description

Set the ColWidth property to control the maximum width of displayed columns in the Output. This property can also be controlled from within the script by using the SET COLWIDTH command.

TOracleScript.CommandByName

Declaration

```
function CommandByName(const Name: string): TOracleCommand;
```

Description

Returns the command with the given name. A name can be specified in a comment line preceding the command. Consider the following text in a script:

```
-- Name = dept_create
create table dept
(
    deptno number(4)    not null,
    dname  varchar2(14) not null,
    loc    varchar2(13)
);
```

This comment can be found by using 'dept_create' for the Name parameter. If we want to check if this command was executed successfully after executing the script, we can use the following code:

```
MyScript.Execute;
if CommandByName('dept_create').ErrorCode <> 0 then
    ShowMessage('Table DEPT has not been created, installation
        failed.');
```

Note that this will only work for comment lines that start with 2 hyphens.

TOracleScript.CommandIndex

Declaration

```
property CommandIndex: Integer;
```

Description

Returns the index of the currently executed command. You can set this index in the OnCommand or AfterCommand event to control the next executed command.

TOracleScript.Commands

Declaration

```
property Commands: TOracleCommands;
```

Description

This property provides access to all individual commands of the script.

TOracleScript.CurrentCommand

Declaration

property CurrentCommand: TOracleCommand;

Description

Returns the currently executed command, which is most useful in the various event handlers of the script.

TOracleScript.Cursor

Declaration

property Cursor: TCursor;

Description

Determines the shape of the mouse cursor while executing the script. Only crDefault, crHourGlass and crSQLWait are useful here.

TOracleScript.Debug

Declaration

property Debug: Boolean;

Description

If you set this property to True, the text of SQL and Non-SQL commands will be displayed in a message box before execution.

TOracleScript.Execute

Declaration

procedure Execute;

Description

Executes the script.

TOracleScript.ExitOnError

Declaration

property ExitOnError: Boolean;

Description

If this property is set to True, script execution will stop whenever a SQL command fails. If this property is set to False, the error message will be written to the output (if the OutputOptions are set accordingly), and execution will continue with the next command.

This property can also be controlled from within the script by using the SET EXITONERROR ON and SET EXITONERROR OFF commands.

TOracleScript.Finished

Declaration

```
property Finished: Boolean;
```

Description

Indicates if the script is finished. If you set this property to True, execution of the script will stop.

TOracleScript.GetVariable

Declaration

```
function GetVariable(const Name: string): string;
```

Description

Returns the value of the given substitution variable. If the variable does not exist, an empty string will be returned.

TOracleScript.Lines

Declaration

```
property Lines: TStrings;
```

Description

This property contains the plain text of the commands of SQL script, which basically follows the rules of the SQL*plus syntax. Individual SQL commands should be separated by a semicolon or a slash. SQL commands that contain PL/SQL (like the creation of a procedure) can only be terminated by a slash. Non-SQL commands do not need to be separated by semicolons or slashes, but are always placed on a single line. Comment can be included as - comment, /* comment */ , or rem comment.

The following example demonstrates these basic syntax rules:

```

/* drop the dept table if it already exists */
prompt Dropping table DEPT...
drop table dept;

/* create the dept table */
prompt Creating table DEPT...
create table dept
(
    deptno number(4)    not null,
    dname  varchar2(14) not null,
    loc    varchar2(13)
);

/* create the deptcount function */
prompt Creating function DEPTCOUNT...
create or replace function deptcount return integer as
    result integer;
begin
    select count(*) into result from dept;
    return(result);
end;
/
prompt Finished.

```

Modifications made to the Lines property will immediately be reflected in the Commands property, and vice versa. During and after execution of the script you will use the Commands property instead of the Lines to access the individual commands.

TOracleScript.OnCommand

Declaration

```

type TOracleScriptCommandEvent = procedure (Sender: TOracleScript; var
    Handled: Boolean) of object;
property OnCommand: TOracleScriptCommandEvent;

```

Description

This event fires just before a command would be executed. You can use the CurrentCommand property to access the properties of the current command. If you set the Handled parameter to True, you indicate that you have handled this command. As a result, the command will subsequently not be processed internally.

The following example implements a 'beep' command:

```

procedure TMainForm.ScriptOnCommand(Sender: TOracleScript; var
    Handled: Boolean);
var FirstWord: string;
begin
    FirstWord := UpperCase(Sender.CurrentCommand.Words[0]);
    if FirstWord = 'BEEP' then
        begin
            MessageBeep(MB_OK);
            Handled := True;
        end;
    end;
end;

```

TOracleScript.OnData

Declaration

```

type TOracleScriptEvent = procedure(Sender: TOracleScript) of object;
property OnData: TOracleScriptEvent;

```

Description

This event fires for each row that is fetched for a select statement. You can use the CurrentCommand property to access the properties of the current command. You can use the Query property to access the field data of the current record.

TOracleScript.OnError

Declaration

```

type TOracleScriptEvent = procedure(Sender: TOracleScript) of object;
property OnError: TOracleScriptEvent;

```

Description

This event fires whenever an error occurs during execution of a SQL command. Use the CurrentCommand property to access the ErrorCode and ErrorMessage properties of the current command.

TOracleScript.OnOutput

Declaration

```

type TOracleScriptOutputEvent = procedure(Sender: TOracleScript;
    const Msg: string) of object;
property OnOutput: TOracleScriptOutputEvent;

```

Description

This event fires for each line that is written to the Output. You can use it to capture the output in real-time, line by line.

TOracleScript.Output

Declaration

```
property Output: TStrings;
```

Description

This property contains the output lines that were generated by the executed commands. The OutputOptions property controls which information will be written to the output.

TOracleScript.OutputOptions

Declaration

```
type TScriptOutputOptions = set of (ooSQL, ooNonSQL, ooData,  
    ooFeedback, ooError);  
property OutputOptions: TScriptOutputOptions;
```

Description

This property controls which information should be written to the output (the equivalent commands between parenthesis):

- ♦ ooSQL: The text of SQL commands (SET ECHO ON|OFF)
- ♦ ooNonSQL: The text of non-SQL commands (SET ECHO ON|OFF)
- ♦ ooData: The formatted result data of SQL select statements (SET TERMOUT ON|OFF)
- ♦ ooFeedback: The feedback (e.g. "Table created") of SQL commands (SET FEEDBACK ON|OFF)
- ♦ ooError: The error messages of failed SQL commands

TOracleScript.Query

Declaration

```
property Query: TOracleQuery;
```

Description

The TOracleQuery instance that is used internally to execute SQL statements. You can use it to perform your own SQL processing in the OnCommand event, and you can use it in the OnData event to access the field data of the current record.

TOracleScript.ScanVariables

Declaration

```
property ScanVariables: Boolean;
```

Description

If you set this property to False, substitution variables in the script will not be replaced. This property can also be controlled from within the script by using the SET SCAN ON and SET

SCAN OFF commands.

TOracleScript.Session

Declaration

```
property Session: TOracleSession;
```

Description

The session in which the script will run.

TOracleScript.SetVariable

Declaration

```
procedure SetVariable(const Name, Value: string);
```

Description

Sets the value of the given substitution variable. A substitution variable can be used in a script by preceding it with an ampersand:

```
drop table &table_name;
```

If 'table_name' is set to 'dept', then the dept table will be dropped. A variable name can be terminated with a period if they are part of an identifier:

```
drop table &app._codes;
```

If variable 'app' is set to 'sys', then the sys_codes table will be dropped. Another example:

```
drop table &owner..&table;
```

If 'owner' is set to 'scott', and 'table' is set to 'dept', then the scott.dept table will be dropped. Note the double period.

Variables can also be set from within the script:

```
define table_name = dept  
drop table &table_name;
```

TOracleCommands object

Unit

Oracle

Description

This object provides access to the individual commands of a TOracleScript component. The Items property is the default array property that can be used to access a single command. The text of the commands can be declared at design time, and the other properties can be accessed at run time after the commands have been executed.

The Count property indicates the number of commands. You can define commands at run time by using the Add, Delete and Clear methods.

TOracleCommands reference

This chapter describes all properties and methods of the TOracleCommands object.

TOracleCommands.Add

Declaration

```
function Add: TOracleCommand;
```

Description

Adds a command at the end of the array. Use the Index property of a newly added command to move it to a specific position.

TOracleCommands.Clear

Declaration

```
procedure Clear;
```

Description

Deletes all commands.

TOracleCommands.Count

Declaration

```
property Count: Integer;
```

Description

Returns the number of commands.

TOracleCommands.Delete

Declaration

```
procedure Delete(Index: Integer);
```

Description

Deletes the command with the given Index.

TOracleCommands.Items

Declaration

```
property Items[Index: Integer]: TOracleCommand; default;
```

Description

This default array property returns the TOracleCommand at the given index. Because it is the default array property, the following 2 statements are equivalent:

```
Command := MyScript.Commands.Items[3];  
Command := MyScript.Commands[3];
```

TOracleCommand object

Unit

Oracle

Description

This object represents a single command in a TOracleScript component. The Text property is the text from the script lines that make up this command. The Index property is the zero-based position of the command in the script. The CommandType indicates if this is a SQL or Non-SQL command.

After the command is executed you can use various properties that provide information about the results. The ErrorCode and ErrorMessage indicate if a SQL command was executed successfully. The RowsProcessed property can be used to find out how many rows were processed by a select, update, delete or insert statement. The FunctionType property indicates what type of SQL command was executed.

TOracleCommand reference

This chapter describes all properties and methods of the TOracleCommand object.

TOracleCommand.CommandType

Declaration

```
type TScriptCommandType = (ctSQL, ctPLSQL, ctNonSQL);
property CommandType: TScriptCommandType;
```

Description

Indicates the type of the command:

- ♦ ctSQL: the command is a plain SQL statement
- ♦ ctPLSQL: the command is a SQL statement with a PL/SQL section (e.g. create procedure or an anonymous PL/SQL Block)
- ♦ ctNonSQL: the command is a non-SQL command (e.g. prompt or define)

TOracleCommand.CommentProperty

Declaration

```
function CommentProperty(const Name: string): string;
```

Description

Returns a property value from the comment of a command. Consider the following command text:

```
-- Create the DEPT table
-- Name = dept_create
-- Critical = Yes
create table dept
(
  deptno number(4)    not null,
  dname  varchar2(14) not null,
  loc    varchar2(13)
);
```

This command contains 2 comment properties: Name and Critical. The 'Name' comment property has a special meaning, as it is also used by the formal Name property and by the CommandByName function. The value of the 'Critical' property can be obtained through the CommentProperty function:

```
Critical := (MyScript.Commands[i].CommentProperty('Critical') =
  'Yes');
if Critical and (MyScript.Commands[i].ErrorCode <> 0) then
  raise Exception.Create('Installation script failed!');
```

Note that this will only work for comment lines that start with 2 hyphens.

TOracleCommand.ErrorCode

Declaration

```
property ErrorCode: Integer;
```

Description

The error code for a SQL command. For a successfully executed SQL command this will be 0. For failed SQL commands this will be the error code you can find in the "Oracle Error Messages Guide" (e.g. 942 for "Table or view does not exist").

TOracleCommand.ErrorMessage

Declaration

```
property ErrorMessage: string;
```

Description

The error message for a SQL command. For a successfully executed SQL command this will be an empty string. For failed SQL commands this will be the error message you can find in the "Oracle Error Messages Guide" (e.g. "ORA-00942: Table or view does not exist").

TOracleCommand.Execute

Declaration

```
function Execute: Boolean;
```

Description

You can use this function to execute a single command. Normally commands get executed by calling the Execute procedure of the script. The result indicates if the command has been handled.

TOracleCommand.FunctionType

Declaration

```
property FunctionType: Integer;
```

Description

Returns the function type for a SQL command after execution. This number corresponds to the numbers you can find in the "Programmer's Guide to the Oracle Call Interface"

TOracleCommand.Index

Declaration

property Index: Integer;

Description

Returns the zero-based index of the command in the commands array of the script. You can also set the index to move a command to a specific position.

TOracleCommand.Name

Declaration

property Name: string;

Description

The Name property helps you identify specific command in your application. It is extracted from the comment lines that precede the actual command in the text. Consider the following text:

```
-- Name = dept_create
create table dept
(
    deptno number(4)    not null,
    dname  varchar2(14) not null,
    loc    varchar2(13)
);
```

In this case the Name property would return 'dept_create'. You can also set the Name property, in which case the comment mentioned above will not be modified. To find a command by name you can use the CommandByName function of the TOracleScript.

Note that this will only work for comment lines that start with 2 hyphens.

TOracleCommand.RowsProcessed

Declaration

property RowsProcessed: Integer;

Description

After execution of a SQL command, this property will return the number of rows that were processed by a select, update, delete or insert statement.

TOracleCommand.ScriptLine

Declaration

```
property ScriptLine: Integer;
```

Description

Returns the line in the script where this command is located.

TOracleCommand.SubstitutedText

Declaration

```
property SubstitutedText: string;
```

Description

Returns the text of the command with replaced substitution variables.

See also

SetVariable

TOracleCommand.Text

Declaration

```
property Text: string;
```

Description

Returns the text of the command.

TOracleCommand.Words

Declaration

```
property Words: TStrings;
```

Description

Returns the words that make up the command text. Any comment in the text will not show up in the words. This can be useful if you wish to parse the command text to process certain commands the OnCommand event.

TOracleDirectPathLoader component

Unit

Oracle

Description

The TOracleDirectPathLoader component allows an application to access the Direct Path Load engine of the Oracle Server, which is also used by SQL*Loader. This provides the ability to load external data from memory into the database at the highest possible speeds. Instead of using Insert SQL statements to insert the individual records, the external data in memory is converted in to a format that can immediately be written to the physical database blocks of a table. This makes it even faster than Array DML, which basically still processes Insert statements.

The drawback of this speed advantage is that there are several restrictions:

- Triggers are not allowed for the table. An attempt to load data into a table with triggers will lead to "ORA-26086 direct path does not support triggers". You can temporarily disable the triggers of the table.
- Check constraints and foreign key constraints are not allowed for the table. Loading data into such a table will lead to "ORA-26087 direct path does not support referential or check constraints". You can temporarily disable the constraints of the table. Primary and unique key constraints are allowed.
- Remote tables cannot be loaded.
- User defined types not allowed for the table.
- The Direct Path Load interface is only available in Net8 8.1 (the Oracle8i client) and later.
- The load operation is not part of a "normal" transaction.

These drawbacks may be a reason to resort to Array DML, which does not suffer from any of these restrictions and still provides excellent batch loading performance. However, the speed advantage of the TOracleDirectPathLoader can be considerable. For example, loading 10,000 records into a table with 2 columns of 40 bytes without any indexes, on a local database configuration on a Pentium III class server shows the following benchmark results:

Single Inserts:	1,500 records / second
Array Inserts:	15,000 records / second
Direct Path Loading:	60,000 records / second

Results will of course vary for different parameters. For example, if the table has one or more indexes there will be more overhead for each inserted row, and results will be closer.

Using the TOracleDirectPathLoader component

To load external data into a table using the TOracleDirectPathLoader component, you need to perform the following tasks:

1. Setup the TOracleDirectPathLoader component

After creating a TOracleDirectPathLoader instance at design time, you should link it to a Session, and provide the TableName. Next you can define the columns that you want to load through the Columns property editor. For each column you need to define the external data

type and data size, and for dates you can also define the external date format.

2. Prepare the TOracleDirectPathLoader for loading

A call to the Prepare procedure will create an array for a batch of records. The number of records that fit into this array depends on the BufferSize property and on the number and data size of the Columns.

3. Fill and load data arrays

You need to divide your external data into batches that fit into the data array. After you have called Prepare the size of the data array is defined by the MaxRows property. Each batch must be completely read into memory, because the Direct Path Load engine can only load data directly from the array in memory. Fill each element in the array by calling the Columns[Index].SetData procedure. Now you can call the Load procedure to load the data from the array into the table.

4. Finish the load process

After you have processed all external data as individual batches, you can call the Finish procedure to commit the loaded data. To undo the load operation, call the Abort procedure instead.

Example - Direct Path Loading

The following example dynamically creates a TOracleDirectPathLoader instance, sets the necessary properties, creates default column definitions, and loads the data. This data is located in a Records structure, and is left out of the example to keep it simple. Each record consists of an integer value 'Line' and a string value 'Text'.

```

// Perform the Direct Path Load
procedure LoadRecords;
var Loader: TOracleDirectPathLoader;
    i, Row: Integer;
begin
    // Create a Loader at run time
    Loader := TOracleDirectPathLoader.Create(nil);
    try
        // Set the session and table name
        Loader.Session := MainSession;
        Loader.TableName := 'record_data';
        // Get the default columns for the record_data table
        Loader.GetDefaultColumns(False);
        // Prepare the loader
        Loader.Prepare;
        // Process all data in batches of <MaxRows> records
        Row := 0;
        for i := 0 to Records.Count - 1 do
            begin
                // Copy one record to the array
                Loader.Columns[0].SetData(Row, @Records[i].Line, 0);
                loader.Columns[1].SetData(Row, @Records[i].Text[1],
                                          Length(Records[i].Text));

                Inc(Row);
                // The array is filled, or we have preprocessed all records:
                // load this batch of records
                if (Row = Loader.MaxRows) or (i = Records.Count - 1) then
                    begin
                        try
                            Loader.Load(Row);
                        except
                            // In case of an error: show where things went wrong
                            // and abort the load operation
                            on E:EOraclError do
                                begin
                                    ShowMessage(E.Message + #13#10 +
                                                  'Row = ' + IntToStr(Loader.LastRow) + #13#10
+
                                                  'Col = ' + IntToStr(Loader.LastColumn));
                                    Loader.Abort;
                                    raise;
                                end;
                            end;
                        end;
                        Row := 0;
                    end;
                end;
                // Commit the loaded data
                Loader.Finish;
            finally
                Loader.Free;
            end;
        end;
    end;

```

TOracleDirectPathLoader reference

This chapter describes all properties and methods of the TOracleDirectPathLoader component.

TOracleDirectPathLoader.Abort

Declaration

```
procedure Abort;
```

Description

Aborts the load operation, and cancels any data that was loaded. Note that this is not the same as a rollback, because the load operation is not part of a normal transaction.

TOracleDirectPathLoader.BufferSize

Declaration

```
property BufferSize: Integer;
```

Description

The size in bytes of the buffer that contains the formatted data that will be written to the database blocks. This will determine the number of rows that can be transferred in one Load. The MaxRows property reflects this value after calling the Prepare procedure. The default buffer size is 64KB, but you must make sure that it can contain at least one record. Make sure that you take the maximum size of any Long, Long Raw, CLOB or BLOB column into account.

TOracleDirectPathLoader.ColumnByName

Declaration

```
function ColumnByName(const ColumnName: string): TDirectPathColumn;
```

Description

Returns the column instance of the given name. If the column does not exist, this function returns nil. The name is case insensitive. Note that you should not use this function to access the columns to fill the data array (e.g. Loader.ColumnByName('ename').SetData(...), because this will affect performance in a negative way.

TOracleDirectPathLoader.Columns

Declaration

```
property Columns: TDirectPathColumns;
```

Description

This property provides access to the column definitions and values of the data array. Instead of accessing columns by index, you can alternatively use the ColumnByName function to

access a column by name.

TOracleDirectPathLoader.DateFormat

Declaration

```
property DateFormat: string;
```

Description

Defines the default external date format for date columns. The date format can be overridden at the column level. If you do not define a date format, then the NLS date format of the session will be used.

TOracleDirectPathLoader.Finish

Declaration

```
procedure Finish;
```

Description

Finishes the load operation, and makes the loaded data permanent. Note that this is not the same as a commit, because the load operation is not part of a normal transaction.

TOracleDirectPathLoader.GetDefaultColumns

Declaration

```
procedure GetDefaultColumns(StringsOnly: Boolean);
```

Description

Removes the current columns and defines default columns based on the column definition of the table. The StringsOnly parameter determines if integer and float columns should be defined with a DataSize of dpString or as dpInteger / dpFloat. If your external data contains string representations for numeric values, you can set this parameter to True.

TOracleDirectPathLoader.LastColumn

Declaration

```
property LastColumn: Integer;
```

Description

The last column that was processed by the conversion of the data array to the load format during the Load operation. In case of an exception, this property indicates the column that failed.

See also

LastRow

TOracleDirectPathLoader.LastRow

Declaration

```
property LastRow: Integer;
```

Description

The last row that was processed by the conversion of the data array to the load format during the Load operation. In case of an exception, this property indicates the row that failed.

See also

LastColumn

TOracleDirectPathLoader.Load

Declaration

```
procedure Load(Rows: Integer);
```

Description

Load the specified number of rows from the data array into the table. This procedure can raise an EOracleError exception, in which case the LastColumn and LastRow properties indicate the column and row that lead to the error.

TOracleDirectPathLoader.LogMode

Declaration

```
type TDirectPathLogMode = (lmDefault, lmNoLogging);  
property LogMode: TDirectPathLogMode;
```

Description

Determines whether redo log information is generated. Setting this property to lmDefault uses the (NO)LOGGING property for the table. Setting it to lmNoLogging will not generate redo log information for the load operation.

TOracleDirectPathLoader.MaxRows

Declaration

```
property MaxRows: Integer;
```

Description

The maximum number of rows in the data array. This read-only property is only valid after calling the Prepare procedure, and depends on the BufferSize and the DataSize of the columns.

TOracleDirectPathLoader.Parallel

Declaration

```
property Parallel: Boolean;
```

Description

Setting this property to True will allow multiple sessions to load the same table simultaneously, at the cost of a small performance trade-off.

TOracleDirectPathLoader.PartitionName

Declaration

```
property PartitionName: string;
```

Description

Name of the partition or subpartition to be loaded. If you don't specify this property, the entire table can be loaded. The name must be a valid partition or subpartition that belongs to the table.

TOracleDirectPathLoader.Prepare

Declaration

```
procedure Prepare;
```

Description

Prepare the TOracleDirectPathLoader to set and load data. You can call this procedure after setting the Session, TableName and BufferSize properties, and after defining the Columns. This procedure will allocate buffers for the array data and for the formatted data, so that you can subsequently fill the data array and load the data.

TOracleDirectPathLoader.Prepared

Declaration

```
property Prepared: Boolean;
```

Description

Indicates if the Prepare procedure has been called.

TOracleDirectPathLoader.Session

Declaration

```
property Session: TOracleSession;
```

Description

The session in which the load operation will take place.

TOracleDirectPathLoader.TableName

Declaration

```
property TableName: string;
```

Description

The name of the table to be loaded. You can specify the owner of the table in this property as well, if it is not owned by the user of the session (e.g. SCOTT.EMP).

TDirectPathColumns object

Unit

Oracle

Description

This object provides access to the column definitions and data of a TOracleDirectPathLoader component. The Items property is the default array property that can be used to access a single column. The definitions can be declared at design time, and the data can be set at run time through the SetData procedure:

```
Loader.Columns[3].SetData(RowIndex, DataPointer, DataLength);
```

The Count property indicates the number of columns. You can define the columns at run time by using the Clear and Add methods.

TDirectPathColumns reference

This chapter describes all properties and methods of the TDirectPathColumns object.

TDirectPathColumns.Add

Declaration

```
procedure Add(const ColumnName: string): TDirectPathColumn;
```

Description

Adds a new Direct Path Column with the given name at the end of the array. This can be useful if you want to define the columns at run time. To change the position of a newly added column, set its Index property. To remove a column you can simply free it.

TDirectPathColumns.Clear

Declaration

```
procedure Clear;
```

Description

Deletes all items. This can be useful if you want to define the columns at run time, in which case it can be used together with the Add function. To remove a single column, you can simply free the corresponding instance.

TDirectPathColumns.Count

Declaration

```
property Count: Integer;
```

Description

Returns the number of items.

TDirectPathColumns.Items

Declaration

```
property Items[Index]: TDirectPathColumn; default;
```

Description

This default array property returns the TDirectPathColumn at the given index. Because it is the default array property, the following 2 statements are equivalent:

```
Col := Loader.Columns.Items[3];  
Col := Loader.Columns[3];
```

TDirectPathColumn object

Unit

Oracle

Description

This object defines the external format of a column of the TOracleDirectPathLoader component. You can specify the name, data type, data size, and date format of the external representation of the data.

The TDirectPathColumn object additionally provides access to the row data that is to be loaded, by using the SetData procedure.

TDirectPathColumn reference

This chapter describes all properties and methods of the TDirectPathColumn object.

TDirectPathColumn.DataSize

Declaration

```
property DataSize: Integer;
```

Description

Defines the maximum length of the external data. This can only be used for dpString and dpBinary data types. The data size of the columns and the BufferSize of the TOracleDirectPathLoader affects the maximum number of rows (MaxRows) of the data array.

TDirectPathColumn.DataType

Declaration

```
property DataType: Integer;
```

Description

The external data type of the column. This does not need to match the internal data type, but the external and internal data type must obviously be compatible so that the Direct Load Engine can convert it. The following values can be specified:

- ♦ dpString - This is the most common external data type. If you are loading data from a text file, you can probably use this data type in most cases. Dates must always be loaded as string values.
- ♦ dpInteger - Use this data type if the external data type is a 4 byte integer value.
- ♦ dpFloat - Use this data type if the external data type is an 8 byte double precision floating point value.
- ♦ dpBinary - Use this data type for binary values of Long Raw and BLOB columns. The most important aspect of this data type is that, unlike the dpString data type, no character set conversion between client and server will take place.

For dpInteger or dpFloat data types you cannot specify a DataSize. It will always be 4 or 8 respectively. The dpString data type is the only one that allows a DateFormat.

TDirectPathColumn.DateFormat

Declaration

```
property DateFormat: string;
```

Description

Defines the external date format for date columns. The date format can also be globally defined through the TOracleDirectPathLoader.DateFormat property. If you do not define a date format, then the NLS date format of the session will be used.

TDirectPathColumn.Index

Declaration

property Index: Integer;

Description

The index of the column in the array. You can set this property to move a column to a specific position.

TDirectPathColumn.Name

Declaration

property Name: string;

Description

The name of the column in the table that is to be loaded.

TDirectPathColumn.SetData

Declaration

procedure SetData(Row: Integer; Data: Pointer; Size: Integer);

Description

Use the SetData procedure to associate an element in the data array with a piece of data in memory. The Row parameter indicates the row position in the array, and must be between 0 and MaxRows - 1. The Data parameter points to the data in memory. The Size parameter should contain the size of the string or binary data. For columns with a DataType of dpInteger or dpFloat you do not need to specify the size, which is respectively 4 or 8 by definition.

To pass an integer value for a row of the Empno column, and a string value for the Ename column, you could use the following code:

```
EmpnoCol.SetData(RowIndex, @EmpRec.Empno, 0);  
EnameCol.SetData(RowIndex, @EmpRec.Ename[1], Length(EmpRec.Ename));
```

TOracleQueue component

Unit

Oracle

Description

The TOracleQueue component encapsulates the basic functionality of the DBMS_AQ package. It provides a convenient way to enqueue messages into a queue or to dequeue messages from a queue.

Enqueueing a message

To enqueue a message, connect the TOracleQueue to a Session, set the QueueName property to the name of the queue, set the message information through the Payload or the RawPayload property, specify the EnqueueOptions, and call the Enqueue function.

Dequeueing a message

To dequeue a message, connect the TOracleQueue to a Session, set the QueueName property to the name of the queue, specify the DequeueOptions, call the Dequeue function, and obtain the message information through the Payload or the RawPayload property. To dequeue messages in a background thread of your application, call the StartThread procedure instead of the Dequeue function. In this case the OnThreadDequeued event handler will be called when a message has been dequeued.

Queue Administration

The TOracleQueue does not encapsulate the Queue Management functionality from the DBMS_AQADM package, such as queue creation, starting, stopping, and so on. This functionality has to be explicitly programmed by calling the corresponding packaged functions and procedures, and by creating the corresponding types.

Additional information

For more information about Oracle's Advanced Queueing, see the following Oracle documentation:

- *Application Developer's Guide - Advanced Queueing*
- *Supplied PL/SQL Packages and Types Reference*

TOracleQueue reference

This chapter describes all properties, methods and events of the TOraclePackage component.

TOracleQueue.Cursor

Declaration

```
property Cursor: TCursor;
```

Description

Determines the shape of the mouse cursor while enqueueing or dequeueing messages. Only

crDefault, crHourGlass and crSQLWait are useful here.

TOracleQueue.Debug

Declaration

```
property Debug: Boolean;
```

Description

When set to true all SQL statements that are executed by the TOracleQueue component will be displayed.

TOracleQueue.Dequeue

Declaration

```
function Dequeue: string;
```

Description

Call the Dequeue function to dequeue a message from the queue.

Before calling this function you can set the DequeueOptions to control the dequeue operation.

The dequeue function returns the message identifier (MsgId) of the message. After the call you can use the Payload property to inspect the message information for an object queue (QueueType = qtObject), or you can inspect the RawPayload property in case of a raw queue (QueueType = qtRaw). The MessageProperties contain additional information about the dequeued message.

If the DequeueOptions.Wait property is set and as a result a time out has occurred before a message was dequeued, the return value will be an empty string and MessageProperties.TimeOut will be True.

If the Threaded property is set to True, the Dequeue function will immediately return with an empty string as result, and dequeue processing will occur in a background thread as if StartThread had been called.

TOracleQueue.DequeueOptions

Declaration

```
property DequeueOptions: TAQDequeueOptions;
```

Description

Specifies the options available for the Dequeue operation

```
property Condition: string;
```

A conditional expression based on the message properties, the payload properties, and PL/SQL functions. Only messages that match this condition will be dequeued.

A condition is specified as a Boolean expression using syntax similar to the where clause of a SQL query. This Boolean expression can include conditions on MessageProperties, payload

properties (object queues only), and PL/SQL or SQL functions (as specified in the where clause of a SQL query). MessageProperties include priority, corrid and other columns in the queue table.

To specify dequeue conditions on a message payload, use attributes of the object type. You must prefix each attribute with *tab.user_data* as a qualifier to indicate the specific column of the queue table that stores the payload. The Condition cannot exceed 4000 characters.

property ConsumerName: **string**;

Only those messages matching the ConsumerName are dequeued. If a queue is not set up for multiple consumers, then this field should be set to an empty string.

property Correlation: **string**;

Specifies the correlation identifier of the message to be dequeued. Special pattern matching characters, such as the percent sign (%) and the underscore (_) can be used. If more than one message satisfies the pattern, then the order of dequeuing is undetermined.

property DequeueMode: TAQDequeueMode;

type TAQDequeueMode = (dmBrowse, dmLocked, dmRemove, dmRemoveNoData);

Specifies the locking behavior associated with the dequeue. The possible settings are:

dmBrowse	Read the message without acquiring any lock on the message. This specification is equivalent to a select statement.
dmLocked	Read and obtain a write lock on the message. The lock lasts for the duration of the transaction. This setting is equivalent to a select for update statement.
dmRemove	Read the message and update or delete it. This setting is the default. The message can be retained in the queue table based on the retention properties.
dmRemoveNoData	Mark the message as updated or deleted. The message can be retained in the queue table based on the retention properties.

property MsgId: **string**;

Specifies the message identifier of the message to be dequeued.

property Navigation: TAQDequeueNavigation;

type TAQDequeueNavigation = (dnNextMessage, dnNextTransaction, dnFirstMessage);

Specifies the position of the message that will be retrieved. First, the position is determined. Second, the search criteria are applied. Finally, the message is retrieved. The possible settings are:

dnNextMessage	Retrieve the next message that is available and matches the search criteria. If the previous message belongs to a message group, then AQ retrieves the next available message that matches the search criteria and belongs to the message group. This setting is the default.
dnNextTransaction	Skip the remainder of the current transaction group (if any) and retrieve the first message of the next transaction group. This setting can only be used if message grouping is enabled for the current queue.
dnFirstMessage	Retrieves the first message which is available and matches the

search criteria. This setting resets the position to the beginning of the queue.

property Transformation: **string**;

Specifies a transformation that will be applied after dequeuing the message. The source type of the transformation must match the type of the queue.

property Visibility: TAQDequeueVisibility;

type TAQDequeueVisibility = (dvImmediate, dvOnCommit);

Specifies whether the new message is dequeued as part of the current transaction. The visibility parameter is ignored when using the dmBrowse DequeueMode. The possible settings are:

dvOnCommit	The dequeue will be part of the current transaction. This setting is the default.
dvImmediate	The dequeue is not part of the current transaction. It constitutes a transaction on its own.

property Wait: **Integer**;

Specifies the wait time in seconds if there is currently no message available which matches the search criteria. Specify -1 (AQForever) to wait forever. Specify 0 (AQNoWait) to return immediately if there is no message. If the wait time expires before a message is dequeued, MessageProperties.TimeOut will be set to True on dequeue.

TOracleQueue.Enqueue

Declaration

function Enqueue: **string**;

Description

Call this function to enqueue a message into the queue.

The enqueue function will enqueue the message information from the Payload property for an object queue (QueueType = qtObject), or the RawPayload property in case of a raw queue (QueueType = qtRaw). The MessageProperties contains additional information about the enqueued message.

Before calling this function you can set the EnqueueOptions to control the enqueue operation.

The enqueue function returns the message identifier (MsgId) of the message.

TOracleQueue.EnqueueOptions

Declaration

property EnqueueOptions: TAQEnqueueOptions;

Description

Specifies the options available for the Enqueue operation.

property RelativeMsgId: **string**;

Specifies the message identifier of the message which is referenced in the

SequenceDeviation. This property is valid only if esBefore is specified in SequenceDeviation. This parameter is ignored if SequenceDeviation is esDefault or esTop.

property SequenceDeviation: TAQEnqueueSequence;

type TAQEnqueueSequence = (esDefault, esBefore, esTop);

Specifies whether the message being enqueued should be dequeued before other messages already in the queue. The possible settings are:

esBefore The message is enqueued ahead of the message specified by RelativeMsgld.

esTop The message is enqueued ahead of any other messages.

esDefault The message is enqueued as the last message.

property Transformation: **string**;

Specifies a transformation that will be applied before enqueueing the message. The return type of the transformation function must match the type of the queue.

property Visibility: TAQEnqueueVisibility;

type TAQEnqueueVisibility = (evImmediate, evOnCommit);

Specifies the transactional behavior of the enqueue request. The possible settings are:

evOnCommit The enqueue is part of the current transaction. The operation is complete when the transaction commits. This setting is the default.

evImmediate The enqueue is not part of the current transaction. The operation constitutes a transaction on its own. This is the only value allowed when enqueueing to a non-persistent queue.

TOracleQueue.MessageProperties

Declaration

property MessageProperties: TAQMessageProperties;

Description

Describes the information that is used to manage individual messages. These are set at Enqueue time, and their values are returned at Dequeue time.

property Attempts: Integer;

The number of attempts that have been made to dequeue the message. This property cannot be set at enqueue time.

property Correlation: **string**;

The identification supplied by the producer for a message at enqueue time

property Delay: Integer;

Specifies the delay of the enqueued message in seconds. Specify 0 (AQNoDelay) for no delay. The delay represents the number of seconds after which a message is available for dequeuing. Dequeuing by Msgld overrides the delay specification. A message enqueued with delay set is in the dsWaiting State, and when the delay expires, the message goes to the dsReady state. Delay processing requires the queue monitor to be started.

property EnqueueTime: TDateTime;

Specifies the time the message was enqueued. This value is determined by the system and cannot be set at enqueue time.

property ExceptionQueue: **string**;

Specifies the name of the queue into which the message is moved if it cannot be processed successfully. Messages are moved automatically in the following cases:

- The number of unsuccessful dequeue attempts has exceeded the specification for the `max_retries` parameter in the `dbms_aqadm.create_queue` procedure during queue creation. You can view the `max_retries` for a queue in the `all_queues` data dictionary view.
- All messages in the exception queue are in the `dsExpired` State.

The default is the exception queue associated with the queue table. If the exception queue specified does not exist at the time of the move, then the message is moved to the default exception queue associated with the queue table, and a warning is logged in the alert file. If the default exception queue is specified, then this property returns a empty string at dequeue time.

property Expiration: **Integer**;

The value of this property specifies the number of seconds that the message remains in the `dsReady` State. During this time the message is available for dequeuing. If the message is not dequeued before it expires, then it is moved to the exception queue in the `dsExpires` State. If you specify -1 (`AQNever`) then the message will never expire. This property is an offset from the Delay. Expiration processing requires the queue monitor to be running.

property MsgId: **string**;

The message identifier at dequeue time.

property OriginalMsgId: **string**;

This property is used by Oracle AQ for propagating messages.

property Priority: **Integer**;

Specifies the priority of the message. A smaller number indicates higher priority. The priority can be any number, including negative numbers.

property SenderId: **TAQAgent**;

Specifies the sender identification. You must specify `SenderId` to enqueue messages to secure queues. The `SenderId` has the following properties:

Name (string)	Name of a producer or consumer of a message. The name must follow object name guidelines in the Oracle9i SQL Reference with regard to reserved characters.
Address (string)	Protocol-specific address of the recipient. If the protocol is 0, then the address is of the form [schema.] queue[@ dblink]. For example, a queue named <code>emp_messages</code> in the <code>hr</code> queue at the site <code>dbs1.net</code> has the following address: <code>hr.emp_messages@dbs1.net</code>
Protocol (integer)	Protocol to interpret the address and propagate the message.

property State: **TAQDequeueState**;

type TAQDequeueState = (`dsReady`, `dsWaiting`, `dsProcessed`, `dsExpired`);

Specifies the state of the message at the time of the dequeue. This property cannot be set at enqueue time. The possible states are:

<code>dsReady</code>	The message is ready to be processed.
<code>dsWaiting</code>	The message delay has not yet been reached.

dsProcessed The message has been processed and is retained.

dsExpired The message has been moved to the exception queue.

property `TimeOut`: `Boolean`;

This property indicates at dequeue time that the dequeue operation did not receive a message that matches the search criteria within the Wait time.

See also

`VariableName`

TOracleQueue.OnThreadDequeued

Declaration

```
type TOracleQueueEvent = procedure(Sender: TOracleQueue) of Object;
property OnThreadDequeued: TOracleQueueEvent;
```

Description

If the queue is running in Threaded mode by calling the `StartThread` procedure, all dequeue operations will trigger an `OnThreadDequeued` event. In this event handler you can inspect the `Payload` or `RawPayload` properties to obtain the message information, and inspect the `MessageProperties` to obtain additional information about the dequeued message.

You can call the `StopThread` procedure within this event handler if you do not want to continue to wait for additional messages.

Example

The following example displays the message identifier, the enqueue time, and the `message_text` attribute of the payload object type:

```
procedure TQueueDemoForm.ReceiveQueueThreadDequeued(Sender:
    TOracleQueue);
begin
    Display('Message Identifier: ' + Sender.MessageProperties.Msgid);
    Display('Enqueue Time      : ' + FormatDateTime('c',
        Sender.MessageProperties.EnqueueTime));
    Display('Message Text      : ' +
        Sender.Payload.GetAttr('message_text'));
end;
```

Note

If `ThreadSynchronized` is `True`, this event will be synchronized with the main thread of your application, which implicitly makes your application thread safe.

TOracleQueue.OnThreadError

Declaration

```
type TOracleQueueErrorEvent = procedure(Sender: TOracleQueue;  
    ErrorCode: Integer; const ErrorMessage: string) of Object;  
property OnThreadError: TOracleQueueErrorEvent;
```

Description

If the queue is running in Threaded mode by calling the StartThread procedure, the OnThreadError event is triggered whenever an exception occurs during the dequeue operation. If the exception is an EOracleError, the ErrorCode will be the Oracle error number. For other exception classes the ErrorCode will be 0. The ErrorMessage is the Message of the exception.

You can call the StopThread procedure within this event handler if you do not want to continue to wait for additional messages.

Note

If ThreadSynchronized is True, this event will be synchronized with the main thread of your application, which implicitly makes your application thread safe.

TOracleQueue.OnThreadStart

Declaration

```
type TOracleQueueEvent = procedure(Sender: TOracleQueue) of Object;  
property OnThreadStart: TOracleQueueEvent;
```

Description

If the queue is running in Threaded mode by calling the StartThread procedure, the OnThreadStart event is triggered when the background thread starts.

Note

If ThreadSynchronized is True, this event will be synchronized with the main thread of your application, which implicitly makes your application thread safe.

TOracleQueue.OnThreadStop

Declaration

```
type TOracleQueueEvent = procedure(Sender: TOracleQueue) of Object;  
property OnThreadStop: TOracleQueueEvent;
```

Description

If the queue is running in Threaded mode by calling the StartThread procedure, the OnThreadStop event is triggered when the background thread stops.

Note

If ThreadSynchronized is True, this event will be synchronized with the main thread of your application, which implicitly makes your application thread safe.

TOracleQueue.OnThreadTimeOut

Declaration

```
type TOracleQueueEvent = procedure(Sender: TOracleQueue) of Object;  
property OnThreadTimeOut: TOracleQueueEvent;
```

Description

If the queue is running in Threaded mode by calling the StartThread procedure, the OnThreadTimeOut event is triggered when the dequeue operation dequeue operation does not receive a message that matches the search criteria within the Wait time.

You can call the StopThread procedure within this event handler if you do not want to continue to wait for messages.

Note

If ThreadSynchronized is True, this event will be synchronized with the main thread of your application, which implicitly makes your application thread safe.

TOracleQueue.Payload

Declaration

```
property Payload: TOracleObject;
```

Description

Use the Payload run-time property to set the message information before an Enqueue operation, or to get the message information after a Dequeue operation for object queues (QueueType = qtObject). The Payload property is a TOracleObject instance, so you can use the GetAttr and SetAttr methods to get and set the attribute values.

You must not free the Payload instance. The TOracleQueue is responsible for this.

The Payload property is nil for raw queues (QueueType = qtRaw). In that case you need to use the RawPayload property instead.

Example

The following example sets the message_text and message_type attributes before enqueueing a message and committing it:

```
procedure TQueueDemoForm.SendButtonClick(Sender: TObject);  
begin  
    // Create the message  
    SendQueue.Payload.SetAttr('message_text', SendMessageEdit.Text);  
    SendQueue.Payload.SetAttr('message_type', 'Info');  
    // Enqueue the message  
    SendQueue.Enqueue;  
    // Commit the enqueue operation  
    SendSession.Commit;  
end;
```

Note

This property is only available in the Object version of Direct Oracle Access.

TOracleQueue.PayloadType

Declaration

```
property PayloadType: string;
```

Description

This run-time property contains the name of the object type (owner.name) of the Payload of the queue. This corresponds to the payload type that was specified when the queue table was created through the `dbms_aqadm.create_queue_table` procedure (the `queue_payload_type` parameter).

For raw queues (QueueType = `qtRaw`) the value of this property will be 'RAW'.

TOracleQueue.QueueName

Declaration

```
property QueueName: string;
```

Description

Use this property to indicate the name of the queue. This must match the name that was used when the queue was created through the `dbms_aqadm.create_queue` procedure (the `queue_name` parameter).

Note

You must set the QueueName property and the Session property and connect the session, before you can use most of the other TOracleQueue methods and properties.

TOracleQueue.QueueType

Declaration

```
type TAQQueueType = (qtObject, qtRaw);  
property QueueType: TAQQueueType;
```

Description

This run-time property indicates whether the queue is an object queue or a raw queue. For object queues (QueueType = `qtObject`) you need to access the Payload property to get or set the message information. For raw queues you need to access the RawPayload property.

TOracleQueue.RawPayload

Declaration

```
property RawPayload: string;
```

Description

Use the RawPayload run-time property to set the message information before an Enqueue operation, or to get the message information after a Dequeue operation for raw queues

(QueueType = qtRaw). The RawPayload property is a string that contains the message information in binary form.

The RawPayload property cannot be used for object queues (QueueType = qtObject). In that case you need to use the Payload property instead.

TOracleQueue.Session

Declaration

```
property Session: TOracleSession;
```

Description

Set the Session property to assign the TOracleQueue instance to a session. If you are using the queue in Threaded mode, you should dedicate this session to just the queue instance it is assigned to, because the session will be locked while the queue is waiting for messages. Only during the OnThread events (OnThreadDequeued, OnThreadTimeOut, and so on) is the session available for other database access operations.

Note

You must set the QueueName property and the Session property and connect the session, before you can use most of the other TOracleQueue methods and properties.

TOracleQueue.StartThread

Declaration

```
procedure StartThread;
```

Description

Use the StartThread procedure to dequeue messages in a background thread of your application. Whenever a successful dequeue operation occurs, the OnThreadDequeued event handler will be called so that your application can process the message. If an error occurs during the dequeue operation, the OnThreadError event handler will be called. In case of a time out (as defined by the DequeueOptions.Wait property), the OnThreadTimeOut event handler will be called.

To stop the background thread, call StopThread (requires that DequeueOptions.Wait > 0).

Note

If ThreadSynchronized is True, all OnThread event handler calls will be synchronized with the main thread of your application, which implicitly makes your application thread safe.

TOracleQueue.StopThread

Declaration

```
procedure StopThread;
```

Description

To stop the background dequeueing thread that was started by calling StartThread or by calling Dequeue while Threaded is True, you can call the StopThread procedure. Note that this procedure cannot interrupt the current dequeue operation. It depends on a timeout (requires that DequeueOptions.Wait > 0) or the actual dequeueing of a message. If DequeueOptions.Wait = 5, then it can take up to 5 seconds before the background thread is actually terminated.

TOracleQueue.TableName

Declaration

```
property TableName: string;
```

Description

The name of the queue table, as defined through the dbms_aqadm.create_queue procedure (the queue_table parameter). The name does not include the table owner.

TOracleQueue.TableOwner

Declaration

```
property TableOwner: string;
```

Description

The owner of the queue table, as defined through the dbms_aqadm.create_queue procedure (the owner of the queue_table parameter).

TOracleQueue.Threaded

Declaration

```
property Threaded: Boolean;
```

Description

When set to True, subsequent calls to the Dequeue function will be processed in a background thread. This is equivalent to calling StartThread.

TOracleQueue.ThreadIsRunning

Declaration

property ThreadIsRunning: Boolean;

Description

Indicates if the background thread for dequeuing is currently running, either by calling StartThread or by calling Dequeue while Threaded is True.

TOracleQueue.ThreadSynchronized

Declaration

property ThreadSynchronized: Boolean;

Description

When True, all OnThread event handler calls (OnThreadDequeued, OnThreadTimeOut, and so on) will be synchronized with the main thread of your application, which implicitly makes your application thread safe. When False, the event handlers will be called asynchronously, and you have to take care of thread safety issues in your application.

TOracleSessionPool component

Unit

Oracle

Description

In server applications that frequently need to create sessions to process requests from client applications, it may be useful to use connection pooling. Without a pooling concept, each request could lead to an actual database logon and logoff. By using the TOracleSessionPool component, you can maintain a pool of database sessions that can be used and reused for different requests. You can define the minimum and maximum number of sessions in the pool, define the TimeOut behavior for idle sessions in the pool, and preset the username, password and database.

Once you have created a pool, you can assign a TOracleSession to this pool by setting its Pool property.

The global pool

Instead of creating a specific session pool and assigning sessions to it, you can alternatively make use of the global pool. The Oracle unit contains a global SessionPool variable that will be used for pooled sessions that do not have their Pool property set. Using the global pool will be sufficient for most applications. Note however that you need to set the SessionPool properties before connecting any pooled sessions.

Default pooling vs Oracle pooling (Oracle 9.2 or later)

Oracle 9.2 introduced its own session pooling mechanism. You can make use of this functionality by setting the PoolType property to ptOracle. Oracle 9.2 pooling is more efficient, but also has the restriction that you cannot have sessions for more than one database in the same pool. If this restriction is not a problem, you should use Oracle pooling.

TOracleSessionPool reference

This chapter describes all properties, methods and events of the TOracleSessionPool component.

TOracleSessionPool.AfterReserve

Declaration

```
type TSessionPoolSessionEvent = procedure(Sender: TOracleSessionPool;  
    Session: TOracleSession) of Object;  
property AfterReserve: TSessionPoolSessionEvent;
```

Description

This event is fired after a session reserves a connection from the pool.

TOracleSessionPool.BeforeRelease

Declaration

```
type TSessionPoolSessionEvent = procedure(Sender: TOracleSessionPool;  
    Session: TOracleSession) of Object;  
property BeforeRelease: TSessionPoolSessionEvent;
```

Description

This event is fired just before a session releases a connection back into the pool.

TOracleSessionPool.Compress

Declaration

```
procedure Compress;
```

Description

This procedure will remove all currently unused connections from the pool. The number of connections will not sink beneath the value indicated by the Min property though.

TOracleSessionPool.CompressOld

Declaration

```
procedure CompressOld;
```

Description

This procedure will remove connections from the pool that have been unused longer than indicated by the TimeOut value. The number of connections will not sink beneath the value indicated by the Min property though.

TOracleSessionPool.Count

Declaration

```
property Count: Boolean;
```

Description

This property indicates the number of (used and unused) connections in the pool. You can use this value to access the Sessions array.

TOracleSessionPool.Homogeneous

Declaration

```
property Homogeneous: Boolean;
```

Description

Indicates that all connections in the pool should use the same LogonUsername,

LogonPassword and LogonDatabase. If a session connects through a Homogeneous pool, its logon properties will be overruled by the logon properties of the pool.

TOracleSessionPool.Lock

Declaration

```
procedure Lock;
```

Description

This procedure will lock the pool. Until you call Unlock, other application threads cannot connect or disconnect a session through the pool.

TOracleSessionPool.LogonDatabase

Declaration

```
property LogonDatabase: string;
```

Description

The logon database in case of a Homogeneous pool.

TOracleSessionPool.LogonPassword

Declaration

```
property LogonPassword: string;
```

Description

The logon password in case of a Homogeneous pool.

TOracleSessionPool.LogonUsername

Declaration

```
property LogonUsername: string;
```

Description

The logon username in case of a Homogeneous pool.

TOracleSessionPool.Max

Declaration

```
property Max: Integer;
```

Description

The maximum allowed number of sessions in the pool. If a session tries to connect through

the pool while this maximum has been reached, it will wait until another application thread disconnects a session, thereby releasing its connection back into the pool.

TOracleSessionPool.Min

Declaration

```
property Min: Integer;
```

Description

The minimum number of connections in the pool. When connections are removed from the pool (due to calls to `Compress`, `CompressOld`, or due to the `TimeOut` value), this is the minimum number of connections that will remain.

TOracleSessionPool.PoolName

Declaration

```
property PoolName: string;
```

Description

In case of an Oracle pool (`PoolType = ptOracle`), this property indicates the unique name of the pool.

TOracleSessionPool.OnClose

Declaration

```
type TSessionPoolEvent = procedure(Sender: TOracleSessionPool) of  
Object;  
property OnClose: TSessionPoolEvent;
```

Description

This event is fired just before the pool closes. This occurs when the pool instance is freed.

TOracleSessionPool.OnOpen

Declaration

```
type TSessionPoolEvent = procedure(Sender: TOracleSessionPool) of  
Object;  
property OnOpen: TSessionPoolEvent;
```

Description

This event is fired after the pool has opened. This occurs when the first session tries to reserve a connection from the pool.

TOracleSessionPool.PoolType

Declaration

```
type TSessionPoolType = (ptDefault, ptOracle);  
property PoolType: TSessionPoolType;
```

Description

Oracle 9.2 introduced its own session pooling mechanism. You can make use of this functionality by setting this property to ptOracle. Oracle 9.2 pooling is more efficient, but also has the restriction that you cannot have sessions for more than one database in the same pool. If this restriction is not a problem, you should use Oracle pooling.

Note

If you set this property to ptOracle when using Oracle 9.0 or earlier, the pool will fall back to ptDefault.

TOracleSessionPool.Sessions

Declaration

```
property Sessions: [Index: Integer]: TOracleSession;
```

Description

This array property contains a session for each connection index. If the connection at this index is currently not being used by a session, the array element will be nil.

Note

If you want to make use of the Sessions array, make sure that you call Lock before accessing it, and that you call Unlock afterwards. Otherwise the array elements may be changed by other application threads while you are processing them.

TOracleSessionPool.StatementCache

Declaration

```
property StatementCache: Boolean;
```

Description

This property indicates whether or not sessions that connect through the pool will use a client side statement cache. It requires Oracle 9.2 or later, and will be ignored for older Oracle versions.

TOracleSessionPool.TimeOut

Declaration

```
property TimeOut: Integer;
```

Description

The TimeOut property defines how long (in seconds) an unused connection may remain in the

pool. If a connection has been unused longer, it will be removed from the pool, unless the number of connections will sink below the value defined by the Min property.

The actual process of removing unused sessions will occur when other sessions connect or disconnect through the pool.

TOracleSessionPool.Unlock

Declaration

```
procedure Unlock;
```

Description

This procedure will unlock the pool after locking it with the Lock procedure.

TOracleTimestamp object

Unit

Oracle

Description

This object encapsulates the Oracle8i Timestamp datatype. It contains properties for all separate date and time elements, and also allows you to manipulate the value as a TDateTime or string. The string can either be formatted in the Windows / Delphi format, or in Oracle format.

A TOracleTimestamp instance can be used in the following ways:

- ♦ It will be returned by the TOracleQuery.TimestampField function.
- ♦ You can set a Timestamp SQL or PL/SQL variable by passing it to the SetComplexVariable procedure, and get or set its properties before and after executing the SQL or PL/SQL.
- ♦ You can get a Timestamp attribute through the TOracleObject.TimestampAttr function, and get or set its properties.

TOracleTimestamp reference

This chapter describes all properties and methods of the TOracleTimestamp object.

TOracleTimestamp.AsDateTime

Declaration

```
property AsDateTime: TDateTime;
```

Description

Creates a new

TOracleTimestamp.AsOracleString

Declaration

```
property AsOracleString: string;
```

Description

Creates a new

TOracleTimestamp.Assign

Declaration

```
procedure Assign(Source: TOracleTimestamp);
```

Description

Creates a new

TOracleTimestamp.AsString

Declaration

```
property AsString: string;
```

Description

Creates a new

TOracleTimestamp.Clear

Declaration

```
procedure Clear;
```

Description

Creates a new

TOracleTimestamp.Create

Declaration

```
constructor Create(ASession: TOracleSession; ADataType: Integer);
```

Description

Creates a new TOracleTimestamp instance for the given session and data type. The data type can be one of the following values:

otTimestamp	TIMESTAMP
otTimestampTZ	TIMESTAMP WITH TIME ZONE
otTimestampLTZ	TIMESTAMP WITH LOCAL TIME ZONE

This constructor is primarily useful when assigning a Timestamp variable through SetComplexVariable.

TOracleTimestamp.DataType

Declaration

```
property DataType: Integer;
```

Description

The data type of a timestamp can be one of the following values:

otTimestamp	TIMESTAMP
otTimestampTZ	TIMESTAMP WITH TIME ZONE
otTimestampLTZ	TIMESTAMP WITH LOCAL TIME ZONE

TOracleTimestamp.Day

Declaration

```
property Day: Byte;
```

Description

The Day (1..31) of the Month.

TOracleTimestamp.Hour

Declaration

```
property Hour: Byte;
```

Description

The Hour (0..23) of the Day.

TOracleTimestamp.IsNull

Declaration

```
property IsNull: Boolean;
```

Description

Indicates that the timestamp value is null. A timestamp will be null after the instance is created, after calling Clear, or after retrieving a timestamp field, variable or attribute that is set to null on the server.

TOracleTimestamp.Minute

Declaration

```
property Minute: Byte;
```

Description

The Minute (0..59) of the Hour.

TOracleTimestamp.Month

Declaration

```
property Month: Byte;
```

Description

The Month (1..12) of the Year.

TOracleTimestamp.NanoSeconds

Declaration

```
property NanoSeconds: Cardinal;
```

Description

The NanoSeconds (0..999,999,999) of the Second.

TOracleTimestamp.Second

Declaration

```
property Second: Byte;
```

Description

The Second (0..59) of the Minute.

TOracleTimestamp.Session

Declaration

property Session: TOracleSession;

Description

The Session for which the timestamp was created. This can explicitly be specified when created, or is implicitly defined when the timestamp is obtained from a field or attribute.

TOracleTimestamp.SetValues

Declaration

procedure SetValues(AYear: SmallInt; AMonth, ADay, AHour, AMinute,
 ASecond: Byte; ANanoSeconds: Cardinal);

Description

Sets the Year, Month, Day, Hour, Minute, Second and NanoSeconds values of the timestamp. If the date/time is invalid, an exception will be raised.

TOracleTimestamp.Year

Declaration

property Year: SmallInt;

Description

The Year (-9999..9999) of the timestamp.

TXMLType object

Unit

Oracle

Description

The SYS.XMLTYPE object was introduced in Oracle9 to encapsulate XML in the Oracle database. In Oracle9.2 this object type can be accessed from the Oracle Net 9.2 client. The TXMLType provides an easy interface to the SYS.XMLTYPE object type. The TXMLType object descends from TOracleObject, and can therefore be used anywhere that you can use a TOracleObject (for example, when assigning an object instance to a variable through SetComplexVariable).

The XML text of a TXMLType instance can be set during creation:

```
var
    XML: TXMLType;
begin
    XML := TXMLType.Create(MainSession, '<message>Hello
        World</message>');
    try
        XMLQuery.SetComplexVariable('xmlvar', XML);
        XMLQuery.Execute;
    finally
        XML.Free;
    end;
end;
```

The XML text of a TXMLType instance can be retrieved through the XML read-only property.

The TOracleQuery.XMLField and TOracleObject.XMLAttr functions return a TXMLType instance for a field or attribute.

Note

The XMLType object can only be used with Oracle9.2.

TXMLType reference

This chapter describes all properties and methods of the TXMLType object.

TXMLType.Create

Declaration

```
constructor Create(ASession: TOracleSession; const AnXML: string);
```

Description

Creates a new TXMLType instance for the specified session and XML text.

TXMLType.XML

Declaration

```
property XML: string;
```

Description

Retrieves the XML Text for the TXMLType instance.

The Package Wizard

The Package Wizard is a powerful and easy to use utility that generates Delphi classes that encapsulate your Oracle Packages. Without the Package Wizard, you have to define a TOracleQuery with the right PL/SQL Block, define variables for the parameters, set the variable values, execute the query, and get the variable values for the output parameters or function result. The TOraclePackage component has already made this process easier, as you can simply call the program unit and pass the parameters in one call. However, this component has some limitations because of its simplified interface.

The Package Wizards creates classes for a selection of Oracle Package that contain exactly the procedures and functions of these packages. If the parameters are record types, it will also generate classes to encapsulate these record types. Let's assume the following Department package as an example:

```
create or replace package Department is
  function Employee_Count(Deptno in dept.deptno%type) return
    binary_integer;
  procedure Select_Record(Dept_Record in out dept%rowtype);
  procedure Insert_Record(Dept_Record in dept%rowtype);
  procedure Update_Record(Dept_Record in dept%rowtype);
  procedure Delete_Record(Deptno in dept.deptno%type);
  procedure Get_Description(Deptno in dept.deptno%type, Description
    out CLOB);
end Department;
```

By a simple press of a button, the Package Wizard will generate the following 2 classes for this package:

```
type
  DeptRowtype = class(TPLSQLRecord)
  public
    Deptno: Integer;
    Dname: string;
    Loc: string;
    procedure Assign(Source: TPLSQLRecord); override;
  end;
  TDepartment = class(TOracleCustomPackage)
  public
    function EmployeeCount(Deptno: Integer): Integer;
    procedure SelectRecord(var DeptRecord: DeptRowtype);
    procedure InsertRecord(DeptRecord: DeptRowtype);
    procedure UpdateRecord(DeptRecord: DeptRowtype);
    procedure DeleteRecord(Deptno: Integer);
    procedure GetDescription(Deptno: Integer; out Description:
      TLOBLocator);
  published
    property Name;
    property Session;
    property Cursor;
    property PackageSpecification;
  end;
```

The DeptRowType class encapsulates the dept%rowtype parameters used by the procedures.

The TDepartment class encapsulates the Department package, and contains an exact representation of the 6 program units. To call these stored program units, you merely need to create an instance of the TDepartment class, set the Session property, and call the corresponding functions or procedures:

```
Department := TDepartment.Create(nil);
Department.Session := MainSession;
EmpCount := Department.EmployeeCount(10);
Department.Free;
```

As you can see the TDepartment class descends from the TOracleCustomPackage class, which in turn is a TComponent descendant. This implies that you can install these Oracle Packages into a Delphi or C++Builder package and place them on a component palette. Now you can conveniently place the TDepartment component on a data module or form at design time, link it to a TOracleSession component at design time, and call the functions and procedures at run time without any additional code to create, setup or free the TDepartment instance:

```
EmpCount := MainDataModule.Department.EmployeeCount(10);
```

We can't possibly make it easier than this. The Oracle Server software has now become a natural extension of your Delphi or C++Builder programming language!

Advantages

Using the Package Wizard has many advantages over the use of TOracleQuery or TOraclePackage components. Some of these advantages are obvious, and some will occur to you as you use the generated packages:

- No need to create and maintain components to call the packaged program units. You can simply generate and regenerate your packages with the Package Wizard. This will be a tremendous time-saver.
- Delphi and C++Builder's code completion will of course work for the packages. Have you forgot the name of a program unit or parameter? Have you forgot if a parameter is input and/or output? You can now find this information as easily as you would find information about other Delphi or C++Builder classes.
- As you don't access any function, procedure or parameter by name, you can't possibly make a mistake that wouldn't be reported at compile-time. If it can compile it can run. For the TOracleQuery or TOraclePackage approach you would find these errors at run-time.
- The Package Wizard will generate classes that encapsulate PL/SQL Record types. Program units that have record type parameters can now be used as easy as any other program unit.
- The Package Wizard makes use of a special class that encapslates scalar PL/SQL table types, which makes these parameter types easier to use.
- In Delphi or C++Builder 4 and later, you will see overloaded program units just like they are.
- If you put some form of documentation into your Package Specifications, then you can see this information in the generated classes. If you are using the packages as components, the package specification is visible as a read-only PackageSpecification property.
- Calls to packaged functions and procedures are automatically thread safe without the need

to explicitly program critical sections. This means that you can call a single package instance from multiple threads simultaneously.

- Because the package classes are generated into separate units, you can automatically reuse them in multiple projects.
- It's fun to generate code and to show your boss that you have 'produced' thousands of lines of code in just 1 day.

Using the Package Wizard

To start the Package Wizard, you can select the 'File | New' menu item in the Delphi or C++Builder IDE, and double click on the 'Oracle Package' icon. You can also go to the 'Oracle' menu and select the 'Package Wizard' item. This will display the first of the 4 steps of the package generation.

Step 1 - Select the packages

Connect to the database, using an Oracle account that has access to the Oracle Packages for which you want to generate the classes. After you make a connection you will be presented with a selection list of packages. The 'Show all packages' checkbox can be used to limit the selection to just the packages that are owned by the current Oracle account.

Simply select one or more packages and press the 'Next' button.

Step 2 - Define interface translation rules

You can define the following translation rules for the packages:

Always use variants as parameters

Instead of string, integer, double, and TDateTime parameters you can alternatively generate variant parameters. This has the advantage that the parameters can always represent null values. For an integer or double parameter there is no difference between null and zero. The disadvantage of this option is that you can't generate overloaded methods, which will now be distinguished by adding an overload identifier to the name (1, 2, and so on). Another disadvantage is that you can't quickly determine the parameter data types at design time, the Code Completion will always show 'Variant' as the data type.

Generate overloaded methods

Enable this option to generate overloaded methods. This is only possible for Delphi or C++Builder 4 and later. If this option is disabled, overloaded program units will be distinguished by adding an overload identifier to the name (1, 2, and so on)

Case

This option controls the use of upper- and lowercase characters in object, method, and parameter names. There are 4 choices:

- Unchanged - Use the names as they are defined in the Oracle Package. Note that this will typically be all uppercase characters, unless you have used "quoted" identifiers!
- Capitalize - Starts each word with a capital, and makes rest of the word lowercase. Words are separated by underscores, so INSERT_EMPLOYEE would become Insert_Employee. This is the default setting, as it converts between typical Oracle naming conventions and Delphi / C++Builder naming conventions, if you have also enabled the 'Remove underscores' option.

- ♦ Uppercase - Convert all characters to uppercase.
- ♦ Lowercase - Convert all characters to lowercase.

Remove underscores

If you enable this option, the underscores in Oracle identifiers are removed. This is typically used in combination with the 'Capitalize' Case option. Using this combination would convert INSERT_EMPLOYEE to InsertEmployee.

Prefix objects with T

In Delphi and C++Builder you expect class names to start with a T. If you enable this option, then this prefix will be added to the Oracle package name. Package DEPARTMENT would result in a TDepartment class if used in combination with the 'Capitalize' Case option.

Prefix parameters with A

You may have adopted the coding style to add 'A' to method parameters. Enabling this option will apply this style to the generated parameter names. If, for example, a method has a DEPTNO parameter, this will be translated to ADeptno. Using this option will also eliminate most of the 'reserved word' conflicts for parameters (see below).

Some of these preferences affect the names of the generated classes. An example is displayed at the bottom of the page.

Step 3 - View and rename elements

After you have defined the translation rules, you are presented with the resulting packages, methods, and parameters. At this point you may choose to rename any of these names, and to deselect individual methods. You can of course also modify the resulting source later.

If your packages contain program units that have PL/SQL Table of records as parameters, then these program units will be deselected because they cannot be called by Direct Oracle Access.

Some of the names may have been implicitly changed by the Package Wizard to avoid 'reserved word' conflicts. An identifier name like 'object', 'type' or 'program' is perfectly legitimate in PL/SQL, but will lead to a compilation error for the generated class. When such a reserved word is encountered, a '1' will be added to the name.

Step 4 - Generate the source file

Before generating the source file you can set the following options:

Prefix database objects with schema name

If you enable this option each database object will be prefixed with the schema name. If database user SCOTT owns package DEPARTMENT, then a resulting call might look like this:

```
begin
    :result := scott.department.employee_count(deptno => :deptno);
end;
```

If the Oracle accounts of the end-users of your application do not have (private or public) synonyms for these objects, and if in each database your packages are owned by SCOTT, then this may be an appropriate setting. If you disable this option, the following code will be generated:

```
begin
  :result := department.employee_count(deptno => :deptno);
end;
```

Now you must make sure that either all users connect to the database using the SCOTT account, or that the Oracle accounts of the end-users have (private or public) synonyms for the SCOTT.DEPARTMENT package

Generate thread safe code

Enable this option to generate thread safe code. This will result in package classes that can be called from multiple threads simultaneously without explicitly programming critical sections. There is of course a (very small) performance trade-off involved with this option.

Include package specification

The source of a package specification will typically contain some documentation about the program units and types that it contains. You can include the package specification in the generated source file by enabling this option. If the packages are not generated as components (see below), then it will be included as comment in the source file. If the packages are generated as components, the specification will be included as a PackageSpecification (read-only) property, which you can of course inspect at design-time.

Besides having some documentation about the package close by, there is a second advantage to this option. You will always have an exact image of the package that the classes were generated against. If your packages are changed with some frequency, this can be useful information.

Generate as components

Enable this option to generate registration code for the package components. Doing so will allow you to add them to a Delphi or C++Builder package and to add them to a component palette. Treating packages as components can be very convenient, as it eliminates the need to explicitly create and free package instances, and you can link them to a session at design-time. It also has the advantage that you can view the PackageSpecification property at design time.

Create component dcr file

Enable this option to generate a resource file with a default icon (green package symbol) for each package class. You can use the Image Editor to make the icons visually represent the functionality of the packages.

Component palette

If you want to add the package components to a component palette, you can make a selection out of the existing palettes or enter the name of a new palette, which will automatically be created during installation of the Delphi or C++Builder package.

Path

Enter or select the path of the source file you are about to generate. If you leave this field empty, the source file will be placed in the directory of the project that is currently opened.

Filename

Enter the filename of the source file you are about to generate. The default extension is .pas.

Add to project

Enable this option to automatically add the generated source file to the current project.

Open in IDE

Enable this option to automatically open the generated source file in the IDE.

Using the generated classes

Each Oracle package is implemented in a class that descends from a `TOracleCustomPackage`. This base class does not contain any functionality that you would typically use, except for the following 2 properties:

- ♦ **Cursor** - The shape of the mouse cursor during execution of a function or procedure
- ♦ **Session** - The `TOracleSession` instance that this package instance will use

The `TOracleCustomPackage` class descends from the `TComponent` class, so each package inherits methods and properties from `TComponent` as well. To create a `TDepartment` instance, link it to a session, call a function, and free the package instance again, you could use the following code:

```
Department := TDepartment.Create(nil);
Department.Session := MainSession;
Department.Cursor := crSQLWait;
EmpCount := Department.EmployeeCount(10);
Department.Free;
```

During execution of the `EmployeeCount` function, the shape of the mouse cursor will be the SQL hourglass.

Parameter types

Scalar PL/SQL parameter types are represented by string, integer, double and `TDateTime` data types in the package class. If you have enabled the 'Always use variants for parameters' option, scalar parameters will instead be represented by `Variants`. Complex PL/SQL parameter types are represented by specific classes, and are discussed in the following chapters.

For complex input or input/output parameters, you need to create an instance of the corresponding class and pass it to the method. For output parameters the corresponding instance will be created inside the method, so you should not create an instance in this situation. This would just lead to memory leaks, as the instance will be overwritten. In all situations the application is responsible for freeing the instance.

Ref Cursor

This parameter type is represented by the `TOracleQuery` class. After calling the method, you need to execute the `TOracleQuery` before fetching the rows. If the `GetEmployees` procedure returns a cursor for all employees with a department, then the code to call this method could look like this:

```

Department := TDepartment.Create(nil);
Department.Session := MainSession;
Department.GetEmployees(10, EmpQuery);
EmpQuery.Execute;
while not EmpQuery.Eof do
begin
    ShowMessage(EmpQuery.Field('ename'));
    EmpQuery.Next;
end;
EmpQuery.Free;
Department.Free;

```

CLOB, BLOB and BFILE

These 3 LOB data types are represented by the TLOBLocator class. For input or input/output LOB parameters you must make sure that you create a TLOBLocator instance of the correct type (otCLOB, otBLOB, otBFILE). Assuming that the Department package has a GetDescription procedure that returns a description CLOB for a department, the code to retrieve this CLOB could look like this:

```

Department := TDepartment.Create(nil);
Department.Session := MainSession;
Department.GetDescription(10, Description);
ShowMessage(Description.AsString);
Description.Free;
Department.Free;

```

Objects and References

Oracle8 object types and references are represented by the TOracleObject and TOracleReference classes. For input or input/output objects and references, you must make sure that the instance you created has the right type name:

```

Department := TDepartment.Create(nil);
Department.Session := MainSession;
Address := TOracleObject.Create(MainSession, 'TAddress', '');
Address.SetVariable('City', 'New York');
Department.SetAddress(10, Address);
Address.Free;
Department.Free;

```

PL/SQL Tables

The Package Wizard only supports PL/SQL Tables of scalar data types. The TPLSQLTable class represents a PL/SQL Table for the TOracleCustomPackage.


```

Department := TDepartment.Create(nil);
Department.Session := MainSession;
EmpTable := TPLSQLTable.Create(10, 0);
EmpTable.Count := 3;
EmpTable[0] := 7389;
EmpTable[1] := 6711;
EmpTable[2] := 8556;
Department.DeleteEmployees(EmpTable);
EmpTable.Free;
Department.Free;

```

In this example a PL/SQL Table is created for up to 10 elements, filled with 3 elements, and passed to the DeleteEmployees procedure. Note that you do not explicitly create PL/SQL Table instances for output parameters, so you cannot control the maximum size of the table. The maximum string size is known within the package class, but the maximum table size depends on the implementation. The unit of the package class contains a DefaultPLSQLTableSize variable, which defines the table size for output parameters. The default value is 100.

Note: the values of a PL/SQL Table will always be represented by Variants, even if you have disabled the 'Always use variants for parameters' option.

Record types

For each record type that is used as a parameter within a package, a corresponding class is created in the unit. The name of this class is derived from the package specification, and descends from the abstract TPLSQLRecord class. This base class defines the Create constructor and Assign method. If, for example, you use a dept%rowtype parameter, a corresponding DeptRowtype class is declared that contains the necessary components:

```

DeptRowtype = class(TPLSQLRecord)
public
    Deptno: Integer;
    Dname: string;
    Loc: string;
    procedure Assign(Source: TPLSQLRecord); override;
end;

```

If you have enabled the 'Always use variants for parameters' option, scalar record components will be represented by Variants. Complex record components are represented by the classes discussed in this chapter. The Create constructor of the record type takes the session as a parameter, so that it can automatically create TLOBLocator, TOracleObject and TOracleReference instances within this record type instance. If it contains record type components, the corresponding instances will be created as well. To call the SelectRecord procedure, you could use the following code:

```
Department := TDepartment.Create(nil);  
Department.Session := MainSession;  
DeptRecord := DeptRowtype.Create(MainSession);  
DeptRecord.Deptno := 10;  
Department.SelectRecord(DeptRecord);  
ShowMessage(DeptRecord.Dname);  
DeptRecord.Free;  
Department.Free;
```

The DeptRecord.Free destructor will also destroy the instances that may have been created for complex record components during the Create procedure.

TOracleCustomPackage component

Unit

Oracle

Description

The TOracleCustomPackage component is the base class for packages generated with the Package Wizard. These generated package components encapsulate the functions and procedures within the Oracle package, so that you can simply call them as if they were methods of normal Delphi or C++Builder classes. The TOracleCustomPackage is derived from TComponent, and therefore inherits its methods and properties.

If you are using generated packages as design time components, you can simply place your custom packages on a data module or main form, and link it to a TOracleSession instance through the Session property. In this case you can simply call the functions and procedures at run time without worrying about creating and freeing custom package instances.

If you are creating TOracleCustomPackage descendants at run time, you must use the standard Create method inherited from TComponent. It has an Owner as parameter, which can be nil or another TComponent like a form or data module. In the first case you need to free the instance later, in the second case the Owner will free the instance.

After creating the instance you need to set the Session property before you can call any of the functions or procedures.

The following example creates a custom package, assigns the session, calls a function, and frees the instance.

```
Department := TDepartment.Create(nil);
try
  Department.Session := MainSession;
  EmpCount := Department.EmployeeCount(10);
finally
  Department.Free;
end;
```

During the process of generating the custom package, you can specify many options that control several interface and implementation aspects of the resulting class.

TOracleCustomPackage reference

This chapter describes all properties and methods of the TOracleCustomPackage component.

TOracleCustomPackage.Create

Declaration

```
constructor Create(AOwner: TComponent);
```

Description

Because the TOracleCustomPackage descends from the TComponent class, it has inherited its constructor. The AOwner parameter can either be another component that will free the TOracleCustomPackage instance when the owner itself is freed, or it can be nil, in which case you have to explicitly free the instance yourself.

If you create custom package instances at design time by placing them on a form or data module, you never need to create or free these instances.

Note: Before you can call any function or procedure of a dynamically created custom package, you must set its Session property.

TOracleCustomPackage.Cursor

Declaration

```
property Cursor: TCursor;
```

Description

Determines the shape of the mouse cursor while calling the package's functions or procedures. Only crDefault, crHourGlass and crSQLWait are useful here.

TOracleCustomPackage.Name

Declaration

```
property Name: string;
```

Description

Use the Name property to change the name of a custom package component at design time to reflect the actual Oracle package it encapsulates. By default, Delphi or C++Builder assigns sequential names based on the type of the component.

TOracleCustomPackage.PackageSpecification

Declaration

property PackageSpecification: TStrings;

Description

If you have used the Package Wizard with both the 'Include package specification' option and the 'Generate as components' option enabled, then this read-only property contains the source of the specification of the package against which the custom package was generated.

If you have added some form of documentation to the package specification, then this property can serve as reference information. You can also use this property to check if the package in the database still matches the generated code, or if you need to regenerate it.

TOracleCustomPackage.Session

Declaration

property Session: TOracleSession;

Description

The session in which the package will execute. You must set this property before you can call any of the package's functions or procedures.

TPLSQLRecord object

Unit

Oracle

Description

The TPLSQLRecord object is the abstract base class for PL/SQL record type parameters for the TOracleCustomPackage component, generated through the Package Wizard. If, for example, the Oracle package contains a function with a dept%rowtype parameter, the following class would be generated:

```
type
  DeptRowtype = class(TPLSQLRecord)
  public
    Deptno: Integer;
    Dname: string;
    Loc: string;
    procedure Assign(Source: TPLSQLRecord); override;
  end;
```

The abstract TPLSQLRecord base class does not have any properties. All properties are defined in the generated classes, and will exactly correspond to the components that make up the PL/SQL record. The following example creates a DeptRowtype instance, sets the component values, and passes it to the TDepartment.InsertRecord procedure:

```
var
  DeptRow: TDeptRowtype;
begin
  DeptRow := TDeptRowtype.Create(DataModule.MainSession);
  try
    DeptRow.Deptno := 40;
    DeptRow.Dname := 'OPERATIONS';
    DeptRow.Loc := 'BOSTON';
    DataModule.Department.InsertRecord(DeptRow);
  finally
    DeptRow.Free;
  end;
end;
```

TPLSQLRecord reference

This chapter describes all properties and methods of the TPLSQLRecord object.

TPLSQLRecord.Assign

Declaration

```
procedure Assign(Source: TPLSQLRecord);
```

Description

Assigns the Source PL/SQL Record to this instance. This will perform a deep copy of all components in the record, so that the source and destination are completely separate. If the record type contains LOB's, Objects, References, PL/SQL Tables, or other PL/SQL Records, these objects will be copied as well.

TPLSQLRecord.Create

Declaration

```
constructor Create(ASession: TOracleSession);
```

Description

Creates a new PL/SQL Record. If the record type contains complex components (LOB's, Objects, References, PL/SQL Tables or other PL/SQL Records), then these instances will be created as well. When the PL/SQL Record is freed, these complex component instances will also be freed.

TPLSQLTable object

Unit

Oracle

Description

The TPLSQLTable object encapsulates scalar PL/SQL Table parameters for the TOracleCustomPackage component, generated through the Package Wizard. To create a PL/SQL Table instance, call its Create constructor and define the TableSize and, optionally, the StringSize. To access the individual PL/SQL Table elements, you can access the Values array property.

TPLSQLTable reference

This chapter describes all properties and methods of the TPLSQLTable object.

TPLSQLTable.Assign

Declaration

```
procedure Assign(Source: TPLSQLTable);
```

Description

Assigns the Source PL/SQL Table to this instance. This will effectively copy the StringSize, TableSize and Values.

TPLSQLTable.Count

Declaration

```
property Count: Integer;
```

Description

The number of elements in the Values array. The following example will copy all PL/SQL Table values to a memo:

```
for i := 0 to Table.Count - 1 do  
    Memo.Lines.Add(Table[i]);
```

If you assign Values beyond the current count, the Values array will automatically extend. You can however also explicitly set the Count property. It is faster to preset the Count to 100 and subsequently assign 100 values, compared to implicitly extending the array. The following example copies the contents of a memo to a PL/SQL Table:

```
Table.Count := Memo.Count;  
for i := 0 to Memo.Count - 1 do  
    Table[i] := Memo.Lines[i];
```

TPLSQLTable.AsString

Declaration

```
property AsString: string;
```

Description

Represents the Values of the PL/SQL Table as a multi line string. Each element of the table will occupy a line. You can also assign a multi line string to the AsString property.

TPLSQLTable.Create

Declaration

```
constructor Create(ATableSize, AStringSize: Integer);
```

Description

Creates a new PL/SQL Table with the TableSize and StringSize as specified.

TPLSQLTable.StringSize

Declaration

```
property StringSize: Integer;
```

Description

If a PL/SQL Table holds string elements, then you must specify the maximum length of the strings. This equivalent to a PL/SQL Table on the server, which has the same requirement. The size of a string is pre-allocated for each element in the table when it is passed to the server. If the StringSize is 100, and the TableSize is 50, then a total of $100 \times 50 = 5000$ bytes will be allocated.

If the PL/SQL Table holds integer, float or date elements, this property is ignored.

TPLSQLTable.TableSize

Declaration

```
property TableSize: Integer;
```

Description

This property determines the maximum number of elements that a PL/SQL Table can hold. This is only important for input/output and output parameters, because you need to determine how many table elements you can expect. For input parameters this property is not important, because if you assign Values beyond the TableSize, it will automatically be adjusted.

TPLSQLTable.ValueArray

Declaration

```
property ValueArray: Variant;
```

Description

This property represents the PL/SQL Table as a Variant array of Variants. You should normally not use this property.

TPLSQLTable.Values

Declaration

```
property Values[Index: Integer]: Variant;
```

Description

The zero-based array of elements of the PL/SQL Table. The Count property indicates the number of elements in the array. You can also set the Count property to extend or decrease the array. If you set Count larger than the current value, the values array will be extended with null values.

The Values property is the default array property of a PL/SQL Table. Therefore the following 2 statements are equivalent:

```
Table[3] := 'Hello';  
Table.Values[3] := 'Hello';
```

TOracleProvider component

Unit

OracleProvider

Description

The TOracleProvider component can be used in a server application of a multi-tiered application. It descends from the abstract TCustomProvider and functions exactly the same as Delphi's standard TProvider, except that you can only assign a TOracleDataSet to its DataSet property. Furthermore, the UniqueFields property allows you to define the fields that the TOracleProvider must use to identify a record. When a TClientDataSet sends records to the TOracleProvider, it will use these fields to match the records. If the UniqueFields property is empty, the TOracleProvider will automatically determine a primary or unique key for the updating table.

Note that multi-tiered applications can only be developed with the Client/Server or Enterprise edition of Delphi or C++Builder. For more information about multi-tiered applications, you can read the Delphi / C++Builder manuals and help files. The 3Tier demo project can also provide some information.

Note

In Delphi 5 the TOracleProvider component is only provided for backward compatibility. For new projects you should use the standard TDataSetProvider component instead. After upgrading a multi-tiered application to Delphi 5, you should consider replacing the TOracleProvider components with TDataSetProvider components.

The UniqueFields property no longer works in the Delphi 5 version of the TOracleProvider. You must convert this information to the ProviderFlags of the fields of the dataset.

TOracleProvider reference

This chapter describes all properties, methods and events of the TOracleProvider component.

TOracleProvider.DataSet

Declaration

```
property DataSet: TOracleDataSet;
```

Description

The dataset is used by the provider to get data from the database to which the component is connected, and to write client application updates to the database.

TOracleProvider.UniqueFields

Declaration

```
property UniqueFields: string;
```

Description

The UniqueFields will be used by the provider to identify a record as it is transferred between client dataset and server application. You can separate multiple fields with semicolons.

Note

This property no longer works in the Delphi 5 version of the TOracleProvider. You must convert this information to the ProviderFlags of the fields of the dataset.

Unit reference

This chapter lists all relevant units in the Direct Oracle Access package, as well as the types, variables, procedures and functions that are available.

Oracle Unit

The Oracle Unit contains the declarations of the DOA components that can be used in any edition of Delphi and C++Builder. It is automatically added to the **uses** clause when you use one of the components.

Components

TOracleSession
TOracleLogon
TOracleQuery
TOraclePackage
TOracleEvent
TOracleCustomPackage
TOracleDirectPathLoader
TOracleScript
TOracleQueue
TOracleSessionPool

Objects

TLOBLocator
TOracleObject
TOracleReference
TOracleTimestamp
TXMLType
TPLSQLTable
TPLSQLRecord
TDirectPathColumns
TDirectPathColumn
TOracleCommands
TOracleCommand

OracleData Unit

The OracleData Unit contains the declarations for the TOracleDataSet component. It is automatically added to the **uses** clause when you use this component.

Components

TOracleDataSet

Objects

TSequenceField
 TOracleDictionary
 TQBDefinition
 TQBField

OracleNavigator Unit

The OracleNavigator Unit contains the declarations for the TOracleNavigator component. It is automatically added to the **uses** clause when you use this component.

Components

TOracleNavigator

OracleProvider Unit

The OracleProvider Unit contains the declarations for the TOracleProvider component. It is automatically added to the **uses** clause when you use this component.

Components

TOracleProvider

OracleCI Unit

The OracleCI unit contains the Oracle Call Interface (OCI). The OCI is part of SQL*Net or Net8 and is provided as a DLL. This DLL is located in Oracle's bin directory and is named something like ora73.dll (for SQL*Net 2.3), ora803.dll (for Net8 8.0.3), or oci.dll (for Net8 8.1 and later). The OCI DLL will be initialized (once) at the first attempt to logon to the database.

The following variables are related to the OCI DLL:

```
OCI70: Boolean = False;           // OCI 7.0 functions detected
OCI72: Boolean = False;           // OCI 7.2 functions detected
OCI73: Boolean = False;           // OCI 7.3 functions detected
OCI80: Boolean = False;           // OCI 8.0 functions detected
OCI81: Boolean = False;           // OCI 8.1 functions detected
OCI90: Boolean = False;           // OCI 9.0 functions detected
OCI92: Boolean = False;           // OCI 9.2 functions detected
OracleHomeName: string = '';      // Name of Oracle Home
OCIDLL: string = '';              // Name of OCI DLL
```

Never set the OCI70..OCI81 booleans yourself! They are used internally to determine which functions are available. The OCI80 boolean can be used to determine if you can use Oracle8 features (LOB's and Objects) in your application, you also have to know if you're connected to an Oracle8 database however.

The OracleHomeName variable can be set to the name of the Oracle Home you wish to use. Note that this is not the registry path or the directory path of the Oracle Home, it is the value of the ORACLE_HOME_NAME registry key. This is equivalent to the name that is used with

Oracle's Home Selector utility. You can obtain a list of oracle home names through the OracleHomeList function.

You can use the OCIDLL string to determine the exact filename of the OCI DLL file that is being used. You can also use this variable to force a specific version of SQL*Net to be loaded. Simply set it to the desired DLL filename before DLLInit is called. The OCIDLL string is to Direct Oracle Access what the Vendor Init is to the BDE.

The following functions are related to the OCI DLL:

procedure InitOCI;

This function initializes the OCI by calling DLLInit, checking the result, and raising an appropriate exception if it failed.

function DLLLoaded: Boolean;

Indicates if the OCI DLL is loaded and initialized.

function DLLInit: Integer;

Loads and initializes the OCI DLL. The following things will happen:

1. If already initialized then exit
2. Check if a OCIDLL= parameter is entered on the command line, skip to point 9
3. If the OCIDLL string has a value, skip to point 9
4. Check if a ORACLEHOME= parameter is entered on the command line, skip to point 7
5. If the OracleHomeName string has a value, skip to point 7
6. Determine the Primary Oracle Home
7. Check the registry to determine the ORACLE_HOME directory
8. Add '\bin\' and search for ora*.dll or oci.dll
9. Set the OCIDLL string to the file with the highest number
10. Call LoadLibrary(OCIDLL)
11. If not successful then set the current path to Oracle's bin directory and try again.
12. Determine all functions by name

The DLLInit function returns an integer with the following possible values:

dllOK	Everything went OK
dllNoRegistry	The registry key was not found (SQL*Net not installed)
dllNoFile	No files were found at step 5 or LoadLibrary() failed
dllMismatch	No functions were found at step 9

procedure DLLExit;

Frees the OCI DLL that was initialized by DLLInit.

procedure OracleHomeList: TStringList;

Returns a TStringList instance with oracle home names. Before OCI is initialized, you can set the OracleHomeName variable to one of these names.

function OracleAliasList: TStringList;

Returns a TStringList instance with database aliases as defined in the tnsnames.ora file. Don't free this list, it is used internally by the TOracleSession.LogonDatabase property and the TOracleLogon dialog. This function first initializes OCI by calling InitOCI, because it needs to know if Net8 is used.

function TNSNames: string;

Returns the name of the tnsnames.ora file. This function first initializes OCI by calling InitOCI, because it needs to know if Net8 is used.

OracleMonitor Unit

The Oracle Monitor is a utility that allows you view all database activities generated by an application that uses the Direct Oracle Access components. The main goals of the Oracle Monitor are:

- ♦ Find performance bottlenecks in your application
- ♦ Measure the performance effects of changes in your application or database
- ♦ Find application errors (what SQL was executed? What were the variable values?)

The Oracle Monitor is a separate application that runs independently from the applications that you want to monitor. It displays the currently running Direct Oracle Access applications, an hierarchical view of the object instances that generate database activities (TOracleSession, TOracleQuery, TOracleDataSet, and so on), and the activities themselves. For each activity it displays a description, timestamp, duration, SQL, error message, variables values before and after execution, database statistics, and the query plan. You can modify the SQL and optimizer goal to quickly test the effects of these changes on the query plan.

Enabling the Oracle Monitor

Include the OracleMonitor unit in the uses clause in some main unit (e.g. the main form or data module) of your application. As long as you do not include the OracleMonitor unit your application cannot be monitored, as it takes care of the following tasks:

- ♦ Enlist the application with the Oracle Monitor when the application starts
- ♦ Send information about objects and activities to the Oracle Monitor when they occur

Including the OracleMonitor unit will affect the performance of your application, because sending this information to the Oracle Monitor will obviously take some time. However, as long as you do not run the OracleMonitor itself, performance degradation will be minimal. Therefore you can decide to keep the OracleMonitor unit included during all test phases, and maybe even in the production version of your application.

Direct Oracle Access Preferences

The 'Preferences' item in the 'Oracle' main menu in the Delphi or C++Builder IDE, allows you to specify several settings that affect the design-time behavior. None of these preferences affect the generated executable. The Preferences dialog is also accessible by right-clicking on a TOracleSession instance and selecting the 'Preferences' item.

General

- ♦ Oracle main menu enabled
The Oracle main menu contains items for the Package Wizard, Explorer, Monitor, PL/SQL Developer, Help file, Documentation, Preferences and Info. If your main menu is getting too crowded, you can disable it through this preference. If you want to enable it again, the preferences can be accessed by right-clicking on a TOracleSession instance and selecting the 'Preferences' item.
- ♦ Wizards enabled
The Direct Oracle Access Wizards are displayed after selecting the 'New' item from the 'File' menu. You can remove the Direct Oracle Access Wizards from this 'New items' dialog through this preference. The Wizards will still be accessible through the 'Oracle' menu.

- ♦ Oracle Home
If you have multiple oracle home's installed, then Direct Oracle Access will select the primary oracle home by default. This is the oracle home that is listed first in the PATH environment variable. You can select a different primary oracle home through the Oracle Home Selector. If you want to use a different oracle home at design time, then you can select it from this list.

At run time you can set the OracleHomeName variable in the OracleCI unit before making a connection to the database, or you can set the oraclehome run time parameter on (e.g. oraclehome=ora815).

- ♦ OCIDLL
If the oracle home has multiple versions of SQL*Net or Net8 installed, you can select an OCI (Oracle Call Interface) DLL from this list, which will be used for design time connections. Direct Oracle Access will use the most recent version by default.

At run time you can set the OCIDLL variable in the OracleCI unit before making a connection to the database, or you can specify the ocidll run time parameter (e.g. ocidll=ora73.dll).

Logon Parameters

- ♦ Username, Password & Database
Specify these preferences for the preferred design time connection.
- ♦ Always use these parameters
If you enable this preference, the username, password and database will always be used when making a connection to the database at design time in the Explorer, SQL Editor and Command Editor.
- ♦ Only use these parameters as a preference
If you enable this preference, the username, password and database will only be presented as a default, but can still be overridden.

Logon History

- ♦ Enabled
If you enable logon history, the Explorer, SQL Editor and Command Editor will present a history list of previous connections during logon.
- ♦ Store with password
If you enable this preference, the connection information will be stored with (encrypted) password. When you recall such a connection, you will immediately be logged on without specifying a password.
- ♦ History size
Determines the number of connections that will be remembered.

Exceptions

By default all Oracle Errors will be raised as EOracleError exceptions. On this tab page you can specify which error numbers will be raised as an EOracleWarning exception, which is a straight descendant from EOracleError (so it still is an EOracleError!).

Making this distinction is useful if you run your application from within the Delphi or C++Builder IDE (version 4 or later), because you can use the Debugger Options to ignore these errors. For example, if you add 54 to the list, all locking exceptions caused by a 'select for update' will not bring you back into the IDE with an exception notification. Most likely you have handled ORA-00054, but the IDE does not know this. By using this preference, you can tell the debugger which exceptions you will always handle.

Making changes to the exception list will not immediately affect a running application. It must be restarted before it knows about these changes. Also note that the EOracleError - EOracleWarning distinction will only occur when running an application from within the IDE. It is a design time preference that does not affect the generated application in its normal run time environment. Never explicitly use EOracleWarning in your application!

Direct Oracle Access Designtime Property Defaults

The 'Property Defaults' item in the 'Oracle' main menu in the Delphi or C++Builder IDE, allows you to specify the default property values for the Direct Oracle Access components at designtime. It additionally allows you to specify that related components (such as a `TOracleSession` instance for the `Session` property of a `TOracleDataSet`) are automatically selected when dropped on a form or data module.

At the top of the Designtime Defaults screen you can select each Direct Oracle Access component for which you want to define a default. Below the component set, you see all designtime properties for the selected component. For each property you can set a default, or leave it empty to indicate that you do not want to define your own default. For properties that can have a restricted set of values (such as a boolean or an enumerated type), you can select a specific value by pressing on the (...) button to the right of the property value. For properties that link to a related component, you can only specify *Autodetect* as a default. In that case the component will automatically select the closest component of the correct type for the property when the component is dropped on a form or data module.

At the bottom of the form you can enable or disable the designtime default properties.

Note

None of these settings affect the default property values when components are created at runtime.

Multi-threaded applications

In multi-threaded applications, multiple threads might be accessing the database. Sometimes this is necessary to perform tasks in the background while the user keeps on working with the application. Multi-threading can also be used to optimize the application's use of resources of client, network and server. Finally, multi-threading is necessary if an application needs to be able to break long running queries on the server.

There are basically two configurations for a multi-threaded application:

Multi-threaded, single-session database access

In this case, multiple threads access one `TOracleSession` simultaneously. Because the database server can only handle one request for a specific session at a time, all database accesses must be serialized. This is achieved by setting the `TOracleSession.ThreadSafe` property to true. As a result, all threads block each other when they are accessing the database. This configuration only makes sense if the threads do a significant amount of client processing relative to the sum of the network and server processing.

Session management needs some special attention in this configuration. The thread that logs on and off, does transaction control or session control statements, must obviously also be in control of the other thread(s) to synchronize these actions.

Multi-threaded, multi-session database access

In this case, each thread uses its own dedicated `TOracleSession`. The client, network and server can do the processing for these threads parallel, which results in an optimized use of resources and a non-blocking behavior of the application. No special properties need to be set, and session management is clear. This is obviously the preferred configuration for a multi-threaded application, but may not always be possible.

For both configurations, access to a `TOracleQuery`, `TOraclePackage` and `TOracleDataSet` must always be restricted to a single thread.

Any executing query can be aborted by calling the `TOracleSession.BreakExecution` method. This causes an "ORA-01013, user requested cancel of current operation" exception for the currently executing query in that session. Beware that `BreakExecution` might not work on all versions of Personal Oracle.

Dynamic Link Libraries

When creating Dynamic Link Libraries that access the database, you can distinguish four different situations:

DLL-only database access

In this case the host program does not do any database access, so the DLL owns the session and controls the log on and log off. There is no difference between such a DLL and a normal DOA application.

DOA DLL sharing a session with a DOA host program

The host program owns the `TOracleSession` and can pass it to the DLL through an external initialization procedure in the DLL. This DLL initialization procedure can call the `Share` procedure to share the physical database session between the host session and a session in a local data module, for example:

```
procedure InitSession(HostSession: TOracleSession);  
begin  
    DLLDataModule := TDLLDataModule.Create(nil);  
    HostSession.Share(DLLDataModule.Session);  
end;
```

There is a strict order in logging on and off in the host program and DLL:

1. Host program log on
2. Share the session with the DLL
3. (Application processing)
4. Log off or free the session in the DLL
5. Host program log off

DOA DLL sharing a session with a non-DOA host program

If you are writing a DLL that should share a session with a non-DOA host program (for example, written in Pro*C), you cannot pass a `TOracleSession` because it does not exist in the host program. To allow you to share a session, you can pass a pointer to the LDA (SQL*Net: Logon Data Area) or SVC (Net8: Service Context) to a `TOracleSession` in the DLL. When you assign this pointer to the `TOracleSession.ExternalLDA` or `TOracleSession.ExternalSVC` property, the session is virtually logged on. All components linked to this session will now work with this LDA or SVC. When you assign a `nil` pointer to `ExternalLDA` or `ExternalSVC`, the session is considered logged off.

There is a strict order in logging on and off in the host program and DLL:

1. Host program log on
2. Assign a value to `ExternalLDA` or `ExternalSVC`
3. (Application processing)
4. Assign `nil` to `ExternalLDA` or `ExternalSVC`
5. Host program log off

If you log off in the host program before assigning a `nil` to `ExternalLDA` or `ExternalSVC`, the components in the DLL will cause a "not logged on" error when freed, because the `TOracleSession.Connected` property does not reflect the actual status.

Non-DOA DLL sharing a session with a DOA host program

If you are writing an application that needs to share a session with a non-DOA DLL (for example, written in Pro*C), you can get a pointer to the LDA (Logon Data Area) or SVC (Service Context) of the `TOracleSession` through the `ExternalLDA` or `ExternalSVC` property. The DLL needs to provide a procedure to pick up this LDA or SVC.

Translating standard messages from English

Direct Oracle Access uses string constants for all messages that are generated. You can overrule these messages in your application to translate them from English to your native language if necessary.

TOracleDataSet messages

To allow translation of the standard messages generated by the TOracleDataSet, the following string constants are defined in the OracleData unit:

```
const // Allow translation of dataset messages
  dmRecordLocked: string = 'Record is locked by another user';
  dmRecordChanged: string = 'Record has been changed by another
    user';
  dmRecordDeleted: string = 'Record has been deleted by another
    user';
```

To enforce the same translated messages in all your application, you can create a unit that overrules the default messages in its initialization section and simply include it in all your projects:

```
unit DOADutch;

interface

implementation

uses OracleData;

initialization
begin
  // Translate dataset messages to Dutch
  dmRecordLocked := 'Record is gereserveerd door een andere
    gebruiker';
  dmRecordChanged := 'Record is gewijzigd door een andere gebruiker';
  dmRecordDeleted := 'Record is verwijderd door een andere
    gebruiker';
end;

end.
```

TOracleLogon messages

The following string constants are defined in the OracleLogon unit and can be overruled to translate the Logon dialog:


```
const // Allow translation of the Logon dialog
  ltLogonTitle:      string = 'Oracle Logon';
  ltPasswordTitle:   string = 'Change password';
  ltConfirmTitle:    string = 'Confirm';
  ltUsername:        string = 'Username';
  ltPassword:        string = 'Password';
  ltDatabase:        string = 'Database';
  ltConnectAs:       string = 'Connect as';
  ltNewPassword:     string = 'New password';
  ltOldPassword:     string = 'Old password';
  ltVerify:          string = 'Verification';
  ltVerifyFail:      string = 'Verification failed';
  ltChangePassword:  string = 'Do you wish to change your password
    now?';
  ltExpired:         string = 'Your password has expired';
  ltOKButton:        string = 'OK';
  ltCancelButton:    string = 'Cancel';
```

The following code translates the logon dialog text to German and then executes a TOracleLogon component. Note that you must add the OracleLogon unit to the uses clause of the unit that performs this task.

```
begin
  ltLogonTitle      := 'Anmeldung';
  ltUsername        := 'Benutzer';
  ltPassword        := 'Passwort';
  ltDatabase        := 'Datenbank';
  ltOKButton        := 'OK';
  ltCancelButton    := 'Abbruch';
  MyLogonDialog.Execute;
end;
```

Oracle Net compatibility issues

If Direct Oracle Access detects Oracle Net 8 or later, the new Net8 interface will be used. If Oracle Net 8 or later is not detected, the old SQL*Net functions will be used to maintain full compatibility with SQL*Net 1.x or 2.x, Oracle 7.x and all previous Direct Oracle Access releases.

Theoretically, Oracle Net 8 is 100% compatible with previous versions of SQL*Net. However, if an application wishes to use features such as LOB's and Objects, it needs to use the new Oracle Net 8 interface. Because of this new interface, the following incompatibilities exist:

- ♦ Long and Long Raw columns can no longer be randomly and piecewise fetched from the server using the `TOracleQuery.GetLongField` method. The method still works, but the entire Long is immediately fetched after execution, and the `GetLongField` method retrieves the requested piece from memory.
- ♦ The `ErrorPosition` and `ErrorLine` of a `TOracleQuery` and `TOracleDataSet` are not supported on Net8 8.0.3 and 8.0.4, because this information is missing. They always return 0. In Net8 8.0.5 and later these functions perform correctly.
- ♦ The `TOracleQuery.WarningFlags` are not available for Net8, except for the "Compiled with errors" bit (\$20).
- ♦ `TOracleSession.ExternalLDA` is no longer available, because Net8 no longer uses an LDA structure.

To force Direct Oracle Access to use the old SQL*Net interface on Net8 clients, thereby removing the above incompatibilities, you can set the `TOracleSession.Preferences.UseOCI7` boolean to True. This way, you achieve 100% compatibility for your application on SQL*Net and Oracle Net 8 clients. Obviously, Oracle Net 8 specific features are not available to your application. These features are:

- ♦ The `TLOBLocator` object
- ♦ The `TOracleObject` object
- ♦ The `TOracleReference` object
- ♦ Oracle8 password expiration
- ♦ MTS session pooling

The following functionality requires Oracle Net 8.1 or later

- ♦ The `TOracleDirectPathLoader` component
- ♦ Temporary LOB's
- ♦ Timestamps

Features that require Oracle Net 9.0 or later:

- ♦ Scrollable queries

Features that require Oracle Net 9.2 or later:

- ♦ The `TXMLType` object
- ♦ Oracle Session Pooling

- ♦ Client side statement caching

Personal Oracle Lite compatibility issues

Direct Oracle Access can be used to create applications that run on Personal Oracle Lite. Features from Oracle 7 that are missing in POLITE (like PL/SQL) will not be discussed here. Check your POLITE documentation for more info. The following chapters discuss compatibility issues that arise when using Direct Oracle Access on a POLITE database.

SQL*Net / Net8 versions

POLITE ships with a specific version of SQL*Net or Net8 that needs to be used to connect to the database. Version 3.0 requires that SQL*Net 2.3 is used, whereas version 3.5 can be accessed though Net8. When Direct Oracle Access detects a POLITE connection, it implicitly uses the required SQL*Net or Net8 version.

Connect strings

To access a POLITE database through a SQL*Net or Net8 client, you must specify an ODBC datasource as the connect string. If, for example, the POLITE database is named MYLITEDB, then you must specify ODBC:MYLITEDB as the connect string in the `TOracleSession.LogonDatabase` property.

Error handling

The error codes that POLITE returns are different than those returned by an Oracle 7 or 8 server. The Oracle error code will usually be 30021, and the second line of the message text will explain what went wrong. If, for example, you insert a duplicate key, you will get the following error text:

OCA-30021: error preparing/executing SQL statement

[POL-3220] duplicate keys in primary/unique index

The error code that Oracle reports will always be 30021, so your application needs to analyze the error text if it needs to respond to specific error situations.

The `OnTranslateMessage` event of a `TOracleDataSet` also suffers from this incompatibility. Not only will you always receive error code 30021, but also will the constraint name always be empty. If you wish to use this event handler than again you need to analyze the secondary message in the error text.

The `EnforceConstraints` option of the `TOracleDataSet` will only work for primary, unique and foreign key constraints. Check constraints cannot be evaluated due to the fact that PL/SQL blocks are not supported by POLITE.

Updateable datasets

To make a `TOracleDataSet` updateable for an Oracle 7 database, you needed to include the rowid in the SQL select statement. The support for rowid's in POLITE is not completely up to Oracle 7 level, so that you also need to include all the primary key columns in the dataset. Alternatively, you can include the columns from a unique key. If you do include the rowid, but not the primary/unique key columns, the dataset will issue an error when you try to open it.

Index

A

Abort, 206
Add, 196; 212
AddOutput, 187
AfterCommand, 187
AfterFetchRecord, 92
AfterLogOn, 11
AfterQBE, 93
AfterQuery, 60; 93
AfterRefreshRecord, 94
AfterReserve, 229
AliasDropDownCount, 45
AllowFileWildCards, 132
AllowOperators, 132
Append, 157
ApplyUpdates, 11
Array DML, 53
AsDateTime, 236
AsOracleString, 236
Assign, 157; 166; 181; 236; 255; 257
AsString, 157; 236; 257
Attempts, 220
AttrIsNull, 166
AutoCommit, 12; 187
AutoContains, 136
AutoPartialMatch, 136

B

BackgroundColor, 132
BeforeLogOn, 12
BeforeQBE, 94
BeforeQuery, 60; 94
BeforeRelease, 230
BreakExecution, 12
BreakThread, 60
Buffering, 157
BufferSize, 206
BytesPerCharacter, 13

C

CachedUpdates, 95

CallBooleanFunction, 141
CallComplexMethod, 166
CallDateFunction, 141
CallFloatFunction, 141
CallIntegerFunction, 142
CallMethod, 167
CallProcedure, 142
CallStringFunction, 142
Cancel, 61
CancelUpdates, 13
Caption, 45
CaseInsensitive, 136
CheckConnection, 13
Clear, 61; 158; 168; 181; 196; 212; 236
ClearAttr, 168
ClearQBE, 95
ClearVariables, 61; 95
Close, 61
CloseAll, 96
ColumnByName, 206
Columns, 206
ColWidth, 187
CommandByName, 188
CommandIndex, 188
Commands, 188
CommandType, 199
CommentProperty, 199
Commit, 14
CommitOnPost, 96
CompareQueryVariables, 96
Compress, 230
CompressBLOBs, 97
CompressionFactor, 97
CompressOld, 230
Condition, 217
ConnectAs, 14
Connected, 15
ConsumerName, 217
ConvertCRLF, 36
Copy, 158
Correlation, 217; 220
Count, 196; 212; 230; 257
CountQueryHits, 98
Create, 158; 168; 181; 237; 241; 252; 258
CreateObject, 98
CreateTemporary, 159
CurrentCommand, 189
Cursor, 15; 62; 99; 142; 189; 216; 252
Cursor variables, 55

D

DataSet, 133; 261
DataSize, 214
DataType, 214; 237
DateFormat, 207; 214
Day, 237
DBMS_Alert, 16
DBMS_Application_Info, 16
DBMS_Job, 17
DBMS_Output, 18
DBMS_Pipe, 19
Debug, 62; 99; 143; 189; 217
DeclareAndSet, 62; 99
DeclareQueryVariables, 99
DeclareVariable, 62; 100
Defaults, 268
DefaultValues, 114
Delay, 220
Delete, 169; 196
DeleteElement, 169
DeleteVariable, 63; 100
DeleteVariables, 63; 100
Dequeue, 217
DequeueMode, 217
DequeueOptions, 217
Describe, 63
DesignActivation, 101
DesignConnection, 20
Designtime Property Defaults, 268
Detachable, 101
DetailFields, 101
DimPLSQLTable, 63
Direct Oracle Access, 5
Direct Oracle Access Designtime
 Property Defaults, 268
Direct Oracle Access Installation, 7
Direct Oracle Access Preferences, 266
Direct Path Columns property editor, 213
Directory, 159
Disabled Constraints property editor, 114
DisabledConstraints, 114
DisplayFormats, 114
DOA, 5
Dynamic Link Libraries, 270
DynamicDefaults, 114

E

ElementCount, 170

ElementExists, 170
Elements, 170
EnforceConstraints, 114
Enqueue, 219
EnqueueOptions, 219
EnqueueTime, 220
Eof, 64
Erase, 159
ErrorCode, 200
ErrorLine, 64; 102
ErrorMessage, 21; 200
ErrorPosition, 65; 102
ExceptionQueue, 220
ExecSQL, 102
Execute, 45; 65; 189; 200
ExecuteArray, 65
ExecuteQBE, 102
ExitOnError, 189
Expiration, 220
ExpirationMessage, 21
ExternalAUT, 21
ExternalCursor, 103
ExternalENV, 21
ExternalLDA, 22; 270
ExternalSRV, 22
ExternalSVC, 22
ExtProcRaise, 23
ExtProcShare, 23

F

Field, 66
FieldAsDate, 66
FieldAsFloat, 67
FieldAsInteger, 67
FieldAsString, 67
FieldByName, 133
FieldCount, 68; 133
FieldIndex, 68
FieldIsNull, 68
FieldKinds, 114
FieldName, 69; 136
FieldOptional, 69
FieldPrecision, 69
Fields, 133
FieldScale, 70
FieldSize, 70
FieldType, 70
FileExists, 159
Filename, 160
Filter, 103

Filtered, 104
FilterOptions, 104
Finish, 207
Finished, 190
First, 71
FloatPrecision, 36
Flush, 170
FlushBuffer, 160
FlushObjects, 27
FontColor, 134
FunctionType, 71; 200

G

GetAttr, 171
GetBooleanVariable, 143
GetComplexVariable, 71; 105
GetCursor, 71
GetDateVariable, 143
GetDefaultColumns, 207
GetFloatVariable, 143
GetIntegerVariable, 143
GetLongField, 72
GetParameter, 144; 171
GetQueryVariables, 105
GetStringVariable, 144
GetVariable, 72; 105; 190

H

Hex, 181
HistoryIniFile, 45
HistoryRegSection, 46
HistorySize, 46
HistoryWithPassword, 46
Homogeneous, 230
Hour, 237

I

IgnoreTime, 137
Index, 201; 215
IntegerPrecision, 36
InternalSession, 147
InTransaction, 27
IsArray, 172
IsCollection, 172

IsLocked, 172
IsNull, 160; 172; 182; 238
IsolationLevel, 27
IsTable, 173
Items, 197; 212

K

KeepConnection, 147

L

Last, 72
LastColumn, 207
LastRow, 208
LastValue, 137
Lines, 190
Load, 208
LoadFromFile, 160
LOBAttr, 173
LOBField, 72
Lock, 174; 231
LockingMode, 106
LogMode, 208
LogOff, 27; 147
LogOn, 28
LogonDatabase, 28; 231
LogonPassword, 29; 231
LogonUsername, 29; 231

M

Master, 106
MasterFields, 108
Max, 231
MaxElements, 174
MaxRows, 208
MaxStringFieldSize, 36
MessageProperties, 220
MessageTable, 29
Min, 232
Minute, 238
Modified, 175
MonitorMessage, 30
Month, 238
MoveBy, 73
MoveTo, 73

MsgId, 217; 220
MTSOptions, 31
Multi-threaded applications, 269

N

Name, 161; 175; 182; 201; 215; 252
NanoSeconds, 238
Navigation, 217
Net8, 263
Next, 73
NullLOBsEmpty, 36
NullObjectIsEmpty, 36
NullValue, 31

O

ObjAttr, 175
ObjectNames, 147
ObjectType, 148
ObjElements, 176
ObjField, 73
OCI, 263
OCIDateCreate, 32
OCIDateFree, 32
OCIDateFromDateTime, 32
OCIDateToDateTime, 32
OCIDLL, 263
OCINumberCreate, 33
OCINumberFree, 33
OCINumberFromFloat, 33
OCINumberFromInt, 33
OCINumberToFloat, 34
OCINumberToInt, 34
OnApplyRecord, 108
OnArrayError, 74
OnCancelQBE, 112
OnChange, 34
OnClose, 232
OnCommand, 191
OnCompressBLOB, 112
OnData, 192
OnDecompressBLOB, 113
OnError, 148; 192
OnEvent, 148
OnOpen, 232
OnOutput, 192
OnStart, 149
OnStop, 149
OnThreadDequeued, 222
OnThreadError, 74; 223
OnThreadExecuted, 74
OnThreadFinished, 75
OnThreadRecord, 75
OnThreadStart, 223
OnThreadStop, 223
OnThreadTimeOut, 224
OnTimeOut, 149
OnTranslateMessage, 113
Optimize, 75; 114; 144
OptimizerGoal, 34
Options, 46
Oracle, 262
Oracle Net compatibility issues, 274
ORACLE_HOME, 263
OracleAliasList, 263
OracleCI, 263
OracleDictionary, 114
OracleHomeList, 263
OracleHomeName, 263
OracleMonitor, 265
OriginalMsgId, 220
Output, 193
OutputOptions, 193

P

Package Wizard, 242
PackageName, 144
PackageSpecification, 253
Parallel, 209
ParameterMode, 144
PartitionName, 209
Payload, 224
PayloadType, 225
Personal Oracle Lite compatibility issues, 276
Picture, 47
Pin, 182
PL/SQL Tables, 56
POLite, 34
Pool, 35
Pooling, 35
PoolName, 232
PoolType, 233
Position, 161
Preferences, 36; 266
Prepare, 209
Prepared, 209
Prior, 76

Priority, 220
Property Defaults, 268
ProviderOptions, 116

Q

QBE Definition property editor, 116
QBEDefinition, 116
QBEMode, 118
QBEModified, 120
Query, 193
Queryable, 137
QueryAllRecords, 120
QueueName, 225
QueueType, 225

R

RangeValues, 114
RawPayload, 225
Read, 161
ReadBuffer, 76; 121
ReadOnly, 121
RefAttr, 176
RefElements, 177
Reference, 177
RefField, 76
Refresh, 178
RefreshOptions, 121
RefreshRecord, 123
RelativeMsgid, 219
RequiredFields, 114
Retries, 47
ReturnCode, 37; 76
Rollback, 37
RollbackOnDisconnect, 38
RollbackToSavepoint, 38
RowCount, 77
RowId, 77; 123
RowsProcessed, 77; 201

S

Savepoint, 38
SaveQBESValues, 134
SaveToFile, 161
ScanVariables, 193

ScriptLine, 202
Scrollable, 77
ScrollPosition, 78
SearchRecord, 124
Second, 238
Seek, 162
SenderId, 220
SequenceDeviation, 219
SequenceField, 125
SequenceField property editor, 125
ServerVersion, 39
Session, 47; 78; 125; 145; 150; 194; 209; 226; 239; 253
Sessions, 233
SetAttr, 178
SetComplexVariable, 78; 125
SetData, 215
SetEmpty, 162
SetLongVariable, 79; 126
SetPassword, 39; 48
SetQueryVariables, 126
SetTransaction, 40
SetValues, 239
SetVariable, 79; 126; 145; 194
Share, 41
Size, 162
SmallIntPrecision, 36
SQL, 80; 127
SQL*Net, 263
SQLTrace, 41
Start, 150
Started, 150
StartThread, 226
State, 80; 220
StatementCache, 41; 233
StatementCacheSize, 42
Stop, 150
StopThread, 227
stored procedures, 80
stored program units, 80
StringFieldsOnly, 80; 127
StringSize, 258
SubstitutedSQL, 81; 127
SubstitutedText, 202
Synchronized, 150

T

TableName, 210; 227
TableOwner, 227
TableSize, 258

- TAAfterFetchRecordAction, 92
 - TAQAgent, 220
 - TAQDequeueMode, 217
 - TAQDequeueNavigation, 217
 - TAQDequeueOptions, 217
 - TAQDequeueState, 220
 - TAQDequeueVisibility, 217
 - TAQEnqueueOptions, 219
 - TAQEnqueueSequence, 219
 - TAQEnqueueVisibility, 219
 - TAQMessageProperties, 220
 - TAQQueueType, 225
 - TBytesPerCharacterOption, 13
 - TCheckConnectionResult, 13
 - TCompressionEvent, 112; 113
 - TConnectAsOption, 14
 - TDirectPathColumn, 213
 - TDirectPathColumns, 211
 - Temporary, 163
 - TemporaryLOB, 36
 - TEventObjectType, 148
 - Text, 202
 - Threaded, 81; 227
 - ThreadIsRunning, 82; 228
 - ThreadSafe, 42; 269
 - ThreadSynchronized, 82; 228
 - TimeOut, 151; 220; 233
 - TimestampAttr, 178
 - TimestampField, 83
 - TIsoolationLevelOption, 27
 - TLOBLocator, 152
 - TLockingModeOption, 106
 - TLogonOption, 46
 - TLogonOptions, 46
 - TNSNames, 263
 - TNullValueOption, 31
 - TOptimizerGoalOption, 34
 - TOracleCommand, 198
 - TOracleCommands, 195
 - TOracleCustomPackage, 242; 251
 - TOracleDataSet, 89
 - TOracleDictionary, 114
 - TOracleDirectPathLoader, 203
 - TOracleEvent, 146
 - TOracleLogon, 44
 - TOracleNavigator, 138
 - TOracleObject, 164
 - TOraclePackage, 139
 - TOracleProvider, 260
 - TOracleQuery, 49
 - TOracleQueue, 216
 - TOracleReference, 180
 - TOracleScript, 184
 - TOracleSession, 9
 - TOracleSessionPool, 229
 - TOracleTimestamp, 235
 - TParameterModeOption, 144
 - TPinLockOption, 182
 - TPinOption, 182
 - TPLSQLRecord, 242; 254
 - TPLSQLTable, 242; 256
 - TQBDefinition, 131
 - TQBField, 135
 - TQueryState, 80
 - Transformation, 217; 219
 - Trim, 163
 - TrimElements, 179
 - TrimStringFields, 36
 - TSearchRecordOptions, 124
 - TSessionPoolType, 233
 - TSessionPreferences, 36
 - TSQLTraceOption, 41
 - TTemporaryLOBOption, 36
 - TTransactionMode, 40
 - TXMLType, 240
-
- ## U
-
- UniDirectional, 127
 - UniqueFields, 128; 261
 - Unlock, 234
 - UpdatesPending, 128
 - UpdateStatus, 128
 - UpdatingTable, 105; 128
 - UseMessageTable, 114
 - UseOCI7, 36
 - UTL_File, 42
-
- ## V
-
- Value, 137
 - ValueArray, 258
 - Values, 259
 - VariableCount, 83; 129
 - VariableIndex, 83; 129
 - VariableName, 84; 129
 - Variables, 84; 130
 - Variables property editor, 84
 - VariableType, 88; 130
 - Visibility, 217; 219

W

Wait, 217
WarningFlags, 88
Words, 202
Write, 163

X

XML, 241
XMLAttr, 179

XMLField, 88

Y

Year, 239

Z

ZeroDatelsNull, 36