Oracle Objects for OLE C++ Class Library

Copyright Other Information Sources

For Help on Help, Press F1

Modification History

Version 1.0.42.0 December 1994

Finalized Oracle Objects for OLE

Version 1.0.53.0 (Internal) March 27,1995

Added Modification History Topic
Modified Troubleshooting Topic
Added Copyright Topic
Synchronized Tuning and Customization Topic with ORACLEO
Fixed underline typo at the end of the Transactions topic
Fixed typo in the Redistributable file topic
Removed "kswnotdone" from ExecuteSQL Method
Added descriptions for the Options argument to the ODynaset and ODatabase Open Methods
Fixed spelling mistake (classess to classes)
Fixed return type of GetFieldServerType and GetParameter Methods
Added descriptions for the DynOpts arguments to the OBinder Open Method
Corrected the return value of the GetDynaset Method
Changed the return value of the GetErrorText Method
Updated Copyright
Added version numbers to the C++ part of the Requirements section
Updated Open(OSession) and GetNamed Session to mention that sessions are not sharable across apps.
Added GetRowsProcessed attribute to ODatabase and updated ExecuteSQL(ODatabase)
Altered code example for GetServerErrorText
Added OBinder class to IsOpen method
Removed references to non-existent attributes of OBinder
Added references to newly implemented methods of Obinder (error handling)
Added detail to BindToBinder, highlighting difference between binding controls first or opening the Binder first,
Changed Usage notes for IsNullOK from IsNull to IsNullOK
Changed return type of GetErrorText and LookupErrorText to be (const char *)

Version 1.0.55.0 (Production) April 14,1995

Added IsFirst and IsLast methods to ODynaset and OBinder

Copyright

Oracle Objects for OLE,   Version 1.0

Other Information Sources

The following Oracle publications contain more information about various topics mentioned here:

- *Oracle7 Server Application Developer's Guide*
- *Oracle7 Server Concepts Manual*
- *Oracle7 Server SQL Language Quick Reference*
- *Oracle7 Server SQL Language Reference Manual*
- *PL/SQL User's Guide and Reference*
- *PL/SQL V2.1 and Precompiler's V1.6 Addendum*
- *Oracle7 Server Documentation Addendum*

Requirements

Design Time

- The *Oracle Object Server* requires an application that supports OLE Automation, such as Visual Basic 3.0 (Standard or Professional), Excel 5.0, or Access 2.0.
- The *Oracle Data Control* requires Visual Basic 3.0 (Standard or Professional).
- The *Oracle Objects for OLE C++ Class Library* requires either Microsoft Visual C++ Version 1.5 (16 bit) or Borland C++ Version 4.x (16 bit). Neither the OLE SDK nor OLE development knowledge is necessary.

Run Time

- Windows 3.1 or an equivalent environment capable of running 16 bit Windows applications such as Windows NT (using WOW) or OS/2 (using Win-OS/2).
- A local or remote Oracle7 database.
- Oracle SQL*Net Version 1.x or 2.x if connecting to a remote Oracle7 database.
- Microsoft OLE 2.0.1 or greater run time files (included with the Oracle Objects for OLE installation).

Getting Started

There are a few things you will need to do before you write a single line of C++ code.

First you will need to place all the Oracle Objects for OLE C++ DLLs, libraries and headers in places that are convenient for you. The Oracle Objects for OLE installation procedure placed the DLLs for the main C++ classes in a subdirectory called *bin*. The DLL is called ORACLM.DLL for the version that works with Microsoft's Visual C++. The DLL for Borland is called ORACLB.DLL. Debug versions are available in the *dbg* subdirectory of *bin*. The .lib file for the main class library was placed in a directory called *lib*. The Borland version of the lib file is called ORACLB.LIB. The Microsoft version of the lib file is called ORACLM.LIB. The headers that you need are in a directory called *include*. The headers work with either Borland or Microsoft. If you are using the user-interface widget bound classes you will find the libraries, headers and source in directories called *OMFC* (for Microsoft's MFC framework) or *OOWL* (for Borland's OWL framework).

Source for the entire class library is provided. It is provided for debugging purposes only. It can be found in the *src* subdirectory.

The next thing you will need to do is to make sure that you have a good connection to an Oracle database. Please consult your Oracle documentation on how to do this. A working knowledge of the SQL language is important because it is the way that you will be interacting with the Oracle database.

Then you are ready to start writing programs. The Oracle Objects for OLE C++ Class Library will help you build large model programs.

In your program you will need to initialize the class library before using it and uninitialize it before your program exits. This initialization and uninitialization must be done per application. Normally the initialization is done when the program starts (for instance in the application object initialization method) and the uninitialization is done when the program exits (for instance in the application object destructor). Please see the documentation for the **OStartup** and **OShutdown** methods.

The normal use of Oracle Objects for OLE is to obtain access to the data of an Oracle database using a dynaset object. If you consider the relations between the objects in the class library (Object Relationship) you will see that the dynaset is not the topmost object. You will have to create or instantiate some other objects that a dynaset is dependent on. You typically will construct an **ODatabase** object, which gives you a connection to the database, and then construct an **ODynaset** object, which will give you access to the data of the database. The A Simple Example topic demonstrates this kind of simple access to the database.

An alternative approach is to use managed dynasets. In this case you use the **OBinder** class to take care of the database connection and dynaset. You will use bound objects, objects that are instances of subclasses of **OBound**, to access the data. The Workbook document gives some examples of using the class library this way. It is particularly suitable for programs that provide a user interface to the database's data.

This on-line help system contains a section called Introduction to Class Library which contains introductory material on the class library, including explanatory subtopics.   This system is also the reference manual for the class library, documenting and explaining all the classes and methods.
In addition there is a *Workbook* document that provides some worked-through examples of using the class library.   The user-interface widget libraries (OMFC and OOWL) are documented in separate documents.

Using Oracle Objects for OLE with Visual Basic or VBA

Please see the *Oracle Objects for OLE on-line help system* for further details on using Oracle Objects for with Visual Basic (including the Oracle data control) and with other OLE Automation aware applications.

Overview

Oracle Objects for OLE

Oracle Objects for OLE is a collection of programmable objects that simplifies the development of applications designed to communicate with an Oracle7 database. Oracle Objects for OLE is particularly well suited for any programming environment that supports Visual Basic custom controls (VBX) or OLE Automation.   Oracle Objects for OLE consists of three principle components: the *Oracle Object Server*, the *Oracle Data Control*, and the *Oracle Objects* for OLE *C++ Class Library.*

The Oracle Object Server

The *Oracle Object Server* is an OLE In Process server that supports a collection of programmable objects for Oracle7 databases running either locally or remotely.   An OLE In Process server is a special kind of OLE server, running in a Windows DLL, that supports the OLE Automation interface. An OLE In Process server has no user interface and is not embeddable.   You can access the *Oracle Object Server* through the *Oracle Data Control,* through any application that supports OLE Automation (such as in Visual Basic for Applications, in applications such as Microsoft Excel Version 5.0 and Access 2.0) and through the *Oracle Objects for OLE C++ Class Library.*

The Oracle Data Control

The *Oracle Data Control* is a Visual Basic custom control for use with development tools that support custom controls.   The Oracle Data Control is compatible with the Microsoft data control included with Visual Basic.   If you are familiar with that data control, learning to use the *Oracle Data Control* is quick and easy.

The Oracle Objects for OLE C++ Class Library

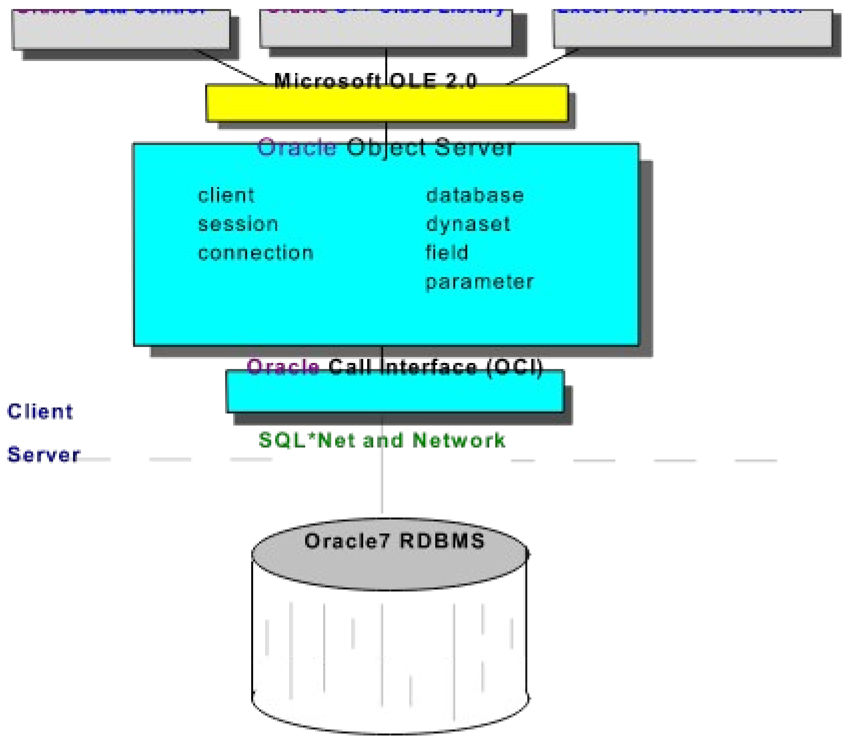
The *Oracle Objects* for OLE *C++ Class Library* is a collection of C++ classes that provide programmatic access to the *Oracle Object Server*.   Although the class library is implemented using OLE Automation, neither the OLE development kit nor any OLE development knowledge is necessary to use it. In addition to the object classes, the class library provides a bound class, which allows controls such as text and list boxes to be linked directly to a field of a dynaset (columns in the database).   The bound class supports late, runtime binding, as is available in Visual Basic.   The *Oracle Objects for OLE C++ Class Library* is supported for Microsoft Visual C++ and the Microsoft Foundation Classes (for the bound class) as well as Borland C++ and the Object Windows Library (for the bound class).

Oracle Objects for OLE, OLE and the Oracle Database

Figure 1 shows the high-level relationship between the *Oracle Data Control,* the *Oracle Objects for OLE C++ Class Library*, the *Oracle Object Server,* OLE, and the Oracle7 database.

Figure 1 Relationships

Close



Introduction to the Class Library

The Oracle C++ Class Library enables developers writing C++ code to access Oracle7 databases easily, quickly, and with a minimum of programming.   The current release is compatible with Microsofts Visual C++ version 1.5 when creating 16-bit applications on a Windows platform. Borland C++ 4.0 is also supported.

Users of this library should have some knowledge of C++ and their development system.   Since Oracle7 is a relational database, access to the data in the database is most easily accomplished using the SQL language.   Therefore, users of this class library should also have some familiarity with SQL.   However, for simple use of the classes, developers unfamiliar with SQL can learn enough from the examples in the accompanying *Workbook* (in the file WORKBOOK.WRI) to get started.

This class library is designed to provide object-oriented access to a relational database.   It is not intended to make the relational database appear as an object-oriented database.   So, for instance, methods are provided for getting data from the database according to a SQL query.   But no methods are provided for stashing C++ objects directly in the database.

In terms of data types, the level of this interface is low.   The data types supported are the simplest C data types: int, long, double, and char *.   There is no explicit support for higher-level data types, although you will not find it difficult to provide this support for your own data types.   In terms of operations, the level is high.   You can open a connection to an Oracle database simply by constructing an **ODatabase** object.   You can obtain an entire set of record from the database by constructing an **ODynaset** object.   You can navigate forward and backward through sets of records returned from the query.   A high-level interface gives the Visual C++ programmer much of the power of the Visual Basic programmer, with bound objects and an interaction model.

The class library was designed to provide the most effortless possible connectivity to an Oracle database.   The objects will handle most mundane tasks.   For instance, you do not have to log off from the database, the **ODatabase** object destructor will do that for you.   You will not have to manage memory; the only memory you need to free is memory that you allocate.   You do not need to run any special tools to build your code.

See the following topics:

A Simple Example
About Binding
Class Naming Conventions
Error Handling
Initializing and Uninitializing the Library
Layers of the Library
Memory Ownership Conventions
Objects as Handles
Supported Data Types

A Simple Example

Consider an application designed to access an Oracle7 database and look at a personnel database. The program is intended to perform some analysis on the salaries of all the employees. Here is what the code would look like:

```
// construct a database, obtaining a database connection
ODatabase  odb("ExampleDB", "scott", "tiger");
// the ODatabase object allows you to connect to a database and
//    to execute SQL statements.  By constructing it with the
//    connection information we immediately get our
//    database connection.

// construct a dynaset, obtaining the data records
ODynaset   odyn(odb, "select * from emp");
// a dynaset corresponds to a cursor.  It gives you access to the
//    rows that are obtained by an SQL select statement.

double     salary;  // will contain salary information

// now look at each record
while (!odyn.IsEOF())
{   // we haven't gotten to the end of the set yet...

    // get the salary for the current record
    odyn.GetFieldValue("sal", &salary);

    // do something with the salary
    Analyze(salary);

    // go to the next record (perhaps moving past last)
    odyn.MoveNext();
}
```

This example opens a database and selects records from an employee database. It then navigates through the result set of the query one record at a time. On each record it obtains the value of the salary column and processes it. The navigation is performed with the **MoveNext** method. You can tell that you have gone through the entire set when the **IsEOF** method returns TRUE.

When the **ODatabase** and **ODynaset** objects go out of scope, all necessary cleanup is performed. That includes disconnecting from the database. Note that the user of the class library does not need to allocate any memory or free any memory.

More examples are presented in the *Workbook*.

About Binding

The Oracle C++ Class Library takes a somewhat different approach to binding than most other C++ database classes. Most classes favor an early binding approach. In this technique, you define classes that correspond to the query result set. Different columns in the result records are bound to data members of the class. The class is generally a subclass of a more general record class. With this approach, you define your data access when you are writing your code (often running a code-generating tool that accesses the database). As you navigate through the query result set, data is placed in every column as each row becomes current - implying a constant moving around of data, with data conversions from database types to native types. All this happens whether or not the code does, in fact, reference the data.

The lower levels of this library (**ODatabase**, **ODynaset**, **OField**) use a late binding approach. You do not need a tool to create classes at the time you write your code. No data is moved out of the record cache until the code asks for it. If a program never asks for the value of the third field in the current record, it isnt translated from the databases data type. Data is obtained and set by method calls.

The difference between the two methods can be illustrated with an example like the one in A Simple Example. Here we expand that example: in addition to looking at salary data, we set everybody's commission.

An early binding scheme would work like this:

```
// before compiling code, run a tool to create the employee class:
class employee : public records
{
public:
    double   salary;
    double   commission;
    char    *name;
}

employee theEmp;
theEmp.Query();    // use records class method to get data
                   // database connection and SQL query are implicit
theEmp.MoveFirst();  // navigate to first record
while (!theEmp.IsEOF())
{
    // we dont have to explicitly get the salary
    Analyze(theEmp.salary);
    theEmp.StartEdit();   // initiate editing
    theEmp.commission = 500.0; // directly change commission value
    theEmp.Update();      // save the changes
    theEmp.MoveNext();
}
```

Using just the **ODynaset** and **ODatabase** classes, we would have:

```
ODatabase  theDB;
ODynaset   theEmp;
double     salary;

theDB.Open("ExampleDB", "scott", "tiger");
```

```
theEmp.Open(theDB, "select ename, sal, comm from employee");

while (!theEmp.IsEOF())
{
    GetFieldValue("sal", &salary);  // fetch the data
    Analyze(salary);
    theEmp.StartEdit();
    SetFieldValue("comm", 500.0);     // set the commission
    theEmp.Update();
    theEmp.MoveNext();
}
```

The difference is that the late binding case specifies neither the database connection nor the query until run time (the argument strings could have been constructed by the program). Instead, methods are called to connect to the database, to get values and to set values. At the same time, the work of binding the employee name (with all the attendant memory management), which is never used, is avoided. Furthermore, when the update is executed, the late binding code needs to update only the changed fields in the database, because it knows which fields have been changed. The early binding case must update all the fields, because it does not know which data members have been changed.

You can make the late binding code even more convenient by using **OField** objects:

```
ODatabase  theDB("ExampleDB", "scott", "tiger");
ODynaset   theEmp(theDB, "select ename, sal, comm from employee");
OField     salary = theEmp.GetField("salary");
OField     comm = theEmp.GetField("comm");

while (!theEmp.IsEOF())
{
    Analyze(salary);    // implicit cast fetches data from OField
    theEmp.StartEdit();
    comm.SetValue(500.0); // set the bonus
    theEmp.Update();
    theEmp.MoveNext();
}
```

Because the setting of the commission calls a method the dynaset knows which fields have been updated so that it can update only changed fields.

The **OBinder** and **OBound** classes provide support for more automatic binding. Use these classes to specify that an object is tied to a particular columns value. You then supply methods to be called when the value changes or when the bound object changes the value. Using these classes you could, for example, implement the equivalent of the early binding code. A very simple example follows:

```
// declarations
OBinder binder;
OBoundTextEdit ename;
OBoundTextEdit sal;

// set up binder object
binder.Open("ExampleDB", "scott", "tiger", "select * from emp");

// attach the bound edits to the user interface widgets
//   wnd is a framework pointer to the widget's parent window
ename.BindToEdit(wnd, IDC_ENAME);
sal.BindToEdit(wnd, IDC_SAL);

// attach the bound edits to the binder
ename.BindToOBinder(&binder, "ename");
sal.BindToOBinder(&binder, "sal");
```

This works like early binding but the binding is not specified until runtime. The OBound subclass objects are attached to particular dynaset fields at runtime. From then on they will receive the current value of the field as the dynaset is navigated. The binder object (m_binder) functions very much like the data control in Visual Basic. It is opened with the information needed to connect to a database (database name, username, password) and a SQL statement that retrieves a set of records. The bound objects (m_ename and m_sal) function very much like bound controls in Visual Basic. In this case they are bound textedits. Calls need to be made to associate the bound textedit with an item in a window (the **BindToEdit** call) and to associate the bound textedit with a field in a particular binder (**BindToOBinder**). For more information see the documentation for **OBound** and **OBinder**.

Class Naming Conventions

Each class of the Oracle Objects for OLE C++ Class Library is described separately, along with its methods and defines, in this Help file. Please note the following important conventions:

- Class names all begin with the letter O. For classes that are handles, the implementation object (that is, the underlying object that the class object refers to) is referred to in this documentation by the same name without the O. For example, the **ODatabase** instance refers to an underlying database instance. (See Objects as Handles for more information on handles and underlying implementation objects.)
- The class library contains **ODatabase** objects (handles), which refer to underlying database objects (implementation objects). In addition, there is the actual Oracle database, typically a program that is running on a server machine that you connect to by way of some network. In this document the database server program is referred to as the Oracle database to distinguish it from the database objects.
- You *open* a handle to attach it to its reference. When you *close* the handle, it is no longer associated with that reference.
- Methods whose name is Open will create an underlying implementation object. For instance, Opening a database will actually create a connection to a database. Methods whose name begins with Get will return a handle object but will not be creating any new underlying implementations. For instance the method **ODynaset::GetDatabase** will return an **ODatabase** object. That object will be a handle on an already existing database object.

Error Handling

The simplest kind of error handling is to know whether a method succeeded in its operation or not. This level of error handling is accomplished by looking at the return value of the method. Most methods in the library return a result of type oresult. An oresult will either have the value OSUCCESS, which indicates that the method worked, or OFAILURE, which indicates that some error occurred during the method's execution.

Once you have found out that something has gone wrong you will often want more information about exactly what went wrong. There are two broad categories of problems:
1) Errors that occur in the server. These are things like invalid SQL statements, attempting to access records that are locked, etc.
2) Errors occurring in the class library. These are things like improperly initialized objects, invalid arguments, out of memory conditions and so on.
The two categories of problems are reported in different ways.

The Oracle errors are reported via the **ServerErrorNumber** and **GetServerErrorText** methods of **OSession** and **ODatabase**. Errors that occur when connecting to the database, or in the execution of transaction operations, will be reported via **OSession**. Errors that occur while processing an SQL statement, for instance when opening a dynaset or using **ExecuteSQL**, will be reported via **ODatabase**. The number returned by **ServerErrorNumber** is a standard Oracle error number. An error number of 0 indicates no error. The text returned by **GetServerErrorText** is the standard Oracle error message. For more information about Oracle errors consult your Oracle documentation.

The class library errors are reported via the **ErrorNumber** and **GetErrorText** methods that are supported by most of the objects (all those that inherit these methods from **OOracleObject**). Each object will have available the most recent state of that object. The object's error state is cleared at the beginning of executing each method. **GetErrorText** will not have explanatory text available for all errors.

Because error reporting is done via additional calls rather than through returns of error codes it is possible to obtain error information about the execution of methods that cannot have error return values such as constructors and overloaded assignment operators. After using a constructor, especially a "construct and open" constructor or an assignment operator you should check an object's error state by calling **ErrorNumber**.

Here is a list of all possible error codes returned by **ErrorNumber** and a brief explanation of each:

Constant	Value	Description
OERROR_NONE	0	No error
OERROR_NOINTER	11	Internal Error
OERROR_MEMORY	12	Couldn't allocate necessary memory.
OERROR_BADERR	13	Internal Error
OERROR_INVPARENT	14	An attempt was made to get an object from an unopened object.
OERROR_SYSTEM	15	Internal Error
OERROR_NOTOPEN	16	An attempt was made to use an unopened object
OERROR_BADARG	17	One of the arguments to the method is invalid.

OERROR_INVRECORD	18	The current record is not valid.
OERROR_ADVISEULINK	4096	Internal Error.
OERROR_POSITION	4098	An attempt was made to retrieve a field value from an empty dynaset.
OERROR_NOFIELDNAME	4099	An invalid field name was specified.
OERROR_TRANSIP	4101	A **BeginTransaction** was specified while a transaction is already in progress
OERROR_TRANSNIPC	4104	A **Commit** was specified without first executing **BeginTransaction**.
OERROR_TRANSNIPR	4105	A **Rollback** was specified without first executing **BeginTransaction**.
OERROR_NODSET	4106	Internal Error.
OERROR_INVROWNUM	4108	An attempt was made to reference an invalid row. This will happen when **IsEOF** or **IsBOF** is True or when the current row has been deleted and no record movement has occurred.
OERROR_TEMPFILE	4109	An error occurred while trying to create a temporary file for data caching.
OERROR_DUPSESSION	4110	An attempt was made to create a named session that already exists.
OERROR_NOSESSION	4111	Internal Error.
OERROR_NOOBJECTN	4112	An attempt was made to reference a named object of a collection (other than the fields collection) that does not exist.
OERROR_DUPCONN	4113	Internal Error.
OERROR_NOCONN	4114	Internal Error.
OERROR_BFINDEX	4115	An invalid field index was specified. The range of indices is 0 to Count -1.
OERROR_CURNREADY	4116	Internal Error.
OERROR_NOUPDATES	4117	An attempt was made to change the data of a nonupdatable dynaset.
OERROR_NOTEDITING	4118	An attempt was made to change a fields value without first executing **StartEdit**.
OERROR_DATACHANGE	4119	An attempt was made to edit data in the local cache, but the data on the Oracle server has been changed.
OERROR_NOBUFMEM	4120	Out of memory for data binding buffers.
OERROR_INVBKMRK	4121	An invalid dynaset mark was specified.
OERROR_BNDVNOEN	4122	Internal Error.
OERROR_DUPPARAM	4123	An attempt was made to create a named parameter using **Add**, but that name already exists.
OERROR_INVARGVAL	4124	An invalid offset or length parameters was passed to **GetChunk** or an internal error has occurred using **AppendChunk**.
OERROR_INVFLDTYPE	4125	An attempt was made to use **GetChunk** or **Append Chunk** on a field that was not of the type Long or Long Raw.
OERROR_TRANSFORUP	4127	A SELECT ... FOR UPDATE was specified without first executing **BeginTransaction**
OERROR_NOTUPFORUP	4128	A SELECT ... FOR UPDATE was specified but the query is non-updatable.
OERROR_TRANSLOCK	4129	A **Commit** or **Rollback** was executed while a SELECT ... FOR UPDATE is in progress.
OERROR_CACHEPARM	4130	An invalid cache parameter was specified.

OERROR_FLDRQROWID	4131	An attempt was made to reference a field that requires a ROWID (Long or Long Raw), but the ROWID was not available.

Initializing and Uninitializing the Library

You must initialize the C++ Class Library before use and uninitialize it when you are finished with it.   Normally the initialization is done at the beginning of your program and the uninitialization is done at the end.

Use the **OStartup** routine to initialize the class library and **OShutdown** to uninitialize it.   You must call these routines for every process.   They initialize per-process state in the class library DLL.

OLE users: These routines call OleInitialize and OleUninitialize.   If OLE is already running when **OStartup** is called, **OShutdown** does not call OleUninitialize.

Layers of the Library

The class library can be divided into several portions.

1. The **ODatabase** and **ODynaset** objects are low-level objects that provide the minimum functionality you need to work with the database.

2. The **OField**, **OValue**, **OParameter**, **ODynasetMark** and **OSession** objects are objects that you will often use.

3. The **OClient**, **OConnection**, **OAdvise**, and the collection classes (**OFieldCollection**, **OSessionCollection**, **OConnectionCollection** and **OParameterCollection**) are classes that you will rarely, if ever, use.

4. Built on top of the classes of the preceding three portions are the **OBinder** and **OBound** classes.   These implement a mechanism that allows you to bind objects of your design to database columns in a way similar to Visual Basics binding of data-aware controls. These classes are more general (and more powerful) than the lower level classes upon which they are built (**ODynaset**, **ODatabase**, etc.).

5. Finally, a framework-specific layer implements GUI widgets that are data-aware.   These classes implement text edits, checkboxes, and so forth; they are built with **OBound** and the framework classes.   These are the OMFC and OOWL libraries.

The first four portions are implemented in the main class library code, provided to you in the oraclm or oraclb DLLs, and are documented in this on-line help system. The last layer is provided in the separate libraries omfc.lib and oowl.lib and are documented separately in two Microsoft Write files: OMFC.WRI (for the Microsoft development environment) and OOWL.WRI (for the Borland development environment).

Memory Ownership Conventions

One reason the C++ Class Library is easy to use is that it relieves developers of concerns about memory allocation. All memory used by the objects themselves is handled by their own methods. Users of the classes never have to allocate or free any pointers.

This design has two important consequences:

1. For all strings passed into objects as method arguments, the objects takes care of managing the string. If the object needs the string beyond the execution time of the method it will make a copy of the string. The caller is never responsible for maintaining a copy of a passed-in string pointer.
2. All strings returned by methods are owned by the object that returned it. Callers do not have to free them (should not free them!) because the object destructors will free them. When the object is destroyed or closed, the strings are freed. Strings returned by methods may be freed at other times. For instance when an **OField** returns a database value as a string it owns the string. Later, if you ask the same **OField** object for the database values as a string again (perhaps the dynaset has been navigated to another record) the old string will be freed. The **OField** only keeps the most current string. The valid lifetimes of returned strings is discussed in the documentation for each method that returns a string pointer.

Objects as Handles

All of the classes in the C++ Class Library that derive from **OOracleObject** (that is, **ODatabase**, **ODynaset**, **OSession**, and so forth, but not **OBound**, **OBinder**, and **OValue**) are implemented using handle-reference semantics. This style is often used to implement such things as strings, where the String class that you use is actually a lightweight handle referencing another object that contains the actual memory allocated string. A reference count is kept on the object that contains the actual string, and as long as there is at least one reference to it, it is kept. Using semantics like this allows you to copy String objects quickly (all that is copied is the reference to the underlying object) and saves storage because multiple uses of the same string all refer to a single store.

Each Oracle C++ class (named O*entity*) is a handle referring to an underlying Oracle Objects for OLE object (called *entity* in this documentation), which is implemented in the OLE in-process server. When you use the C++ Class Library, you have no access to the underlying implementation objects. In Windows the underlying objects are implemented in OLE. Using the Class Library frees you from the details of OLE itself, and from concerns about reference counting.

Consider an **OSession** object. The **OSession** object itself is actually a handle referring to an underlying session object. Consequently, the **OSession** object is lightweight. It can be copied, assigned, passed back from routines, and so forth with little concern for the cost. A simple assignment results in two **OSession** objects, each of which refers to the same underlying session object. This is generally desirable. Keep in mind, however, that the operations on the **OSession**, such as a **Commit**, are actually taking place on the underlying session object. As a result, executing a **Commit** method on one **OSession** is the same as executing a **Commit** on any of the **OSessions** that refer to the same session.

The **ODatabase** object is a more complicated case. The **ODatabase** object refers to an underlying database object. But that implementation object is not the database itself; it is actually a reference to the database. This is what you would probably expect. Executing the **ODatabase** destructor decrements the reference count on the database object, which can then be destroyed. But destroying the database object does not destroy the actual Oracle7 database!

The use of classes entails the concept of an open object or a closed object. Open objects have a current, valid reference to an underlying implementation object. Closed objects do not have a valid reference to an underlying implementation object. In addition, closed objects have freed any auxiliary storage, such as copies of strings passed from callers or back to callers. Closed objects cannot do very much except be opened.

Supported Data Types

The Oracle Objects for OLE C++ Class Library supports the simplest types of C data types: int, long, double, and char *. Two additional types are defined by the library:

- oboolean, a standard TRUE or FALSE container, and
- oresult, which contains result codes from methods.

In this release, the routines return oresults of either OSUCCESS or OFAILURE to indicate whether the methods succeeded or not. In cases of OFAILURE, other methods should be called to determine the precise error. See Error Handling for more information.

The Oracle database does not store values using C++ data types. It has its own type system. The types that the database use are:

OTYPE_VARCHAR2	variable length character
OTYPE_NUMBER	number (either integer or fixed point)
OTYPE_LONG	a long piece of text
OTYPE_ROWID	special record identifier
OTYPE_DATE	a date
OTYPE_RAW	short piece of raw bytes
OTYPE_LONGRAW	a large blob or raw bytes
OTYPE_CHAR	fixed length character
OTYPE_MSLABEL	special type for secure databases

You should consult your Oracle documentation for more information about these types. The text types (varchar2, long and char) are special in that Oracle will perform character set translation on them. A number field may store an integer or a fixed point number depending on its scale and precision. It is important to note that calculations that are done in the database are done with decimal (not binary) rounding.

Locks and Editing

One of the defining features of client-server computing is that many clients may be accessing the server simultaneously. This feature means that several clients may access the same table or record simultaneously. In Oracle7 this issue is generally resolved using locks. Locks allow one client to restrict other clients use of a table or record. Locks are placed temporarily on database entities to prevent confusion and data corruption.

When you use Oracle Objects for OLE, locks are not placed on data until an **ODynaset** executes the **StartEdit** method. The **StartEdit** method attempts to obtain a lock (using "SELECT ... FOR UPDATE") on the current record of the dynaset. This is done as late as possible to minimize the time that locks are placed on the records. The **StartEdit** method can fail for several reasons:

- The SQL query violates Oracle SQL updatability rules, for instance, by using calculated columns or table joins.
- The user does not have the privileges needed to obtain a lock.
- Another user has already locked the record. The **ODatabase::Open** method has an option so that you can decide whether to wait on locks.

Transactions

A transaction is a logical unit of work that comprises one or more SQL statements executed by a single user. A typical example is transferring money from one bank account to another. Two operations take place:

1. Money is taken out of one account.
2. Money is put into the other account.

These operations need to be performed together. If one were to be done and the other not done (for example, if the network connection went down), the banks books would not balance correctly.

Normally, when you execute an **Update** method on a dynaset, the changes are committed to the database immediately. Each operation is treated as a distinct transaction. Using the **BeginTransaction**, **Commit**, and **Rollback** transactional control methods of the **OSession** object allow operations to be grouped into larger transactions. **BeginTransaction** tells the session that you are starting a group of operations. **Commit** makes the entire group of operations permanent. **Rollback** cancels the entire group. **Commit** and **Rollback** end the transaction and the program returns to normal operation: one transaction per operation. Experienced Oracle users should note the following differences between the operation of Oracle Objects for OLE and many Oracle tools:

- Oracle tools such as SQL*Plus execute as if the **BeginTransaction** method was called when the tool was started. This means that updates are not committed immediately, but are held until a commit or rollback is executed.
- SQL*Plus always starts a new transaction every time a commit or rollback is executed.
- The autocommit setting in SQL*Plus results in behavior similar to the default of the Oracle Objects for OLE.

If you are connected to more than one database and use the transaction methods, you should understand that Oracle Objects for OLE commits each database separately. This is *not* the same as the two-phase commit that Oracle7 provides. If your application needs to guarantee data integrity across databases, you should connect to a single database and then access additional databases via the Oracle7 database link feature. This method gives you the benefit of Oracle7s two-phase commit. Consult your Oracle7 documentation for more information about two-phase commit, database links and distributed transactions.

Transactions apply only to the Data Manipulation Language (DML) portion of the SQL language (such as INSERT, UPDATE, and DELETE). Transactions do not apply to the Data Control Language (DCL) or Data Definition Language (DDL) portions (such as CREATE, DROP, ALTER, etc.) of the SQL language. DCL and DDL commands always force a commit, which in turn commits everything done before them.

Long and Long Raw Columns

Putting data into the database

Long and long raw columns of an Oracle database may contain up to 2 gigabytes of data. You can either put the data into the field piecewise (a piece having a maximum size of 64K) by using **ODynaset::AppendFieldChunk** or **OField::AppendChunk**, or you can put the data into the field in one piece using the **ODynaset::SetFieldValue** or **OField::SetValue** methods. If the data's length is less than 64K you can use the **SetValue** method which takes a const char * as an argument. If the data's length is greater than 64K you will need to use the special **SetValue** methods for which you specify the data length. See SetValue for more information.

Fetching

Because these long columns can contain up to 2 gigabytes of data, it is impractical to automatically retrieve all data from a long or long raw column when it is selected. Instead, the first 64K bytes is retrieved and the Oracle ROWID is cached locally so that the row containing the long or long raw column can be located and the long data retrieved using one of the methods **ODynaset::GetFieldChunk**, **ODynaset::GetFieldValue**, **OField::GetChunk**, **OField::GetValue**. Oracle ROWIDs are only available on rows which are updatable so any dynaset which contains a long or long raw column greater than 64K bytes must be updatable.

Editing

When a **StartEdit** is executed, a column's locally cached value is compared to it's current database value. If the values match, then the edit will proceed, else an error is generated. Since long and long raw columns may contain up to 2 gigabytes of data, no comparison of the long and long raw columns is done before an **StartEdit** is executed.

Select for Update

Normally, when a dynaset is created, rows are not locked in the database until **StartEdit** is invoked.   If this is not desirable, the SQL SELECT statement could include the FOR UPDATE construct.   Unfortunately, the FOR UPDATE construct undermines the normal dynaset operations.   You may use FOR UPDATE , but it is not recommended.

Dynasets created with FOR UPDATE are handled correctly in most cases by scanning the SQL statement for the FOR UPDATE construct (This is necessary because the Oracle database functions do not distinguish between SELECT and SELECT FOR UPDATE SQL statements.   It is possible that some exotic FOR UPDATE SQL statements will be treated as not FOR UPDATE - this means that rows are not locked during the lifetime of the dynaset.   If the FOR UPDATE is not recognized, rows will be locked only during an **StartEdit**/**Update** sequence.   However, during the **StartEdit**/**Update** sequence, the row is verified as unchanged before the **StartEdit** is permitted.

The use of FOR UPDATE on dynasets requires that a session transaction be in progress at the time the dynaset is created.   Further, before the session can be committed or rolled back, the dynaset must be closed or an error is returned.   The dynaset is closed when all of the ODynaset objects that refer to it are closed or are destroyed.

If an error results and the application terminates, uncommitted data is rolled back, including pending FOR UPDATE dynasets.

Tuning and Customization

A number of working parameters of Oracle Objects for OLE can be customized. Access to these parameters is provided through the Oracle initialization file, by default named ORAOLE.INI. Each entry currently available in that file is described below. The location of the ORAOLE.INI file is specified by the ORAOLE environment variable. Note that this variable should specify a full pathname to the Oracle initialization file, which is not necessarily named ORAOLE.INI. If this environment variable is not set, or does not specify a valid file entry, then Oracle Objects for OLE looks for a file named ORAOLE.INI in the Windows directory. If this file does not exist, all of the default values listed will apply.

You can customize the following sections of the ORAOLE.INI file:

[Cache Parameters]

A cache consisting of temporary data files is created to manage amounts of data too large to be maintained exclusively in memory. This cache is needed primarily for dynaset objects, where, for example, a single LONG RAW column can contain more data than exists in physical (and virtual) memory.

The default values have been chosen for simple test cases, running on a machine with limited Windows resources. Tuning with respect to your machine and applications is recommended.

Note that the values specified below are for a single cache, and that a separate cache is allocated for each object that requires one. For example, if your application contains three dynaset objects, three independent data caches are constructed, each using resources as described below.

SliceSize = 256 (default)

This entry specifies the minimum number of bytes used to store a piece of data in the cache. Items smaller than this value are allocated the full *SliceSize* bytes for storage; items larger than this value are allocated an integral multiple of this space value. An example of an item to be stored is a field value of a dynaset.

PerBlock = 16 (default)

This entry specifies the number of *Slices* (described in the preceding entry) that are stored in a single block. A block is the minimum unit of memory or disk allocation used within the cache. Blocks are read from and written to the disk cache temporary file in their entirety. Assuming a *SliceSize* of 256 and a *PerBlock* value of 16, then the block size is 256 * 16 = 4096 bytes.

CacheBlocks = 20 (default)

This entry specifies the maximum number of blocks held in memory at any one time. As data is added to the cache, the number of used blocks grows until the value of *CacheBlocks* is reached. Previous blocks are swapped from memory to the cache temporary disk file to make room for more blocks. The blocks are swapped based upon recent usage. The total amount of memory used by the cache is calculated as the product of (*SliceSize* * *PerBlock* * *CacheBlocks*).

<u>Recommended Values</u>: You may need to experiment to find optimal cache parameter values for your applications and machine environment. Here are

some guidelines to keep in mind when selecting different values:

- The larger the (*SliceSize* * *PerBlock*) value, the more disk I/O is required for swapping individual blocks.
- The smaller the (*SliceSize* * *PerBlock)* value, the more likely it is that blocks will need to be swapped to or from disk.
- The larger the *CacheBlocks* value, the more memory is required, but the less likely it is that swapping will be required.

A reasonable experiment for determining optimal performance might proceed as follows:

- Keep the *SliceSize* >= 128 and vary *PerBlock* to give a range of block sizes from 1K through 8K.
- Vary the *CacheBlocks* value based upon available memory. Set it high enough to avoid disk I/O, but not so high that Windows begins swapping memory to disk.
- Gradually decrease the *CacheBlocks* value until performance degrades or you are satisfied with the memory usage. If performance drops off, increase the *CacheBlocks* value once again as needed to restore performance.

[Fetch Parameters]

FetchLimit = 20 (default)

This entry specifies the number of elements of the array into which data is fetched from Oracle. If you change this value, all fetched values are immediately placed into the cache, and all data is retrieved from the cache. Therefore, you should create cache parameters such that all of the data in the fetch arrays can fit into cache memory. Otherwise, inefficiencies may result.

Increasing the *FetchLimit* value reduces the number of fetches (calls to the database) calls and possibly the amount of network traffic. However, with each fetch, more rows must be processed before user operations can be performed. Increasing the *FetchLimit* increases memory requirements as well.

FetchSize = 4096 (default)

This entry specifies the size, in bytes, of the buffer (string) used for retrieved data. This buffer is used whenever a long or long raw column is initially retrieved.

[General]

TempFileDirectory = [*Path*]

This entry provides one method for specifying disk drive and directory location for the temporary cache files. The files are created in the first legal directory path given by:

1. The drive and directory specified by the TMP environment variable (this method takes precedence over all others);
2. The drive and directory specified by this entry (*TempFileDirectory*) in the [general] section of the ORAOLE.INI file;
3. The drive and directory specified by the TEMP environment variable; or
4. The current working drive and directory.

HelpFile = [*Path and File Name*]

This entry specifies the full path (drive/path/filename) of the Oracle Objects for OLE help file as needed by the Oracle Data Control.   If this entry cannot be located, the file ORACLEO.HLP is assumed to be in the directory where ORADC.VBX is located (normally \WINDOWS\SYSTEM).

About Sample Code and Applications

For the sample applications shipped with Oracle Objects for OLE and most of the sample code in this file, the following rules apply:

- The user scott with password tiger (scott/tiger) are used to connect to the database.
- The SQL*Net alias ExampleDb is used as the database name.
- The data tables referenced are the standard Oracle demonstration tables that can be created by the script **DEMOBLD7.SQL**. (These tables and views can be dropped by the script **DEMODRP7.SQL**.
- The stored procedures referenced can be created by the script **ORAEXAMP.SQL**.

Those examples that do not use the ExampleDB database reference other databases, for one illustrative purpose or another.

Redistributable Files

Oracle Objects for OLE

The following redistributable files are part of Oracle Objects for OLE and must be distributed with your application developed using Oracle Objects for OLE:

ORAIPSRV.DLL
ORAIPSRV.REG
ORAIPSRV.TLB

These files should be installed in the \WINDOWS\SYSTEM directory. In addition to including these three files with your application, you must register in the Windows registration database the information found in ORAIPSRV.REG.

Finally, you must also distribute the file from the following list corresponding to the development software you used to build your application:

ORACLB.DLL (for Borland C++ 4.0)
ORACLB45.DLL (for Borland C++ 4.5)
ORACLM.DLL (for Microsoft C++)
ORADC.VBX (for Visual Basic 3.0)

Microsoft OLE 2.0

The following files are part of Microsoft OLE 2.0 and are required by Oracle Objects for OLE. Please refer to the OLE 2 Programmer's Reference or the Visual Basic documentation for details on installing and distributing these files.

COMPOBJ.DLL
OLE2.DLL
OLE2.REG
OLE2CONV.DLL
OLE2DISP.DLL
OLE2NLS.DLL
OLE2PROX.DLL
STDOLE.TLB
STORAGE.DLL
TYPELIB.DLL

Visual Basic 3.0

The following files are part of Visual Basic and are required by Oracle Objects for OLE. If you are shipping a Visual Basic application that uses Oracle Objects for OLE, you should include at least the following files (in addition to other Visual Basic runtime files that you may need). Please refer to the Visual Basic documentation for details on installing and distributing these files.

VBOA300.DLL
VBRUN300.DLL

Troubleshooting

OLE Initialization or OLE Automation Errors

The most frequent cause of OLE Initialization and Automation errors is missing or incorrectly installed software. Please ensure correct installation of the software specified below. Then make sure that you have specified method and property names correctly and that you have declared all "Oracle objects" as type "object".

Possible Cause	**Solution**
Your system does not contain the Microsoft OLE 2.0 runtime files or these files are out of date. Note that Visual Basic 3.0 does not include the OLE file TYPELIB.DLL.	Reinstall Oracle Objects for OLE and select the "Microsoft OLE 2.0 Libraries" option.
OLE 2.0 information was not registered in the Windows registration database.	Run REGEDIT.EXE and merge the information from the file OLE2.REG (normally located in \WINDOWS\SYSTEM).
The Oracle Objects OLE object information was not registered in the Windows registration database.	Reinstall Oracle Objects for OLE and select the "Oracle Objects Server" option.
The Oracle Objects OLE object information was not registered in the Windows registration database.	Run REGEDIT.EXE and merge the information from the file ORAIPSRV.REG (normally located in \WINDOWS\SYSTEM).
Your system does not contain the Oracle Required Support Files (ORA71WIN.DLL, CORE3WIN.DLL, NLS23WIN.DLL, etc.) or these files are not on the PATH.	Reinstall Oracle Objects for OLE and select the "Oracle Required Support Files" option or add to your PATH environment variable the directory where these files are located.
Your system does not contain Oracle SQL*Net or its file are not on the PATH..	Install Oracle SQL*Net or add to your PATH environment variable the directory where these files are located.
You have misspelled a method or property name.	Check the documentation to determine the correct spelling.
You have referenced a method or property from the wrong object.	Check the documentation to determine the correct object.

Oracle SQL*Net Errors

The most frequent cause of Oracle SQL*Net errors is incorrectly specified connection information. The connection information for Oracle Objects for OLE is specified differently than when using ODBC. Please verify that you have specified connection information correctly, and then make sure your SQL*Net connection is working properly before using Oracle Objects for OLE. The appropriate Oracle SQL*Net documentation contains information about testing your connection and about any Oracle SQL*Net error that you may receive.

Possible Cause	**Solution**
Incorrect **Connect** property or argument to the **OpenDatabase** method.	See the topics on the **Connect** property or the **OpenDatabase** method for examples.
Incorrect **DatabaseName** property or argument to the **OpenDatabase** method.	See the topics on the **DatabaseName** property or the **OpenDatabase** method for examples.
Your system does not contain Oracle SQL*Net.	Install Oracle SQL*Net.

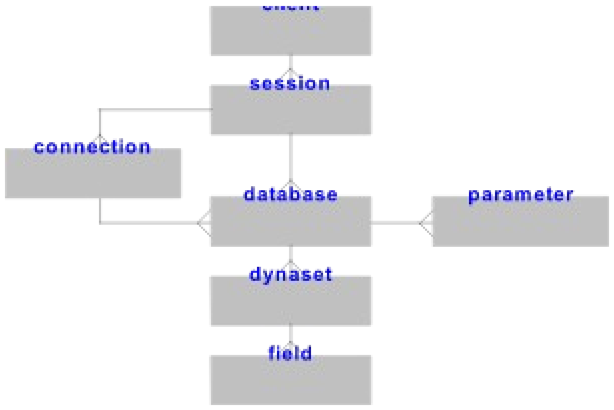
General Protection Faults

The most frequent cause of GPFs is installing Oracle Objects for OLE while other applications are running that require the Oracle Object Server, Oracle Required Support Files or OLE 2.0. To avoid this, install Oracle Objects for OLE immediately after starting Windows and before running any other application.

Possible Cause	**Solution**
Duplicate Oracle Objects for OLE files exist in the \WINDOWS or \WINDOWS\SYSTEM directories or along the PATH.	Remove any duplicate files. The files ORAIPSRV.DLL, ORAIPSRV.TLB and ORAIPSRV.REG should be located in \WINDOWS\SYSTEM.
Duplicate Oracle Required Support Files DLLs exist in the \WINDOWS or \WINDOWS\SYSTEM directories or along the PATH.	Remove any duplicate files. Typically, the Oracle Required Support Files DLLs (ORA7*.DLL, CORE*.DLL, NLS*.DLL) are located in \ORAWIN\BIN.
Duplicate OLE 2.0 DLLs exist in the \WINDOWS or \WINDOWS\SYSTEM directories or along the PATH.	Remove any duplicate files. The OLE 2.0 DLLs (listed in the Redistributable Files topic) should be located in \WINDOWS\SYSTEM.

Object Relationships

An operational hierarchy of the objects expresses has-a and "belongs-to" relationships. This hierarchy can be drawn as follows:



The "crows feet" indicate the many ends of one-to-many or many-to-one relationships.

Each **client** object can have many **session** objects.
Each **session** object can only be associated with one **client** object.

Each **session** object can have many **connection** objects.
Each **connection** object may be shared by many **database** objects although these must be within the same **session** object.

Each **database** object belongs to only one **session** object.
Each **dynaset** object belongs to only one **database** object.

Each **field** object belongs to only one **dynaset** object.
Each **parameter** object belongs to only one **database** object.

Some of these objects can be explicitly created (using Open methods) while others are implicitly created as necessary. The **session**, **database**, and **dynaset** objects can be explicitly created.

One **client** object exists per workstation.   This object is created when the first **session** object is created.   A **connection** object may be created when a **database** is created if that **database** is not sharing a previously created connection.

Classes

Close

- OAdvise
- OBinder
- OBound
- OClient
- OConnection
- OConnectionCollection
- ODatabase
- ODynaset
- ODynasetMark
- OField
- OFieldCollection
- OOracleCollection
- OOracleObject
- OParameter
- OParameterCollection
- OSession
- OSessionCollection
- OValue

Classes

- OAdvise
- OBinder
- OBound
- OClient
- OConnection
- OConnectionCollection
- ODatabase
- ODynaset
- ODynasetMark
- OField
- OFieldCollection
- OOracleCollection
- OOracleObject
- OParameter
- OParameterCollection
- OSession
- OSessionCollection
- OValue

OAdvise

The **OAdvise** class enables you to set up callbacks that attach to a dynaset. When operations occur on the dynaset attached **OAdvise** instances are notified of the operations. You will not declare any instances of **OAdvise** yourself. Instead, you write a new class that is a subclass of **OAdvise**, and your subclass then receives calls to its methods. **OAdvise** is a subclass of **OOracleObject**.

When an operation occurs on a dynaset, the dynaset

1. calls all attached advisories before the operation occurs and allows them to veto the operation; and
2. calls all attached advisories after the operation to tell them that it occurred.

In addition, when the status of the dynaset changes, it notifies the advisory. In this release, the only dynaset status change is that which occurs when the dynaset has found the last record.

The advisories are given a message that tells them what is happening. These messages are one of three types: navigational advisories, other advisories, and status changes. The specific defines are as follows:

```
// navigation advisories
OADVISE_MOVE_FIRST
OADVISE_MOVE_PREV
OADVISE_MOVE_NEXT
OADVISE_MOVE_LAST
OADVISE_MOVE_TOMARK          // move to mark

// other advisories
OADVISE_REFRESH              // dynaset being refreshed
OADVISE_DELETE               // record being deleted
OADVISE_ADDNEW               // new record being added
OADVISE_UPDATE               // dynaset being updated
OADVISE_ROLLBACK             // session being rolled back
OADVISE_OTHER                // undefined advisories

// status changes
OADVISE_FOUNDLAST            // dynaset knows that the last record has been read
```

The **ActionRequest** method is called before the operation. The advisory can cancel an operation by returning FALSE from the **ActionRequest** method; it must return TRUE from **ActionRequest** to allow the operation. The **ActionNotify** method is called after the operation. The **StatusChange** method is called when the dynaset status changes.

The **OAdvise** class does nothing. Its **ActionRequest** method always returns TRUE. To obtain other behavior, create a subclass of **OAdvise** and override the **ActionRequest**, **ActionNotify**, and **StatusChange** methods. Declare an instance of your subclass, and then open it to attach it to a dynaset.

It is often useful for your subclass to have its own members for reference to some sort of application context. You can add these, along with methods to set the context, in your own class. An example of an **OAdvise** subclass is provided in the *Workbook*.

The **OAdvise** class supports the following methods:

Construction and destruction:

OAdvise
~OAdvise
operator=

Attributes:

operator==
operator!=
IsOpen

Operations:

ActionNotify	GetDynaset
ActionRequest	Open
Close	StatusChange

ErrorHandling:

ErrorNumber
ErrorReset
GetErrorText
LookupErrorText

OBinder

Suppose you are writing an application that accesses data in an Oracle database. You have obtained a dynaset on a set of records and are writing a graphical user interface program to access those records. As you navigate through, add, or delete records, you want the value of some column to be reflected in some object such as a variable or, more typically, in an edit control user interface widget. Much bookkeeping is involved in such programming: when to get new values for the column objects, when to clear them, when to put their changed values into the Oracle database, and so forth.

The **OBinder** and **OBound** classes solve this problem. An instance of the **OBinder** class functions as the bookkeeper. You tell it the SQL query and it manages the dynaset. If you want, it can also manage the database object. Then, you bind instances of **OBound** objects to the **OBinder** instance. Each **OBound** object becomes bound to a particular column in the query. At appropriate times, the **OBound** object is told its new value (for example, after navigation to a new record). The **OBound** object tells the **OBinder** when the value has been changed (for example, when the text of a edit control has been modified). Later, again at the appropriate time, the **OBound** object is told when to save the data to the Oracle database.

OBound objects display data, allow changes to data, and note when changes are made. At the time the first change occurs, an **ODynaset::StartEdit** call is made to begin editing the record. This may fail (for example, for reasons of privilege; see the **StartEdit** documentation). If a record has been changed and the dynaset attempts to move to another record, the values in the changed record are set, by way of calls to **OBound::SaveChange**, and the record updated.

Users familiar with Visual Basic will notice that the **OBinder** instance functions as a VB data control (though it has no user interface), and the **OBound** instances function as VB bound controls.

You never actually declare instances of **OBound** objects. **OBound** is always subclassed. (See the **OBound** documentation for more information.)

Often you need to modify the specific behavior of the **OBinder** class. The power of C++ virtual functions are valuable in such situations. For most common operations, a trigger function is called both before and after the operation is performed. You can change the behavior of the **OBinder** class by overriding various trigger functions. The **OBinder** class and the **OBound** class have separate trigger functions, so that behavior can be overridden at either the query level or at the individual bound object level. (Triggers will be familiar to users who have developed using Oracle Forms. The **OBinder** triggers correspond to block-level triggers; the **OBound** triggers correspond to item-level triggers.)

Consider a typical example:
You have built a form in which each column in a table is represented by an edit control - that is, each column is bound to a text widget subclass of **OBound** (such subclasses of **OBound** are provided in the OMFC and OOWL libraries). When a new record is added, you want a default value set for the color column. You would make your own subclass of the **OBound** class, overriding the **PostAdd** trigger method. You would then change your code so that the color column is bound to this new subclass. Then, after a new record is added, your method would be called. (More

concrete examples are provided in the *Workbook*.)

The times that the various trigger methods are called is documented in the section OBinder Trigger Methods. **OBinder** trigger methods are always called before **OBound** triggers. Then, every **OBound** objects equivalent trigger is called. The object that was bound first has its trigger called first, the second object bound has its trigger called second, and so on. If at any time a trigger method returns an OFAILURE return, no more triggers are called. If this is a pretrigger, an OFAILURE return also cancels the action.

There is no method in **OBinder** to bind **OBound** objects. To bind **OBound** objects, use the **BindToBinder** method of the **OBound** class.

The **OBinder** class supports the following methods:

Construction and destruction:

- OBinder
- ~OBinder

Attributes:

- GetChangedError
- IsChanged
- IsFirst
- IsLast
- IsOpen

Operations:

- AddNewRecord
- Changed
- Close
- DeleteRecord
- DiscardChanges
- DuplicateRecord
- GetDatabase
- GetDynaset
- MoveFirst
- MoveLast
- MoveNext
- MovePrev
- MoveToMark
- OnChangedError
- Open
- Refresh
- RefreshQuery
- SetSQL
- UnbindObj
- Update

Triggers:

- PostAdd
- PostDelete
- PostMove
- PostQuery
- PostRollback
- PostUpdate
- PreAdd
- PreDelete
- PreMove
- PreQuery
- PreRollback
- PreUpdate
- Shutdown
- Startup

ErrorHandling:

- ErrorNumber
- ErrorReset
- GetErrorText
- LookupErrorText

OBound

You use **OBound** objects in conjunction with an instance of the `OBinder` class.   (See the description of the **OBinder** class for details on how these two classes are used, including discussion of the trigger methods.)

OBound is an abstract class; you never declare any instances of it. This release contains a set of **OBound** subclasses that implements data-aware user interface widgets.   You can subclass those subclasses, or you can subclass **OBound** directly to implement other bound objects.   The *Workbook* contains examples of creating **OBound** subclasses, and how to use the provided subclasses.

OBound subclasses must implement two methods: **Refresh** and **SaveChange**.   The **Refresh** method transfers a value from the dynaset to the **OBound** instance.   It is called, for example, whenever the dynaset navigates to a new record.   The **SaveChange** method is called when it is time to set the value in the Oracle database, for example, after a change has been made and the dynaset is trying to move to a different record.   It is usually implemented using the **SetValue** method of **OBound**.

OBound subclasses may also override the default triggers. If you do this, you should know that the overloaded trigger you write should call the default trigger in the **OBound** class. This will cause the bound object to be refreshed (have its value updated) when the database record is altered. Alternatively, the overloaded PostAdd, PostMove, PostRollback and PostQuery triggers can call RefreshBound() directly. In either case, the **Refresh** method you wrote above is called to refresh the bound control.

OBound instances generally allow some form of editing of the data that they contain. When a change is made, the **OBound** instance must call the **Changed** method to inform the **OBinder**/**OBound** bookkeeping machinery that the object has been changed.

The **OBound** class supports the following methods:

Construction and destruction:

OBound
~OBound
operator=

Attributes:

operator==	IsChanged
operator!=	IsOpen
Changed	

Operations:

BindToBinder	GetValue
Close	Refresh
GetDatabase	SaveChange
GetDynaset	SetValue
GetName	Unbind

Triggers:

PostAdd	PreDelete

PostDelete	PreMove
PostMove	PreQuery
PostQuery	PreRollback
PostRollback	PreUpdate
PostUpdate	Shutdown
PreAdd	Startup

ErrorHandling:

- ErrorNumber
- ErrorReset
- GetErrorText
- LookupErrorText

OClient

The client object is primarily an internal bookkeeper.   One client object is created automatically for each workstation.   You may never need to declare an **OClient** object.   You can use an **OClient** object to obtain a list of the sessions on the workstation.   **OClient** is a subclass of **OOracleObject**.

You get OClient objects from an OSession object using the GetClient method.

The OClient class supports the following methods:

Construction and destruction:

OClient
~OClient
operator=

Attributes:

operator==
operator!=
IsOpen

Operations:

Close
GetName
GetSessions

ErrorHandling:

ErrorNumber
ErrorReset
GetErrorText
LookupErrorText

OConnection

A connection object is primarily an internal bookkeeper. Multiple database objects within the same session that use the same connection information (database name, username, and password) share a single connection to the Oracle database. You can use an **OConnection** object to ask for the database name and connection string that were used to start the connection. **OConnection** is a subclass of **OOracleObject**.

You can get **OConnection** objects, using the **GetConnection** method, from an **OConnectionCollection** object, an **ODatabase** object, or an **ODynaset** object.

The **OConnection** class supports the following methods:

Construction and destruction:

OConnection
~OConnection
operator=

Attributes:

operator==
operator!=
IsOpen

Operations:

Close
GetConnectString
GetDatabaseName
GetSession

ErrorHandling:

ErrorNumber
ErrorReset
GetErrorText
LookupErrorText

OConnectionCollection

This class is a convenient wrapper for a collection of **OConnections**. It is a subclass of **OOracleCollection**.

You can get **OConnectionCollection** objects from an **OSession** object with the **GetConnections** method. The **OConnectionCollection** is dynamic: it always reflects the current set of connections of the session it came from, even if the connections are added after you get the collection. It is read-only. You cannot add **OConnections** through the collection.

You can get a particular **OConnection** object from the **OConnectionCollection** using the **GetConnection** method.

The **OConnectionCollection** class supports the following methods:

Construction and destruction:

OConnectionCollection
~OConnectionCollection
operator=

Attributes:

operator==
operator!=
IsOpen

Operations:

Close
GetConnection
GetCount

ErrorHandling:

ErrorNumber
ErrorReset
GetErrorText
LookupErrorText

ODatabase

A database object represents an Oracle database. If more than one database is opened on a single session with the same database name and connect information (username and password), those database objects share a connection (see **OConnection**).

Use **ODatabase** to log on to an Oracle database. You can then execute SQL statements directly with the **ExecuteSQL** method or you can get records from the Oracle database by using the **ODatabase** object to create **ODynaset** objects (see **ODynaset**). **ODatabase** is a subclass of **OOracleObject**.

Two options are available for the **ODatabase** at the time it is opened. These options are ORed together on the Open method call. They affect operations on dynasets attached to this database. The options are:

```
ODATABASE_PARTIAL_INSERT        // turn on requerying on insert
ODATABASE_EDIT_NOWAIT           // turn on no waiting on StartEdit calls
```

By default these options are both off. The default setting of the options are reflected by:

```
ODATABASE_DEFAULT               // the default settings
```

ODATABASE_PARTIAL_INSERT affects the way that **AddNew**, **Edit**, and **Update** put values into the Oracle database and how they change the values of the local data cache. (See **ODynaset** for a description of the local data cache.) The "partial insert" option of the database affects all dynasets that are created on the database.

If ODATABASE_PARTIAL_INSERT is off (the default), dynasets behave the way they do in Visual Basic:

- When adding a new record, all fields that were selected to create the dynaset are inserted into the Oracle database. Fields that have not been explicitly set by the user are set to NULL. Any fields that are actually in the table, but do not appear in the dynaset, are set to the default values that Oracle would use for the column.
- When updating a record, only fields that have been modified are updated. The local cache is set to the values that have been inserted into the Oracle database.
- For both adding and updating, the local data cache is set to the values inserted into the Oracle database (including the NULLs).

If ODATABASE_PARTIAL_INSERT is on, dynasets will behave slightly differently:

- When adding new records, only fields that are NOT NULL are inserted into the Oracle database.
- After adding a new record or updating an existing record, the local cache is set by requerying the Oracle database for that record after the add or update. This allows the Oracle database to set values for fields, perhaps by way of database triggers (which are a kind of stored procedure).

If you are accessing tables that have default values for their fields, or that have triggers that set values on Insert or Update, then you will probably want to user the ODATABASE_PARTIAL_INSERT option. This will cause the dynaset to refetch records after it makes changes to the data, ensuring that the dynaset and the database will always agree on the record's data. Please see StartEdit for more information.

ODATABASE_EDIT_NOWAIT affects what happens when **ODynaset::StartEdit** method is called. When **StartEdit** is called, the dynaset attempts to obtain a lock on the record that is being edited. (See Locks and Editing). It also affects **ExecuteSQL** when the SQL statement being executed performs inserts, updates, or deletes.

- If ODATABASE_EDIT_NOWAIT is off and some other user has a lock on the record, your process waits until the lock is removed. This results in possible deadlock if the caller of **StartEdit** has outstanding locks that are in turn holding up the user with a lock on the record you want now.
- If ODATABASE_EDIT_NOWAIT is on and another user has a lock on the record, the **StartEdit** call fails. It is up to the caller to handle the failure properly.

The **ODatabase** class supports the following methods:

Construction and destruction:

ODatabase
~ODatabase
operator=

Attributes:

operator==
operator!=
IsOpen

Operations:

Close
ExecuteSQL
GetConnection
GetConnectString
GetName
GetOptions
GetParameters
GetRowsProcessed
GetSession
Open

ErrorHandling:

ErrorNumber
ErrorReset
GetErrorText
GetServerErrorText
LookupErrorText
ServerErrorNumber
ServerErrorReset
ServerErrorSQLPos

ODynaset

The **ODynaset** class creates, manages, and accesses records of data from the database. It corresponds to a scrollable cursor. **ODynaset** is a subclass of **OOracleObject**.

An **ODynaset** is opened by executing a query against an Oracle database in the form of a SQL select statement. Any legal select statement is acceptable. All the database records that this query returns are referred to as the dynaset's *result set*. Records from the result set are fetched to the client as needed and cached locally. You then operate on the records of the dynaset one at a time. The record you are currently working with is referred to as the *current record*.

You can obtain field values from the current record (**GetFieldValue**), edit the current record (**StartEdit**, **SetFieldValue**, **Update**), delete the current record (**DeleteRecord**), or duplicate the current record (**DuplicateRecord**).

You can change the current record, navigating through the dynaset's result set, by using one of the Move methods. Execution of a Move method changes which record is current. The records in a dynaset's result set will be in some order. To specify a particular order, use an order by clause in your SQL statement to order the results. Records that are not explicitly ordered may be returned in different order on different queries. Records that are added are added at the end of the result set.

Records that have been fetched from the Oracle database and placed in the local cache do not reflect changes made to the data in the database until the dynaset is refreshed.

When operations - either navigations or operations such as Updates - are properly performed on the dynaset, messages are sent to any attached advisory objects. These messages may cancel the operation. (See the **OAdvisory** class for more information.) No advisories are attached to dynasets by default.

You can use the **Clone** method to open a separate dynaset (on the same record cache) with an independent current record mark. Cloned dynasets are read-only.

You can access data in the result set either directly (through **ODynaset** methods such as **SetFieldValue** and **GetFieldValue**) or by getting an **OField** object and using it to access the data. The **OField** object always contains the data from the current record. Columns can be referred to either by name or by index. The index starts at 1 and is the position of the column in the select statement. A column name is either the database column name or an aliased name if the SQL select statement aliases a column name. For example:

```
select ename, emp.sal, nvl(comm,0) "commission" from emp
```

will result in three columns. Column ename will have index 1, column sal will have index 2, and column commission will have index 3. It is more efficient to refer to columns by index. You can use the method **GetFieldIndex** to translate a field name into its index.

It is often important to know whether the opening select statement will result in an updatable dynaset. In general, queries that perform joins, have aliased column names, or have calculated columns are not updatable. In addition, queries that perform a select distinct are not updatable. Dynasets based on nonupdatable queries will fail the **AddNew**, **Duplicate**, Sta**r**tEdit, and **Update** methods.

Several options are set for a dynaset at the time it is opened. These options are ORed together on the **Open** method call. The options are:

ODYNASET_NOBIND	// do not use bindable parameters
ODYNASET_KEEP_BLANKS	// do not strip trailing blanks in values
ODYNASET_READONLY	// make this dynaset nonupdatable
ODYNASET_NOCACHE	// do not make a local record cache for this dynaset

By default all the options are off. This state is represented by:

ODYNASET_DEFAULT	// the default settings

By default, a dynaset attempts to use any available bindable parameters when it is being opened (see OParameter for more information on parameters). Therefore, when issuing a SQL statement that does not refer to any parameters, you can make it more efficient by specifying nobind.

By default, values returned by a dynaset are stripped of trailing blanks. You can retain the trailing blanks by specifying keep_blanks.

By default, dynaset objects are updatable, which requires some processing overhead. Therefore, if you know that you will be using a dynaset only for reading data, you can make it more efficient by specifying readonly.

By default, a local record cache is created for each dynaset to allow reverse scrolling. The record cache requires considerable overhead. If you know that you will simply be reading through the returned values of a dynaset, you can improve performance by turning on the "nocache" option. Dynasets without a record cache will not be able to move backwards or move to a dynaset mark.

A dynaset is opened with a particular SQL statement, which specifies the values to return from the Oracle database. The **Refresh** method of **ODynaset** is handy for reexecuting the SQL statement. This is useful in two circumstances:
1. the values in the database are changing and you want to obtain the new values, or
2. the SQL statement that you have used is parameterized (uses syntax of the form :parameter) and the value of the parameter has changed.

Refreshing a dynaset resets the local record cache and moves the current record to the first record of the new result set.

The **ODynaset** class supports the following methods:

Construction and destruction:

ODynaset
~ODynaset
operator=

Attributes:

operator==
operator!=
CanMark

GetFieldServerTyp
e
GetFieldSize

- CanRefresh
- CanScroll
- CanTransact
- CanUpdate
- GetEditMode
- GetFieldCount
- GetFieldIndex
- GetFieldPrecision
- GetFieldScale
- GetFieldServerSize
- GetOptions
- GetRecordCount
- GetSQL
- IsBOF
- IsEOF
- IsFieldNullOK
- IsFieldTruncated
- IsFirst
- IsLast
- IsOpen
- IsValidRecord

Operations:

- AddNewRecord
- AppendFieldChunk
- CancelEdit
- Clone
- Close
- DeleteRecord
- DuplicateRecord
- GetConnection
- GetDatabase
- GetField
- GetFieldChunk
- GetFields
- GetFieldValue
- GetLastModifiedMark
- GetMark
- GetSession
- MoveFirst
- MoveLast
- MoveNext
- MovePrev
- MoveToMark
- Open
- Refresh
- SetFieldValue
- SetSQL
- StartEdit
- Update

ErrorHandling:

- ErrorNumber
- ErrorReset
- GetErrorText
- LookupErrorText

ODynasetMark

Sometimes it is desirable to navigate directly back to some previously seen record. Dynaset marks provide you with this functionality. You obtain a mark from a dynaset and then later you can use the mark on the dynaset to reset the current record. ODynasetMarks can be used across **ODynasets** that refer to the same underlying query result sets (that is, all **ODynasets** that are copies or clones of the **ODynaset** that the mark was originally taken from). **ODynasetMark** is a subclass of **OOracleObject**.

See the **ODynaset** class methods **GetMark** and **GetLastModifiedMark** to learn how to obtain an **ODynasetMark**. See the **ODynaset** class method **MoveToMark** to learn how to use an **ODynasetMark** to navigate to the marked record.

The **ODynasetMark** class supports the following methods:

Construction and destruction:

ODynasetMark
~ODynasetMark
operator=

Attributes:

operator==
operator!=
IsOpen

Operations:

Close

ErrorHandling:

ErrorNumber
ErrorReset
GetErrorText
LookupErrorText

OField

A field object represents a single column of data in a single record. **OField** objects are not independently opened; rather, you get them from an **ODynaset**. When navigation occurs on the dynaset, the **OField** objects value changes to the value of the column in the new current record. **OFields** are convenient because they always reflect the value of the field in the current record. (In some cases, however, you may find it more straightforward to manipulate field values directly from the **ODynaset** using the **GetFieldValue** and **SetFieldValue** methods.) OF**i**eld is a subclass of **OOracleObject**.

You can get **OField** objects from either an **ODynaset** or **OFieldCollection** object, using the **GetField** method.

The **OField** class supports the following methods:

Construction and destruction:

- OField
- ~OField
- operator=

Attributes:

- operator==
- operator!=
- GetName
- GetPrecision
- GetScale
- GetServerSize
- GetServerType
- GetSize
- IsNullOK
- IsOpen
- IsTruncated

Operations:

- AppendChunk
- Close
- GetChunk
- GetDynaset
- GetValue
- operator const char *
- operator double
- operator int
- operator long
- SetValue

ErrorHandling:

- ErrorNumber
- ErrorReset
- GetErrorText
- LookupErrorText

OFieldCollection

The **OFieldCollection** object is the set of fields in a record in a dynaset query result set. You can get **OFieldCollection** objects from an **ODynaset** object using the **GetFields** method.   **OFieldCollection** is a subclass of **OOracleCollection**.

The **OFieldCollection** is dynamic: it always reflects the current set of fields of the dynaset.   It is read-only.   You cannot add **OFields** through the collection.

You can get a particular field from the **OFieldCollection** with the **GetField** method.

The **OFieldCollection** class supports the following methods:

Construction and destruction:

OFieldCollection
~OFieldCollection
operator=

Attributes:

operator==
operator!=
GetCount
IsOpen

Operations:

Close
GetField

ErrorHandling:

ErrorNumber
ErrorReset
GetErrorText
LookupErrorText

OOracleCollection

OOracleCollection is a base class from which the collection classes are derived (see **OConnectionCollection**).   You never declare an **OOracleCollection** instance; rather, **OOracleCollection** supplies the **GetCount** method on collections.

OOracleObject

OOracleObject is a base class from which most of the l database classes are derived (all except **OValue**, **OBound** and **OBinder**).   You never declare an **OOracleObject** instance.   All **OOracleObject** instances support the error-reporting methods **ErrorNumber**, **LookupErrorText**, **GetErrorText**, and **ResetError**. **OOracleObject** also supplies the **IsOpen** method.

OParameter

A parameter is an object that enables you to place variables within a SQL statement. The simplest use would be in the SQL statement that opens a dynaset:

```
select * from mytable where column = :pvalue
```

In this example *:pvalue* is a parameter. (In SQL syntax, a parameter name is prefixed with a colon.) When the SQL statement is used, the current value of the parameter is substituted for *:pvalue*. Such a parameter can be used wherever a literal value can be placed: values within update and insert statements and values that are part of where clauses.

Parameters are also used to represent arguments in calls to stored procedures. A stored procedure is a PL/SQL program that is stored in the Oracle database. Parameters can be used both as input and as output variables. See the example in ExecuteSQL for a sample of calling a stored procedure.

Parameters are managed as **OParameter** objects. **OParameter** objects are managed by way of an **OParameterCollection** that exists for every database object. You attach parameter objects to databases by using the **Add** method of the **OParameterCollection** class. **OParameter** is a subclass of **OOracleObject**.

The primary benefit of using **OParameter** is efficiency. The Oracle database knows how its records are to be fetched by remembering each SQL statement. When a precisely equal SQL statement is handed to the server, it can be processed quickly, because the database simply reuses the existing information. By using parameters instead of explicit values, you do not have to change the SQL statement when the database values change. Therefore, the SQL statement can be reused. This reuse is also convenient for you, because you also can modify the SQL statement without having to retain the entire string.

Parameters are attached to an **ODatabase** object. By default, whenever an **ODynaset** is opened or refreshed with a new SQL statement, all parameters that are attached to the parent **ODatabase** (the **ODatabase** on which the **ODynaset** is being opened) are bound to the **ODynaset**. A bound **OParameter** is said to be enabled. By default **OParameters** are autoenabled. This means that they can bind to any **ODynasets**. By using the **AutoEnable** method, you can turn this default behavior off and on. This can be useful if there are a large number of **OParameter** objects, which could result in many parameters being unnecessarily bound. Such a condition does not cause errors, but may be inefficient.

When the parameter is created (using the **OParameterCollection::Add** method) you need to specify whether the parameter is used for input, output or both. Use one of the following defines:

```
OPARAMETER_INVAR              // input variable
OPARAMETER_OUTVAR             // output variable
OPARAMETER_INOUTVAR           // input and output variable
```

You can query the status of a parameter, in which case the parameter returns a long that contains bits set to indicate the status. The bits are defined as:

```
OPARAMETER_STATUS_IN          // this is an input variable
```

OPARAMETER_STATUS_OUT	// this is an output variable
OPARAMETER_AUTOENABLED	// this parameter is bound automatically
OPARAMETER_ENABLED	// this parameter is ready to be bound

When the parameter is created you also need to specify the server type of the parameter.   The server type is one of the following Oracle types:

OTYPE_VARCHAR2
OTYPE_NUMBER
OTYPE_LONG
OTYPE_ROWID
OTYPE_DATE
OTYPE_RAW
OTYPE_LONGRAW
OTYPE_CHAR
OTYPE_MSLABEL

Please consult the *Oracle SQL Language Reference Manual* for more information about these types.   In general you can use OTYPE_VARCHAR2 for strings and OTYPE_NUMBER for numbers.

The **OParameter** class supports the following methods:

Construction and destruction:

OParameter
~OParameter
operator=

Attributes:

operator==
operator!=
GetName
GetServerType
GetStatus
IsOpen

Operations:

AutoEnable
Clear
Close
GetValue
operator const char *
operator double
operator int
operator long
SetValue

ErrorHandling:

ErrorNumber
ErrorReset
GetErrorText
LookupErrorText

OParameterCollection

An **OParameterCollection** object represents the set of parameters currently attached to a database. You can use an **OParameterCollection** object to add a parameter to a database by using the **Add** method. The collection object is dynamic: it always reflects the current state of the database, which is not necessarily the state of the database at the time you got the **OParameterCollection**. **OParameterCollection** is a subclass of **OOracleObject**.

You get an **OParameterCollection** instance from an **ODatabase** instance by using the **GetParameters** method. Use the **Add** method to add parameters and the **Remove** method to remove parameters.

The **OParameterCollection** class supports the following methods:

Construction and destruction:

OParameterCollection
~OParameterCollection
operator=

Attributes:

operator==
operator!=
GetCount
IsOpen

Operations:

Add
Close
GetParameter
Remove

ErrorHandling:

ErrorNumber
ErrorReset
GetErrorText
LookupErrorText

OSession

The **OSession** object controls the behavior of an entire database session. You can access more than one database in a session. Typically an application has a single session, but applications can have more. A session is the unit on which transactions occur.

Please see Transactions for more complete information on transactions.

Database objects share the connection to the database if they are created on the same session (by using the same session as an argument in the **ODatabase::Open** method), and if they have the same connection information (database name, username, and password).

You can get **OSession** objects from **OConnection**, **ODatabase**, **ODynaset**, or **OSessionCollection** objects using the **GetSession** method.

Normally you will use the *default session*, for which you provide no name. You can also open sessions with specific names. Later you can get a list of sessions from the client object and identify sessions by name.

The **OSession** class supports the following methods:

Construction and destruction:

- OSession
- ~OSession
- operator=

Attributes:

- operator==
- operator!=
- GetName
- GetVersion
- IsOpen

Operations:

- BeginTransaction
- Close
- Commit
- GetClient
- GetConnections
- GetNamedSession
- Open
- ResetTransaction
- Rollback

ErrorHandling:

- ErrorNumber
- ErrorReset
- GetErrorText
- GetServerErrorText
- LookupErrorText
- ServerErrorNumber
- ServerErrorReset

OSessionCollection

The **OSessionCollection** object is the set of sessions that exists for a given client. You can get **OSessionCollection** objects from an **OClient** object using the **GetSessions** method. **OSessionCollection** is a subclass of **OOracleCollection**.

The **OSessionCollection** is dynamic: it always reflects the current set of sessions of the client it came from, even if the sessions are added after you get the collection.   It is read-only.   You cannot add sessions through the collection.

You can obtain individual **OSessions** from an **OSessionCollection** using the **GetSession** method.

The **OSessionCollection** class supports the following methods:

Construction and destruction:

OSessionCollection
~OSessionCollection
operator=

Attributes:

operator==
operator!=
GetCount
IsOpen

Operations:

Close
GetSession

ErrorHandling:

ErrorNumber
ErrorReset
GetErrorText
LookupErrorText

OValue

An **OValue** object is a convenient place to store data of a variety of different types. It is also useful as a way to convert between different types.   You can use **OValues** to get and set field values in a type-independent fashion.   **OValue** is a subclass of **OOracleObject**.

OValue has a rich set of constructors and **SetValue** methods to set values using different types and cast operators to obtain the value from an **OValue** instance.   The **Clear** method sets the value to a database NULL value (which is not the same as a C++ NULL).   The **IsNull** method checks to see whether the current value is NULL.

The **OValue** class supports the following methods:

Construction and destruction:

OValue
~OValue
operator=

Attributes:

operator==
operator!=
IsNull
IsOpen

Operations:

operator const char *
operator double
operator int
operator long
Clear
SetValue

Methods

Close

ActionNotify
ActionRequest
Add
AddNewRecord
AppendChunk
AppendFieldChunk
AutoEnable
BeginTransaction
BindToBinder
CancelEdit
CanMark
CanRefresh
CanScroll
CanTransact
CanUpdate
Changed
Clear
Clone
Close
Close(OBinder)
Commit
DeleteRecord
DiscardChanges
DuplicateRecord
ErrorNumber
ErrorReset
ExecuteSQL
GetChangedError
GetChunk
GetClient
GetConnection
GetConnection(Collection)
GetConnections
GetConnectString
GetCount
GetDatabase
GetDatabaseName
GetDynaset
GetEditMode
GetErrorText
GetField
GetFieldChunk
GetFieldCount
GetFieldIndex
GetFieldPrecision
GetFields
GetFieldScale
GetFieldServerSize
GetFieldServerType
GetFieldSize
GetMark
GetName
GetNamedSession
GetOptions
GetParameter
GetParameters
GetPrecision
GetRecordCount
GetRowsProcessed
GetScale
GetServerErrorText
GetServerSize
GetServerType
GetSession
GetSession(Collection)
GetSessions
GetSize
GetSQL
GetStatus
GetValue
GetValue(OBound)
IsBOF
IsChanged
IsEOF
IsFieldNullOK
IsFieldTruncated
IsFirst
IsLast
IsNull
IsNullOK
IsOpen
IsTruncated
IsValidRecord
LookupErrorText
MoveFirst
MoveLast
MoveNext
MovePrev
MoveToMark
OAdvise
~OAdvise
OBinder
~OBinder
OBound
~OBound
OClient
~OClient
OConnection
~OConnection
OConnectionCollection
ODynaset
~ODynaset
ODynasetMark
~ODynasetMark
OField
~OField
OFieldCollection
~OFieldCollection
OnChangedError
OParameter
~OParameter
OParameterCollection
~OParameterCollection
Open(OAdvise)
Open(OBinder)
Open(ODatabase)
Open(ODynaset)
Open(OSession)
operator const char *
operator double
operator int
operator long
operator=
operator==
operator!=
OSession
~OSession
OSessionCollection
~OSessionCollection
OShutdown
OStartup
OValue
~OValue
Refresh(OBinder)
Refresh(OBound)
Refresh(ODynaset)
RefreshQuery
Remove
ResetTransaction
Rollback
SaveChange
ServerErrorNumber
ServerErrorReset
ServerErrorSQLPos
SetFieldValue
SetSQL
SetValue
SetValue(OBound)
StartEdit
StatusChange

GetFieldValue
GetLastModifiedMark

~OConnectionCollection
ODatabase
~ODatabase

Unbind
UnbindObj
Update

Trigger Methods

Close

PostAdd
PostDelete
PostMove
PostQuery
PostRollback
PostUpdate
PreAdd

PreDelete
PreMove
PreQuery
PreRollback
PreUpdate
Shutdown
Startup

Methods

ActionNotify
ActionRequest
Add
AddNewRecord
AppendChunk
AppendFieldChunk
AutoEnable
BeginTransaction
BindToBinder
CancelEdit
CanMark
CanRefresh
CanScroll
CanTransact
CanUpdate
Changed
Clear
Clone
Close
Close(OBinder)
Commit
DeleteRecord
DiscardChanges
DuplicateRecord
ErrorNumber
ErrorReset
ExecuteSQL
GetChangedError
GetChunk
GetClient
GetConnection
GetConnection(Collection)
GetConnections
GetConnectString
GetCount
GetDatabase
GetDatabaseName
GetDynaset
GetEditMode
GetErrorText
GetField
GetFieldChunk
GetFieldCount
GetFieldIndex
GetFieldPrecision
GetFields
GetFieldScale
GetFieldServerSize
GetFieldServerType
GetFieldSize
GetFieldValue
GetLastModifiedMark
GetMark
GetName
GetNamedSession
GetOptions
GetParameter
GetParameters
GetPrecision
GetRecordCount
GetRowsProcessed
GetScale
GetServerErrorText
GetServerSize
GetServerType
GetSession
GetSession(Collection)
GetSessions
GetSize
GetSQL
GetStatus
GetValue
GetValue(OBound)
IsBOF
IsChanged
IsEOF
IsFieldNullOK
IsFieldTruncated
IsFirst
IsLast
IsNull
IsNullOK
IsOpen
IsTruncated
IsValidRecord
LookupErrorText
MoveFirst
MoveLast
MoveNext
MovePrev
MoveToMark
OAdvise
~OAdvise
OBinder
~OBinder
OBound
~OBound
OClient
~OClient
OConnection
~OConnection
OConnectionCollection
~OConnectionCollection
ODatabase
ODynaset
~ODynaset
ODynasetMark
~ODynasetMark
OField
~OField
OFieldCollection
~OFieldCollection
OnChangedError
OParameter
~OParameter
OParameterCollection
~OParameterCollection
Open(OAdvise)
Open(OBinder)
Open(ODatabase)
Open(ODynaset)
Open(OSession)
operator const char *
operator double
operator int
operator long
operator=
operator==
operator!=
OSession
~OSession
OSessionCollection
~OSessionCollection
OShutdown
OStartup
OValue
~OValue
Refresh(OBinder)
Refresh(OBound)
Refresh(ODynaset)
RefreshQuery
Remove
ResetTransaction
Rollback
SaveChange
ServerErrorNumber
ServerErrorReset
ServerErrorSQLPos
SetFieldValue
SetSQL
SetValue
SetValue(OBound)
StartEdit
StatusChange
Unbind
UnbindObj

Trigger Methods

- PostAdd
- PostDelete
- PostMove
- PostQuery
- PostRollback
- PostUpdate
- PreAdd
- PreDelete
- PreMove
- PreQuery
- PreRollback
- PreUpdate
- Shutdown
- Startup

ActionNotify Method

Applies To

OAdvise

Description

The **ActionNotify** method is called by a dynaset when that dynaset has performed an operation. You do not call **ActionNotify**; the **ActionNotify** method of your **OAdvise** subclass is called by the dynaset.

Usage

void **ActionNotify(**int *actiontype*)

Arguments

actiontype

actiontype will have one of the following values:

OADVISE_MOVE_FIRST	// dynaset moving to first record
OADVISE_MOVE_PREV	// dynaset moving to previous record
OADVISE_MOVE_NEXT	// dynaset moving to next record
OADVISE_MOVE_LAST	// dynaset moving to last record
OADVISE_MOVE_TOMARK	// dynaset moving to dynaset mark
OADVISE_REFRESH	// dynaset refreshing
OADVISE_DELETE	// dynaset deleting current record
OADVISE_ADDNEW	// dynaset adding a new record
OADVISE_UPDATE	// dynaset updating
OADVISE_ROLLBACK	// session (that dynaset is part of) is rolling back

Remarks

When you subclass **OAdvise**, you can override the **ActionNotify** method. After an instance of your **OAdvise** subclass is attached to a dynaset (by way of the **OAdvise::Open** method) your instance receives calls to its **ActionNotify** method. Use an **ActionNotify** method to perform processing after a dynaset has performed an action.

The unoverridden **ActionNotify** method of **OAdvise** does nothing.

Example

This example puts up a notification dialog whenever a record is updated.

```
void YourOAdvise::ActionRequest(int actiontype)
{
    if (actiontype == OADVISE_UPDATE)
    {
        NotifyDialog("Record has been updated");
    }
    return;
}
```

ActionRequest Method

Applies To

OAdvise

Description

The **ActionRequest** method is called by a dynaset when that dynaset is about to start an operation. You do not call **ActionRequest**; the **ActionRequest** method of your **OAdvise** subclass is called by the dynaset.

Usage

oboolean **ActionRequest**(int *actiontype*)

Arguments

actiontype

actiontype will have one of the following values:

OADVISE_MOVE_FIRST	// dynaset moving to first record
OADVISE_MOVE_PREV	// dynaset moving to previous record
OADVISE_MOVE_NEXT	// dynaset moving to next record
OADVISE_MOVE_LAST	// dynaset moving to last record
OADVISE_MOVE_TOMARK	// dynaset moving to dynaset mark
OADVISE_REFRESH	// dynaset refreshing
OADVISE_DELETE	// dynaset deleting current record
OADVISE_ADDNEW	// dynaset adding a new record
OADVISE_UPDATE	// dynaset updating
OADVISE_ROLLBACK	// session (that dynaset is part of) is rolling back

Remarks

When you subclass **OAdvise,** you can override the **ActionRequest** method. After an instance of your **OAdvise** subclass is attached to a dynaset (by way of the **OAdvise::Open** method), your instance receives calls to its **ActionRequest** method. Use an **ActionRequest** method to control whether certain dynaset actions should be allowed to proceed, or to do your own processing before dynaset actions occur.

The unoverridden **ActionRequest** method of **OAdvise** always returns TRUE, allowing all dynaset operations to proceed immediately.

Return value

TRUE	tells the dynaset that the action can proceed
FALSE	tells the dynaset to cancel the action

Example

This example attempts to save a change to the current record if there is one. This is an action that needs to be taken before the dynaset does an operation.

```
oboolean YourOAdvise::ActionRequest(int actiontype)
{
    // check whether we have an unsaved change in this record
    int error;
    if (m_havechange)
    { // we have a change, try to save it
        error = m_context->SaveTheChange();
        if (error != 0)
        { // some problem - cancel the action
            return(FALSE);
        }
    }

    // everything is fine, allow the action
    return(TRUE);
}
```

Add Method

Applies To

OParameterCollection

Description

This method adds a parameter to a database.

Usage

OParameter **Add**(const char **name*, const char **value*, int *iotype*, int *servertype*)

OParameter **Add**(const char **name*, double *value*, int *iotype*, int *servertype*)

OParameter **Add**(const char **name*, int *value*, int *iotype*, int *servertype*)

OParameter **Add**(const char **name*, long *value*, int *iotype*, int *servertype*)

Arguments

name	the name to be given to the parameter
value	the initial value of the parameter
iotype	specifies whether this parameter is an input variable, an output variable, or both. The value should be one of:

```
OPARAMETER_INVAR           // in parameter
OPARAMETER_OUTVAR          // out parameter
        OPARAMETER_INOUTVAR      // in/out parameter
```

*servertype*the Oracle type of the parameter.   The value should be one of:

```
OTYPE_VARCHAR2
OTYPE_NUMBER
OTYPE_LONG
OTYPE_ROWID
OTYPE_DATE
OTYPE_RAW
OTYPE_LONGRAW
OTYPE_CHAR
OTYPE_MSLABEL
```

Remarks

These methods attach an **OParameter** to an **ODatabase**. The *name* argument specifies the name of the parameter.   To refer to the value of the **OParameter** within SQL statements, use *:name*. The *value* argument is the initial value of the **OParameter**. The data type of the parameter is set to be the type of the initial value.

The parameter that is created is referenced by the returned **OParameter** object.

Return Value

An **OParameter**, which will be open on success, closed on failure.

Example

This example adds a parameter to an existing **ODatabase** (odb) and uses it to open an **ODynaset**:

```
// open an ODatabase called odb
ODatabase odb("ExampleDB", "scott", "tiger");
```

```
// now add a parameter named ourdeptno to the database
OParameterCollection params = odb.GetParameters();
params.Add("ourdeptno ", 20, OPARAMETER_INVAR, OTYPE_NUMBER);

// now create and open a dynaset using that parameter
ODynaset dyn(odb, "select * from emp where deptno = :ourdeptno");
```

AddNewRecord Method

Applies To

OBinder, ODynaset

Description

This method adds a new record to the dynaset result set.

Usage

oresult **AddNewRecord**(void)

Remarks

This method adds a new record to the dynaset. **OBinder::AddNewRecord** adds a new record to the dynaset that it is managing. The added record becomes the current record. Execution of this method sends OADVISE_ADDNEW messages to all attached advisories.

To add a record to a dynaset, first call **ODynaset::AddNewRecord**, then set the values of whatever fields you wish, then call **ODynaset::Update**.

To add a record to a managed dynaset (an **OBinder** instance) call **AddNewRecord**. The **OBinder** instance takes care of the rest.

Depending on the options that were used to create the database to which this dynaset is attached, the Oracle database may or may not be called to fill values into some of the fields. See the ODATABASE_PARTIAL_INSERT under **ODatabase**.

Note: A call to **StartEdit**, **AddNewRecord**, **DuplicateRecord**, or **DeleteRecord**, will cancel any outstanding **StartEdit**, **AddNewRecord** or **DuplicateRecord** calls before proceeding. Any outstanding changes not saved using **Update** will be lost during the cancellation.

When used with **OBinder**, the method also calls the **PreAdd** and **PostAdd** triggers.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example adds a new record to the emp table. We assume an already open dynaset named empdyn.

```
// add a record to empdyn
empdyn.AddNewRecord();

// set the values of important fields
empdyn.SetFieldValue("empno", 4512);
empdyn.SetFieldValue("ename", "Scott Feline");
empdyn.SetFieldValue("sal", 1000.45);

// save the change to the Oracle database
empdyn.Update();
```

AppendChunk Method

Applies To

OField

Description

This method appends data to a long or long raw field.

Usage

oresult **AppendChunk**(const void **chunkp*, unsigned short *numbytes*)

Arguments

chunkp	a pointer to the data to be appended
numbytes	the number of bytes from *chunkp* to be appended

Remarks

Long and long raw columns in an Oracle7 database can hold large amounts of data. The **AppendChunk** method enables you to add data to a long field piecewise. You can add a maximum of 64K at once. Each call to **AppendChunk** adds data following any other calls. **AppendChunk** calls must be made while the dynaset is in an editing mode started by either **StartEdit**, **AddNewRecord**, or **DuplicateRecord**. This method is valid only for fields with server types of OTYPE_LONG or OTYPE_LONGRAW.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example adds take an image segmented into several chunks and stores it in the database.

```
// we have an open dynaset pictdyn
// get an OField on the "picture" field of the dynaset
OField pictfield = pictdyn.GetField();

// now start editing
pictdyn.StartEdit();

// and loop through the chunks of the picture
int ii;
for (ii=0; ii<nchunks; ii++)
    pictfield.AppendChunk((const void *) picture[ii], plen[ii]);

// and save it to the database
pictdyn.Update();
```

AppendFieldChunk Method

Applies To

ODynaset

Description

This method appends data to a long or long raw field.

Usage

oresult **AppendFieldChunk**(int *index*, const void **chunkp*, unsigned short *numbytes*)

oresult **AppendFieldChunk**(const char **fieldname*, const void **chunkp*,

unsigned short *numbytes*)

Arguments

index	the 0-based index of the field. The index is the position of the field in the SQL query that created the current record set.
fieldname	the name of the field, as expressed in the SQL query
chunkp	a pointer to the data to be appended
numbytes	the number of bytes from *chunkp* to be appended

Remarks

Long and long raw columns in an Oracle7 database can hold large amounts of data. The **AppendFieldChunk** method enables you to add data to a long field piecewise. You can add a maximum of 64K at once. Each call to **AppendFieldChunk** adds data following any other calls. **AppendFieldChunk** calls must be made while the dynaset is in an editing mode started by either **StartEdit**, **AddNewRecord**, or **DuplicateRecord**. This method is valid only for fields with server types of OTYPE_LONG or OTYPE_LONGRAW.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example adds take an image segmented into several chunks and stores it in the database.

```
// we have an open dynaset pictdyn

// now start editing
pictdyn.StartEdit();

// and loop through the chunks of the picture
int ii;
for (ii=0; ii<nchunks; ii++)
    pictdyn.AppendFieldChunk("pfield", (const void *) picture[ii],
plen[ii]);

// and save it to the database
pictdyn.Update();
```

AutoEnable Method

Applies To

OParameter

Description

This method turns the **AutoEnable** status for a parameter on or off.

Usage

oresult **AutoEnable**(oboolean *enable*)

Arguments

enable	If TRUE, enables the parameter. If FALSE, disables the parameter.

Remarks

Parameters can have autoenabling turned on or off. Only parameters that have autoenabling turned on can be bound to newly executed SQL statements. Disabling a parameter does not change its use in an already executed SQL statement.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example adds, enables, and disables a parameter.

```
// add a parameter to an existing ODatabase: odb
OParameterCollection params = odb.GetParameters();
OParameter deptp;

// initial value of 20
deptp = params.Add("dno", 20, OPARAMETER_INVAR, OTYPE_NUMBER);

// disable it
deptp.AutoEnable(FALSE);

// try to use it
ODynaset empdyn;
oresult ores;
ores = empdyn.Open(odb, "select * from emp where deptno = :dno");
// that failed: ores == OFAILURE

// enable the parameter and try again
deptp.AutoEnable(TRUE);
ores = empdyn.Open(odb, "select * from emp where deptno = :dno");
// now ores == OSUCCESS
```

BeginTransaction Method

Applies To

OSession

Description

This method begins a database transaction.

Usage

oresult **BeginTransaction**(void)

Remarks

A database transaction is a way to group database operations so that they all either succeed or fail together. Please see *Transactions* for more information. You start a transaction with **BeginTransaction**. You terminate a transaction either with a **Commit** or a **Rollback**. It is an error to call **BeginTransaction** when a transaction is already in progress.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example starts a transaction and begins a long sequence of operations. If an error occurs along the way, all the changes are discarded with a **Rollback**. If they all succeed, all the changes are made permanent with a **Commit**.

```
// routine to give all employees the same bonus
void Transfer(ODynaset empdyn, double bonus)
{
    // get the session of this dynaset
    OSession empsess = empdyn.GetSession();

    // start a transaction
    empsess.BeginTransaction();

    // edit every record (with StartEdit, SetFieldValue, Update)
    empdyn.MoveFirst();
    while (!empdyn.IsEOF())
    {
        if (empdyn.StartEdit() != OSUCCESS)
            break;
        if (empdyn.SetFieldValue("bonus", bonus) != OSUCCESS)
            break;
        if (empdyn.Update() != OSUCCESS)
            break;

        empdyn.MoveNext();  // go to the next record
    }

    if (!empdyn.IsEOF())
    { // we got out of the loop early.  Get rid of any changes we made
        empsess.Rollback();
    }
    else
    { // everything worked.  Make it all permanent
```

```
        empsess.Commit();
    }
    return;
}
```

BindToBinder Method

Applies To

OBound

Description

This method binds the **OBound** object to a particular **OBinder** object.

Usage

oresult **BindToBinder**(OBinder **binder*, const char **fieldname*)

Arguments

binder	A pointer to the **OBinder** object to which this **OBound** should be bound.
fieldname	The name of the field in the query to which this **OBound** should be bound.

Remarks

Before an **OBound** object is associated with an **OBinder** (a managed dynaset), it is inert. **BindToBinder** sets up the relationship between the **OBound** object and the **OBinder** object that makes the **OBound** machinery work.

This method calls the **Startup** trigger for the **OBound** object and the **OBinder Startup** trigger if this is the first object to be bound.

If the **OBinder** object is already open, **BindToBinder** verifies that the *fieldname* is valid and returns OFAILURE if the *fieldname* does not belong to the bound dynaset. However, your Bound Controls will not receive their values until they are refreshed, either by using the **OBinder::Refresh** method or one of the Move methods.(such as **MoveFirst**) It is also possible to bind all the **OBound** objects before opening the **OBinder**. In this situation, **OBinder Open** fails if there are any **OBound** objects referring to an invalid *fieldname.*

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example does all the work needed to set up several **OBoundVal** objects that will be bound to several fields in the emp table. **OBoundVal** is a subclass of **OBound** that is discussed in the *Workbook*.

```
// open a database object
ODatabase odb("ExampleDB", "scott", "tiger");

// open the managed dynaset
OBinder empblock;
empblock.Open(odb, "select ename, sal, empno from emp");

// declare some OBoundVal objects
OBoundVal ename;
OBoundVal sal;
OBoundVal empno;

// bind the OBoundVals
ename.BindToBinder(&empblock, "ename");
sal.BindToBinder(&empblock, "sal");
empno.BindToBinder(&empbloc, "empno");
```

```
// now we're all set up
```

CancelEdit Method

Applies To

ODynaset

Description

This method cancels the edits made to the current record.

Usage

oboolean **CancelEdit**(void)

Remarks

Editing of the current record is begun with the **StartEdit** method. If you have called **StartEdit** and then decide to discard any changes you have made, call **CancelEdit**.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example gives everybody a raise, except for managers.

```
// open a database object
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset on the employee table
ODynaset empdyn(odb, "select * from emp");

OField salf = empdyn.GetField("sal"); // the salary
OField jobf = empdyn.GetField("job"); // the job

// go through all the records
empdyn.MoveFirst();
while (!empdyn.IsEOF())
{
    // start editing
    empdyn.StartEdit();
    // give a raise
    salf.SetValue(1000.0 + (double) salf);

    // wait a minute, what position does this person have?
    if (0 == strcmp("MANAGER", (const char *) jobf))
    { // forget the raise!
        empdyn.CancelEdit();
    }
    else
    { // go ahead and save the raise
        empdyn.Update();
    }

    empdyn.MoveNext();
}
```

CanMark Method

Applies To

ODynaset

Description

This method returns TRUE if you can obtain marks on this dynaset.

Usage

oboolean **CanMark**(void) const

Remarks

Normal dynasets support the **GetMark** and **GetLastModifiedMark** methods. However, dynasets created with the ODYNASET_NOCACHE option do not. Use this method to determine whether you can get dynaset marks on this dynaset.

Return Value

TRUE if you can obtain a mark on this dynaset; FALSE if not.

CanRefresh Method

Applies To

ODynaset

Description

This method returns TRUE if you can call the **Refresh** method on this **ODynaset**.

Usage

oboolean **CanRefresh**(void) const

Remarks

In this release, the method always returns TRUE.

Return Value

TRUE if the dynaset is refreshable; FALSE if it is not.

CanScroll Method

Applies To

ODynaset

Description

This method returns TRUE if you can move backwards and navigate to a mark.

Usage

oboolean **CanScroll**(void) const

Remarks

ODynasets that return FALSE on **CanScroll** do not maintain a local record cache; they keep the data only for the current record.   By default dynasets are created with local record caches.   Only dynasets created with the ODYNASET_NOCACHE option are not scrollable. Use this method to determine whether the dynaset is scrollable.

Return Value

TRUE if the dynaset can scroll backwards; FALSE if not.

CanTransact Method

Applies To

ODynaset

Description

This method returns TRUE if the session to which this **ODynaset** is attached supports transactions.

Usage

oboolean **CanTransact**(void) const

Remarks

See **OSession** for more information about transactions.   In the current release, all sessions support transactions.

Return Value

TRUE if the dynaset's session supports transactions; FALSE if not.

CanUpdate Method

Applies To

ODynaset

Description

This method returns TRUE if the **ODynaset** is updatable.

Usage

oboolean **CanUpdate**(void) const

Remarks

Depending on the SQL statement that is used to Open or Refresh the **ODynaset**, it may or may not be possible to make changes to the dynaset. Several factors can make a dynaset non-updatable:

- The SQL statement may contain fields that are not database table columns (for example, computed fields or fields whose names have been aliased).
- The SQL statement may reference columns in more than one database field (a join).
- The user may not have privileges to update the table the dynaset is on.
- The dynaset may have been opened with the ODYNASET_READONLY option.

Note that even if the SQL statement references only a single view, the dynaset will not be updatable if that view is created by way of a join, computed columns, or aliased columns.

Return Value

TRUE if the dynaset is updatable; FALSE if not.

Example

Here are examples of dynaset updatability.

```
// we assume the existence of an open ODatabase named odb
ODynaset dyn;  // construct an ODynaset

// now open the ODynaset with various SQL statements
dyn.Open(odb, "select * from emp");
if (dyn.CanUpdate()) ; // is TRUE

dyn.Open(odb, "select sal*1.1 from emp");
if (dyn.CanUpdate()) ; // FALSE because of computed field

dyn.Open(odb, "select emp.ename, dept.dname from emp, dept \
               where emp.deptno = dept.deptno");
if (dyn.CanUpdate()) ; // FALSE because of join
```

Changed Method

Applies To

OBound, **OBinder**

Description

This method tells the object that the value for the current record has (or has not) been changed. Normally only **OBound** or **OBinder** subclasses call this method.

Usage

oresult **Changed**(oboolean *changed* = TRUE)

Arguments

changed	if TRUE (the default), tells the object that it has been changed
	if FALSE, tells the object that it has not been changed (rarely used)

Remarks

When you subclass **OBound** and **OBinder**, you generally create objects that allow field values to be edited in some convenient fashion. In order for **OBinder** and **OBound** to know when to update the database with edited field values, they need to know when values have been changed. Calling this method is how you inform **OBinder** and **OBound** objects that a change has occurred.

Normally the **OBound** subclass calls **Changed** in some routine that notices that the value has been changed. For example, where the **OBound** subclass is a text editing control, **Changed** should be called when the user enters a key in the control. Users of **OBound** and **OBinder** classes and subclasses normally do not call **Changed**.

Calling **Changed** on an **OBound** means that the field value of that **OBound** has been changed in the current record. Calling **Changed** on an **OBinder** means that the current record has been changed.

Marking an **OBound** as changed means that eventually the **OBound**'s **SaveChange** method will be called to save the change to the database.

The first time that something in the current record is marked as changed, either some **OBound** in the record or the whole record, **StartEdit** is called on the dynaset being managed by the **OBinder**. This can fail, for all the reasons **StartEdit** might fail.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE). Failure typically means that the dynaset that **OBinder** is managing is not updatable.

Example

Please see the *Workbook* for the example "OBound of a variable." That example works through the implementation of a subclass of **OBound**. That subclass calls **Changed**.

Clear Method

Applies To

OValue, OParameter

Description

This method clears the value stored in the object, leaving a value of NULL.

Usage

oresult **Clear**(void)

Remarks

Oracle database columns may have a value of NULL.   This database NULL means "no value specified", rather than the NULL pointer of C++.   Sometimes it is desirable to generate a NULL value.   The **Clear** method makes the **OValue** or **OParameter** object hold a NULL value.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example gets a value from a dynaset and then sets it to NULL.

```
// we have a dynaset named empdyn

// get the current commission value
OValue commission;
empdyn.GetFieldValue("comm", &commission);

// if the commission has a value, change it to NULL
if (!commission.IsNull())
{ // make the commission NULL
    commission.Clear();
    empdyn.StartEdit();
    empdyn.SetFieldValue("comm", commission);
    empdyn.Update();
}
```

Clone Method

Applies To

ODynaset

Description

This method returns a clone of the **ODynaset**.

Usage

ODynaset **Clone**(void)

Remarks

A clone of a parent dynaset is a read-only dynaset that refers to the same local data cache as the parent. It is possible to navigate through the clone dynaset, and read from the clone dynaset, without having any side effects on the parent dynaset. If a dynaset has advisories attached to it or is the dynaset of an **OBinder** object (which has advisories placed by the **OBinder**), moving from record to record could be expensive and can even be canceled by an advisory. Cloning the dynaset allows you to look at the data without these side effects.

An **ODynaset** and its clone can share dynaset marks.

Note that two separate **ODynaset** objects can refer to the same underlying dynaset. For example, one dynaset can be assigned (using the = operator) to the other. Then, changing the current record in one changes the current record in the other: the current record is changed in the dynaset and the **ODynasets** are just handles to the dynaset. A **Clone** is an actual different underlying dynaset, but it observes the same data cache as another dynaset.

Return Value

Returns an **ODynaset**. If the returned **ODynaset** is Open (check with the **IsOpen** method), the **Clone** was successful. If the returned **ODynaset** is not Open, the operation failed.

Example

This example creates several ODynaset objects to illustrate relationships between them.

```
// open the database
ODatabase odb("ExampleDB", "scott/tiger", 0);

// Create some ODynasets.

// ODynaset dyn1 and dyn2 are completely separate
ODynaset dyn1(odb, "select * from emp");
dyn1.MoveFirst();

ODynaset dyn2(odb, "select * from emp");

// ODynaset dyn1copy looks at the same dynaset as dyn1
ODynaset dyn1copy;
dyn1copy = dyn1;

// ODynaset dyn1clone is a clone of dyn1 ...
ODynaset dyn1clone;
// ...which is the same as dyn1copy.Clone()
```

```
dyn1clone = dyn1.Clone();

// now dyn1, dyn1copy and dyn1clone are all at the first record
// so now...
dyn1.MoveLast();
// ...dyn1copy is at its last record
// ...but dyn1clone is still at the first

// if we add a record...
dyn1.AddRecord();
/*
Dyn1copy is now current on the new record along with dyn1. dyn1clone is
still at the first record.
*/

dyn1.Update(); // add that new record
/*
If we navigate around in dyn1clone we will see the new record, but
navigating around in dyn2 we will not see the new record until dyn2 is
refreshed.
*/
```

Close Method

Applies To

OAdvise, **OClient**, **OConnection**, **OConnectionCollection**, **ODatabase**, **ODynaset**, **ODynasetMark**, **OField**, **OFieldCollection**, **OParameter**, **OParameterCollection**, **OSession**, **OSessionCollection**

Description

This method closes the object, freeing the connection with the underlying implementation object.

Usage

oresult **Close**(void)

Remarks

All objects that are instances of subclasses of **OOracleObject** are handles that reference some implementations object. This architecture allows the class objects to be very lightweight and easy to use. The relationship between an object and its implementation is established when the object is opened, either explicitly with an **Open** method or implicitly with some constructors or the assignment operator. Normally the relationship between the object and its implementation is dropped when the object is destroyed. It can also be dropped if the object is reopened. It can also be dropped explicitly by using the **Close** method.

Once all handles for an implementation object are closed (or destroyed), the implementation object will be destroyed. Because certain objects consume considerable resource when they are open (databases consume connections and dynasets consume a great deal for their local data cache), it is sometimes desirable to close handle objects (with the **Close** method) before they are destroyed. Use **Close** also if the **OOracleObject** is be a member of some class that you do not want to destroy, but you do want to free the implementation object.

Closed objects generally fail all operations, with the obvious exception of **Open**.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example demonstrates when a database object is released.

```
// open an ODatabase.  This creates an underlying database object
ODatabase odb("ExampleDB", "scott", "tiger");

// now we can do operations with this database object
odb.ExecuteSQL("drop table dontwantit");

// get another handle (ODatabase) on the same database
ODatabase odb2 = odb;

// now we close odb
odb.Close();  // that works

// try to use the database object (with a closed ODatabase)
odb.ExecuteSQL("drop table fooey");  // fails because odb is Closed
```

```
// now use an open ODatabase on the database object
//    that odb referenced
odb2.ExecuteSQL("drop table fooey");
// that succeeded because odb2 is Open

// now close odb2
odb2.Close();
// that dropped odb2.  Now the database connection is dropped because
//    now all the handles on the database object have been closed.
```

Close (OBinder) Method

Applies To

OBinder

Description

This method closes the object.

Usage

oresult **Close**(oboolean *doShutdown* = TRUE)

Arguments

doShutdown	if TRUE (the default), the **Shutdown** triggers are called
	if FALSE, the **Shutdown** triggers are not called

Remarks

Closing an **OBinder** object frees all its resources and makes it unusable (until it is reopened). You might want to do this if the **OBinder** is a member variable of an object that you do not want to destroy yet, but you do not need the **OBinder** anymore.

Normally when you close an **OBinder** object you should let *doShutdown* be TRUE and call the **Shutdown** triggers (both the **OBinder** trigger and the triggers for all **OBound** objects bound to this **OBinder**). Depending on the circumstances and precisely what your **OBound** objects do on the **Shutdown** trigger, this can fail. Calling **Close** with *doShutdown* FALSE will not call the **Shutdown** triggers, but it also does not fail.

Note that once the **OBinder** object has been **Close**d, all of the **OBound** objects that were bound to the **OBinder** are unbound - just as if each had called the **OBound::Unbind** method.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example shows the closing of an OBinder

```
// we have an OBinder object called empblock

// bind an OBoundVal object to it (see the Workbook)
OBoundVal sal;
sal.BindToBinder(&empblock, "sal");

// now close the OBinder
empblock.Close();
// that called OBoundVal.Shutdown()

salary = (int) sal;
// salary is always 0 now because sal is unbound

ODynaset empdyn = empblock.GetDynaset();
// empdyn is closed and empdyn.IsOpen() is FALSE,
//    because empblock is Closed
```

Commit Method

Applies To

[OSession]

Description

This method commits the current transaction.

Usage

oresult **Commit**(oboolean *startnew* = FALSE)

Arguments

startnew	If TRUE a new transaction is begun (as if **BeginTransaction** had been called). If FALSE, no additional work is done after the transaction is committed.

Remarks

A database transaction is a way to group database operations so that they all either succeed or fail together.   Please see *Transactions* for more details.   You start a transaction with **BeginTransaction.**   You terminate the transaction either with a **Commit** or a **Rollback**.   It is an error to call **Commit** when no transaction is in progress.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example starts a transaction and begins a long sequence of operations.   If an error occurs along the way, all the changes are discarded with a **Rollback**.   If they all succeed, all the changes are made permanent with a **Commit**.

```
// routine to give all employees the same salary
void Transfer(ODynaset empdyn, double salary)
{
    // get the session of this dynaset
    OSession empsess = empdyn.GetSession();

    // start a transaction
    empsess.BeginTransaction();

    // edit every record (with StartEdit, SetFieldValue, Update)
    empdyn.MoveFirst();
    while (!empdyn.IsEOF())
    {
        if (empdyn.StartEdit() != OSUCCESS)
            break;
        if (empdyn.SetFieldValue("sal", salary) != OSUCCESS)
            break;
        if (empdyn.Update() != OSUCCESS)
            break;

        empdyn.MoveNext();  // go to the next record
    }

    if (!empdyn.IsEOF())
    { // we got out of the loop early.  There must be a problem.
```

```
      //   Get rid of any changes we made
        empsess.Rollback();
    }
    else
    { // everything worked, so make it all permanent
        empsess.Commit();
    }
    return;
}
```

DeleteRecord Method

Applies To

OBinder, ODynaset

Description

This method deletes the current record from the dynaset result set.

Usage

oresult **DeleteRecord**(void)

Remarks

This method deletes the current record from the dynaset. Execution of this method sends OADVISE_DELETE messages to all attached advisories.   It is not necessary to call **StartEdit** or **Update** to delete a record.

After the record has been deleted, the current record will not be valid (it has been deleted).   If the deletion was from a dynaset, you must navigate yourself to a valid record.   The **OBinder** class will attempt to move to an adjacent valid record by itself.

Note: A call to **StartEdit**, **AddNewRecord**, **DuplicateRecord**, or **DeleteRecord**, will cancel any outstanding **StartEdit**, **AddNewRecord** or **DuplicateRecord** calls before proceeding.   Any outstanding changes not saved using **Update** will be lost during the cancellation.

OBinder calls the **PreDelete** and **PostDelete** triggers when this method is called.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example deletes all managers.

```
// open the database
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset on the employee table
ODynaset empdyn(odb, "select * from emp");

// get an OField object for looking at the job field
OField job = empdyn.GetField("job");

// look through all the employees
while (!empdyn.IsEOF())
{
    if (0 == strcmp((const char *) job, "MANAGER"))
    { // we found a manager; delete that employee
        empdyn.DeleteRecord();
    }

    // go to next record (gets us to valid record)
    //    or past EOF if there are no more records
    empdyn.MoveNext();
}
```

DiscardChanges Method

Applies To

OBinder

Description

This method discards the changes made to the current record.

Usage

oresult **DiscardChanges**(void)

Remarks

OBound subclass instances allow changes to be made to field values. When those changes are made, the subclass code marks itself as changed with the **OBound::Changed** method, which in turn marks the current record (of the dynaset that the attached **OBinder** is managing) as changed with **OBinder::Changed**.

Sometimes you need to discard changes that have been made. To do this, call **DiscardChanges**.

After the changes are discarded, all attached bound objects are refreshed with correct values.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example considers handling an **OBinder** when it is about to be closed.

```
// Here's an OBinder that we are using to edit a table
OBinder tableedit;
// setup of tableedit goes here (see Workbook for a sample).

// now the user is closing the window.  Deal with any changes.
if (tableedit.IsChanged())
{ // the current record has a change in it
    // ask the user if they want to save the change
    int yesno = Message("Do you want to save the change?");
    if (yesno == YES_ANSWER)
        tableedit.Update();
    else
        tableedit.DiscardChanges();
}
/*
Now we can close the window.
By the way, the default behavior when an OBinder is destroyed is for it to
Update().
*/
```

DuplicateRecord Method

Applies To

OBinder, ODynaset

Description

This method creates a new record that is a duplicate of the current record. The current record must be valid.

Usage

oresult **DuplicateRecord**(void)

Remarks

This method works the same way that **AddNewRecord** does, and has the same side effects. OADVISE_ADDNEW messages are sent to all attached advisories and, for **OBinders**, the **PreAdd** and **PostAdd** triggers are called. The only difference is that after the record has been added, the values of the previously current record are used to fill the fields of the new record.

To use this method with an **ODynaset**, you must call **AddNewRecord** and then **Update** to save the changes. When you use **OBinder** you do not need to call **Update**; the **OBinder** machinery does that for you.

Depending on the options that were used to create the database to which this dynaset is attached, the Oracle database may or may not be called to fill values into some of the fields. See the ODATABASE_PARTIAL_INSERT under **ODatabase**.

Note: A call to **StartEdit**, **AddNewRecord**, **DuplicateRecord**, or **DeleteRecord**, will cancel any outstanding **StartEdit**, **AddNewRecord** or **DuplicateRecord** calls before proceeding. Any outstanding changes not saved using **Update** will be lost during the cancellation.

When duplicating records, the order of events is as follows:

1. **ActionRequest** on advisories for add and **PreAdd** trigger (for **OBinders**) in undefined order.
2. Then the new record is added.
3. Then the **ActionNotify** advisory is called.
4. Then the values are copied.
5. Then the **PostAdd** trigger (for **OBinders**) is called.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example duplicates an order.

```
// open the database
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset on the orders table
ODynaset orderdyn(odb, "select * from ord");

// navigate somewhere in the orders dynaset
MoveToInterestingOrder(orderdyn);
/*
```

```
Note that although we are passing the orderdyn by value, when
the routine MoveToInterestingOrder navigates the dynaset, the
current record that we see will be changed.
*/

// duplicate the current order record
orderdyn.DuplicateRecord();

// we need to set a new id (because it must be unique)
orderdyn.SetFieldValue("ordid", orderidseed++);

// save the new record
orderdyn.Update();
```

ErrorNumber Method

Applies To

OAdvise, **OBinder**, **OClient**, **OConnection**, **OConnectionCollection**, **ODatabase**, **ODynaset**, **ODynasetMark**, **OField**, **OFieldCollection**, **OParameter**, **OParameterCollection**, **OSession**, **OSessionCollection**

Description

This method returns an internal class library error number.

Usage

long **ErrorNumber**(void) const

Remarks

This method reports class library error numbers. If there is no error the result will be OERROR_NONE. Otherwise the result will be one of the other OERROR error codes defined in the ORACL.H header file.   These error numbers are discussed in the Error Handling section. Depending on the error, there may be more information available using the **GetErrorText** function.

Oracle database errors (as distinct from error that occur in the use of the class library) are reported through the **OSession** and **ODatabase** methods **ServerErrorNumber** and **GetServerErrorText**.

Return Value

The internal class library error number.

Example

This example shows two different kinds of errors and how they are reported

```
// construct but do not open an ODatabase
OSession sess(0);  // get default session
ODatabase odb;  // construct an unopened ODatabase

// now try to open a dynaset with that
ODynaset empdyn(odb, "select * from emp");
// that failed, so the dynaset is not open

if (!empdyn.IsOpen())
{ // we'll always get here
    long errno = empdyn.ErrorNumber();
    // errno will be OERROR_INVPARENT
    //    because the database was closed
}

// now let's open the database incorrectly
oresult ores = odb.Open(sess, "ExampleDB", "scott", "nottiger");
// the database isn't open because of the bad password

if (ores != OSUCCESS)
{ // we'll always get here
    long errno = sess.ServerErrorNumber();
    // errno will be 1017: invalid username/password
}
```

ErrorReset Method

Applies To

OAdvise, **OBinder**, **OClient**, **OConnection**, **OConnectionCollection**, **ODatabase**, **ODynaset**, **ODynasetMark**, **OField**, **OFieldCollection**, **OParameter**, **OParameterCollection**, **OSession**, **OSessionCollection**

Description

This method resets the internal error state of the object to no error.

Usage

void **ErrorReset**(void) const

Remarks

This method resets the internal error state of the object.   The error reset is the one that reports errors in the use of the class library, not Oracle database errors.   After calling this method **ErrorNumber** will return OERROR_NONE.

ExecuteSQL Method

Applies To

ODatabase

Description

This method sends a SQL statement to the Oracle database to be executed.

Usage

oresult **ExecuteSQL(**const char **sqlstmt*) const

Arguments

sqlstmt	the SQL statement to be executed.

Remarks

This method executes an arbitrary SQL statement specified in the *sqlstmt* argument. The *sqlstmt* should not be a query, but can use select clauses in some other kind of statement. If the SQL statement modifies the data accessed by an open dynaset, the dynaset is not guaranteed to see the change until it is **Refresh**ed.

Note: Some kinds of SQL statements result in an implicit commit. Consult your Oracle documentation.

You can also use the **ExecuteSQL** method to call stored PL/SQL procedures and functions. Any parameters to the procedure or function should be provided with parameter objects.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Several examples follow. They all assume the existence of an open ODatabase named odb.

This example executes a very simple statement. It drops a table.

```
odb.ExecuteSQL("drop table dontwantit");
```

This example gives everybody in Department 20 a 10% raise in their salary by using a SQL statement that updates multiple records. Note that you can obtain the number of rows actually processed with the **GetRowsProcessed** attribute.

```
odb.ExecuteSQL("update emp set sal = sal * 1.1 where deptno = 20");
long numrows = odb.getRowsProcessed();
```

This example calls a stored procedure.

```
// Create a database object
ODatabase odb("ExampleDB", "scott", "tiger");

// Add EMPNO as an Input parameter and set it's initial value
```

```
odb.GetParameters().Add("EMPNO", 7369,
                         OPARAMETER_INVAR, OTYPE_NUMBER);

// Add ENAME as an Output parameter and set it's initial value
odb.GetParameters().Add("ENAME", 0,
                      OPARAMETER_OUTVAR, OTYPE_VARCHAR2);

/*
Execute the Stored Procedure Employee.GetEmpName to retrieve ENAME. This
Stored Procedure can be found in the file ORAEXAMP.SQL
*/

odb.ExecuteSQL("Begin Employee.GetEmpName (:EMPNO, :ENAME); end;")
```

GetChangedError Method

Applies To

OBinder

Description

This method retrieves the last error that occurred when processing a *changed* message.

Usage

oboolean **GetChangedError(**long **serr*, long **cerr*) const

Arguments

serr	pointer to variable to be set to the server error.
cerr	pointer to variable to be set to the class library error.

Remarks

When an **OBound** subclass instance notifies its **OBinder** instance that it has changed its value (by calling **OBound::Changed**) the **OBinder** will call **StartEdit** on its dynaset. This may fail for a variety of reasons. The most common are that another user has a lock on the row, or the current user doesn't have permission to edit the row, or that the data in the database has changed. If the **StartEdit** call fails **OBinder** will call **OnChangedError** to save the error information. That error information can then be retrieved later by calling **GetChangedError**.

A separate routine is needed to check for errors because the changed message is normally sent to the **OBinder** by some indirect process, such as assignment to a variable or a keystroke to a user-interface widget, that does not allow a success return value to be passed up to client code.

Return Value

TRUE if there was an error, FALSE if there was not.

Example

This example sets up a managed dynaset (**OBinder**) and shows *changed* error handling.

```
// construct the OBinder
OBinder empblock;

// here we have several OBoundVal objects (see the Workbook)
OBoundVal salary;
OBoundVal ename;

// bind the OBoundVal objects to the OBinder
salary.BindToBinder(&empblock, "sal");
ename.BindToBinder(&empblock, "ename");

// now open the OBinder
ODatabase odb("ExampleDB", "scott", "tiger");  // open the database
empblock.Open(odb, "select * from emp order by ename");

/*
At this point the OBinder and OBound subclass instances are all set up.
The first record of the dynaset is current.  Now we can try to change a
value.
```

```
*/

salary = 3499.99;
/*
That tried to initiate a database change.  Note that there was no return
value for us to check for success.  We need to call GetChangedError to find
out if that worked.
*/

long servererr;
long classerr;
if (empblock.GetChangedError(&servererr, &classerr))
{
    // error processing here
}
```

GetChunk Method

Applies To

OField

Description

This method fetches a piece of a long or long raw field.

Usage

oresult **GetChunk**(const char ***chunkp*, long *offset*, unsigned short *numbytes*) const

Arguments

chunkp	address of a pointer to be set to point at the resulting data
offset	offset into the long field
numbytes	the number of bytes to get out of the long field

Remarks

Long and long raw fields in an Oracle database can hold a very large amount of information. You may not want to download all of the data from such a field to your client workstation. It is more efficient to simply get the piece that you need (if you only need a piece). **GetChunk** allows you to fetch only a portion of a long field.

The data is read into a memory buffer that is owned by the **OField** object. *Chunkp* is set to point at that memory buffer. The caller should not free what *chunkp* is pointing at; **OField** manages the memory. The memory will be freed when the **OField** object is destroyed, when another **GetChunk** call is made to the same **OField** object, or when a character string is returned by way of a (const char *) cast of the **OField** object.

This method is valid only on fields whose server type is OTYPE_LONG or OTYPE_LONGRAW.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Consider an application where large documents are stored in long fields in an Oracle database. The documents may have excerpts marked on them that are interesting. This example retrieves a single clause from a contract that is stored in a long field.

```
// open the contracts database
ODatabase contrdb("p:legalserver", "solicitor", "murbles");

// get a dynaset on the clause information for clause "a1"
ODynaset clausedyn;  // construct unopened dynaset
// now open it
clausedyn.Open(contrdb, "select * from clauses where cname = "a1");

// what's the contract number for that?
int contractnum;
clausedyn.GetFieldValue("cnumber", &contractnum);

/*
```

```
Set up a parameter on the database. Give it the value of the contract
number we want. (In a real application this parameter would already be
around).
*/
contrdb.GetParameters().Add("cnum", contractnum,
                            OPARAMETER_INVAR, OTYPE_NUMBER);

// get a dynaset on the contracts, selecting the contract we want
ODynaset cdyn(contrdb, "select * from contracts where cnumber = :cnum");
// and an OField on the contract text field
OField contractf = cdyn.GetField("ctext");

// what's the offset and length of that contract clause?
long clauseoffset;
long clauselen;
clausedyn.GetFieldValue("coffset", &clauseoffset);
claysedyn.GetFieldValue("clen", &clauselen);

// get the text
const char *clausetext;
contractf.GetChunk(&clausetext, clauseoffset, clauselen);
```

GetClient Method

Applies To

OSession

Description

This method returns the associated client object by way of an **OClient** handle.

Usage

OClient **GetClient**(void) const

Remarks

Given an **OSession**, you can obtain an **OClient** object, which you can then use to access the client object. Note that this does not create another client object; rather it returns another **OClient** that is a handle for an already existing client object.

Return Value

An **OClient**, which will be open on success, closed on failure.

Example

This example gets a list of the sessions on the workstation.

```
// construct and open an OSession on the default session
OSession defsess(0);

// get the client object
OClient theclient = defsess.GetClient();

// now get the list of sessions
OSessionCollection sessset = theclient.GetSessions();
```

GetConnection

Applies To

ODatabase, **ODynaset**

Description

This method returns the associated connection object by way of an **OConnection** handle.

Usage

OConnection **GetConnection**(void) const

Remarks

This method returns an **OConnection** on the connection that is associated with the object.   Note that this does not create another connection object; rather, it returns another **OConnection** that is a handle for an already existing connection object.

Return Value

An **OConnection**, which will be open on success, closed on failure.

GetConnection (OConnectionCollection)

Applies To

OConnectionCollection

Description

This method returns a specified **OConnection** object.

Usage

OConnection **GetConnection**(int *index*) const

Arguments

index	an index from 0 to **OConnectionCollection**.GetCount()-1

Remarks

This method returns the indexed connection in the collection.

Return Value

An **OConnection**, which will be open on success, closed on failure.

GetConnections Method

Applies To

OSession

Description

This method returns an **OConnectionCollection** object that contains the connections of the session.

Usage

OConnectionCollection **GetConnections**(void) const

Remarks

A session can have multiple connections associated with it.   If you are interested in finding out about the connections of a session as a group, call **GetConnections**.

Return Value

An **OConnectionCollection**, which will be open on success, closed on failure.

Example

This example prints out to a file all the database names in the current session.

```
// we assume an already open FILE * named ofile

// construct and open an OSession on the default session
OSession defsess(0);

// get the connections
OConnectionCollection ccoll = defsess.GetConnections();

// look through all of them
int ii;
for (ii=0; ii<ccoll.GetCount(); ii++)
{  // for each connection, print the database name
    fprintf(ofile, "%s\n", ccoll.GetConnection(ii).GetDatabaseName());
}
```

GetConnectString Method

Applies To

OConnection, ODatabase

Description

This method returns the username that was used to connect to the Oracle database.

Usage

const char ***GetConnectString**(void) const

Remarks

You need three things to connect to an Oracle database: the database name, the username, and the password. **GetConnectString** returns the username. The password is not available because, for security reasons, the class library (and its implementation objects) do not keep the password. **OConnection::GetDatabaseName** and **ODatabase::GetName** return the database name.

The string that is returned is owned by the object. The caller should not free it; it will be freed when the object is destroyed. On error, a NULL is returned.

Return Value

A valid, null terminated const char pointer on success; NULL on failure.

Example

This example opens the database and then looks at the connect string.

```
// construct & open a database object
ODatabase odb("ExampleDB", "scott/tiger", 0);

if (odb.IsOpen())
{ // let's look at the connect string
    const char *cstring = odb.GetConnectString();
    // cstring should be "scott"
}
```

GetCount Method

Applies To

OConnectionCollection, OFieldCollection, OParameterCollection, OSessionCollection

Description

This method returns the number of items in the collection.

Usage

long **GetCount**(void) const

Remarks

The various **Collection** classes inherit the **GetCount** method from their parent **OOracleCollection**.   The **GetCount** method returns the number of items in the collection.   The collection is dynamic; the number of items in the collection reflects the current conditions, not the conditions when the **Collection** object was created.

A closed collection will return 0 items.

Return Value

The number of items in the collection.

Example

This example looks at the number of connections in the current session.

```
// construct and open an OSession on the default session
OSession defsess(0);

// get the connection collection
OConnectionCollection connc = defsess.GetConnections();

// how many connections are there now?
int nconn = connc.GetCount();

// now add a connection by creating a new database object
ODatabase odb("ExampleDB", "scott/tiger", 0);

// how many connections are there now?
int nconn2 = connc.GetCount();
// nconn2 will equal nconn + 1 because of the new connection
/*
Actually, nconn2 == nconn + 1 only if a previous connection was
not shared.  Connection sharing would have occurred if there was
a previous connection, within the same session, that used the
same connection information (database name, user name, password).
*/
```

GetDatabase Method

Applies To

OBinder, ODynaset

Description

This method returns the associated database object by way of an **ODatabase** handle.

Usage

ODatabase **GetDatabase**(void) const

Remarks

Given an **ODynaset** or **OBinder** object, you can obtain an **ODatabase** object, which you can then use to access the **ODynaset**'s or **OBinder**'s database.   Note that this does not create another database object; rather, it returns another **ODatabase** that is a handle for an already existing database object.

Return Value

An **ODatabase**, which will be open on success, closed on failure.

Example

This example assumes that we have been working with an **ODynaset** object and now we want to execute a general SQL statement on the database.

```
// we've got an open ODynaset named workdyn

// get an ODatabase from workdyn
ODatabase workdb = workdyn.GetDatabase();

// now execute an SQL statement
workdb.ExecuteSQL("drop table tempwork");

// to do just one thing, we don't even need an ODatabase instance
// so we could have said
workdyn.GetDatabase().ExecuteSQL("drop table tempwork");
```

GetDatabaseName Method

Applies To

OConnection

Description

This method returns the name of the database to which this connection is connected.

Usage

const char *__GetDatabaseName__(void) const

Remarks

You need three things to connect to an Oracle database: the database name, the username, and the password. **GetConnectString** returns the username. The password is not available because, for security reasons, the class library (and its implementation objects) do not keep the password. **OConnection::GetDatabaseName** and **ODatabase::GetName** return a pointer to the database name.

It is possible for several database objects to share a connection. However, they will not share a connection unless they have the same database name.

The actual memory that the pointer points to is managed by the object. It should not be freed by the caller. It will be freed when the object is destroyed or closed.

Return Value

A valid, null terminated const char pointer on success; NULL on failure.

Example

This example gets the database name.

```
// we start with a dynaset named empdyn (which is open)

// get the dynaset's connection
OConnection tempconn = empdyn.GetConnection();

// now get the database name
const char *dbname = tempconn.GetDatabaseName();
/*
Note that we must use dbname (or copy it) before tempcon goes out of scope,
because tempconn's destructor will free the string.
*/
```

GetDynaset Method

Applies To

OBinder, **OAdvise**

Description

This method returns the associated dynaset object by way of an **ODynaset** handle.

Usage

ODynaset **GetDynaset**(void)

Remarks

When used with **OBinder**, this method returns an **ODynaset** object corresponding to the dynaset that the **OBinder** object is managing.   If the **OBinder** has not successfully executed a SQL query, this method returns a closed **ODynaset** object.

When used with **OAdvise**, this method returns the dynaset that the advisory is on. If the advisory is closed, it returns a closed dynaset.

Return Value

An **ODynaset**, which will be open on success, closed on failure.

GetEditMode

Applies To

ODynaset

Description

This method returns the edit mode of the current record.

Usage

int **GetEditMode**(void) const

Remarks

The edit mode of the current record can be one of the following forms:

ODYNASET_EDIT_NOEDIT	// the current record is not being edited
ODYNASET_EDIT_EDITING	// the current record is being edited
ODYNASET_EDIT_NEWRECORD	// the current record was added, either with **AddNewRecord** or **DuplicateRecord**

If there is no current record, or the **ODynaset** is not open, or there is some other error, ODYNASET_EDIT_NOEDIT is returned.

Return Value

One of the ODYNASET_EDIT_* defines; ODYNASET_EDIT_NOEDIT on error.

Example

This example shows how each of the different edit modes can occur.

```
// construct and open an ODatabase
ODatabase odb("ExampleDB", "scott", "tiger");

// construct an ODynaset but don't open it
ODynaset dyn;

int editmode;
editmode = dyn.GetEditMode();
// editmode is ODYNASET_EDIT_NOEDIT because the dynaset is not open

// now open the dynaset
dyn.Open(odb, "select * from emp");
dyn.MoveFirst();

// What is the edit mode when we're just looking at data?
editmode = dyn.GetEditMode();
// editmode is ODYNASET_EDIT_NOEDIT because the current record has
//    not been changed or added

// What's the edit mode when we're editing?
dyn.StartEdit();  // start editing
editmode = dyn.GetEditMode();
// now editmode is ODYNASET_EDIT_EDITING
dyn.SetFieldValue("sal", 8000);  // set some data
dyn.Update();
// the edit mode is back to ODYNASET_EDIT_NOEDIT after the Update
// now add a record
dyn.AddNewRecord();
editmode = dyn.GetEditMode();
// now editmode is ODYNASET_EDIT_NEWRECORD
dyn.Update();  // save the new record
// edit mode is back to ODYNASET_EDIT_NOEDIT again
/*
By the way, dyn.ErrorNumber() would now report an error because the
update didn't occur.  That's because the emp table requires some
fields to have non-NULL values and we didn't set them.
*/
```

GetErrorText Method

Applies To

OAdvise, **OBinder**, **OClient**, **OConnection**, **OConnectionCollection**, **ODatabase**, **ODynaset**, **ODynasetMark**, **OField**, **OFieldCollection**, **OParameter**, **OParameterCollection**, **OSession**, **OSessionCollection**

Description

This method returns a text description (if one is available) of the current internal class library error.

Usage

const char ***GetErrorText**(void) const

Remarks

This method returns a text description of the most recent internal error on this object. If no error condition exists, or if no text is available for the most recent error, the method returns a NULL. Calling **ErrorNumber** gives you the error number even if no text is available.

The string returned is owned by the object. The caller should not free it; it will be freed when the object is destroyed, closed, or the next time the error is reset. The error is reset whenever you call a method.

Oracle database errors (as distinct from error that occur in the use of the class library) are reported through the **OSession** and **ODatabase** methods **ServerErrorNumber** and **GetServerErrorText**.

Return Value

A test description (if one is available) of the current internal class library error.

GetField Method

Applies To

ODynaset, OFieldCollection

Description

This method returns one of the associated field objects by way of an **OField** handle.

Usage

OField **GetField**(int *index*) const

OField **GetField**(const char **fieldname*) const (for **ODynaset** only)

Arguments

index	the 0-based index of the field. The index is the position of the field in the SQL query that created the current record set.
fieldname	the name of the field, as expressed in the SQL query

Remarks

This method returns an **OField** object on the indicated field. The field object remains valid until the dynaset is refreshed or destroyed.

When getting a field from an **OFieldCollection**, the 0-based index still refers to a SQL query. Specifically, it refers to the SQL query of the dynaset that the **OFieldCollection** was gotten from.

Return Value

An **OField**, which will be open on success, closed on failure.

Example

This example demonstrates getting fields.

```
// open an ODatabase
ODatabase odb("ExampleDB", "scott/tiger", 0);

// open an ODynaset
ODynaset dyn;
dyn.Open(odb, "select empno, ename, nvl(comm,0), hiredate date \
               from emp");

// get a field by index
OField empnofield = dyn.GetField(0);

// get a field by name
OField namefield = dyn.GetField("ename");

// get a computed field by name
//    we use the expression (which is the column name)
OField commfield = dyn.GetField("nvl(comm,0)");

// get a field that has been aliased - the column name is the alias
OField datefield = dyn.GetField("date");
```

GetFieldChunk Method

Applies To

ODynaset

Description

This method fetches a piece of a long or long raw field.

Usage

oresult **GetFieldChunk**(int *index*, void **chunkp*, long *offset*, unsigned short *numbytes*) const

oresult **GetFieldChunk**(const char **fieldname*, void **chunkp*, long *offset*, unsigned short *numbytes*) const

Arguments

index	the 0-based index of the field. The index is the position of the field in the SQL query that created the current record set.
fieldname	the name of the field, as expressed in the SQL query
chunkp	pointer to buffer to be filled with data
offset	offset into the long field
numbytes	the number of bytes to get out of the long field

Remarks

Long and long raw fields in an Oracle database can hold a very large amount of information. You may not want to download all of the data from such a field to your client workstation. It is more efficient to get the piece that you need (if you only need a piece). **GetFieldChunk** enables you to fetch only a portion of a long field.

The data is read into a memory buffer that is provided by the caller. The caller is responsible for ensuring that the buffer is long enough to hold the number of bytes that is asked for. **GetFieldChunk** does not null-terminate the returned data (it is possible that the data is not text).

This method is valid only on fields whose server type is OTYPE_LONG or OTYPE_LONGRAW.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Consider an application where large documents are stored in long fields in an Oracle database. The documents may have excerpts marked on them that are interesting. This example retrieves a single clause from a contract that is stored in a long field.

```
// open the contracts database
ODatabase contrdb("p:legalserver", "solicitor", "murbles");

// get a dynaset on the clause information for clause "a1"
ODynaset clausedyn;  // construct unopened dynaset
// now open it
clausedyn.Open(contrdb, "select * from clauses where cname = "a1");
```

```
// what's the contract number for that?
int contractnum;
clausedyn.GetFieldValue("cnumber", &contractnum);

/*
Set up a parameter on the database. Give it the value of the contract
number we want. (In a real application this parameter would already be
around).
*/
contrdb.GetParameters().Add("cnum", contractnum,
                            OPARAMETER_INVAR, OTYPE_NUMBER);

// get a dynaset on the contracts, selecting the contract we want
ODynaset cdyn(contrdb, "select * from contracts where cnumber = :cnum");
// and an OField on the contract text field
OField contractf = cdyn.GetField("ctext");

// what's the offset and length of that contract clause?
long clauseoffset;
long clauselen;
clausedyn.GetFieldValue("coffset", &clauseoffset);
claysedyn.GetFieldValue("clen", &clauselen);

// get the text
const char *clausetext;
contractf.GetChunk(&clausetext, clauseoffset, clauselen);
```

GetFieldCount Method

Applies To

ODynaset

Description

This method returns the number of fields in each record.

Usage

int **GetFieldCount**(void) const

Return Value

The number of fields in each record; 0 on error.

Example

This example demonstrates **GetFieldCount**.

```
// open a database
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset
ODynaset dyn(odb, "select empno, ename, sal, comm from datatable");

// how many fields was that?
int nfields = dyn.GetFieldCount();
// nfields is 4
```

GetFieldIndex Method

Applies To

ODynaset

Description

This method returns the index of the field indicated by *fieldname*.

Usage

int **GetFieldIndex**(const char **fieldname*) const

Arguments

fieldname	the name of the field as it appears in the SQL statement that the dynaset most recently used

Remarks

Accessing fields of a dynaset by index is more efficient than accessing them by name. Therefore, if you need to access a particular field many times, use this method to translate its name into its index.

Return Value

The 0-based index of the field; -1 on error.

Example

This example looks at the salaries of many employees.

```
// open up the employee database
ODatabase empdb("ExampleDB", "scott", "tiger");

// get the main employee table
ODynaset emps(empdb, "select * from emp");

// now look at all their salaries

// let's get the index of the salary field for speed
int salind = emps.GetFieldIndex("sal");

double salary;  // variable we'll use to get the salary
while (!emps.IsEOF())
{
    emps.GetFieldValue(salind, &salary);
    // do some processing
    emps.MoveNext();
}
```

GetFieldPrecision Method

Applies To

ODynaset

Description

This method returns the precision of the number field.

Usage

int **GetFieldPrecision**(int *index*) const

int **GetFieldPrecision**(const char **fieldname*) const

Arguments

index	the 0-based index of the field. The index is the position of the field in the SQL query that created the current record set.
fieldname	the name of the field, as expressed in the SQL query

Remarks

In an Oracle database, one column data type is *number*. If a field has a data type of number, it has two additional attributes: *scale* and *precision*. The precision is the total number of decimal digits. In Oracle7 this can range from 1 to 38.

Precision has no meaning for non-number fields.

Return Value

The precision of the number field. On error (which includes calling this method on a non-number field) a 0 is returned.

GetFields Method

Applies To

ODynaset

Description

This method returns an **OFieldCollection** object that contains the fields in the dynaset.

Usage

OFieldCollection **GetFields**(void) const

Remarks

An **OFieldCollection** object is a dynamic collection of the fields contained within a single dynaset.

Return Value

An **OFieldCollection**, which will be open on success, closed on failure.

GetFieldScale Method

Applies To

ODynaset

Description

This method returns the scale of the number field.

Usage

int **GetFieldScale**(int *index*) const

int **GetFieldScale**(const char **fieldname*) const

Arguments

index	the 0-based index of the field. The index is the position of the field in the SQL query that created the current record set.
fieldname	the name of the field, as expressed in the SQL query

Remarks

In an Oracle database, one column data type is *number*. If a field has a data type of number, it has two additional attributes: *scale* and *precision*. The scale is the number of decimal digits to the right of the decimal point. It can range (in Oracle7) from -84 to 127.

Scale has no meaning for non-number fields.

Return Value

The scale of the number field. On error (which includes calling this method on a non-number field) a 0 is returned.

GetFieldServerSize Method

Applies To

ODynaset

Description

This method returns the length of a long or long raw field as stored on the server.

Usage

long **GetFieldServerSize**(int *index*) const

long **GetFieldServerSize**(const char **fieldname*) const

Arguments

index	the 0-based index of the field. The index is the position of the field in the SQL query that created the current record set.
fieldname	the name of the field, as expressed in the SQL query

Remarks

This routine is used to obtain the size of long (server type OTYPE_LONG) and long raw (server type OTYPE_LONGRAW) fields as stored on the server.

If this is a long field and the size is larger than 64K, this routine returns -1.

For fields that are neither long nor long raw, this routine will return a 0. To get the size of these fields as stored on the client, use **GetFieldSize**.

Return Value

The size of the field; 0 on error.

GetFieldServerType Method

Applies To

ODynaset

Description

This method returns the Oracle type of the specified field in the database.

Usage

int **GetFieldServerType**(int *index*) const

int **GetFieldServerType**(const char **fieldname*) const

Arguments

index	the 0-based index of the field. The index is the position of the field in the SQL query that created the current record set.
fieldname	the name of the field, as expressed in the SQL query

Remarks

Every column in an Oracle database and every column computed in a SQL statement has a type. This method returns the type of the specified field. It will have one of the following values:

OTYPE_VARCHAR2	varchar2, variable length character
OTYPE_NUMBER	numeric field
OTYPE_LONG	long text (> 2000 bytes)
OTYPE_ROWID	Oracle rowid
OTYPE_DATE	a date
OTYPE_RAW	raw bytes
OTYPE_LONGRAW	long blob of bytes (generally > 255 bytes)
OTYPE_CHAR	fixed-length text
OTYPE_MSLABEL	special type for Trusted Oracle

For more information on these types consult the *Oracle SQL Language Reference Manual*.

Return Value

An integer which identifies the type of the specified field.

GetFieldSize Method

Applies To

ODynaset

Description

This method returns the length of the field as stored locally on the client.

Usage

long **GetFieldSize**(int index) const

long **GetFieldSize**(const char *fieldname) const

Arguments

index	the 0-based index of the field. The index is the position of the field in the SQL query that created the current record set.
fieldname	the name of the field, as expressed in the SQL query

Remarks

This method will return turn number of bytes used to store the field on the client. It will always return a 0 for long or long raw fields.

To get the size of a long or long raw field as stored on the server, use **GetFieldServerSize**.

Return Value

The size of the field; 0 on error.

GetFieldValue Method

Applies To

ODynaset

Description

This method obtains the value of a field in the current record of the dynaset.

Usage

oresult **GetFieldValue**(int *index*, OValue **val*) const

oresult **GetFieldValue**(const char **fieldname*, OValue **val*) const

oresult **GetFieldValue**(int *index*, int **val*) const

oresult **GetFieldValue**(const char **fieldname*, int **val*) const

oresult **GetFieldValue**(int *index*, long **val*) const

oresult **GetFieldValue**(const char **fieldname*, long **val*) const

oresult **GetFieldValue**(int *index*, double **val*) const

oresult **GetFieldValue**(const char **fieldname*, double **val*) const

oresult **GetFieldValue**(int *index*, char **buffer*, unsigned short *maxlen*) const

oresult **GetFieldValue**(const char **fieldname*, char **buffer*, unsigned short *maxlen*) const

oresult **GetFieldValue**(int *index*, void __huge **blobp*, long *bloblen*, long * *blobread*) const

oresult **GetFieldValue**(const char **fieldname*, void __huge * *blobp*, long *bloblen*, long **blobread*) const

Arguments

index	the 0-based index of the field. The index is the position of the field in the SQL query that created the current record set.
fieldname	the name of the field, as expressed in the SQL query
val	a variable, of one of a number of types, that will receive the value
buffer	a caller-provided buffer that will be filled with a text value
maxlen	the maximum number of bytes that can be place in the buffer
blobp	a caller-provided buffer that will be filled with data from a long or long raw field
bloblen	the number of bytes to be read into *blobp*
blobread	to be set to the number of bytes that were read into *blobp*

Remarks

These methods get the value of a particular field specified by *index* (position in the SQL query) or by *fieldname*. Simple data can be extracted into any of the following types: int, long, double, and **OValue**.

If you need to get the value as a string, pass in a pointer to a character buffer. In this case, the length indicated by *maxlen* should include space for a null terminator, which will be added. Alternatively, you can get the string as an **OValue** and then cast the

OValue to const char *.   (See **OValue** for more information).

You should read data from a raw field into a string.   Embedded nulls will be preserved (a null terminator will be added).

You can read data from a long or long raw field as a string if the length is less than 64K. If the length is greater than 64K (or simply if you want to), you can read the field into a buffer that you provide.   The number of bytes that is actually read from the database is returned in the *blobread* argument.   You can use the forms of **GetFieldValue** that read *blobs* only on fields whose server type is OTYPE_LONGRAW or OTYPE_LONG.

The method attempts to convert from one type to another.   For example, asking for the field value as an integer when it is a character string with the value "23" will return the integer 23.

The method returns OSUCCESS if the value could be obtained in the desired type.   It fails, and returns OFAILURE, if the current record is invalid, or the indicated field does not exist, or the data cannot be coerced into the desired type.

It is more efficient to ask for the field's value using the *index* argument than using the *fieldname* argument.   Use the **GetFieldIndex** method to convert a field name to an index.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

An example of a variety of GetFieldValue calls:

```
// open a database object
ODatabase odb("t:123.45.987.06", "scott", "feline");

// open a dynaset on a table with several field types
ODynaset dyn(odb, "select fchar, fnumber, flong from data");
dyn.MoveFirst();

// declare variables to hold data
int ival;  // integer value
char cval[30]; // character string value
char *blobbuff = new char[80000]; // space for a whole document

// now read some data
dyn.GetFieldValue("fnumber", &ival);  // returns 23
dyn.GetFieldValue("fnumber", cval, 30); // returns "23"
dyn.GetFieldValue("fchar", cval, 30);   // returns the string in fchar
dyn.GetFieldValue("fchar", &ival);  // puts a number in ival, if it can

// get the long piece of text
long nread;  // number of bytes actually read
dyn.GetFieldValue("flong", (void *) blobbuff, 80000, &nread);
```

GetLastModifiedMark Method

Applies To

ODynaset

Description

This method returns an **ODynasetMark** on the record that was changed most recently.

Usage

ODynasetMark **GetLastModifiedMark**(void) const

Remarks

This method returns an **ODynasetMark** object referring to the last record modified. The last record modified is the last record that was edited or added.

See **ODynasetMark** for more information on marks.

You can use the mark with the **MoveToMark** method to set the current record back to the marked record.

Return Value

An **ODynasetMark**, which will be open on success, closed on failure.

Example

This example demonstrates GetLastModifiedMark.

```
// open a database
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset
ODynaset empdyn(odb, "select * from emp");

// add a new record
empdyn.AddNewRecord();
empdyn.SetFieldValue("empno", 9991);
empdyn.SetFieldValue("deptno", 10);
empdyn.SetFieldValue("sal", 2500);
empdyn.SetFieldValue("job", "CLERK");
empdyn.Update();

// go to the first record
empdyn.MoveFirst();

// get a mark on the last modified record
ODynasetMark markadd = empdyn.GetLastModifiedMark();

// navigate some more
empdyn.MoveNext();
empdyn.MoveNext();

// now go to the mark
empdyn.MoveToMark(markadd);
// the current record is now the record we added
```

GetMark Method

Applies To

ODynaset

Description

This method returns an **ODynasetMark** on the current record.

Usage

ODynasetMark **GetMark**(void) const

Remarks

This method returns an **ODynasetMark** object referring to the current record. The current record must be valid; you cannot get a mark on a deleted record or when the dynaset is after the last record (IsEOF is TRUE) or before the first record (IsBOF is TRUE).

See **ODynasetMark** for more information on marks.

You can use the mark with the **MoveToMark** method to set the current record back to the marked record.

Return Value

An **ODynasetMark**, which will be open on success, closed on failure.

Example

This example demonstrates GetMark.

```
// open a database
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset
ODynaset empdyn(odb, "select * from emp");

// navigate to an interesting record
empdyn.MoveNext();

// get a mark on this record
ODynasetMark record1 = GetMark();

// go somewhere else
empdyn.MoveLast();
empdyn.MovePrev();

// return to where we were
empdyn.MoveToMark(record1);
```

GetName Method

Applies To

ODatabase, OField, OParameter, OSession

Description

This method returns the object's name.

Usage

const char ***GetName**(void) const

Remarks

Various objects can be referred to, in one context or another, by name.

- The name of a database object is the database name used for connecting to an Oracle database.
- The name of a field is the field name in the SQL query that created the dynaset to which the field is attached.
- The parameter name is the name that is used (with the ":name" syntax) in SQL statements; it is specified when the parameter is created by way of **OParameterCollection::Add**.
- The session name is either an internally generated string (for a default session) or the name specified by the user when the session is created.

GetName returns a pointer to a null-terminated string containing the name.

The actual memory that the pointer points to is managed by the object. It should not be freed by the caller; it will be freed when the object is destroyed or closed.

Return Value

A pointer to a string if successful; NULL if not.

Example

An example of the uses and pitfalls of GetName:

```
// we have connection information from a caller:
//    dname - database name
//    connect - username/password
ODatabase odb;
odb.Open(dname, connect);
if (!odb.IsOpen())
    return;  // the user gave us a bad connect
// odb.GetName will equal dname

// we also have an SQL statement called sqlstmt
// open a dynaset with it
ODynaset dyn(odb, sqlstmt);
if (!dyn.IsOpen())
    return;  // user gave us a bad SQL statement

// What is the name of the first field in the dynaset?
OField f1 = dyn.GetField(0);
const char *fieldname = f1.GetName();
// that works fine
// what if we skipped the declaration of f1?
const char *fname2 = dyn.GetField(0).GetName();
/*
What object is GetName run on?  The temporary OField object returned by
dyn.GetField(0).  It will successfully return a name with GetName() and
then go out of scope.  So GetName() will return a non-NULL pointer that is
pointing to freed memory.  Watch out!
*/
```

GetNamedSession Method

Applies To

OSession

Description

This method returns the session with the specified name by way of an **OSession** handle.

Usage

static OSession **GetNamedSession**(const char **sname*)

Arguments

sname	the name of the desired session

Remarks

When sessions are created they are given a name. The **GetNamedSession** enables you to get a session based on that name. It is not possible to share sessions across applications, only within applications.

This routine is static, so it does not have to be invoked on an **OSession** object. It can be invoked as "OSession::GetNamedSession".

You can obtain the applications default session by passing a NULL for *sname*.

Return Value

An **OSession**, which will be open on success, closed on failure.

Example

Getting sessions by name:

```
// we can obtain the default session:
OSession defsess = OSession::GetNamedSession(0);

// or we can get a session by name.
// Lets create a session by name:
OSession newsess;
newsess.Open("sessname");

// now go get that session
OSession newscopy = OSession::GetNamedSession("sessname");

// by the way
oboolen isequal = (newscopy == newsess);
// isequal is TRUE
```

GetOptions Method

Applies To

ODatabase, ODynaset

Description

This method returns the options that were set on the database or dynaset at the time it was opened.

Usage

long **GetOptions**(void) const

Remarks

Database: When a database is created using **ODatabase::Open**, or using one of the construct and open constructors, some options are specified.   This method returns those options.   The options are a set of flags that are ORed together:

ODATABASE_PARTIAL_INSERT

ODATABASE_EDIT_NOWAIT

If no options have been set the return value is:

ODATABASE_DEFAULT

See the **ODatabase** section for more information on these flags.

Dynaset: When a dynaset is created using **ODynaset::Open** or the construct and open constructor, some options are specified.   This method returns those options.   The options are a set of flags that are ORed together:

ODYNASET_NOBIND

ODYNASET_KEEP_BLANKS

ODYNASET_READONLY

ODYNASET_NOCACHE

If no options have been set, the return value is:

ODYNSET_DEFAULT

See the **ODynaset** section for more information on these flags.

Return Value

The objects options; 0 on error.

GetParameter Method

Applies To

OParameterCollection

Description

This method returns a specified **OParameter** object.

Usage

OParameter **GetParameter**(int *index*) const

OParameter **GetParameter**(const char **pname*) const

Arguments

index	An index from 0 to **OParameterCollection**.GetCount()-1
pname	the name of the parameter, as stated when the parameter was created with Add

Remarks

OParameter objects are obtained either by name or index from an **OParameterCollection** object. The **OParameterCollection** object is obtained from an **ODatabase** with the GetParameters method.

Return Value

An **OParameter** object which will be open on success, closed on failure.

Example

This example checks the status of the "deptno" parameter.

```
// we have an open ODatabase named odb
OParameterCollection params = odb.GetParameters();
OParameter pdeptno = params.GetParameter("deptno");
pstatus = pdeptno.GetStatus();
```

GetParameters Method

Applies To

ODatabase

Description

This method returns an **OParameterCollection** object that contains the parameters of the database.

Usage

OParameterCollection **GetParameters**(void) const

Remarks

To add or remove parameters from a database, for use with dynasets on that database, you must use an **OParameterCollection**.

Return Value

An **OConnectionCollection**, which will be open on success, closed on failure.

Example

This example adds a parameter to an existing ODatabase (odb) and uses it to open an ODynaset:

```
// open a database
ODatabase odb("ExampleDB", "scott", "tiger");

// now add a parameter named ourdeptno to the database
OParameterCollection params = odb.GetParameters();
params.Add("ourdeptno ", 20, OPARAMETER_INVAR, OTYPE_NUMBER);

// now create and open a dynaset using that parameter
ODynaset dyn(odb, "select * from emp where deptno = :ourdeptno");
```

GetPrecision Method

Applies To

OField

Description

This method returns the precision of the field.

Usage

int **GetPrecision**(void) const

Remarks

In an Oracle database, one column data type is *number*. If a field has a data type of number it has two additional attributes: *scale* and *precision*. The precision is the total number of decimal digits. In Oracle7 this can range from 1 to 38.

Precision has no meaning for non-number fields.

Return Value

The precision of the number field. On error (which includes calling this method on a non-number field) a 0 is returned.

GetRecordCount Method

Applies To

ODynaset

Description

This method returns the total number of records in the dynaset's result set.

Usage

long **GetRecordCount**(void) const

Remarks

This method returns the total number of records that the **ODynaset**s query returns.   On error, this method returns 0 (zero).

Attention: When working with a relational database, the only way to determine the total number of records returned by a query is to actually fetch them from the server. Therefore, this method fetches the entire query result set to the local cache. If the result set is large, this takes a long time and uses a large amount of disk space.   It is rare that you really need to know the number of records in a dynaset.

Executing this method with an **ODynaset** that was opened with the ODYNASET_NOCACHE option will cause an implicit **MoveLast** and will make the current record the last record in the dynaset.

Return Value

The total number of records in the result set; 0 on error.

Example

An example of using GetRecordCount:

```
// open the employee database
ODatabase odb("p:us-postoffice", "sam", "uncle");

// get records on all the postal workers in the united states
ODynaset dyn(odb, "select * from employees");

// how many employees are there?
// DON'T DO THIS!!
dyn.GetRecordCount();
// that's just used up all your free disk space, and a lot of time

// do this instead:
ODynaset tempdyn(odb, "select count(*) from employees");
long nemployees;
tempdyn.GetFieldValue(0, &nemployees);
// now the number of records is in nemployees

// we can use the server program to do calculations for us.
// we don't always have to download the records to the client
//    machine to calculate something
```

GetRowsProcessed Method

Applies To

ODatabase

Description

This method returns the number of rows that were processed by the last call to **ExecuteSQL.**

Usage

long **GetRowsProcessed**(void) const

Remarks

This call is only valid following a call to **ExecuteSQL.** Furthermore, the result is only valid if the SQL statement was an insert, delete, or update. Although you can use **ExecuteSQL** to run a select statement, you cannot use **GetRowsProcessed** to obtain the record count. This is because **ExecuteSQL** only executes the SQL statement, but does not fetch any rows. Thus there will always be zero rows processed following **ExecuteSQL** for a select statement.

Return Value

The number of rows processed by the last call to **ExecuteSQL.** If the SQL statement was not Data Manipulation (for example update, insert, or delete), **GetRowsProcessed** returns 0. If there has been no previous call to **ExecuteSQL**, the return value is -1.

GetScale Method

Applies To

OField

Description

This method returns the scale of the number field.

Usage

int **GetScale**(void) const

Remarks

In an Oracle database, one column data type is *number*. If a field has a data type of number, it has two additional attributes: *scale* and *precision*. The scale is the number of decimal digits to the right of the decimal point. It can range (in Oracle7) from -84 to 127.

Scale has no meaning for non-number fields.

Return Value

The scale of the number field. On error (which includes calling this method on a non-number field) a 0 is returned.

GetServerErrorText Method

Applies To

OSession, ODatabase

Description

This method returns a text description of the most recent Oracle error in this session.

Usage

const char ***GetServerErrorText**(void) const

Remarks

This method returns the Oracle error message text for the most recent server error, if available. The error message contains an Oracle error number and may contain a brief description of the problem. Errors that occur while opening a database or in a transactional method will be reported on the session. Other errors are reported on the database.

The string returned is owned by the object. The caller should not free it; it will be freed when the object is destroyed, closed, the error is reset with **ServerErrorReset**, or another call is made to **GetServerErrorText**.

Return Value

A valid, null terminated const char pointer on success; NULL on failure.

Example

An example of a server error:

```
// open an ODatabase object
ODatabase odb("ExampleDB", "scott", "tiger");
if (! odb.IsOpen())
{ // Failed to open the database
    OSession tempsess = odb.GetSession();
    ErrorMessage(tempsess.GetServerErrorText());
}
// try to open a dynaset with a bad column name
ODynaset dyn(odb, "select xx from emp");

// if that didn't work, get the error message
if (!dyn.IsOpen())
{ // give the user a message box explaining the error
    ErrorMessage(odb.GetServerErrorText());
}
```

GetServerSize Method

Applies To

OField

Description

This method returns the length of the field as stored on the server.

Usage

long **GetServerSize**(void) const

Remarks

The size of a field may be different on the client and the server.   This is most notable in the case of long (server type OTYPE_LONG) and long raw (server type OTYPE_LONGRAW) fields.

To get the size of the field as stored on the client, use **GetSize**.

Return Value

The size of the field; 0 on error.

GetServerType Method

Applies To

OField, **OParameter**

Description

This method returns the Oracle7 type of the database field or parameter.

Usage

short **GetServerType**(void) const

Remarks

Every column in an Oracle database and every column computed in a SQL statement has a type.   This method returns the type of the field.   It will have one of the following values:

OTYPE_VARCHAR2	varchar2, variable length character
OTYPE_NUMBER	numeric field
OTYPE_LONG	long text (> 2000 bytes)
OTYPE_ROWID	Oracle rowid
OTYPE_DATE	a date
OTYPE_RAW	raw bytes
OTYPE_LONGRAW	long blob of bytes (generally > 255 bytes)
OTYPE_CHAR	fixed-length text
OTYPE_MSLABEL	special type for Trusted Oracle

For more information on these types consult the *Oracle SQL Language Reference Manual*.

Return Value

The type of the field, or 0 on error.

GetSession Method

Applies To

OSession, **ODatabase**, **ODynaset**

Description

This method returns the associated session object by way of an **OSession** handle.

Usage

OSession **GetSession**(void) const

Remarks

This method returns on **OSession** on the session that is associated with the object. Note that this does not create another session object; rather, it returns another **OSession** that is a handle for an already existing session object.

Return Value

An **OSession**, which will be open on success, closed on failure.

GetSession (OSessionCollection)

Applies To

OSessionCollection

Description

This method returns a specified **OSession** object

Usage

OSession **GetSession**(int *index*) const

Arguments

index	an index from 0 to **OSessionCollection**.GetCount()-1

Remarks

This method returns the indexed session in the collection.

Return Value

An **OSession**, which will be open on success, closed on failure.

GetSessions Method

Applies To

OClient

Description

This method returns an **OSession** object containing the sessions of the client.

Usage

OSessionCollection **GetSessions**(void) const

Remarks

The session collection contains all the sessions of the client. Because the client is a workstation-wide object, the collection contains sessions for all the processes on this workstation, not just those of the current application.

Return Value

An **OSessionCollection**, which will be open on success, closed on failure.

Example

This example gets a list of the sessions on the workstation.

```
// construct and open an OSession on the default session
OSession defsess(0);

// get the client object
OClient theclient = defsess.GetClient();

// now get the list of sessions
OSessionCollection sessset = theclient.GetSessions();
```

GetSize Method

Applies To

OField

Description

This method returns the length of the field as stored locally on the client.

Usage

long **GetSize**(void) const

Remarks

This method will return the number of bytes used to store the field on the client.   It will always return a 0 for long or long raw fields.

To get the size of a long or long raw field as stored on the server, use **GetServerSize**.

Return Value

The size of the field; 0 on error.

GetSQL Method

Applies To

ODynaset

Description

This method returns the dynaset's SQL statement.

Usage

const char ***GetSQL**(void) const

Remarks

When an **ODynaset** is opened, creating a dynaset object, a SQL query is given. Subsequently, the SQL statement may be changed with the **SetSQL** method of **ODynaset**. This routine returns the most recent SQL statement given to the dynaset, either through **ODynaset::Open** or **ODynaset::SetSQL**.

If **SetSQL** has been called and the dynaset has not been refreshed, then the SQL statement returned by this method will not correspond to the SQL statement that gives the current result set.

The pointer that is returned is managed by the object. It should not be freed by the caller; it will be freed when the object is destroyed or closed, or the next time **SetSQL**, **Open**, or **GetSQL** is called.

Return Value

A valid, null terminated const char pointer on success; NULL on failure.

Example

An example showing the use of **GetSQL**:

```
// Open a database
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset
ODynaset dyn(odb, "select * from emp");
const char *sql1 = dyn.GetSQL();
// sql1 is equal to "select * from emp"

// now set the SQL of the dynaset
dyn.SetSQL("select * from dept");
// now the sql1 pointer is invalid

// so get sql1 again
sql1 = dyn.GetSQL();
// sql1 is equal to "select * from dept"
//   but the current result set is select * from emp

// now refresh the dynaset
dyn.Refresh();
// sql1 is still valid
// and now sql1 and the current result set are in synch
```

GetStatus Method

Applies To

OParameter

Description

This method returns a set of flags indicating the status of the **OParameter**.

Usage

int **GetStatus**(void) const

Remarks

You can use this method to query a parameter's state. It reports the state by means of a single integer which contain flags. These flags are ORed together, and can have the following values:

```
OPARAMETER_STATUS_IN            // on if the parameter is an in variable
OPARAMETER_STATUS_OUT           // on if the parameter is an out variable
OPARAMETER_STATUS_AUTOENABLED       // on if the parameter is autoenabled
OPARAMETER_STATUS_ENABLED       // on if the parameter is enabled
```

Autoenabling is set with the **AutoEnable** method. The in and out status of a parameter is set when the parameter is created with **OParameterSet::Add**. A parameter is enabled if it is ready to be used as a parameter, which means that it is autoenabled, has a name and a valid value. If you have created the parameter with **OParameterSet::Add** then it will be enable if it is autoenabled.

Return Value

An integer containing ORed flags, or 0 on error.

GetValue Method

Applies To

OField, **OParameter**

Description

This method gets the current value of the object.

Usage

oresult GetValue(OValue **val*) const

oresult GetValue(int **val*) const

oresult GetValue(long **val*) const

oresult GetValue(double **val*) const

oresult GetValue(const char ***cvalp*) const

oresult GetValue(void __huge **blobp*, long *bloblen*, long **blobread*) const   (for **OField** only)

Arguments

val	a variable of one of a number of types, which will receive the value
cvalp	pointer that will be set to point at a text string
blobp	a caller-provided buffer that will be filled with data from a long or long raw field
bloblen	the number of bytes to be read into *blobp*
blobread	to be set to the number of bytes that were read into *blobp*

Remarks

These methods obtain the current value of the object.   Simple data can be extracted into any of the following types: int, long, double, and **OValue**.

When the value is obtained as a const char *, the pointer *cvalp* is set to point at memory that is managed by the object.   That memory should not be freed by the caller; it will be freed when the object is destroyed, closed, or another **GetValue** call is made to get a string.   The string is null-terminated.

You can read data from a long or long raw field as a string if the length is less than 64K. If the length is greater than 64K (or simply if you want to), you can read the field into a buffer that you provide.   The number of bytes that is actually read from the database is returned in the *blobread* argument. You can use the form of **GetValue** that read blobs only on fields whose server type is OTYPE_LONGRAW or OTYPE_LONG.

The method attempts to convert from one type to another.   For example, asking for the value as an integer when it is a character string with the value "23" will return the integer 23.

The method fails if the data cannot be coerced into the desired type.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example is a routine that copies the values from one field (infield) to another (outfield) for all the records in a dynaset.

```
// open the database
ODatabase odb("t:inetserver", "gopher", "web");

// open the dynaset
ODynaset odyn(odb, "select * from table1");

// Get OField variables on the two fields for transferring values
OField infield = odyn.GetField("infield");
OField outfield = odyn.GetField("outfield");

// declare an OValue variable to hold the values
OValue transferval;

// do the work
while (!odyn.IsEOF())
{  // for every record

    // get the value
    infield.GetValue(&transferval);

    // put the value in the other column
    odyn.StartEdit(); // edit this record
    outfield.SetValue(transferval);
    odyn.Update();  // save the change to the Oracle database

    // go to the next record
    odyn.MoveNext();
}
/*
Note that we didn't have to worry about the types of infield and outfield
(other than the fact that they can go from one to the other) because the
OValue variable can hold anything.
*/
```

GetValue (OBound) Method

Applies To

OBound

Description

This method gets the current value of the object.

Usage

oresult **GetValue**(OValue **val*)

Arguments

val	an **OValue** variable that will receive the value

Remarks

This is a protected **OBound** method. It is called only by subclasses of the **OBound** class. It can be called at any time by a subclass method to obtain the current value of the field to which the instance is bound.

The most common case when an instance needs a value is when it is being **Refresh**ed. In this case the value is handed to it. See **Refresh** (**OBound**).

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Please see the *Workbook* for the example "OBound of a variable." That example works through the implementation of a subclass of OBound.

GetVersion Method

Applies To

OSession

Description

This method returns a string indicating the version of the class library.

Usage

const char ***GetVersion**(void) const

Remarks

This method returns a string which changes for every version of the class library.

The actual memory that the pointer points to is managed by the object.   It should not be freed by the caller; it will be freed when the object is destroyed or closed.

Return Value

A string indicating the class library's version, or NULL on error.

IsBOF Method

Applies To

ODynaset

Description

This routine returns TRUE if the current record is before the first record in the result set.

Usage

oboolean **IsBOF**(void) const

Remarks

This method returns TRUE if the current record is before the first record of the dynaset result set. This can happen if the **MovePrev** method is executed at the time that the first record is current, or if there are no records in the result set.

The name means "Beginning of File" and is a relic from flat-file databases.

Return Value

TRUE if (1) the current record is before the first record, (2) the **ODynaset** is closed, or (3) the dynaset has no records; FALSE otherwise.

Example

This example traverses a dynaset record set from the last record to the first.

```
// open a database
ODatabase odb("ExampleDB", "scott", "tiger");

// construct and open the ODynaset
ODynaset odyn(odb, "select * from emp order by empno");

// go to the end
odyn.MoveLast();

// and go through all the records
while (!odyn.IsBOF())
{
    // process the record

    // go to the previous record
    odyn.MovePrev();
}
```

IsChanged Method

Applies To

OBinder, **OBound**

Description

This method returns TRUE if a change has been noted on the object.

Usage

oboolean **IsChanged**(void)

Remarks

An **OBinder** instance keeps track of whether *the current record* has been marked as changed.   This marking is done with the **Changed** method, which is normally called by **OBound** subclass code rather than client code.

The **OBound** instance keeps track of whether *its particular field* has been marked as changed.   This is marking is done with the **Changed** method, which is normally called by the **OBound** subclass code that makes the change to the **OBound** instance value.

Use **IsChanged** to determine whether the record or field has been changed.

Return Value

TRUE if the object's value has been marked as changed; FALSE otherwise.

Example

This example considers handling an **OBinder** when it is about to be closed.

```
// Here's an OBinder that we are using to edit a table
OBinder tableedit;
// setup of tableedit goes here (see Workbook for a sample).

// now the user is closing the window.  Deal with any changes.
if (tableedit.IsChanged())
{ // the current record has a change in it
    // ask the user if they want to save the change
    int yesno = Message("Do you want to save the change?");
    if (yesno == YES_ANSWER)
        tableedit.Update();
    else
        tableedit.DiscardChanges();
}
/*
Now we can close the window.
By the way, the default behavior when an OBinder is destroyed is for it to
Update().
*/
```

IsEOF Method

Applies To

ODynaset

Description

This routine returns TRUE if the current record is after the last record in the result set.

Usage

oboolean **IsEOF**(void) const

Remarks

This method returns TRUE if the current record is after the last record of the dynaset result set.   This can happen if the **MoveNext** method is executed at the time that the last record is current, if the current record is the last record and it is deleted, or if there are no records in the result set.

The name means "End of File" and is a relic from flat-file databases.

Return Value

TRUE if (1) the current record is after the last record (2) the **ODynaset** is closed or (3) the dynaset has no records; FALSE otherwise.

Example

This example deletes all the managers.

```
// open a database
ODatabase odb("ExampleDB", "scott/tiger", 0);

// open a dynaset
ODynaset odyn(odb, "select * from emp");

// get an OField object for looking at the job field
OField job = empdyn.GetField("job");

// look through all the employees
while (!empdyn.IsEOF())
{
    if (0 == strcmp((const char *) job, "MANAGER"))
    { // we found a manager - delete that employee
        empdyn.DeleteRecord();
    }

    // go to next record (gets us to valid record)
    //    or past EOF if there are no more records
    empdyn.MoveNext();
}
```

IsFieldNullOK Method

Applies To

OValue

Description

This method returns TRUE if the field can accept NULL values.

Usage

oboolean **IsFieldNullOK**(int *index*) const

oboolean **IsFieldNullOK**(const char **fieldname*) const

Arguments

index	the 0-based index of the field. The index is the position of the field in the SQL query that created the current record set.
fieldname	the name of the field, as expressed in the SQL query

Remarks

NULL is a possible value for Oracle database fields. This database NULL is different from a C++ NULL. Database NULLs mean "no value set". Some fields are not allowed to contain NULLS. This routine tells you whether the indicated field is allowed to have NULL values.

Return Value

TRUE if the field can contain NULLs; FALSE otherwise.

IsFieldTruncated Method

Applies To

ODynaset

Description

This method returns TRUE if the contents of the specified field are not the complete contents of the column in the database.

Usage

oboolean **IsFieldTruncated**(int *index*) const

oboolean **IsFieldTruncated**(const char **fieldname*) const

Arguments

index	the 0-based index of the field. The index is the position of the field in the SQL query that created the current record set.
fieldname	the name of the field, as expressed in the SQL query

Remarks

Unlike most types of fields, the values of long and long raw fields may be only partially fetched from the server.

This call is valid only on fields whose server type is OTYPE_LONGRAW or OTYPE_LONG.

Return Value

TRUE if the field's value is incomplete; FALSE otherwise.

IsFirst Method

Applies To

ODynaset, **OBinder**

Description

This method returns TRUE if the current record in a dynaset is the first record in that dynaset. In the case of an OBinder object, the current record is in the bound dynaset.

Usage

oboolean **IsFirst**(void) const

Remarks

When a dynaset is opened; the current record is automatically the first record and IsFirst will be TRUE.

If the current record is invalid, such as when BOF is TRUE; IsFirst will be FALSE. This cannot occur in an OBinder as the underlying dynaset is bound between BOF and EOF.

Return Value

TRUE if the current record is the first; FALSE otherwise.

IsLast Method

Applies To

ODynaset, **OBinder**

Description

This method returns TRUE if the current record in a dynaset is the last record in that dynaset. In the case of an OBinder object, the current record is in the bound dynaset.

Usage

oboolean **IsFirst**(void) const

Remarks

If the current record is invalid, such as when EOF is TRUE; IsLast will be FALSE. This cannot occur in an OBinder as the underlying dynaset is bound between BOF and EOF.

Return Value

TRUE if the current record is the first; FALSE otherwise.

IsNull Method

Applies To

OValue

Description

This method returns TRUE if value of the variable is NULL.

Usage

oboolean **IsNull**(void) const

Remarks

NULL is a possible value for Oracle database fields. This database NULL is different from a C++ NULL. Database NULLs mean "no value set".

An **OValue** can contain a value of NULL. This routine tells you whether the **OValue** contains a NULL. Note that if you cast a NULL to (for example) an integer, you get a result of 0.

Return Value

TRUE if the value is NULL; FALSE otherwise.

Example

NULL and not-NULL values in an **OValue** variable:

```
// construct an OValue with a value of 5
OValue val(5);

// is that NULL?
oboolean isnull = val.IsNull();
// isnull is FALSE

// make it NULL
val.Clear();

isnull = val.IsNull();
// now isnull is TRUE
```

IsNullOK Method

Applies To

OField

Description

This method returns TRUE if the field can accept NULL values.

Usage

oboolean **IsNullOK**(void) const

Remarks

NULL is a possible value for Oracle database fields. This database NULL is different from a C++ NULL. Database NULLs mean "no value set". Some fields are not allowed to contain NULLS. This routine tells you whether the indicated field is allowed to have NULL values.

Return Value

TRUE if the field can contain NULLs; FALSE otherwise.

IsOpen Method

Applies To

OAdvise, **OClient**, **OBinder**, **OConnection**, **OConnectionCollection**, **ODatabase**, **ODynaset**, **ODynasetMark**, **OField**, **OFieldCollection**, **OParameter**, **OParameterCollection**, **OSession**, **OSessionCollection**

Description

This method returns TRUE if the object is open.

Usage

virtual oboolean **IsOpen**(void) const

Remarks

See **Close** for a discussion of what it means for an object to be open or closed.

The most common use for **IsOpen** is to check an object after construction or after it has been returned from a routine.   Closed objects indicate that there was some problem opening the object.

Return Value

TRUE if the object is open; FALSE otherwise.

Example

An example demonstrating when to use **IsOpen**:

```
// we construct and incorrectly open a database
ODatabase odb("p:ntserver", "user", "wrongpassword");
if (!odb.IsOpen())
{  // the database wasn't opened
    // error processing
}
else
{ // the database is open
    // use it
}
```

IsTruncated Method

Applies To

OField

Description

This method returns TRUE if the contents of the field are not the complete contents of the column in the database.

Usage

oboolean **IsTruncated**(void) const

Remarks

Unlike most types of fields, the values of long and long raw fields may be only partially fetched from the server.

This call is valid only on fields whose server type is OTYPE_LONGRAW or OTYPE_LONG.

Return Value

TRUE if the field's value is incomplete; FALSE otherwise.

IsValidRecord Method

Applies To

ODynaset

Description

This method returns TRUE if the current record is valid.

Usage

oboolean **IsValidRecord**(void) const

Remarks

If the current record was deleted, or if the current record is before the first record or after the last record, then the current record is invalid and this method returns FALSE. A closed **ODynaset** also returns FALSE.

Otherwise the current record is valid and this routine returns TRUE.

Return Value

TRUE if the current record is valid; FALSE otherwise.

Example

An example showing invalid record:

```
// open a database
ODatabase odb;
odb.Open("ExampleDB", "scott", "tiger");

// open a dynaset
ODynaset odyn;
odyn.Open(odb, "select * from emp");

// go to the first record
odyn.MoveFirst();

oboolean isvalid = odyn.IsValidRecord();
// isvalid is TRUE, unless the dynaset has no records

// go before first record
odyn.MovePrev();

isvalid = odyn.IsValidRecord();
// isvalid is FALSE now

// delete a record
odyn.MoveFirst();
odyn.MoveNext();
odyn.DeleteRecord();
isvalid = odyn.IsValidRecord();
// isvalid is FALSE now, because the current record is deleted
```

LookupErrorText Method

Applies To

OAdvise, **OBinder**, **OClient**, **OConnection**, **OConnectionCollection**, **ODatabase**, **ODynaset**, **ODynasetMark**, **OField**, **OFieldCollection**, **OParameter**, **OParameterCollection**, **OSession**, **OSessionCollection**

Description

This method returns a text description (if one is available) for the internal class library error corresponding to *errno*.

Usage

const char ***LookupErrorText**(long *errno*) const

Arguments

errno	an error number that was returned by **ErrorNumber**

Remarks

This method returns a text string describing the error represented by *errno*.   If no text is available for the this error code, the method returns a NULL.

The string returned is owned by the object. The caller should not free it; it will be freed when the object is destroyed, closed, or the next time the error is reset. The error is reset whenever you call a method.

Oracle database errors (as distinct from error that occur in the use of the class library) are reported through the **OSession** and **ODatabase** methods **ServerErrorNumber** and **GetServerErrorText**.

Return Value

A valid, null terminated const char pointer on success; NULL on failure.

MoveFirst Method

Applies To

OBinder, ODynaset

Description

This method changes the current record to be the first record in the dynaset's result set.

Usage

oresult **MoveFirst**(void)

Remarks

This method sets the current record of the dynaset (for **OBinder**, the dynaset being managed by the **OBinder** object) to be the first record in the result set.

Execution of this method sends OADVISE_MOVE_FIRST messages to all attached advisories.   One of the advisories could cancel the move, which would result in an OFAILURE return.

If the dynaset is being managed by an **OBinder** object, this method causes **PreMove** and **PostMove** triggers to be called.

By default, when a dynaset is created by opening an **ODynaset**, a **MoveFirst** is performed automatically.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Move to the first record in a dynaset:

```
// we assume that we have an open dynaset named empdyn

// Move to the first record
oresult ores = empdyn.MoveFirst();
// if ores == OSUCCESS we got there
```

MoveLast Method

Applies To

OBinder, ODynaset

Description

This method changes the current record to be the last record in the dynaset's result set.

Usage

oresult **MoveLast**(void)

Remarks

This method sets the current record of the dynaset (for **OBinder**, the dynaset being managed by the **OBinder** object) to be the last record in the result set.

Attention: This action requires that all the records in the query be downloaded from the server, which can be expensive in time and disk space.

Execution of this method sends OADVISE_MOVE_LAST messages to all attached advisories.   One of the advisories could cancel the move, which would result in an OFAILURE return.

If the dynaset is being managed by an **OBinder** object, this method causes **PreMove** and **PostMove** triggers to be called.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Move to the last record in a dynaset:

```
// we assume that we have an open dynaset named empdyn

// Move to the last record
oresult ores = empdyn.MoveLast();
// if ores == OSUCCESS we got there
// and we downloaded all the records too (!)
```

MoveNext Method

Applies To

OBinder, ODynaset

Description

This method changes the current record to be the next record in the dynaset's result set.

Usage

oresult **OBinder::MoveNext**(void)

oresult **ODynaset::MoveNext**(oboolean **gopast** = TRUE)

Arguments

gopast	TRUE when we allow the current record mark to go past the last record in the set

Remarks

This method sets the current record of the dynaset (for **OBinder**, the dynaset being managed by the **OBinder** object) to be the next record in the result set. It is the most common routine used to navigate through the records of the database.

It is possible to **MoveNext** past the last record in the dynaset. The current record then becomes invalid and the **IsEOF** method returns TRUE. This is the default behavior. If you want to restrict dynaset navigation to valid records, pass in a *gopast* argument of FALSE. **OBinder::MoveNext** always restricts navigation to valid records.

Execution of this method sends OADVISE_MOVE_NEXT messages to all attached advisories. One of the advisories could cancel the move, which would result in an OFAILURE return.

If the dynaset is being managed by an **OBinder** object, this method causes **PreMove** and **PostMove** triggers to be called.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example deletes all the managers.

```
// open a database
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset
ODynaset empdyn(odb, "select * from emp");

// get an OField object for looking at the job field
OField job = empdyn.GetField("job");

// look through all the employees
while (!empdyn.IsEOF())
{
    if (0 == strcmp((const char *) job, "MANAGER"))
    { // we found a manager - delete that employee
        empdyn.DeleteRecord();
    }
    // go to next record (gets us to valid record)
    //    or past EOF if there are no more records
    empdyn.MoveNext();
}
```

MovePrev Method

Applies To

OBinder, ODynaset

Description

This method changes the current record to be the previous record in the dynaset's result set.

Usage

oresult **OBinder::MovePrev**(void)

oresult **ODynaset::MovePrev**(oboolean *gopast* = TRUE)

Arguments

gopast	TRUE when we allow the current record mark to go before the first record in the set

Remarks

This method sets the current record of the dynaset (for **OBinder**, the dynaset being managed by the **OBinder** object) to be the previous record in the result set.

It is possible to **MovePrev** before the first record in the dynaset.   The current record then becomes invalid and the **IsBOF** method returns TRUE.   This is the default behavior. If you want to restrict dynaset navigation to valid records, pass in a *gopast* argument of FALSE.   **OBinder::MovePrev** always restricts navigation to valid records.

Execution of this method sends OADVISE_MOVE_PREV messages to all attached advisories.   One of the advisories could cancel the move, which would result in an OFAILURE return.

If the dynaset is being managed by an **OBinder** object, this method causes **PreMove** and **PostMove** triggers to be called.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example traverses a dynaset record set from the last record to the first.

```
// open a database
ODatabase odb("ExampleDB", "scott", "tiger");

// construct and open the ODynaset
ODynaset odyn(odb, "select * from emp order by empno");

// go to the end
odyn.MoveLast();

// and go through all the records
while (!odyn.IsBOF())
{
    // process the record

    // go to the previous record
    odyn.MovePrev();
}
```

MoveToMark Method

Applies To

ODynaset

Description

This method sets the current record to the record indicated by the ODynasetMark *mark*.

Usage

oresult **MoveToMark**(const ODynasetMark &*mark*)

Arguments

mark	an **ODynasetMark** previously returned by **GetMark** or **GetLastModifiedMark**

Remarks

An **ODynasetMark** is a way to remember a particular row and be able to get back to it quickly. An **ODynasetMark** is returned by the two **ODynaset** methods **GetMark** and **GetLastModifiedMark**.

The mark being used must have come from an **ODynaset** that is a handle on the same dynaset object, or a clone of that dynaset object.

Navigating to a marked record skips over any intervening records. Execution of this method sends OADVISE_MOVE_TOMARK messages to all attached advisories. One of the advisories could cancel the move, which would result in an OFAILURE return.

If the dynaset is being managed by an **OBinder** object, this method causes **PreMove** and **PostMove** triggers to be called. It is legal to call **MoveToMark** on the dynaset that an **OBinder** is managing.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example uses a clone to figure out where to jump to in a dynaset. If you have a dynaset that has some overhead for moving (for example, one that has many advisories or is being managed by an **OBinder**), it may be faster to navigate around in a clone.

```
// open a database
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset
ODynaset empdyn(odb, "select * from emp");

double salary;  // the employee's salary
int mgrid;      // the employee's manager's id

// get the employee's salary
empdyn.GetFieldValue("sal", &salary);
if (salary < 1000.0)
{ // this employee is underpaid - let's take care of that
     // who is responsible for this?
     empdyn.GetFieldValue("manager", &mgrid);

     // let's go find that scoundrel
```

```
    ODynaset tempdyn = empdyn.Clone();  // clone the employees
    OField id = tempdyn.GetField("id"); // for speed
    tempdyn.MoveFirst();
    while (!tempdyn.IsEOF())
    {
        if (mgrid == (int) id)
            break; // we found the manager
        tempdyn.MoveNext();
    }
    // either we found the manager or...
    if (tempdyn.IsEOF())
        return;  // we won't deal with this in an example

    // a mark on the manager
    ODynasetMark managerMark = tempdyn.GetMark();

    // a mark on the employee
    ODynasetMark empMark = empdyn.GetMark();

    // now penalize that manager
    empdyn.MoveToMark(managerMark); // go to manager record
    empdyn.StartEdit();  // we're going to change things here
    empdyn.SetFieldValue("sal", 700.0);
    empdyn.Update();

    // and give the employee a raise
    empdyn.MoveToMark(empMark); // go to employee record
    empdyn.StartEdit();
    empdyn.SetFieldValue("sal", 7000.0);
    empdyn.Update();

    // Now everybody is happy. Go process some more.
}
```

OAdvise Method

Applies To

OAdvise

Description

OAdvise constructor

Usage

OAdvise(void)

OAdvise(const OAdvise &*otheradvise*)

OAdvise(const ODynaset &*dyn*)

Arguments

otheradvise	another **OAdvise** object that you are copying
dyn	an **ODynaset** to which you will attach after construction

Remarks

These methods construct a new **OAdvise** instance.

The default constructor constructs an unopened **OAdvise** object.   It cannot fail.   You must open the object before you can use it.

The copy constructor copies another advisory.   If that other advisory is open (attached to a dynaset), the new **OAdvise** will be attached to the same dynaset object. The new **OAdvise** object will be a separate advisory on the same dynaset object.   The copy constructor allows **OAdvise** objects to be passed correctly as arguments to routines and to be return values from routines.   The copy constructor can fail; check whether the new OAdvise is open after the constructor call.

The constructor that takes an **ODynaset** as an argument constructs the **OAdvise** object and attempts to open it with that dynaset.   This construct and open can fail; check whether the new **OAdvise** is open after the constructor call.

Note that instances of the **OAdvise** class itself can be declared, but they do nothing.   To have an interesting advisory you must subclass **OAdvise**.

Example

Construction of an **OAdvise** object:

```
// default constructor
OAdvise adv1;

// open that advisory
adv1.Open(thedynaset);

// construct two other advisories, attaching them to the same dynaset
OAdvise adv2(adv1);  // copy constructor
if (!adv2.IsOpen())
{  // there was some error opening the advisory
    // error processing
}

OAdvise adv3(thedynaset);
if (!adv3.IsOpen())
```

```
{ // there was some error opening the advisory
    // error processing
}
```

~OAdvise Method

Applies To

OAdvise

Description

OAdvise destructor

Usage

~**OAdvise**(void)

Remarks

This method destroys the **OAdvise** and frees all its resources.   If the **OAdvise** object is open, the attached dynaset is informed that this advisory should no longer receive messages.

OBinder Method

Applies To

OBinder

Description

OBinder constructor

Usage

OBinder(void)

Remarks

This method constructs an unopened OBinder instance - one that is unconnected to a database and has not executed a SQL query.

You must open an **OBinder** object with an **Open** call before you can use it.

The constructor cannot fail or cause any errors.

The copy constructor for **OBinder** is defined in the header but is not implemented.   This is done so that the compiler's default implementation will not be used (it would be very incorrect).   If you want an **OBinder** subclass copy constructor, you must implement it.

~OBinder Method

Applies To

OBinder

Description

OBinder destructor

Usage

~**OBinder**(void)

Remarks

This method destroys an **OBinder** object.

Destroying an **OBinder** object **Unbind**s all attached **OBound** objects.   The attached **OBound** objects are not themselves destroyed.

Before the **OBinder** is destroyed - which often happens automatically when an **OBinder** instance goes out of scope - you should call the **Close** method so that the **Shutdown** triggers are called. The **OBinder** destructor does not invoke the **Shutdown** triggers.

OBound Method

Applies To

OBound

Description

OBound constructor

Usage

OBound(void)

Remarks

This constructor creates an unbound **OBound** object.   Because **OBound** is a pure virtual class, you never declare any instances of **OBound**.   This constructor is called by the subclass constructor.   **OBound** objects must be bound with a call to **BindToBinder** before they are useful.

This constructor cannot fail or cause any errors.

The copy constructor for **OBound** is defined in the header but is not implemented.   This is done so that the compiler's default implementation will not be used (it would be very incorrect).   If you want an **OBound** subclass copy constructor, you must implement it.

~OBound Method

Applies To

OBound

Description

OBound destructor

Usage

OBound(void)

Remarks

The destructor frees all of the object's resources.   If it is bound to an **OBinder** object, it informs that **OBinder** object that this **OBound** is no longer bound.

The **OBound Shutdown** trigger will be called.   Any OFAILURE return from the **Shutdown** trigger is ignored.

OClient Method

Applies To

OClient

Description

OClient constructor

Usage

OClient(void)

OClient(const OClient &*otherclient*)

Arguments

otherclient	another **OClient** object that you are copying

Remarks

These methods construct a new **OClient** instance.

The default constructor constructs an unopened **OClient** object.   It cannot fail.

The copy constructor copies another **OClient** object.   If that other **OClient** object is open - which means it is a handle on an implementation client object - the new **OClient** object becomes a handle to that same client object.   The copy constructor can fail; check whether the new **OClient** is open after the constructor call.

There is no **Open** method for the **OClient** class.   To get an open **OClient**, call one of the **GetClient** methods.

~OClient Method

Applies To

OClient

Description

OClient destructor

Usage

~**OClient**(void)

Remarks

This method destroys the **OClient** and frees its resources.   The underlying implementation client object will be freed if this is the last object that is referring to it.

OConnection Method

Applies To

OConnection

Description

OConnection constructor

Usage

OConnection(void)

OConnection(const OConnection &*otherconn*)

Arguments

otherconn	another **OConnection** object that you are copying

Remarks

These methods construct a new **OConnection** instance.

The default constructor constructs an unopened **OConnection** object. It cannot fail. You must open the object before you can use it.

The copy constructor copies another **OConnection** object. If that other **OConnection** object is open - which means it is a handle on an implementation connection object - the new **OConnection** object becomes a handle to that same connection object. The copy constructor copies the reference to the connection object but does not copy any strings that the source OConnection may own. The copy constructor can fail; check whether the new **OConnection** is open after the constructor call.

There is no **Open** method for the **OConnection** class. To get an open **OConnection**, call one of the **GetConnection** methods.

~OConnection Method

Applies To

OConnection

Description

OConnection destructor

Usage

~**OConnection**(void)

Remarks

This method destroys the **OConnection** and frees its resources.   The underlying implementation connection will be freed if this is the last object that is referring to it.

When the **OConnection** is destroyed, any strings that the OConnection owns (such as database name or connection strings) are also freed.

OConnectionCollection Method

Applies To

OConnectionCollection

Description

OConnectionCollection constructor

Usage

OConnectionCollection(void)

OConnectionCollection(const OConnectionCollection &*othercoll*)

Arguments

othercoll	another **OConnectionCollection** object that you are copying

Remarks

These methods construct a new **OConnectionCollection** instance.

Constructing an **OConnectionCollection** does not create any connections or **OConnection** objects.

The default constructor constructs an unopened **OConnectionCollection** object.

The copy constructor copies another **OConnectionCollection** object. If that other **OConnectionCollection** object is open - which means it is a handle on an implementation **ConnectionCollection** object - the new **OConnectionCollection** object becomes a handle to that same **ConnectionCollection** object. The copy constructor can fail; check whether the new **OConnectionCollection** is open after the constructor call.

There is no **Open** method for the **OConnectionCollection** class. To get an open **OConnectionCollection**, call one of the **GetConnections** methods.

~OConnectionCollection Method

Applies To

OConnectionCollection

Description

OConnectionCollection destructor

Usage

~**OConnectionCollection**(void)

Remarks

This method destroys the **OConnectionCollection** and frees its resources.   The underlying implementation ConnectionCollection object will be freed if this is the last object that is referring to it.

Destroying the **OConnectionCollection** object has no effect on any connections or **OConnection** objects, even those gotten from the destroyed collection

ODatabase Method

Applies To

ODatabase

Description

ODatabase constructor

Usage

ODatabase(void)

ODatabase(const ODatabase &*otherdb*)

ODatabase(const OSession &*dbsess*, const char **dbname*, const char **username*, const char **pwd*, long *options* = ODATABASE_DEFAULT)

ODatabase(const char **dbname*, const char **username*, const char **pwd*, long *options* = ODATABASE_DEFAULT)

Arguments

otherdb	another **ODatabase** object that you are copying
dbsess	the session under which you want to open this database
dbname	the name of the database you want to connect to
username	the username you want to use to log into the database
pwd	the database password for the user *username*
options	options to be used to create the database object

Remarks

These methods construct a new **ODatabase** instance.

The default constructor constructs an unopened **ODatabase** object. It cannot fail. You must open the object before you can use it.

The copy constructor copies another **ODatabase** object. If that other **ODatabase** object is open - which means it is a handle on an implementation database object - the new **ODatabase** object becomes a handle to that same database object. The copy constructor copies the reference to the database object but does not copy any strings that the source **ODatabase** may own. The copy constructor can fail; check whether the new **ODatabase** is open after the constructor call.

The remaining two constructors both construct and attempt to open the **ODatabase** object. Opening an ODatabase object creates a new database object and may create other resources such as sessions and connections. Successful **Open**ing of an **ODatabase** results in a connection to the Oracle database.

Use the optional *dbsess* argument to choose the session under which this database should opened. The session under which the database is opened affects connection sharing and transaction processing. If you do not specify a session, the application's default session is used.

Use the *dbname*, *username*, and *pwd* arguments to establish the connection to the database. The *pwd* argument is allowed to be NULL. In that case it is expected that the string passed to *username* will be of the form "*username/password*"; the "slash character must be in the string. The database name will either be a SQL*Net alias, such as "ExampleDB" or a complete Oracle database name such as "p:namedpipe-server" or "t:123.45.987.06:SID" (network protocol identifier, network address, option instance id).

The options affect various aspects of the database's behavior.   See **ODatabase** for more information.

The constructors that construct and open an **ODatabase** can fail; check whether the **ODatabase** object is open after the constructor call.

Example

An example of opening **ODatabase** objects:

```
// construct an unopened ODatabase
ODatabase odb;

// the simplest way to open a database
ODatabase odb("ExampleDB", "scott", "tiger");

// now if we open another database similarly
ODatabase odb2("ExampleDB", "scott", "tiger", ODATABASE_EDIT_NOWAIT);
/*
We have two separate database objects on the same oracle database but
because odb2 and odb are in the same session and have the same connection
information they share a database connection. But because of different
options the two database objects will behave differently.
*/

// open a database on a named session
OSession msess("mysession");
ODatabase odb3(msess, "ExampleDB", "scott", "tiger");
// odb3 does not share a connection with odb, because it is on
//    a different session

// call a routine to open a database for us
//   see the implementation below
ODatabase mydb = OpenADatabase("ExampleDB", "scott", "tiger");
/*
mydb is being constructed with the copy constructor.  It is copying
the temporary ODatabase that is the result of OpenADatabase
*/

if (!mydb.IsOpen())
{ // that didn't work
    return;  // we give up
}
// etc...


// here's the routine OpenADatabase
ODatabase OpenADatabase(const char *db, const char *un,
                        const char *pw)
{
    ODatabase tempdb;

    // get a handle on the default session
    //   we need it for error handling later on
    OSession defsess(0);

    // attempt to open tempdb within the default session
    tempdb.Open(defsess, db, un, pw);

    if (!tempdb.IsOpen())
    { // some kind of error.  Give the user a message
        const char *errmsg = defsess.GetServerErrorText();
        MessageBox(errmsg);
    }

    return(tempdb);
}
/*
We always return tempdb.  Returning an object like this works because of
the copy constructor.  The caller of OpenADatabase can tell if the routine
worked or not by checking whether or not the returned ODatabase is open.
*/
```

~ODatabase Method

Applies To

ODatabase

Description

ODatabase destructor

Usage

~**ODatabase**(void)

Remarks

This method destroys the **ODatabase** and frees its resources.   The underlying implementation database will be freed if this is the last object that is referring to it.

When the **ODatabase** is destroyed, any strings that the ODatabase owns (such as database name and connection strings) are also freed.

Destroying the **ODatabase** object takes care of logging off from the database.

ODynaset Method

Applies To

ODynaset

Description

ODynaset constructor

Usage

ODynaset(void)

ODynaset(const ODynaset &*otherdyn*)

ODynaset(const ODatabase &*odb*, const char **sqlstmt*, long *options* = ODYNASET_DEFAULT)

Arguments

otherdyn	another **ODynaset** object that you are copying
odb	the database on which you want to open this dynaset
sqlstmt	a valid select SQL statement
options	options to be used to create the dynaset

Remarks

These methods construct a new **ODynaset** instance.

The default constructor constructs an unopened **ODynaset** object.   It cannot fail.   You must open the object before you can use it.

The copy constructor copies another **ODynaset** object. If that other **ODynaset** object is open - which means it is a handle on an implementation dynaset object - the new **ODynaset** object becomes a handle to that same dynaset object.   The copy constructor copies the reference to the dynaset object but does not copy any strings that the source **ODynaset** may own.   The copy constructor can fail; check whether the new **ODynaset** is open after the constructor call.

The remaining constructor both constructs and attempts to open the **ODynaset** object. Opening a **ODynaset** object creates a new dynaset object.   An **ODatabase** and a SQL statement must be given to open the **ODynaset**.   These specify which Oracle database to get the records from and which records to get.   The options affect various aspects of the dynaset's behavior; see **ODynaset** section for more information.   When the **ODynaset** is opened, it moves to the first record automatically. The copy and open constructor can fail; check whether the **ODynaset** object is open after the constructor call.

Using a "FOR UPDATE" clause in the SQL statement that opens the dynaset requires some special attention.   Please refer to Select for Update.

Example

Examples of opening ODynasets:

```
// first we need a database
ODatabase odb("ExampleDB", "scott", "tiger");

// default constructor
ODynaset odyn;
oboolean isopen = odyn.IsOpen();
```

```
// isopen is FALSE

// copy constructor
ODynaset odyn2(odyn);
isopen = odyn2.IsOpen();
// isopen is FALSE because odyn was not open

// create and open a dynaset
ODynaset odyn3(odb, "select * from emp");
isopen = odyn3.IsOpen();
// isopen is TRUE - the open was successful

// and now if we use a copy constructor
ODynaset odyn4(odyn3);
isopen = odyn4.IsOpen();
// isopen is TRUE

odyn4.MoveLast();
// now odyn3 also refers to the last record, because
// odyn3 and odyn4 are handles to the same dynaset
```

~ODynaset Method

Applies To

ODynaset

Description

ODynaset destructor

Usage

~**ODynaset**(void)

Remarks

This method destroys the **ODynaset** and frees its resources. The underlying implementation dynaset will be freed if this is the last object that is referring to it.

When the **ODynaset** is destroyed, any strings that the **ODynaset** owns (such as the SQL statement) are also destroyed..

If a dynaset has advisories on it, destroying the **ODynaset -** even if it is the only **ODynaset** that referred to that dynaset - will not destroy the dynaset, because the advisories also refer to the dynaset. The dynaset is destroyed only when *all* the objects that refer to the dynaset are destroyed.

ODynasetMark Method

Applies To

ODynasetMark

Description

ODynasetMark constructor

Usage

ODynasetMark(void)

ODynasetMark(const ODynasetMark &*othermark*)

Arguments

othermark another **ODynasetMark** object that you are copying

Remarks

These methods construct a new **ODynasetMark** instance.

The default constructor constructs an unopened **ODynasetMark** object.   It cannot fail.

The copy constructor copies another **ODynasetMark** object.   If that other **ODynasetMark** object is open - which means it contains a valid mark on a dynaset - that mark will be copied.   The copy constructor can fail; check whether the new **ODynasetMark** is open after the constructor call.

There is no **Open** method for the **ODynasetMark** class.   To get an open **ODynasetMark**, call either **GetMark** or **GetLastModifiedMark**.

Example

This example finds the employee with the lowest salary and then gives that person a big commission.

```
// open the employee database
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset on the employee's table
ODynaset odyn(odb, "select sal, comm from emp");

// the low employee mark
ODynasetMark lowmark;  // default constructor.  lowmark is not open
double lowsalary = 100000.;

// find the lowest salary
OField salf = odyn.GetField(0);  // salary is the 0th field
odyn.MoveFirst();
while (!odyn.IsEOF())
{
    if ((double) salf < lowsalary)
    { // the lowest we've seen yet
        // get a mark.  odyn.GetMark returns an open ODynasetMark
        lowmark = odyn.GetMark();
        lowsalary = (double) salf;
    }
}
if (lowmark.IsOpen())
{ // we found a lowest - give them a big commission
```

```
    odyn.MoveToMark();
    odyn.StartEdit();
    odyn.SetFieldValue("comm", 2000.0);
    odyn.Update();
}
```

~ODynasetMark Method

Applies To

ODynasetMark

Description

ODynasetMark destructor

Usage

~**ODynasetMark**(void)

Remarks

This method destroys an **ODynasetMark** and frees its resources.

OField Method

Applies To

OField

Description

OField constructor

Usage

OField(void)

OField(const OField &*otherfield*)

Arguments

otherfield another **OField** object that you are copying

Remarks

These methods construct a new **OField** instance.

The default constructor constructs an unopened **OField** object. It cannot fail.

The copy constructor copies another **OField** object. If that other **OField** object is open - which means it is a handle on an implementation field object - the new **OField** object becomes a handle to that same field object. The copy constructor copies the reference to the field object but does not copy any strings that the source **OField** may own. The copy constructor can fail; check whether the new **OField** is open after the constructor call.

There is no **Open** method for the **OField** class. To get an open **OField**, call one of the **GetField** methods.

Example

This method sums all the salaries of employees.

```
// open the employee database
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset on the employee's table
ODynaset odyn(odb, "select sal, comm from employees");

// get a field on the salary for speed
OField salf = odyn.GetField("sal");
/*
By using the = operator in the declaration of salf we are invoking the copy
constructor.  It is copying the temporary object that is returned by the
GetField method.
*/

// sum the salaries
double sumsal = 0.0;
odyn.MoveFirst();
while (!odyn.IsEOF())
{
    sumsal += (double) salf;
    odyn.MoveNext();
}
```

```
// of course, we could have done the same thing (faster) with:
ODatabase odb("ExampleDB", "scott", "tiger");
ODynaset odyn(odb, "select sum(sal) from employees");
odyn.GetFieldValue(0, &sumsal);
// the server is good at that kind of bulk calculation
```

~OField Method

Applies To

OField

Description

OField destructor

Usage

~**OField**(void)

Remarks

This method destroys the **OField** and frees its resources.   The underlying implementation field will be freed if this is the last object that is referring to it.

When the **OField** is destroyed, any strings that the **OField** owns (such as field name) are also freed.

OFieldCollection Method

Applies To

OFieldCollection

Description

OFieldCollection constructor

Usage

OFieldCollection(void)

OFieldCollection(const OField &*othercoll*)

Arguments

othercoll	another **OFieldCollection** object that you are copying

Remarks

These methods construct a new **OFieldCollection** instance.

Constructing an OFieldCollection does not create any fields or **OField** objects.

The default constructor constructs an unopened **OFieldCollection** object. It cannot fail.

The copy constructor copies another **OFieldCollection** object. If that other **OFieldCollection** object is open - which means it is a handle on an implementation FieldCollection object - the new **OFieldCollection** object becomes a handle to that same FieldCollection object. The copy constructor can fail; check whether the new **OFieldCollection** is open after the constructor call.

There is no **Open** method for the **OFieldCollection** class. To get an open **OFieldCollection**, call one of the **GetFields** methods.

~OFieldCollection Method

Applies To

OFieldCollection

Description

OFieldCollection destructor

Usage

~**OFieldCollection**(void)

Remarks

This method destroys the OFieldCollection and frees its resources.   The underlying implementation fieldcollection object will be freed if this is the last object that is referring to it.

Destroying the **OFieldCollection** object has no effect on any fields or **OField** objects, even those gotten from the destroyed collection

OnChangedError Method

Applies To

OBinder

Description

This method is called by OBinder when there is an error processing a *changed* message.

Usage

virtual void **OnChangedError**(void) const

Remarks

When an **OBound** subclass instance notifies its **OBinder** instance that it has changed its value (by calling **OBound::Changed**) the **OBinder** will call **StartEdit** on its dynaset. This may fail for a variety of reasons. The most common are that another user has a lock on the row, or the current user doesn't have permission to edit the row, or that the data in the database has changed. If the **StartEdit** call fails **OBinder** will call **OnChangedError**.

This routine is supplied separately because often the call sequence that causes the error is very indirect. For instance the **OBound** instance may change the value when an assignment operator is called, or when some user-interface widget is used. **OnChangedError** is a callback mechanism allowing your code to obtain control when an error has occurred.

OnChangedError is a virtual function. The implementation in **OBinder** saves the current server error number and class library error number. These are available using the routine **OBinder::GetChangedError**. You can subclass **OBinder** and override **GetChangedError** with your own error handling method. That overriding method should call **OBinder::OnChangedError** so that the **OBinder** will be able to obtain the server error numbers for use by **GetChangedError**.

Example

This example sets up a managed dynaset (**OBinder**) and shows *changed* error handling.

```
// construct the OBinder
OBinder empblock;

// here we have several OBoundVal objects (see the Workbook)
OBoundVal salary;
OBoundVal ename;

// bind the OBoundVal objects to the OBinder
salary.BindToBinder(&empblock, "sal");
ename.BindToBinder(&empblock, "ename");

// now open the OBinder
ODatabase odb("ExampleDB", "scott", "tiger");  // open the database
empblock.Open(odb, "select * from emp order by ename");

/*
At this point the OBinder and OBound subclass instances are all set up.
The first record of the dynaset is current.  Now we can try to change a
value.
*/
```

```
salary = 3499.99;
/*
That tried to initiate a database change.  Note that there was no return
value for us to check for success.  We need to call GetChangedError to find
out if that worked.
*/

long servererr;
long classerr;
if (empblock.GetChangedError(&servererr, &classerr))
{
    // error processing here
}
```

OParameter Method

Applies To

OParameter

Description

OParameter constructor

Usage

OParameter(void)

OParameter(const OParameter &*otherparam*)

Arguments

otherparam	another **OParameter** object that you are copying

Remarks

These methods construct a new **OParameter** instance.

The default constructor constructs an unopened **OParameter** object. It cannot fail.

The copy constructor copies another **OParameter** object. If that other **OParameter** object is open - which means it is a handle on an implementation parameter object - the new **OParameter** object becomes a handle to that same parameter object. The copy constructor copies the reference to the parameter object but does not copy any strings that the source **OParameter** may own. The copy constructor can fail; check whether the new **OParameter** is open after the constructor call.

There is no **Open** method for the **OParameter** class. To get an open **OParameter**, call the **GetParameter** method of the **OParameterCollection** class. The way that a parameter is actually created is by the **OParameterCollection**::**Add** method.

Example

This example creates a parameter and constructs an **OParameter** object.

```
// open a database
ODatabase odb("ExampleDB", "scott", "tiger");

OParameterCollection params = odb.GetParameters();
OParameter deptp;
deptp = params.Add("dno", 20, OPARAMETER_INVAR, OTYPE_NUMBER);

/*
By using the = operator we are invoking the copy constructor.  It is
copying the temporary object that is returned by the Add method.
*/

// use it
ODynaset empdyn;
empdyn.Open(odb, "select * from emp where deptno = :dno");
```

~OParameter Method

Applies To

OParameter

Description

OParameter destructor

Usage

~**OParameter**(void)

Remarks

This method destroys the OParameter and frees its resources.   The underlying implementation field will be freed if this is the last object that is referring to it.

When the **OParameter** is destroyed, any strings that the OParameter owns (such as parameter name) are also freed.

OParameterCollection

Applies To

OParameterCollection

Description

OParameterCollection constructor

Usage

OParameterCollection(void)

OParameterCollection(const OParameterCollection &*othercoll*)

Arguments

othercoll	another **OParameterCollection** object that you are copying

Remarks

These methods construct a new **OParameterCollection** instance.

Constructing an **OParameterCollection** does not create any parameters, or **OParameter** objects.

The default constructor constructs an unopened **OParameterCollection** object. It cannot fail.

The copy constructor copies another **OParameterCollection** object. If that other **OParameterCollection** object is open - which means that it is a handle on an implementation parametercollection object - the new **OParameterCollection** object becomes a handle to that same parametercollection object. The copy constructor can fail; check whether the new **OParameterCollection** is open after the constructor call.

There is no **Open** method for the **OParameterCollection** class. To get an open **OParameterCollection**, call the **GetParameters** method.

~OParameterCollection

Applies To

OParameterCollection

Description

OParameterCollection destructor

Usage

~OParameterCollection(void)

Remarks

This method destroys the **OParameterCollection** and frees its resources.   The underlying implementation parametercollection object will be freed if this is the last object that is referring to it.

Destroying the **OParameterCollection** object has no effect on any parameters or **OParameter** objects, even those gotten from the destroyed collection

Open (OAdvise) Method

Applies To

OAdvise

Description

This method opens an OAdvise **object**, attaching it to a dynaset.

Usage

oresult **Open**(const ODynaset &*odyn*)

Arguments

odyn	an **ODynaset** to which you attach

Remarks

This method attaches the advisory to the dynaset *odyn*. After this call, the dynaset of *odyn* delivers messages to this advisory.

It is legal to **Open** an already open **OAdvise** object. The object is closed and then opened again.

Note that the instances of the **OAdvise** class do nothing. You will want to subclass **OAdvise** to get behavior that you want.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Opening an **OAdvise** object:

```
// create a database
ODatabase odb("ExampleDB", "scott", "tiger");

// create a dynaset
ODynaset thedynaset(odb, "select * from emp");

// default constructor of OAdvise
OAdvise adv1;

// open that advisory, attaching it to dynaset
adv1.Open(thedynaset);
```

Open (OBinder) Method

Applies To

OBinder

Description

This method opens an **OBinder** object, making it useful.

Usage

OBinder: oresult **Open**(const char **dbname*, const char **username*, const char **pwd*,
const char **sqls*, long *dynopts*)

OBinder: oresult **Open**(const ODatabase &*odb*, const char **sqls*, long *dynopts*)

Arguments

dbname	the name of the database to which you want to connect
username	the username you want to use to log into the database
pwd	the database password for the user *username*
odb	the database with which you want to open the **OBinder's** dynaset
sqls	a valid select SQL statement
dynopts	options to be used to create the dynaset

Remarks

To **Open** an **OBinder** object for work, you need to connect it to an Oracle database and select a set of records. Opening an **OBinder** object is like opening both an **ODatabase** and an **ODynaset**.

The **OBinder** object needs to connect to a database. You can supply the database directly with an **ODatabase** argument, or you can give the connection information: database name, username, and password. If you choose the latter method, note that the database will be created with ODATABASE_DEFAULT set for its options.

The **OBinder** object also needs a dynaset. The dynaset is created using the database specified by the arguments listed above, and by the *sqlstmt* and *dynopts* arguments. These arguments act identically as the arguments to an **ODynaset Open**. Note that whenever an **OBinder** opens a dynaset, it always moves immediately to the first record.

The *dynopts* argument can have the following values:

Constant	Value	Description
ODYNASET_DEFAULT	0	Accept the default behavior.
ODYNASET_NOBIND	1	Do not perform automatic binding of database parameters.
ODYNASET_KEEP_BLANKS	2	Do not strip trailing blanks from character string data retrieved from the database.
ODYNASET_READONLY	4	Force dynaset to be read-only.
ODYNASET_NOCACHE	8	Do not create a local dynaset data cache. Without the local cache, previous rows within a dynaset are unavailable; however, increased performance results during retrieval of data from the database (move operations) and from the rows

(field operations). Use this option in applications that make single passes through the rows of a dynaset for increased performance and decreased resource usage.

Options may be combined by adding their respective values.

These values can be found in the file **ORCL.H**.

Open(OBinder) calls the **PreQuery** and **PostQuery** triggers. The **OBinder Startup** trigger will not be called until the first **OBound** object is bound to the **OBinder**.

It is legal (though unusual) to open an already open **OBinder**. The **OBinder's** dynaset is closed, then reopened. Note that this does not affect an **OBinder's** bound **OBound** objects - they are still bound.

If **OBound** objects are already bound to an **OBinder**, but do not represent valid column names in the SQL query, **Open** will fail. This can happen when **OBound** objects are bound prior to calling **OBinder::Open** or when opening an already open **OBinder**.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

An example of setting up an OBinder object:

```
// construct the OBinder
OBinder empblock;

// we normally then bind OBound objects to the OBinder

// here we have several OBoundVal objects (see the Workbook)
OBoundVal salary;
OBoundVal ename;

// bind them
salary.BindToBinder(&empblock, "sal");
/*
That is the first thing bound, so the OBinder::Startup trigger is called.
*/
ename.BindToBinder(&empblock, "ename");
/*
The OBound::Startup trigger is called on each OBound as it is bound.
*/

// now open the OBinder
ODatabase odb("ExampleDB", "scott", "tiger");  // open the database
empblock.Open(odb, "select * from emp order by ename");
// that calls all the PreQuery and PostQuery triggers
```

Open (ODatabase) Method

Applies To

ODatabase

Description

This method opens an **ODatabase**, making it useful.

Usage

oresult **Open**(const OSession &*dbsess*, const char **dbname*, const char **username*, const char **pwd*, long *options* = ODATABASE_DEFAULT)

oresult **Open**(const char **dbname*, const char **username*, const char **pwd*, long *options* = ODATABASE_DEFAULT)

Arguments

dbsess	the session under which you want to open this database
dbname	the name of the database to which you want to connect
username	the username you wish to use to log into the database
pwd	the database password for the user *username*
options	options to be used to create the database object

Remarks

These methods open the **ODatabase** object. Opening an **ODatabase** object establishes a connection to an Oracle database and specifies certain kinds of behavior.

You can specify the session in which you want the database to work; if you do not specify a session, the database works in the default session. The session under which the database is opened affects connection sharing and transaction processing. Use *dbsess* to specify the session.

Use the *dbname*, *username*, and *pwd* arguments to establish the connection to the database. The *pwd* argument is allowed to be NULL. In that case it is expected that the string passed to *username* will be of the form "*username/password*"; the "slash" character must be in the string. The database name will either be a SQL*Net alias, such as "ExampleDB" or a complete Oracle database name such as "p:namedpipe-server" or "t:123.45.987.06:SID" (network protocol identifier, network address, option instance id).

The options argument can have the following values:

Constant	**Value**	**Description**
ODATABASE_DEFAULT	0	Accept the default behavior.
ODATABASE_PARTIAL_INSERT	1	Let Oracle set default field (column) values.
ODATABASE_EDIT_NOWAIT	2	Do not wait on row locks when executing a "SELECT ... FOR UPDATE".

Options may be combined by adding their respective values.

These values can be found in the file **ORCL.H**.

Two separate database objects share an Oracle database connection if they are in the same session and have the same connection information (database name, username, and password).

The options affect various aspects of the database's behavior. See **ODatabase** for more information.

It is legal to **Open** an already open **ODatabase**. The **ODatabase** is closed and then opened with the new arguments. Note that if you do this, any **ODynaset**s that were opened with the old **ODatabase** still work and are still connected to the old database object. Only the **ODatabase** handle object is affected.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

An example of opening ODatabase objects:

```
// the simplest way to open a database
ODatabase odb;
odb.Open("ExampleDB", "scott", "tiger");

// now if we open another database similarly
ODatabase odb2;
odb2.Open("ExampleDB", "scott", "tiger", ODATABASE_EDIT_NOWAIT);
/*
We have two separate database objects on the same oracle database. But
because odb2 and odb are in the same session and have the same connection
information they share a database connection. But because of different
options the two database objects will behave differently.
*/

// open a database on a named session
OSession msess("mysession");
ODatabase odb3;
odb3.Open(msess, "ExampleDB", "scott", "tiger");
/*
odb3 does not share a connection with odb, because it is on a different
session.
*/
```

Open (ODynaset) Method

Applies To

ODynaset

Description

This method asks the Oracle database for a set of records and sets up a dynaset to access them.

Usage

oresult **Open**(const ODatabase &*odb*, const char * *sqlstmt*, long *options* = ODYNASET_DEFAULT)

Arguments

odb	the database with which you want to open this dynaset
sqlstmt	a valid select SQL statement
options	options to be used to create the dynaset

Remarks

This method opens the **ODynaset** object, creating an underlying dynaset object. The dynaset is formed on records retrieved from the database represented by *odb* and the SQL select statement in *sqlstmt*. The **ODynaset** is automatically positioned at the first record after opening. The **ODynaset** copies the SQL statement, so the caller does not have to retain it.

The options argument can have the following values:

Constant	Value	Description
ODYNASET_DEFAULT	0	Accept the default behavior.
ODYNASET_NOBIND	1	Do not perform automatic binding of database parameters.
ODYNASET_KEEP_BLANKS	2	Do not strip trailing blanks from character string data retrieved from the database.
ODYNASET_READONLY	4	Force dynaset to be read-only.
ODYNASET_NOCACHE	8	Do not create a local dynaset data cache. Without the local cache, previous rows within a dynaset are unavailable; however, increased performance results during retrieval of data from the database (move operations) and from the rows (field operations). Use this option in applications that make single passes through the rows of a dynaset for increased performance and decreased resource usage.

Options may be combined by adding their respective values.

These values can be found in the file **ORCL.H**.

It is legal to **Open** an already open **ODynaset**. The **ODynaset** is closed and then opened with the new arguments.

Using a "FOR UPDATE" clause in the SQL statement that opens the dynaset requires some special attention. Please refer to Select for Update.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Examples of opening **ODynasets**:

```
// first, open an ODatabase
ODatabase odb("ExampleDB", "scott", "tiger");

// create and open a dynaset
ODynaset odyn;
odyn.Open(odb, "select * from emp");
isopen = odyn3.IsOpen();
// isopen is TRUE; the open was successful
```

Open (OSession) Method

Applies To

OSession

Description

This method sets up a session.

Usage

oresult **Open**(void);

oresult **Open**(const char **sessname*)

Arguments

sessname The name to be given to the new session

Remarks

This method opens an **OSession** object.

Every application that uses Oracle Objects for OLE is assigned a default session. The name of the default session is internally generated. By using the **Open** method with no arguments, you open an **OSession** object that refers to the default session for your application. **Open** can be called multiple times with a void argument and each time it gives you a handle of the default session. This is the only case in the class library where multiple **Opens** give you the same object.

To create another session, to have a separate transaction group, or to refer to a session by name, open the session and specify its name with the *sessname* argument. The **Open** attempts to create a new session of that name. To obtain an open **OSession** object on an existing named session, get it from an **OSessionCollection** object. It is not possible to share sessions across applications, only within applications.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Open a session:

```
// open the default session
OSession defaultsess;
defaultsess.Open();

// and open a separate session
OSession sess2;
sess2.Open("session2");

// Now if we create databases on these . . .
ODatabase odb(defaultsess, "ExampleDB", "scott", "tiger");
ODatabase odb2(sess2, " ExampleDB", "scott", "tiger ");
// . . . performing transactions on odb will not affect odb2
//  and vice versa, because they are in separate sessions.
```

operator const char *

Applies To

OField, OParameter, OValue

Description

This method returns the object's value as a string.

Usage

operator **const char ***() const

Remarks

This methods hands the value of the object back to the caller as a null-terminated string. If the object's current value is not a string, the method attempts to convert the value to a string. This can fail, resulting in the return of a NULL pointer.

The actual memory to which the returned pointer points is managed by the object. It should not be freed by the caller; it will be freed when the object is destroyed or closed or when another const char * cast is made.

Return Value

A valid, null-terminated const char pointer on success; NULL on failure.

Example

This example deletes all managers. We determine which employees are managers by casting the job field to a string.

```
// open the database
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset on the employee table
ODynaset empdyn(odb, "select * from emp");

// get an OField object for looking at the job field
OField job = empdyn.GetField("job");

// look through all the employees
while (!empdyn.IsEOF())
{
    if (0 == strcmp((const char *) job, "MANAGER"))
    { // we found a manager; delete that employee
        empdyn.DeleteRecord();
    }

    // go to next record (gets us to valid record)
    //    or past EOF if there are no more records
    empdyn.MoveNext();
}
```

operator double

Applies To

OField, OParameter, OValue

Description

This method returns the object's value as a double.

Usage

operator **double**() const

Remarks

This method hands the value of the object back to the caller as a double. If the object's current value is not a double, the method attempts to convert the value. This can fail., resulting in a return of the value 0.0.

Return Value

The value of the field as a double; 0.0 on failure.

Example

Sum all the salaries of employees:

```
// open the employee database
ODatabase odb("ExampleDB", "scott", "tiger");

// open a dynaset on the employee's table
ODynaset odyn(odb, "select sal, comm from employees");

// get a field on the salary for speed
OField salf = odyn.GetField("sal");
/*
By using the = operator in the declaration of salf we are invoking the copy
constructor.  It is copying the temporary object that is returned by the
GetField method.
*/

// sum the salaries
double sumsal = 0.0;
odyn.MoveFirst();
while (!odyn.IsEOF())
{
    sumsal += (double) salf;
    odyn.MoveNext();
}

// of course, we could have done the same thing (faster) with:
ODatabase odb("ExampleDB", "scott", "tiger");
ODynaset odyn(odb, "select sum(sal) from employees");
odyn.GetFieldValue(0, &sumsal);
// the server is good at that kind of bulk calculation
```

operator int

Applies To

OField, OParameter, OValue

Description

This method returns the object's value as an int.

Usage

operator **int**() const

Remarks

This method hands the value of the object back to the caller as an int. If the object's current value is not an int, the method attempts to convert the value. This can fail, resulting in a return of the value 0.

Return Value

The value of the field as an int; 0 on failure.

Example

Look for the employee with a certain employee number:

```
// open the ODatabase
ODatabase odb("ExampleDB", "scott", "tiger")

// open the dynaset
ODynaset odyn(odb, "select * from emp");

// get a field on the id
OField enof = odyn.GetField("empno");

// now look for the id we want
while (!odyn.IsEOF())
{
    // we'll examine the value of the id field in this record simply
    //   by casting the enof OField variable
    if ((int) enof == targetid)
        break;
}
```

operator long

Applies To

OField, **OParameter**, **OValue**

Description

This method returns the object's value as a long.

Usage

operator **long**() const

Remarks

This method hands the value of the object back to the caller as a long.   If the object's current value is not a long, the method attempts to convert the value.   This can fail, resulting in the return of the value 0.

Return Value

The value of the field as a long; 0 on failure.

Example

Look for the employee with a certain employee ID:

```
// open the ODatabase
ODatabase odb("ExampleDB", "scott", "tiger")

// open the dynaset
ODynaset odyn(odb, "select * from emp");

// get a field on the id
OField enof = odyn.GetField("empno");

// now look for the id we want
while (!odyn.IsEOF())
{
    // we'll examine the value of the id field in this record simply
    //   by casting the enof OField variable
    if ((long) enof == targetid)
        break;
}
```

operator=

Applies To

OAdvise, OClient, OConnection, OConnectionCollection, ODatabase, ODynaset, ODynasetMark, OField, OFieldCollection, OParameter, OParameterCollection, OSession, OSessionCollection, OValue

Description

This method assigns one object to another.

Usage

OAdvise &OAdvise::**operator =**(const OAdvise &*other*)

OClient &OClient::**operator =**(const OClient &*other*)

OConnection &OConnection::**operator =**(const OConnection &*other*)

OConnectionCollection &OConnectionCollection::**operator =**(const OConnectionCollection &*other*)

ODatabase &ODatabase::**operator =**(const ODatabase &*other*)

ODynaset &ODynaset::**operator =**(const ODynaset &*other*)

ODynasetMark &ODynasetMark::**operator =**(const ODynasetMark &*other*)

OField &OField::**operator =**(const OField &other)

OFieldCollection &OFieldCollection::**operator =**(const OFieldCollection &*other*)

OParameter &OParameter::**operator =**(const OParameter &*other*)

OParameterCollection &OParameterCollection::**operator =**(const OParameterCollection &*other*)

OSession &OSession::**operator =**(const OSession &*other*)

OSessionCollection &OSessionCollection::**operator =**(const OSessionCollection &*other*)

OValue &OValue::**operator =**(const OValue &*other*)

Arguments

other	the object that is the source of the assignment

Remarks

The assignment operator makes the object a copy of another object.

For the classes **OAdvise**, **OClient**, **OConnection**, **OConnectionCollection**, **ODatabase**, **ODynaset**, **OField**, **OFieldCollection**, **OParameter**, **OParameterCollection**, **OSession**, and **OSessionCollection**, a copied object becomes another handle that refers to the same underlying implementation object. Strings owned by the source object are not copied to the destination object (although the same information - such as database name and SQL statement - are available from the new object).

Note: For **OField**, what is being copied is *not* the value of the field, but the **OField** handle itself.

ODynasetMark and **OValue** are simpler objects. The data of the source object is simply copied.

If the object is already open, it is closed before the assignment. As a result, if the assignment fails, the return value of the operation will be a closed object.

The work that is done by assigning is the same as for a copy constructor.

OBinder and **OBound** have **operator=** defined in the header file, but the operator is not implemented. This is done so that the compiler's default implementation will not be used (it would be incorrect). If you want an assignment operator for your **OBinder** or **OBound** subclass, you must implement it.

Return Value

The object that was assigned to.

Example

An illustration of the meaning of assignment:

```
// open an ODatabase
ODatabase odb("ExampleDB", "scott", "tiger");

// open an ODynaset
ODynaset odyn(odb, "select ename, sal, comm from employees");

// get a field on sal and commission
OField salfield = odyn.GetField("sal");
OField commfield = odyn.GetField("comm");

// declare some OValue variables
OValue salval;
OValue commval;

// now look at the values of the first record
odyn.GetFieldValue("sal", &salval);
odyn.GetFieldValue("comm", &commval);
// let us say that salval contains 5000 and commval contains 300

salval = bonusval;  // assign commission value to salary
// now salval contains 300

// can we do the same with OFields? NO!
int isal = (int) salfield;  // isal is now 5000
int ibonus = (int) commfield; // ibonus is now 300

salfield = commfield; // assign comm OField to salary OField
// NOTE: we have only assigned the OField variable
int isal2 = (int) salfield;  // isal2 is 300
/*
isal2 is 300 because salfield is now referring to the field "comm" in the
record.
*/

// now update the record
odyn.StartEdit()
salfield.SetValue(4000);
odyn.Update();
// we have just set the "comm" field to 4000, not the "sal" field
```

operator==

Applies To

OAdvise, **OClient**, **OConnection**, **OConnectionCollection**, **ODatabase**, **ODynaset**, **ODynasetMark**, **OField**, **OFieldCollection**, **OParameter**, **OParameterCollection**, **OSession**, **OSessionCollection**, **OValue**

Description

Equivalence operator

Usage

int OOracleObject::**operator==**(const OOracleObject &*other*)

int ODynasetMark::**operator==**(const ODynasetMark &*other*)

int OValue::**operator==**(const OValue &*other*)

Arguments

other	the other object to which this object is being compared

Remarks

For the subclasses of **OOracleObject** - namely **OAdvise**, **OClient**, **OConnection**, **OConnectionCollection**, **ODatabase**, **ODynaset**, **ODynasetMark**, **OField**, **OFieldCollection**, **OParameter**, **OParameterCollection**, **OSession**, and **OSessionCollection** - two objects are the same if they refer to the same underlying implementation object. If one object was assigned from the other or copy constructed from the other, or if they were both obtained from some other object in the same way, they are equal.

If either of the objects is closed (and even if they are both closed), they are considered unequal.

Two **OValue** objects are equal if their values are equal. This equality crosses over type boundaries: integer 34 equals double 34.0 equals string "34".

Return Value

1 if the objects are equal; 0 if they are not.

operator!=

Applies To

OAdvise, **OClient**, **OConnection**, **OConnectionCollection**, **ODatabase**, **ODynaset**, **ODynasetMark**, **OField**, **OFieldCollection**, **OParameter**, **OParameterCollection**, **OSession**, **OSessionCollection**, **OValue**

Description

Nonequivalence operator

Usage

int OOracleObject::**operator!=**(const OOracleObject &*other*)

int ODynasetMark::**operator!=**(const ODynasetMark &*other*)

int OValue::**operator!=**(const OValue &*other*)

Arguments

other	the other object to which this object is being compared

Remarks

This routine returns the opposite of **operator==**.

Return Value

1 if the objects are not equal; 0 if they are equal.

OSession Method

Applies To

OSession

Description

OSession constructor

Usage

OSession(void)

OSession(const OSession &*othersess*)

OSession(const char **sname*)

Arguments

othersess	another **OSession** object that you are copying
sname	the name you wish to give to this session

Remarks

These methods construct a new **OSession** instance.

The default constructor constructs an unopened **OSession** object.   It cannot fail.   You must open the object before you can use it.

The copy constructor copies another **OSession** object. If that other **OSession** object is open - which means that it is a handle on an implementation session object - the new **OSession** object becomes a handle to that same session object.   The copy constructor copies the reference to the session object but does not copy any strings that the source **OSession** may own.   The copy constructor can fail; check whether the new **OSession** is open after the constructor call.

The constructor that takes an argument of *sname* constructs the **OSession** and then attempts to open it.   It will be opened with the name *sname*.   *Sname* can be NULL, in which case the default **OSession** is returned.   This construct and open constructor can fail; check whether the new **OSession** is open after the constructor call.

Example

Construct and open the application's default session:

```
OSession sess(0);
```

~OSession Method

Applies To

OSession

Description

OSession destructor

Usage

~**OSession**(void)

Remarks

This method destroys the **OSession** and frees its resources.   The underlying implementation session will be freed if this is the last object that is referring to it.

When the **OSession** is destroyed any strings that the **OSession** owns (such as session name) are also freed.

OSessionCollection

Applies To

OSessionCollection

Description

OSessionCollection constructor

Usage

OSessionCollection(void)

OSessionCollection(const OSessionCollection &*othercoll*)

Arguments

othercoll	another **OSessionCollection** object that you are copying

Remarks

These methods construct a new **OSessionCollection** instance.

Constructing an **OSessionCollection** does not create any sessions or **OSession** objects.

The default constructor constructs an unopened **OSessionCollection** object.

The copy constructor copies another **OSessionCollection** object.   If that other **OSessionCollection** object is open - which means it is a handle on an implementation sessioncollection object - the new **OSessionCollection** object becomes a handle to that same sessioncollection object.   The copy constructor can fail; check whether the new **OSessionCollection** is open after the constructor call.

There is no **Open** method for the **OSessionCollection** class.   To get an open **OSessionCollection**, call the **GetSessions** method.

~OSessionCollection

Applies To

OSessionCollection

Description

OSessionCollection destructor

Usage

~**OSessionCollection**(void)

Remarks

This method destroys the **OSessionCollection** and frees its resources.   The underlying implementation sessioncollection object will be freed if this is the last object that is referring to it.

Destroying the **OSessionCollection** object has no effect on any sessions or **OSession** objects, even those gotten from the destroyed collection

OShutdown Method

Applies To

No class.   This is a standalone routine.

Description

This method cleans up and shuts down the C++ class library.

Usage

void **OShutdown**(void)

Remarks

Before your application exits it should call OShutdown to clean up the C++ class library. This routine performs the per-process cleanup.

OLE users: If **OStartup** successfully called OleInitialize, **OShutdown** will call OleUninitialize.

OStartup Method

Applies To

No class.   This is a standalone routine.

Description

This method initializes the C++ class library

Usage

oboolean **OStartup**(void)

Remarks

This routine initializes the C++ class library for this process.   It must be called for every process.

OLE users: **OStartup** calls OleInitialize.   It remembers whether OLE was already running. **OShutdown** calls OleUnitialize only if **OStartup** was the call that started OLE.

Example

Start the C++ class library:

```
OStartup()
```

OValue Method

Applies To

OValue

Description

OValue constructor

Usage

OValue(void)

OValue(const OValue &*otherval*)

OValue(int *val*)

OValue(long *val*)

OValue(double *val*)

OValue(const char **val*)

Arguments

otherval	another **OValue** object whose value you are copying
val	a value that you are placing into the **OValue**

Remarks

These methods construct **OValue** objects.

The default constructor method constructs an **OValue** with a value of NULL.

The copy constructor copies the value of the other **OValue** object.   The copy constructor can fail, in which case the constructed **OValue** has a value of NULL.

The rest of the constructors allow the **OValue** to be initialized with values of various types.   The initialization with a string value can fail (because of memory allocation failure), in which case the **OValue** has a value of NULL.

Example

Construct several **OValues**:

```
OValue str45("45");
OValue int45(45);
OValue long45(45L);
OValue double45(45.0);
OValue val45(str45);
OValue valnull;

// str45, int45, long45, double45 and val45 are all
//    equal according to operator==
```

~OValue Method

Applies To

OValue

Description

OValue destructor

Usage

~**OValue**(void)

Remarks

This method destroys the **OValue** and frees its resources. When the **OValue** is destroyed, any strings that it owns (such as strings that have been obtained by way of a *const char* * cast) are also freed.

Refresh (OBinder) Method

Applies To

OBinder

Description

This method causes all the **OBound** objects to be given their values again.

Usage

oresult **Refresh**(void)

Remarks

The method calls the **Refresh** method on every bound **OBound** object. This causes them to get their values again. In normal processing you do not need to call **Refresh**.

This **Refresh** method does not refresh the dynaset that is being managed by the **OBinder**. To refresh such a dynaset, use the **RefreshQuery** method.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Refresh (OBound) Method

Applies To

OBound

Description

This method sets the value of the **OBound** object with a new value from the database.

Usage

virtual oresult **Refresh**(const OValue &*val*)

Arguments

val	the new value of the **OBound** object

Remarks

The **OBinder** (to which this **OBound** object is bound) calls this method whenever a value needs to be transferred from the dynaset to the **OBound** object. The *val* argument contains the new value.

Refresh is not implemented in the base **OBound** class. Your **OBound** subclass must implement this method

Refresh is the inverse of **OBound::SaveChange**.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Please see the *Workbook* for the example "OBound of a variable." That example works through the implementation of a subclass of **OBound**. That subclass implements **Refresh**.

Refresh (ODynaset) Method

Applies To

ODynaset

Description

This method executes the dynaset's SQL statement and fetches a new set of records.

Usage

oresult **Refresh**(void)

Remarks

This method executes the current SQL statement, which is set by the **Open** method and reset by the **SetSQL** method. Execution of this method discards the current result set and the current local cache and returns to the database to get new values. After refreshing, the current record becomes the first record in the result set (which may not be the same as the first record of the previous result set). Execution of this method sends an OADVISE_REFRESH message to all attached advisories.

If the dynaset is being managed by an **OBinder**, this method calls the **PreQuery** and **PostQuery** triggers.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

A powerful use of **Refresh** is with parameters. When the value of the parameter changes, we can **Refresh** the dynaset and get a different set of records.

```
// open an ODatabase
ODatabase odb("ExampleDB", "scott", "tiger");

// get the parameter collection
OParameterCollection params = odb.GetParameters();

// add a parameter for department number
params.Add("dno", 10, OPARAMETER_INVAR, OTYPE_NUMBER);

// now set up a dynaset that uses that parameter
ODynaset odyn(odb, "select * from emp where deptno = :dno");

// do some processing with that dynaset

// now we want to look at records from another department

// get the parameter and set its value to 20
params.GetParameter("dno").SetValue(20);

// and refresh the dynaset to get the new records
odyn.Refresh();
/*
We can get different sets of records without manipulating SQL statements.
*/
```

RefreshQuery Method

Applies To

OBinder

Description

This method refreshes the dynaset that the **OBinder** is managing.

Usage

oresult **RefreshQuery**(void)

Remarks

This method calls the **ODynaset::Refresh** method on the dynaset that the **OBinder** is managing.   See the description of **ODynaset::Refresh** for a description of the side effects.

Execution of this method calls the **PreQuery** and **PostQuery** triggers.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Remove Method

Applies To

OParameterCollection

Description

This method removes a parameter from a database.

Usage

oresult **Remove**(int *index*) const

oresult **Remove**(const char **pname*) const

Arguments

index	An index from 0 to **OParameterCollection**.GetCount()-1
pname	the name of the parameter, as stated when the parameter was created with Add

Remarks

Parameters are attached to a database using **OParameterCollection::Add**. Once attached they will stay in existence. When you no longer need a parameter you can remove it using the **Remove** method. This will reduce overhead in the processing of SQL statements.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Adding and removing a parameter

```
// open a database
ODatabase odb("ExampleDB", "scott", "tiger");

OParameterCollection pcoll = odb.GetParameters();

// add a parameter
pcoll.Add("param1", 34, OPARAMETER_INVAR, OTYPE_NUMBER);

// now remove it
pcoll.Remove("param1");
```

ResetTransaction Method

Applies To

OSession

Description

This method rolls back the current transaction unconditionally.

Usage

oresult **ResetTransaction**(void)

Remarks

ResetTransaction is the same as **Rollback**, with the exception that no advisory messages are issued.   As a result, **ResetTransaction** cannot be canceled by an advisory.

Please see **Rollback** for more information.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Rollback Method

Applies To

OSession

Description

This method rolls back the current transaction.

Usage

oresult **Rollback**(oboolean *startnew* = FALSE)

Arguments

startnew	If TRUE a new transaction is begun (as if **BeginTransaction** had been called). If FALSE, no additional work is done after the transaction is committed.

Remarks

A database transaction is a way to group database operations so that they all either succeed or fail together. Please see "Transactions" for more details. **BeginTransaction** starts a transaction. You can terminate the transaction either with a **Commit** or a **Rollback**. It is an error to call **Rollback** when no transaction is in progress.

Calling **Rollback** results in OADVISE_ROLLBACK messages being sent to all advisories attached to all dynasets within this session. Any **OBinder** or **OBound** objects within the session will have **PreRollback** and **PostRollback** triggers called.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example starts a transaction and begins a long sequence of operations. If an error occurs along the way, all changes are discarded with a **Rollback**. If they all succeed, the changes are made permanent with a **Commit**.

```
// routine to give all employees the same salary
void Transfer(ODynaset empdyn, double newsal)
{
    // get the session of this dynaset
    OSession empsess = empdyn.GetSession();

    // start a transaction
    empsess.BeginTransaction();

    // edit every record (with StartEdit, SetFieldValue, Update)
    empdyn.MoveFirst();
    while (!empdyn.IsEOF())
    {
        if (empdyn.StartEdit() != OSUCCESS)
            break;
        if (empdyn.SetFieldValue("sal", newsal) != OSUCCESS)
            break;
        if (empdyn.Update() != OSUCCESS)
            break;
        empdyn.MoveNext();  // go to the next record
    }

    if (!empdyn.IsEOF())
    { // we got out of the loop early.  Get rid of
      //   any changes we made
        empsess.Rollback();
    }
    else
    { // everything worked.  Make it all permanent
        empsess.Commit();
    }
    return;
}
```

SaveChange Method

Applies To

OBound

Description

This method saves the value of the **OBound** object to the database.

Usage

virtual oresult **SaveChange**(void)

Remarks

The **OBinder** (to which this **OBound** object is bound) calls this method when it is time to save the object's value to the database.   The **OBinder** object knows that the value needs to be saved because of some previous call to **Changed**.

SaveChange is not implemented in the base **OBound** class.   Your **OBound** subclass must implement this method.   Normally you can implement **SaveChange** by calling the protected **OBound::SetValue** method.

The inverse of **SaveChange** is **OBound::Refresh**.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Please see the *Workbook* for the example "OBound of a variable."   That example works through the implementation of a subclass of **OBound**.   That subclass implements **SaveChange**.

ServerErrorNumber Method

Applies To

OSession, **ODatabase**

Description

This method returns the Oracle7 error number for the last database-related error in the session.

Usage

long **ServerErrorNumber**(void) const;

Remarks

This method returns the Oracle7 error number for the most recent database-related error, if there has been an error since the session was created or the server error was reset with **ServerErrorReset**. Errors that occur while opening a database or in a transactional method will be reported on the session. Other errors are reported on the database.

A value of 0 means no error.

Return Value

Returns the Oracle error number.

Example

An example of a server error:

```
// open an ODatabase object
ODatabase odb("ExampleDB", "scott", "tiger");

// try to open a dynaset with a bad column name
ODynaset dyn(odb, "select xx from emp");

// if that didn't work, get the error number
if (!dyn.IsOpen())
{
    long errno = odb.ServerErrorNumber();
}
```

ServerErrorReset Method

Applies To

OSession, **ODatabase**

Description

This method clears the last remembered Oracle7 database-related error for the session.

Usage

oresult **ServerErrorReset**(void);

Remarks

After calling this method **ServerErrorNumber** will return 0 (indicating no error) until the next database-related error occurs.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

ServerErrorSQLPos Method

Applies To

ODatabase

Description

When an error occurs during parsing of a SQL statement, this method returns the position within the SQL statement where the error occurred.

Usage

int **ServerErrorSQLPos**(void) const;

Remarks

Whenever the server detects an error while parsing an SQL statement, for instance during **ODatabase::ExecuteSQL** or **ODynaset::Open**, the position within the SQL string where the error occurred is remembered and can be obtained using this method. The position is 0-based; the first character in the SQL statement is character 0.   If no error has occurred, or no SQL statements have been parsed by this database or since the last call to **ServerErrorReset**, then -1 is returned.

Return Value

Returns the character position of the SQL parse error (0 based) or -1 if there was no error.

Example

An example of a SQL parse error:

```
// open an ODatabase object
ODatabase odb("ExampleDB", "scott", "tiger");

// execute a bad SQL statement
odb.ExecuteSQL("xxzzz");

int sqlpos = odb.ServerErrorSQLPos();
// sqlpos will be 0

// execute another bad SQL statement
odb.ExecuteSQL("drop zzz");
// the zzz doesn't make sense
sqlpos = odb.ServerErrorSQLPos();
// sqlpos will be 5
```

SetFieldValue Method

Applies To

ODynaset

Description

This method sets the value of a field.

Usage

oresult **SetFieldValue**(int *index*, const OValue &*val*)

oresult **SetFieldValue**(const char **fieldname*, const OValue &*val*)

oresult **SetFieldValue**(int *index*, int *val*)

oresult **SetFieldValue**(const char **fieldname*, int *val*)

oresult **SetFieldValue**(int *index*, long *val*)

oresult **SetFieldValue**(const char **fieldname*, long *val*)

oresult **SetFieldValue**(int *index*, double *val*)

oresult **SetFieldValue**(const char **fieldname*, double *val*)

oresult **SetFieldValue**(int *index*, const char **val*)

oresult **SetFieldValue**(const char **fieldname*, const char **val*)

oresult **SetFieldValue**(int *index*, const void __huge **blobp*, long *bloblen*)

oresult **SetFieldValue**(const char **fieldname*, const void __huge **blobp*, long *bloblen*)

Arguments

index	the 0-based index of the field. The index is the position of the field in the SQL query that created the current record set.
fieldname	the name of the field, as expressed in the SQL query
val	the new value for the field, in one of various types
blobp	pointer to the long data to be placed in the field
bloblen	the number of bytes from *blobp* to be placed into the field

Remarks

These methods set the values of fields of data in the query result set. They change the values in the current record. You should call the **StartEdit** method on the record before any **SetFieldValue** calls are made. The field is specified either by *index* (position in the SQL query) or by *fieldname*. The data values can be set using data of any of the following types: **OValue**, int, long, double, and const char *. Setting a fields value to an empty string (namely "") sets the field to NULL.

The last two methods (the ones with the *blobp* arguments) are used to place more than 64K bytes of data into a long or long raw field. They are valid only for fields with server types of OTYPE_LONG or OTYPE_LONGRAW. Such fields can also have their values set with (const char *), but only up to 64K.

The **SetFieldValue** methods can fail under several conditions:

- if the current record is not valid
- if the indicated field does not exist
- if **StartEdit** has not been called on the current record (or **AddNewRecord**, or

DuplicateRecord)

- if the type of the data being set cannot be coerced into the type needed by the database (for example trying to put a boring string into a number).

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

A snippet of code to give somebody a 7% raise:

```
// given an open ODynaset named empdyn that is on the right record

// get the current salary
double salary;
empdyn.GetFieldValue("sal", &salary);

// change it
empdyn.StartEdit();
empdyn.SetFieldValue("sal", salary*1.07);
empdyn.Update();
```

SetSQL Method

Applies To

OBinder, ODynaset

Description

This method sets the SQL statement that will be used the next time the dynaset is refreshed.

Usage

oresult **SetSQL**(const char **sqls*)

Arguments

sqls	the new SQL statement

Remarks

OBinder: This method sets the SQL statement to be used the next time **RefreshQuery** is called.

ODynaset: This method sets the SQL statement to be used on the next **Refresh**. The **ODynaset** copies the SQL statement, so the calling routine does not have to retain it. (See **GetSQL** for more discussion.)

The result set of a dynaset is determined by the SQL statement that is executed to fetch the records. This method enables you to reset the SQL statement of the dynaset. Resetting the SQL statement does not immediately change the dynaset's result set.

The dynaset's result set can be changed to correspond to the new SQL statement by calling **ODynaset::Refresh** or **OBinder::RefreshQuery**.

Setting the SQL statement invalidates any pointers to previously returned SQL statements (from **GetSQL**).

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Change the SQL statement of a dynaset:

```
// open an ODatabase object
ODatabase odb("ExampleDB", "scott", "tiger");

// open an ODynaset
ODynaset odyn(odb, "select * from emp order by ename");

// ... do some processing ...

// now get the records again, but in a different order
odyn.SetSQL("select * from emp order by empno");
odyn.Refresh();
```

SetValue Method

Applies To

OField, **OParameter**, **OValue**

Description

This method sets the value of the object. Setting an **OField** value also sets a database field value.

Usage

oresult **SetValue**(const OValue &*val*)

oresult **SetValue**(int *val*)

oresult **SetValue**(long *val*)

oresult **SetValue**(double *val*)

oresult **SetValue**(const char *val)

oresult **SetValue**(const void __huge *blobp, long bloblen) (**OField** only)

Arguments

val	the new value, in one of a variety of types.
blobp	pointer to the long data to be placed in the field
bloblen	the number of bytes from *blobp* to be placed into the field

Remarks

These methods set the value of the object.

If the new value is passed in as a string, the object copies the string. The caller does not have to retain the string.

Setting the value invalidates any pointers returned from a previous (const char *) cast.

Setting the value of an **OField** is legal only when the related dynaset has an edit mode of ODYNASET_EDIT_NEWRECORD (from an **AddNewRecord** or **DuplicateRecord**) or ODYNASET_EDIT_EDITING (from **StartEdit**). Setting the value of an **OField** sets the value of the field the **OField** is on, in the current record, in the Oracle database.

The last method (the one with the *blobp* argument) is used to place more than 64K bytes of data into a long or long raw field. It is valid only for fields with server types of OTYPE_LONG or OTYPE_LONGRAW. Such fields can also have their values set with (const char *), but only up to 64K.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Set the values of some objects:

```
// open a database
ODatabase odb("ExampleDB", "scott", "tiger");

// add a parameter to odb
OParameterCollection params = odb.GetParameters();
OParameter deptp;
```

```
deptp = params.Add("dno", 20, OPARAMETER_INVAR, OTYPE_NUMBER);

// set the parameter's value to 10 instead
deptp.SetValue(10);

// use it
ODynaset empdyn;
oresult ores;
empdyn.Open(odb, "select * from emp where deptno = :dno");

// get an OField on the salary field
OField salf = empdyn.GetField("sal");

// and change the salary of the current record
empdyn.StartEdit();
salf.SetValue(4500.0);
empdyn.Update();
```

SetValue (OBound) Method

Applies To

OBound

Description

This method sets the value of the database field to which this bound object is bound, in the current record.

Usage

OBound: oresult **SetValue**(const OValue &*val*)

Arguments

val	new value of the database field

Remarks

This protected method is provided as a convenience for **OBound** subclasses. It is normally used to implement the subclass's **SaveChange** method.

This method should be called only in response to a call to **SaveChange**.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

Please see the *Workbook* for the example "OBound of a variable." That example works through the implementation of a subclass of **OBound**. That subclass uses **SetValue** in its implementation of **SaveChange**.

StartEdit Method

Applies To

ODynaset

Description

This method begins the process of editing the current record.

Usage

oresult **StartEdit**(void)

Remarks

Editing an existing record in a dynaset consists of three steps:

1. Call StartEdit.
2. Change field values, either with **SetFieldValue** or **SetValue**.
3. Call Update.

Calling **StartEdit** informs the **ODynaset** that you are going to edit the values of the current record. The **ODynaset** attempts to obtain a lock on the record from the Oracle database so that no other user can edit the record at the same time. The precise behavior depends on the database ODATABASE_EDIT_NOWAIT option (see **ODatabase**).

Once a lock is obtained the values of the record's fields in the database are compared to the what the dynaset thinks the values are (with the exception of long fields). If there is a difference it is assumed that some other user has changed the data in the database since the dynaset fetched the record. If a difference is found **StartEdit** will fail with an OERROR_DATACHANGE error.

This can be misleading. Consider a table where some of the records have default values set in the database or whose values are set by database triggers. When the records of such a table are updated to the database (after being added or edited by the dynaset) the values of some records may change in the database. If we then execute **StartEdit** on this we will fail with OERROR_DATACHANGE because the database values do not match the dynaset's values. To avoid this problem use the ODATABASE_PARTIAL_INSERT option on the database (see **ODatabase**). With this option turned on the dynaset will refetch the record after the database has had a chance to change it.

Note: A call to **StartEdit**, **AddNewRecord**, **DuplicateRecord**, or **DeleteRecord**, will cancel any outstanding **StartEdit**, **AddNewRecord** or **DuplicateRecord** calls before proceeding. Any outstanding changes not saved using **Update** will be lost during the cancellation.

If the current query for the database resulted in a nonupdatable dynaset, this method fails.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example sets the salary in the current record to $9985.

```
// we have a dynaset named empdyn

// edit the salary
empdyn.StartEdit();
empdyn.SetFieldValue("sal", 9985.0);
empdyn.Update();
```

StatusChange Method

Applies To

OAdvise

Description

The **StatusChange** method is called by a dynaset when that dynaset's status changes.

Usage

void **StatusChange**(int *statustype*)

Arguments

statustype

statustype will have one of the values:

OADVISE_FOUNDLAST	// dynaset has come to last record

Remarks

You do not call **StatusChange**; rather, the **StatusChange** method of your **OAdvise** subclass is called.

When you subclass **OAdvise**, you can override the **StatusChange** method. After an instance of your **OAdvise** subclass is attached to a dynaset (by way of the **OAdvise::Open** method), your instance receives calls to its **StatusChange** method. Use a **StatusChange** method to perform processing when a dynaset's status has changed.

In the current release, the only status change is that which occurs when the dynaset finds the last record. This can occur on a **MoveLast** or on a **MoveNext** that attempts to move past the last record, or on a **GetRecordCount**. The unoverridden **StatusChange** method does nothing.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example notifies the application that the entire dynaset has been downloaded to the client.

```
void YourOAdvise::StatusChange(int statustype)
{
    if (statustype == OADVISE_FOUNDLAST)
        m_appcontext->HaveAllRecords();
    return;
}
```

Unbind Method

Applies To

OBound

Description

This method breaks the connection between the **OBound** object and the **OBinder** to which it has been bound.

Usage

oresult **Unbind** (void)

Remarks

This method unbinds the **OBound** object and calls its **Shutdown** trigger.   The **Shutdown** trigger can return OFAILURE, in which case the **Unbind**ing is canceled.

If the **Unbind**ing is successful, neither the **OBinder** nor the **OBound** is destroyed.   Only their relationship is terminated.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

UnbindObj Method

Applies To

OBinder

Description

This method breaks the connection between the **OBinder** and one of its **OBound** objects.

Usage

oresult **UnbindObj**(OBound **bobj*, oboolean *nofail* = FALSE)

Arguments

bobj	the **OBound** object that is being unbound
nofail	TRUE if you unbind despite **Shutdown** trigger failure; FALSE if you are willing to have the **Unbind** fail.

Remarks

This method unbinds the **OBound** object **bobj* and calls the **Shutdown** trigger on **bobj*. If *nofail* is TRUE, this method unbinds the object even if the **Shutdown** trigger returns OFAILURE - that is, the unbinding cannot fail.   If *nofail* is FALSE and the **Shutdown** trigger returns OFAILURE, the object is not unbound.

If the **Unbind**ing is successful, neither the **OBinder** nor the **OBound** is destroyed.   Only their relationship is terminated.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Update Method

Applies To

ODynaset, OBinder

Description

This method saves dynaset changes to the Oracle database.

Usage

oresult **Update**(void)

Remarks

For **ODynaset**, **Update** is the end of the three-part record editing sequence:

1. Call StartEdit.
2. Change field values, either with **SetFieldValue** or **SetValue**.
3. Call Update.

Calling **Update** saves to the Oracle database the changes that have been made. If you are in the middle of a transaction (that is, your session had **BeginTransaction** called on it), the changes are not made permanent until a **Commit** is done. Alternatively, the changes can be canceled with a **Rollback**.

For **OBinder**, **Update** saves any changes that have been marked in the current record. This results in the same behavior as **ODynaset::Update**.

ODATABASE_PARTIAL_INSERT determines precisely what happens to values of fields that have not been explicitly set. See **ODatabase** for details.

Calling **Update** sends OADVISE_UPDATE messages to all advisories attached to the dynaset. If the dynaset is being managed by an **OBinder** object, **PreUpdate** and **PostUpdate** triggers are called.

Return Value

An oresult indicating whether the operation succeeded (OSUCCESS) or not (OFAILURE).

Example

This example sets the salary in the current record to $9985.

```
// we have a dynaset named empdyn

// edit the salary
empdyn.StartEdit();
empdyn.SetFieldValue("salary", 9985.0);
empdyn.Update();
```

PostAdd Trigger Method

Applies To

OBinder, **OBound**

Description

Called after a new record is added

Usage

virtual oresult **PostAdd**(void)

Remarks

The default **OBound PostAdd** trigger will refresh the value of the **OBound**.

The default **OBinder PostAdd** trigger does nothing.

Adding a record may happen either because of **AddNewRecord** or **DuplicateRecord**.

PostDelete Trigger Method

Applies To

OBinder, **OBound**

Description

Called after a record is deleted

Usage

virtual oresult **PostDelete**(void)

Remarks

The default **OBound PostDelete** trigger does nothing.

The default **OBinder PostDelete** trigger does nothing.

PostMove Trigger Method

Applies To

OBinder, **OBound**

Description

Called after moving to a new record

Usage

virtual oresult **PostMove**(void)

Remarks

The default **OBound PostMove** trigger will refresh the value of the **OBound**.

The default **OBinder PostMove** trigger does nothing.

PostQuery Trigger Method

Applies To

OBinder, OBound

Description

Called after an SQL statement has been executed

Usage

virtual oresult **PostQuery**(void)

Remarks

The default **OBound PostQuery** trigger will refresh the value of the **OBound**.

The default **OBinder PostQuery** trigger does nothing.

PostRollback Trigger Method

Applies To

OBinder, **OBound**

Description

Called after a session rollback

Usage

virtual oresult **PostRollback**(void)

Remarks

The default **OBound PostRollback** will refresh the value of the **OBound**.

The default **OBinder PostRollback** trigger does nothing.

PostUpdate Trigger Method

Applies To

OBinder, **OBound**

Description

Called after the database is updated

Usage

virtual oresult **PostUpdate**(void)

Remarks

The default **OBound PostUpdate** trigger does nothing.

The default **OBinder PostUpdate** trigger does nothing.

PreAdd Trigger Method

Applies To

OBinder, OBound

Description

Called before a new record is added

OBinder: This trigger method is called before a new record is added. By default, if changes have been made, the method saves them (by calling **OBound::SaveChange** on every object) and updates the record (which as a side effect calls the update triggers).

OBound: This trigger method is called before a new record is added.

Usage

virtual oresult **PreAdd**(void)

Remarks

The default **OBound PreAdd** trigger does nothing.

The default **OBinder PreAdd** trigger saves to the database any changes to the current record.   If there were any changes that needed saving this will call the update triggers as a side effect.

Adding a record my happen either because of **AddNewRecord** or **DuplicateRecord**.

PreDelete Trigger Method

Applies To

OBinder, **OBound**

Description

Called before a record is deleted.

Usage

virtual oresult **PreDelete**(void)

Remarks

The default **OBound PreDelete** trigger does nothing.

The default **OBinder PreDelete** trigger does nothing.

PreMove Trigger Method

Applies To

OBinder, **OBound**

Description

Called before moving to another record

Usage

virtual oresult **PreMove**(void)

Remarks

The default **OBound PreMove** trigger does nothing.

The default **OBinder PreMove** trigger saves to the database any changes to the current record.   If there were any changes that needed saving this will call the update triggers as a side effect.

PreQuery Trigger Method

Applies To

OBinder, **OBound**

Description

Called before an SQL statement is executed

Usage

virtual oresult **PreQuery**(void)

Remarks

The default **OBound PreQuery** trigger does nothing.

The default **OBinder PreQuery** trigger does nothing.

PreRollback Trigger Method

Applies To

OBinder, **OBound**

Description

Called before a session rollback

Usage

virtual oresult **PreRollback**(void)

Remarks

The default **OBound PreRollback** trigger does nothing.

The default **OBinder PreRollback** trigger does nothing.

PreUpdate Trigger Method

Applies To

OBinder, **OBound**

Description

Called before a record is saved to the database

Usage

virtual oresult **PreUpdate**(void)

Remarks

The default **OBound PreUpdate** trigger does nothing.

The default **OBinder PreUpdate** trigger does nothing.

Shutdown Trigger Method

Applies To

OBinder, OBound

Description

Called when the **OBinder** is closed, or on an **OBound** when it is unbound

Usage

virtual oresult **Shutdown**(void)

Remarks

The default **OBound Shutdown** trigger does nothing.

The default **OBinder Shutdown** trigger saves to the database any changes to the current record.   If there were any changes that needed saving this will call the update triggers as a side effect.

Startup Trigger Method

Applies To

OBinder, OBound

Description

Called when the object binding starts.

OBinder: This trigger method is called at the time the first bound object is added to the binder. It is guaranteed only be called only once for the **OBinder** instance.

OBound: This trigger method is called immediately after a bound object is bound to a binder.   If the binder has not been opened, then **GetValue** cannot be called by this method.

Usage

virtual oresult **Startup**(void)

Remarks

For an **OBinder** object this method will be called the first time that a bound object is successfully added to the **OBinder**.   It will only be called once.

For an **OBound** this method will be called when the object is bound to an **OBinder**. Note that if the **OBinder** object has not been opened yet (usually the case) then **GetValue** cannot be called by this method.

The default **OBound Startup** trigger does nothing.

The default **OBinder Startup** trigger does nothing.