15

Oracle8 and Oracle8i

Wouldn't life be a lot easier if everyone used the same operating system, the same Internet browser software and the same relational database? Unfortunately things aren't that simple, so we need to be as flexible as possible. This book introduces all the main databases and data sources that you, as an ASP developer, may come across. Here, we will cover one of the most important relational databases – Oracle.

In this chapter we will cover the basics of configuring a web server to connect to a remote Oracle8 or Oracle8i database, although many of the techniques apply equally well to previous versions of Oracle 7.x. We'll show you how to use the common **ODBC Drivers** and **OLE DB Providers** in conjunction with ActiveX Data Objects (ADO) to manipulate data stored within an Oracle database from within an ASP application.

We'll build upon each area by creating a collection of ASP scripts that will use the sample `scott` database schema showing how to retrieve and update data in a flexible and more importantly, scalable manner. We will finally cover the (unfortunately), *advanced* concept of retrieving **recordsets** from an Oracle **stored procedure**. Don't worry though, it's relatively easy to use scalar `INPUT` and `OUTPUT` parameters to return individual parameters from an Oracle stored procedure – which we'll also be covering in this chapter.

> **All of the SQL and ASP code for this sample application can be downloaded from the Wrox web site at** `http://www.wrox.com`**.**

A Brief History of Oracle

Way back in June 1970, Dr E F Codd published a paper *entitled A Relational Model of Data for Large Shared Data Banks*. This relational model, sponsored by IBM, then came to be accepted as the definitive model for relational database management systems – RDBMS. The language developed by IBM to manipulate the data stored within Codd's model was originally called Structured English Query Language, or SEQUEL, with the word 'English' later being dropped in favor Structured Query Language – SQL.

In 1979 a company called Relational Software, Inc. released the first commercially available implementation of SQL. Relational Software later came to be known as Oracle Corporation.

Oracle Version 8

You'll find versions of Oracle8 available for many of today's popular computing environments, in particular Windows, UNIX and Linux. This is one of the reasons why it's so popular, and luckily for us as developers, doesn't make that much of a difference which platform it is running on.

The original version of Oracle8 was designed to support applications in the up-and-coming network-computing era, ranging from a small departmental application right up to a high-volume enterprise-wide system. In order to provide this level of flexibility, Oracle8 comes in two different editions:

- Oracle8
- Oracle8 Enterprise Edition

Both editions actually share the same code base, but the difference is that the standard edition (referred to simply as Oracle8) is aimed at smaller applications, whereas its big brother, the Enterprise Edition, comes with a number of high-end features that allows it to support the thousands of users of larger enterprise-wide applications. The Enterprise Edition provides greater support for very large databases containing hundreds of terabytes of data, whilst the number of columns per table and maximum database size, for both editions, has been increased compared to previous versions of Oracle.

In order to support large numbers of users, both the Oracle8 and Oracle8 Enterprise Edition servers provide a method of connection pooling that temporarily drops and then re-uses a physical connection for those users that are idle, in conjunction with its networking software **Net8**. With this type of technology there is no reason why an Oracle server cannot support many thousands of concurrent users.

With that said, it's worth remembering that our ASP-based applications, if designed correctly, should connect and disconnect from the Oracle database as soon as they have completed a certain task, rather than hold onto a database connection for the life of the user's session. Don't forget that Microsoft Transaction Server and OLE DB also offer connection-pooling techniques to save the valuable time taken to initialize database connections.

> *Traditional client/server applications that maintain a user's connection, until the application has closed, will more than likely utilize Oracle's own connection pooling rather than that of OLE DB which pools connections based on the same username and password combination.*

The actual edition of Oracle8 that we connect to again makes little practical difference to the front-end applications that we develop, as we use the same query language and networking software to manipulate the data.

Both editions of Oracle8 provide support for the emerging SQL-3 standard for object-type definition. SQL-3 allows us to create object types that, for example, define a person's address that we could then use directly in our database, and access through our programs.

For a full investigation into Oracle's object support, check out the Oracle TechNet at `http://technet.oracle.com`.

Oracle Version 8i

Oracle8i is the latest incarnation of the Oracle8 data server. If you hadn't already guessed it, the 'i' in Oracle8i refers to the **Internet**. Oracle Corporation bills Oracle8i as "the database for Internet computing".

All of the above Oracle8 features apply just the same to the new Oracle8i, with the Oracle8i data server also being available in two editions – the standard edition, Oracle8i, and the high-end version, Oracle8i Enterprise Edition.

The major change to the Oracle8i Enterprise Edition is the inclusion of support for the **Java Virtual Machine** allowing developers to execute Java code directly from within the database engine. Whereas previously, the only way to procedurally manipulate Oracle data was through its PL/SQL language, you can now use Java to do exactly the same job.

So are they doing away with PL/SQL? This does seem to be the general trend if you consider that the next version of SQL Server (the one after version 2000) will allow stored procedures written in any .Net language and that DB2 already provides support for Java stored procedures. SQL is here for the foreseeable future, but maybe for the benefit of ODBC and OLE DB access.

Oracle8i includes the Internet File System, **iFS**, a Java application that brings the combination of an integrated file system and database into one server to provide text searches and querying of files and data stored within iFS.

Another new technology in Oracle8i is **Oracle WebDB** that allows dynamic web sites to be built and deployed from within the Oracle database. WebDB provides an HTML interface so that non-programmers can develop their own web-based database applications. It includes a lightweight HTTP listener that can act as a web server and a PL/SQL interface to the database. As web developers, we might want to discourage non-programmers from developing their own web sites; this is not just because of our own job security, but also due to the fact that small in-house projects have a tendency to grow into large projects that may not have been designed with scalability in mind through poor programming techniques.

Oracle8i's new **interMedia** feature provides additional support for multimedia content such as image, video, text, and audio. interMedia allows users to query data held within common document formats such as HTML, Adobe Acrobat (PDF) and the Microsoft Office applications such as Word documents and Excel spreadsheets, and provides support for the delivery of *streaming* media in conjunction with common streaming servers such as Oracle Video Server and RealVideo.

Release 2 of Oracle8i (version 8.1.6) also brings native support for XML.

If you've used Oracle8 prior to Oracle8i, then you'll notice that the developer's tools such as the Schema Manager, Oracle Installer and Net8 Assistant have had a radical interface makeover. I personally find them slightly slower to use than their previous counterparts owing to the fact that most are now written in Java, but they do appear more user friendly.

Installing Oracle Client Components

In this section we will be installing the Oracle client components on an IIS-based web server to enable our ASP scripts to communicate with an Oracle database server. Once the client programs have been installed we will be using Oracle's configuration utilities to configure our web server to connect to Oracle.

> *With the exception of cosmetics, there are very few differences between the Oracle8 and Oracle8i installation programs, so we will be showing screen shots from the Oracle8i installation.*

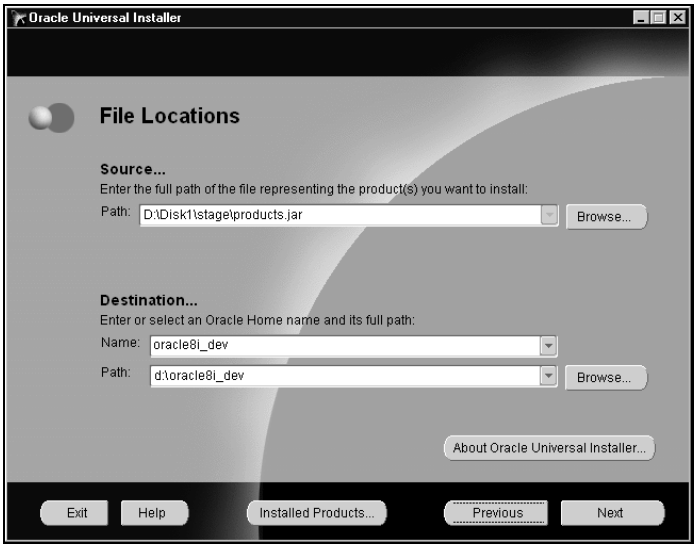
In order to access an Oracle database, a number of software components need to be installed on a client computer. Oracle8 uses its networking component **Net8** to provide client-server and server-server connectivity for many common protocols and platforms.

In versions prior to Oracle8, the forerunner to Net8 was **SQL*Net** version 2 – you'll find lots of documentation that still refers to SQL*Net. Net8 is backwardly compatible with SQL*Net version 2, allowing Net8 to access both Oracle7 and Oracle8 databases. It is possible, however, to connect to an Oracle8 database using SQL*Net but some of the new network features will not be available.

Once you've installed the Oracle client components, the **Net8 Easy Config** and **Net8 Assistant** applications can be used to configure your Net8 settings. Both applications use a number of `.ora` configuration files that you can, if you know what you're doing, edit yourself in Notepad. We'll go through the installation of the client components before we go into the details of the applications.

> **The Oracle client components are supported on all 32-bit Windows platforms: Windows 95, Windows 98, Windows NT 3.5 and NT 4 Server and Workstation, and Windows 2000.**

By running the familiar `setup.exe` file you will be presented with a screen welcoming you to the **Oracle Universal Installer**, or **Oracle Installer** as Oracle8 calls it. After clicking Next your first choice is to tell the Installer where to put the physical files that it installs. This location is known as the **Oracle Home** setting:



This allows you to install multiple versions of the Oracle products onto the same machine without an installation conflicting with any other installation. Oracle Home essentially defines the location of a folder into which the software is installed. If you only plan to have one set of Oracle products installed on the machine, which is very often the case, then choose `DEFAULT_HOME` for the Name. In my case, I have a number of Oracle products installed, so I have given it a name of oracle8i_dev with the files being located in the d:\oracle8i_dev folder.

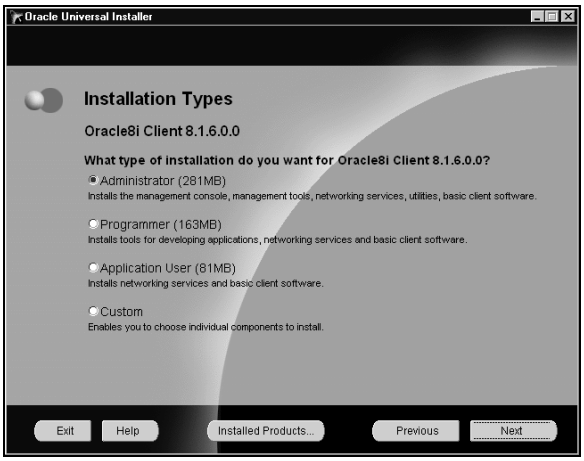
Clicking the Next button takes you to the Available Products screen:

Here you must choose the actual product to install: the database server, client software, or management infrastructure software. In many cases you will be connecting to an Oracle database running on a different server to that of your web server, so you should select the second option, Oracle8i Client. If your web server also happens to be your database server then you will need to select the first option to install the actual database sever and, optionally, a starter database accessed via the `scott` account. The Management Infrastructure option installs the client components along with directory services components.

In my case, I am installing the Oracle client on a web server that will connect to Oracle on a remote server, so I selected Oracle8i Client.

Once you've clicked Next, you will be asked about the Installation Type (the Oracle8 installation calls this the Primary Function). The list of options shown is dependent upon the item selected from the previous screen. If you had chosen to install the Oracle server then you will see a list of options, such as whether to include the pre-configured starter `scott` database.

In the case of the Oracle Client installation you can specify the type of installation required for the client components depending upon the features that the client machine needs:



If you need to perform DBA tasks such as creating and backing-up databases, and stopping the server then choose the Administrator option. This will install all of the utilities required to administer an Oracle server.

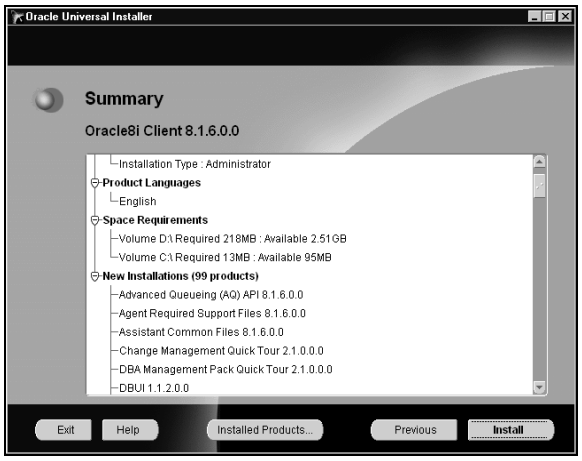
If the machine is used as your development server then it's a good idea to choose the Programmer option to install a subset of the Administrator tools. However, you won't get utilities such as the **Enterprise Manager Console** used to administer an Oracle server.

The Application User option should be selected if the machine is used as your web server. This will only install the basic networking and client components and none of the admin or programming tools.

The last option, Custom, allows you to specify exactly which components should be installed.

> *You can decide which items should be installed on your machine, but some organizations do not allow developers to perform traditional DBA functions such as stopping servers - for very good reasons. You can always add or remove components using the Installer at a later date. Personally I'd want everything that's available so I'd choose* Administrator *anytime!*

After you've clicked Next (for the last time) the Installer will show a summary page confirming the options that you have selected:



Now it's just a case of pressing the Install button to install all of the required programs. Once the installation has been completed you can move onto configuring Net8, the client software, to connect to your Oracle server.

Configuring Net8

As mentioned earlier, we need to install a layer of network software on our web server that allows us to communicate with Oracle. By selecting the **Client** installation, the Oracle Installer will have installed Net8, which we now have to configure.

Net8 supports standard network protocols, such as TCP/IP, to connect to Oracle8 servers through the use of user-friendly aliases called **service names**. A service name is simply a name used to refer to an Oracle server much as we use URLs in preference for hard-to-remember IP addresses.

You have a number of ways in which to store these lists of service names:

- Domain Name System (DNS)
- Local client configuration files
- Oracle Name Server
- Non-Oracle name server

Net8 uses **Oracle Protocol Adaptors** to map the following industry-standard network protocols into a standard that it can recognize internally:

TCP/IP	Widely-used Internet network protocol
SPX	Another commonly used network protocol
Named Pipes	Microsoft's networking protocol specific to PC-based LANs
Bequeath	Used for local Windows 95 and 98 Personal Oracle8 installations
Logical Unit Type	Part of IBM's peer-to-peer SNA network

Oracle8 comes with two utilities to configure Net8: **Net8 Easy Config**, to edit our list of service names, and **Net8 Assistant**, an advanced utility that allows us to configure service names, network listeners, Oracle Names Servers and local configuration files. Configuration using Net8 Assistant is primarily a DBA role so we won't cover it here. We will be using the Net8 Easy Config application to configure our client.

There are a number of ways to store the list of service names with the two most commonly used methods being:

- **Host Naming** – Uses existing DNS-based or a centrally maintained `HOSTS` file for name resolution. By simply using the host's network name, no client configuration is required.
- **Local Naming** – Uses a local configuration file, `TNSNAMES.ORA`, to resolve names.

`Host Naming` does not require any client configuration so we will take a look at `Local Naming` using the Net8 Easy Config program. Net8 Easy Config edits a file called `TNSNAMES.ORA` in the *`installation_folder`*`\Net80\Admin` folder, which can be edited manually using Notepad. In many Oracle sites it is a common practice to simply copy the `TNSNAMES.ORA` file from the Oracle server machine onto the client.

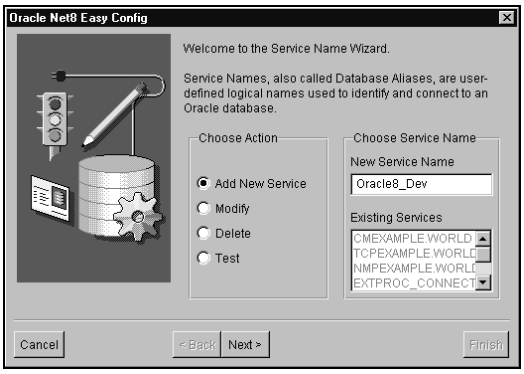
*TNS stands for **Transparent Network Substrate** (TNS). This is a non- proprietary low-level interface that manages the opening and closing of sessions and the sending or receiving of requests.*

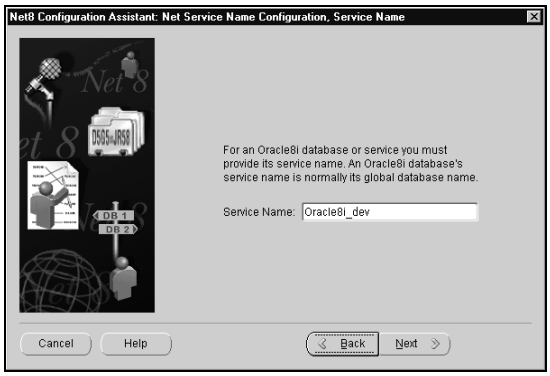
The screens that make up the Oracle8 and Oracle8i Net8 Easy Config (Net8 Configuration Assistant in Oracle8i) applications do differ somewhat so we'll work through both versions to configure connecting to an Oracle8 and Oracle8i server.

If you are configuring an Oracle8 client then start the Oracle Net8 Easy Config and select Add New Service. Our DBA has called the Oracle8 server `Oracle8_Dev`, so we'll type that name in – you'll have to use the name of your own Oracle8 server. You may find that this is the name of the actual server, provided that it is only running one instance of Oracle.

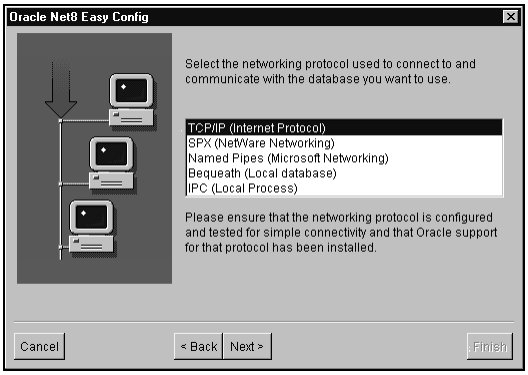
You may see a dialog box warning you that Net8 Easy Config has found a number of comments in the configuration file `TNSNAMES.ORA`. It is generally safe to ignore this warning message.

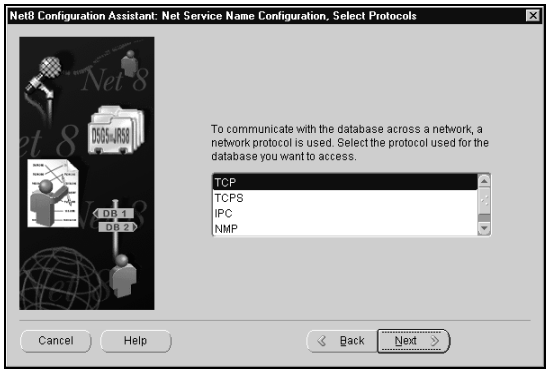
In the case of our Oracle8i database, we have a server called `Oracle8i_Dev` so we'll use that for the name. Before you can enter the Oracle8i service name, choose the Local Net Service Name configuration option, click Next and then choose the Add item to add a new service name before pressing Next again. Finally you must tell the Oracle8i Configuration Assistant that you want to access an Oracle8i database. Clicking Next will take you to the Service Name screen:



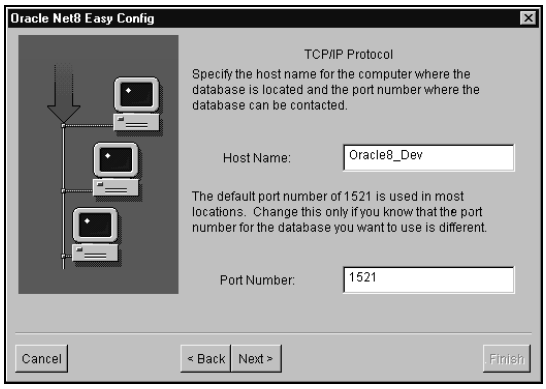


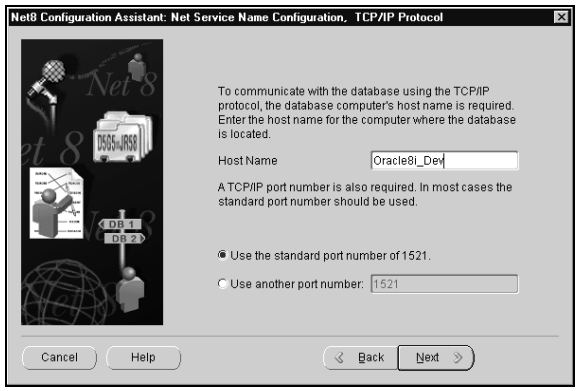
The next step is to choose the type of network protocol used to communicate with the server. Typically this will most likely be TCP/IP:



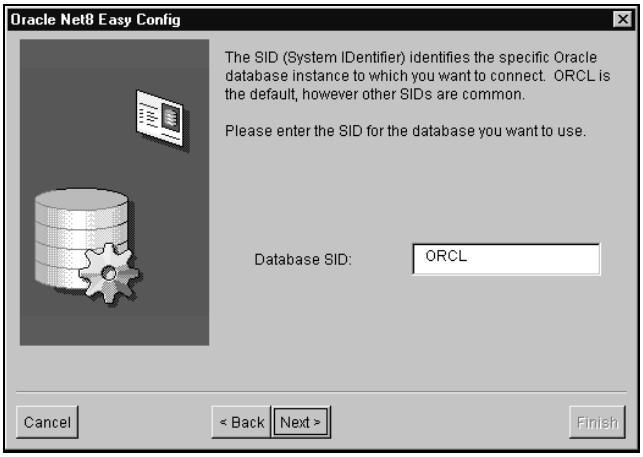


The host name is the resolved name used to refer to the server, which in our case is the same name given to this service name – for ease. It is possible to install the Oracle server software to listen on a different TCP/IP port number. By default, port number **1521** is used for Oracle installations, in much the same way that port number 80 is used for HTTP requests. Unless your DBA has used a different port number for additional security, select the default option:





There is one additional step to complete before testing an Oracle8 connection: you have to type in the name of the database System Identifier, or **SID** to connect to. It is possible to run more than one **instance** of Oracle on the same server by giving each instance a unique SID by which it can be identified. If there is only one instance installed, then the Oracle server installation will default to calling it ORCL, which your DBA can confirm:



If you do have a number of SIDs per server then it might be a good idea to use the SID as the name for each service.

To test the new service name, you must enter a valid user name and password when using Oracle8. Typically you can enter the `scott/tiger` username/password combination provided that the pre-configured `scott` database has been installed. The Oracle8i version actually defaults to using `scott/tiger` for you.

If you've entered the correct host name and username/password then you should receive a message saying that the connection test was successful. If you receive the error message ORA-12545: connect failed because target host or object does not exist, you need to recheck the values of your host name, port number and SID. You should also confirm that Oracle is actually running on the host specified.

The message ORA-01017: invalid username/password; logon denied is a lot more encouraging; it means that you successfully communicated with the Oracle server, but you entered the wrong username or password.

ORA-12545 and ORA-01017 are the common error messages that you are likely to come across, but you may receive any of the following messages as well:

ORA-12154: "TNS:could not resolve service name"	Net8 could not find the service name specified in your `TNSNAMES.ORA` file. Make sure that the `TNSNAMES.ORA` file actually exists and that you do not have multiple copies of the `TNSNAMES.ORA` file. Make sure that you do not have duplicate copies of the `SQLNET.ORA` file. When using domain names ensure that your `SQLNET.ORA` file contains a `NAMES.DEFAULT_DOMAIN` value.
ORA-12198: "TNS:could not find path to destination" and ORA-12203: "TNS:unable to connect to destination"	The client could not find the required database. Is the service name spelled correctly? Is `TNSNAMES.ORA` file in the correct folder? Check that the service name `ADDRESS` parameter in the connect descriptor of your `TNSNAMES.ORA` file is correct. Get your DBA to check that the Oracle Listener on the remote server has started and is running.
ORA-12224: "TNS:no listener"	Could not connect because the listener is not running. Does the destination address match one of the addresses used by the listener. Are you running the correct version of Net8 or SQL*Net?

Now that we've covered some of the differences between Oracle8 and Oracle8i, from now on we'll refer to them both collectively as Oracle8. When we come across a distinction between the two, we'll highlight it.

Much like SQL Server's **user spaces**, Oracle groups database objects, such as tables, indexes and procedures, into what is called a **schema**. A schema maps to an actual login name. So, in the case of the `scott` login name you will find a whole host of database objects under the `scott` account. `scott` is the sample database schema created by the Oracle Installer when you first install the Oracle server. Typically, a new default Oracle installation will have the following logins created:

Username	Password	Password
scott	tiger	Sample login.
sys	change_on_ install	Database administrator. Can perform all operations such as stopping and starting the database.
system	manager	Operations user that can perform operational tasks such as database backups.

We've gone through the process of installing Oracle's client networking software, Net8, then added and tested a new Net8 service name to connect to an Oracle8 server called `Oracle8_dev` and an Oracle8i server called `Oracle8i_dev`. Now it's time to look at how we connect to an Oracle database through ASP.

Connecting to an Oracle Database

There are a number of ways in which we can connect to an Oracle database in order to manipulate its data from within our ASP scripts. Which one you use rather depends what you are trying to achieve and whether your organization prefers access through stored procedures, as the features supported by one method may not be supported in another.

As well as Oracle Corporation, there are many third-party vendors such as Microsoft and Intersolv that provide a number of products to communicate with Oracle. The following list represents the more commonly used tools:

- Microsoft OLE DB Provider for Oracle
- Microsoft OLE DB Provider for ODBC
- Microsoft ODBC Driver for Oracle
- Oracle ODBC Driver
- Intersolv's Merant range of OLE DB providers and ODBC drivers
- Oracle Objects for OLE by Oracle
- Oracle Provider for OLE DB by Oracle

Microsoft's Universal Data Access (UDA) initiative contains a set of tools that we can use to communicate with an Oracle database. With the integrated Microsoft Data Access Components (MDAC) suite we can use **ActiveX Data Objects** (ADO) in conjunction with the **Microsoft OLE DB Provider For Oracle** (`MSDAORA.DLL`) or the **Microsoft ODBC Driver for Oracle** (`MSORCL32.DLL`) to communicate effectively with Oracle in a way that is reliable, scalable and offers high performance when using ADO.

Microsoft also offers the universal **OLE DB Provider for ODBC Drivers** (`MSDASQL.DLL`) that allows any ODBC data source to make use of the improvements in OLE DB. This, the default provider used by ADO, was developed so that any existing ODBC-based data could fit into the UDA environment efficiently and without losing an organization's ODBC investment.

As if this didn't give us enough flexibility, we also have the universal **Merant** range of OLE DB providers and ODBC drivers from Intersolv (www.merant.com/products/datadirect/oledb/Connect/factsheet.asp), and **Oracle Objects for OLE** (OO4O).

We've discussed how Oracle8's Net8 networking component is used to communicate with an Oracle8 database, but we haven't mentioned the **Oracle Call Interface** (OCI) library. We won't go into much detail except to say that this low-level layer exposes certain procedures that the OLE DB providers and ODBC drivers call in order to communicate with the database, in much the same way as DBLib for SQL Server databases.

After that brief overview, it is now time to show you how to connect to an Oracle8 database using the more popular technologies so that you can see the relative pros and cons of each in terms of feature support, performance, and ease.

There are bound to be times when you need the ability to fetch recordsets from an Oracle stored procedure with ADO. At the time of writing you have no choice but to use the ODBC driver for Oracle or Oracle's Oracle Provider for OLE DB, both of which will be covered later.

OLE DB Provider for Oracle

The OLE DB Provider for Oracle supports most of the Oracle8 data types:

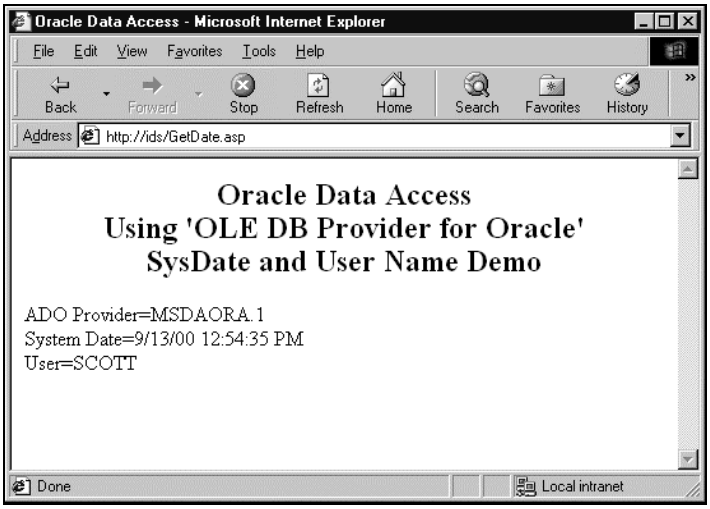
Data Type	Supported	Data Type	Supported
BFILE		LONG RAW	Yes
BLOB		NCHAR	
CHAR	Yes	NCLOB	
CLOB		NUMBER	Yes
DATE	Yes	NVARCHAR2	
FLOAT	Yes	RAW	Yes
INTEGER	Yes	VARCHAR2	Yes
LONG	Yes	MLSLABEL	

This table shows that many of the standard data types are supported but those such as the `LOB` (Large Object) and object-based extensions are not supported.

The provider is a *native* provider, in that it accesses the Oracle's API directly rather than through ODBC. This provides us with *generally* the best performance when compared to other methods of connecting to Oracle, but does mean that some functionality is not available.

In order to use the provider, you must set its name in the `ConnectionString` property of the ADO `Connection` object or as the `ConnectionString` argument to the `Open` method. As with any provider for ADO, unpredictable results can occur if you specify the name of the provider in more than one place.

Let's start by connecting to the Oracle database using the `scott` username to execute two simple built-in Oracle functions to retrieve the system date, `sysdate`, and current username, `user`:



Create a new ASP script called `GetDate.asp`:

```
<% Option Explicit %>
<HTML>
<HEAD><TITLE>Oracle Data Access</TITLE></HEAD>
<BODY>
<CENTER>
   <H2>
      Oracle Data Access<BR>
      Using 'OLE DB Provider for Oracle'<BR>
      SysDate and User Name Demo<BR>
   </H2>
</CENTER>
<%
Dim objConnection
Dim objRecordset

Set objConnection = Server.CreateObject("ADODB.Connection")
With objConnection
   .ConnectionString = "Provider=MSDAORA;Data Source=Oracle8_dev;" & _
                       "User ID=scott; Password=tiger;"
   .Open
   Response.Write "ADO Provider=" & .Provider & "<BR>"

   Set objRecordset = .Execute("SELECT sysdate, user FROM dual")
End With
```

We use the `Connection` object's `ConnectionString` property to tell ADO how to connect to our Oracle database before calling the `Open` command to attempt to connect to the database. Don't forget that the `Data Source` property, `Oracle8_dev`, is the **service name** that we created earlier, rather than the actual machine name – but in my case, both are actually the same value.

The `Provider` section tells ADO to use the OLE DB Provider for Oracle. We can use either the class name of the provider, in this case `MSDAORA`, or the full provider name: '`OLE DB Provider for Oracle`'. As we want to use the `scott` account, we need to set the `User ID` and `Password` accordingly.

> *Our Oracle8 server is located on a server called* `Oracle8_dev`. *You'll have to change this to reflect your own Oracle database server.*

If you've not used the `With...End With` statement, it serves as a way to call multiple methods on a single object without having to refer to the name explicitly every time. It makes your code easier to read and actually runs slightly faster as the ASP processor doesn't have to do extra processing to establish the address of the `objConnection` object.

By calling the `Open` method, we should get a connection to the Oracle database. By way of a confirmation, we write out the name of the `Provider` property. This shows us the name as defined in the Registry along with any version number if there are multiple versions installed on the server.

The `Execute` method returns back a `Recordset` representing the records that were fetched from the database, in this case a single record with a column containing the current system date and the current user name. The argument passed to `Execute` is the command that we want Oracle to run for us.

> *Notice the word* `dual` *in our* `SELECT` *statement? Oracle does not allow you to execute a* `SELECT` *statement without an accompanying* `FROM` *clause;* `dual` *is a logical pseudo-table, available to all accounts, provided for that purpose. It is not a physical table that you can alter.*

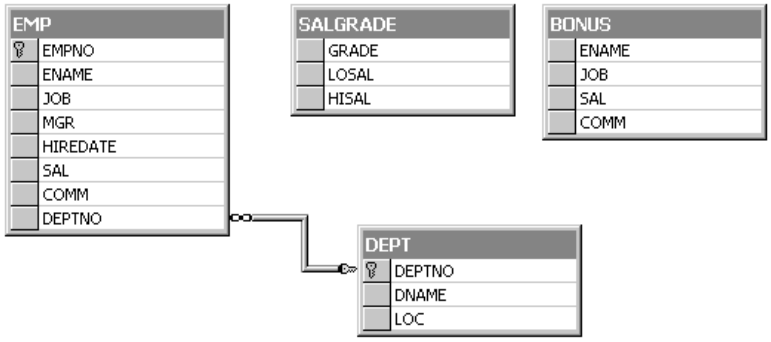
```
Response.Write "System Date=" & objRecordset.Fields("sysdate") & "<BR>" & _
               "User="        & objRecordset("user")

Set objConnection = Nothing
Set objRecordset = Nothing
%>
</BODY>
</HTML>
```

We finish off by reading the `Fields` collection of our `objRecordset` object to get the value for the `sysdate` and `user` functions. In the case of the `user` field we've left out the `.Fields` statement as this is the `Recordset` object's default property, though you can make your code run faster if you use it.

There's no need to navigate through the `objRecordset`, as there will only be one record returned. With any objects that we create in our scripts, it's always a good idea to shut them down explicitly as soon as possible using the `Set ... = Nothing` statement in order to free up server resources.

> **As we mentioned earlier, if you received the Oracle error message ORA-12545: connect failed because target host or object does not exist then you need to recheck the values of your host name, port number and SID that were entered when your created the new service name using Net8 Easy Config.**

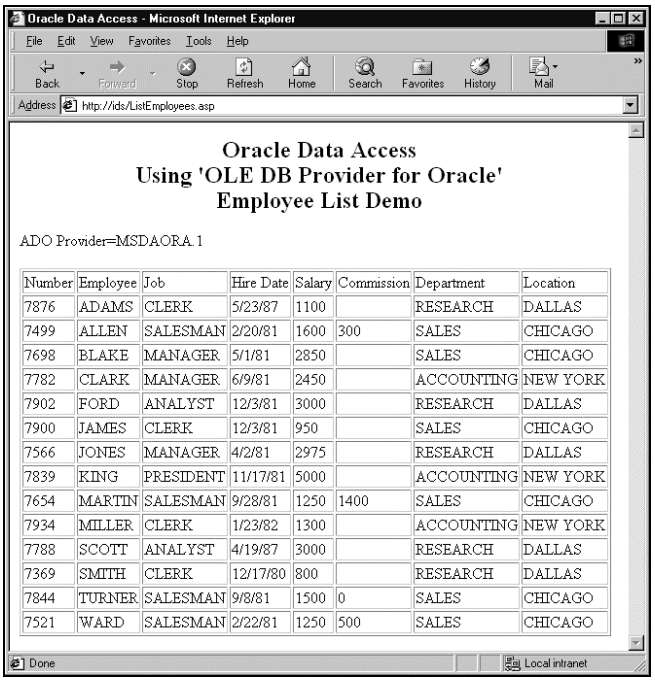
That was a relatively easy example, so let's have a look at a more complex statement in which we return a number of records. The `scott` schema comes with four sample tables that you can look at yourself. The tables opposite represent an employee's bonus and salary tracking system:



This isn't the best schema that Oracle could have used as their pre-configured sample database. The SALGRADE and BONUS tables are not referenced by any other tables and contain no primary keys.

Table Name	Purpose
DEPT	Stores a list of department names
SALGRADE	Stores a list of salary grades
EMP	Stores a list of employees
BONUS	Stores a list of employee bonuses

Our example ASP script will be using the DEPT and EMP tables to show a list of all employees sorted by their name (later on we'll be using them in our sample application):



The code used to produce the previous screenshot looks like this:

```
<% Option Explicit %>
<HTML>
<HEAD><TITLE>Oracle Data Access</TITLE></HEAD>
<BODY>
<CENTER>
   <H2>
      Oracle Data Access<BR>
      Using 'OLE DB Provider for Oracle'<BR>
      Employee List Demo<BR>
   </H2>
</CENTER>
<%
Dim objConnection
Dim objRecordset
Dim varSQL

Set objConnection = Server.CreateObject("ADODB.Connection")
With objConnection
   .ConnectionString = "Provider=MSDAORA; Data Source=Oracle8_dev; " & _
                       "User ID=scott; Password=tiger;"
   .Open
   Response.Write "ADO Provider=" & .Provider & "<P>"
```

We start off as before by defining two variables for our `Connection` and `Recordset` objects and then connect to the database using the `scott/tiger` combination. We've added a new variable, `varSQL`, to hold a nicely formatted SQL statement:

```
   varSQL = "SELECT emp.empno, emp.ename, emp.job, emp.hiredate," & _
            "       emp.sal, emp.comm, dept.dname, dept.loc" & _
            " FROM emp, dept" & _
            " WHERE emp.deptno = dept.deptno" & _
            " ORDER BY emp.ename"

   Set objRecordset = .Execute(varSQL)
End With
```

The SQL statement joins the employee table, `EMP`, to the department, `DEPT`, to return a list of employees and their departments. Again we use the `Execute` command to return back a `Recordset` of data:

```
Response.Write "<TABLE BORDER=1><TR>" & _
               "   <TD>Number</TD>" & _
               "   <TD>Employee</TD>" & _
               "   <TD>Job</TD>" & _
               "   <TD>Hire Date</TD>" & _
               "   <TD>Salary</TD>" & _
               "   <TD>Commission</TD>" & _
               "   <TD>Department</TD>" & _
               "   <TD>Location</TD>" & _
               "</TR>"
```

We use `Response.Write` to write out the start of our table of results:

```
Do While Not objRecordset.EOF

   Response.Write "<TR>" & _
                  "   <TD>" & objRecordset("empno")    & "</TD>" & _
                  "   <TD>" & objRecordset("ename")    & "</TD>" & _
                  "   <TD>" & objRecordset("job")      & "</TD>" & _
                  "   <TD>" & objRecordset("hiredate") & "</TD>" & _
                  "   <TD>" & objRecordset("sal")      & "</TD>" & _
                  "   <TD>" & objRecordset("comm")     & " </TD>" & _
                  "   <TD>" & objRecordset("dname")    & "</TD>" & _
                  "   <TD>" & objRecordset("loc")      & "</TD>" & _
                  "</TR>"

   objRecordset.MoveNext
Loop
Response.Write "</TABLE>"
```

Now it's just a case of writing out each record by retrieving the value for each column from the `Fields` collection of our `Recordset` object `objRecordset` and moving to the next record using the `MoveNext` method. We loop through using a `Do While...Loop` that will stop as soon as it gets to the end of the `Recordset`.

Some of the records in the `comm` column contain a null value, so we add the HTML non-breaking space tag (` `) to ensure that the browser draws the cell border correctly.

```
Set objConnection = Nothing
Set objRecordset = Nothing
%>
</BODY>
</HTML>
```

As with our previous example, it's a good idea to explicitly close our `objConnection` and `objRecordset` objects as soon as we've finished with them.

We've now managed to connect to an Oracle8 database using the OLE DB Provider for Oracle to retrieve a single record of the current system date and username and a full list of employees in the `scott` database's `emp` table. It is suggested that the OLE DB Provider for Oracle be used for the majority of Oracle data access as it executes faster and supports Microsoft's new direction in data access – OLE DB.

Microsoft ODBC Driver for Oracle

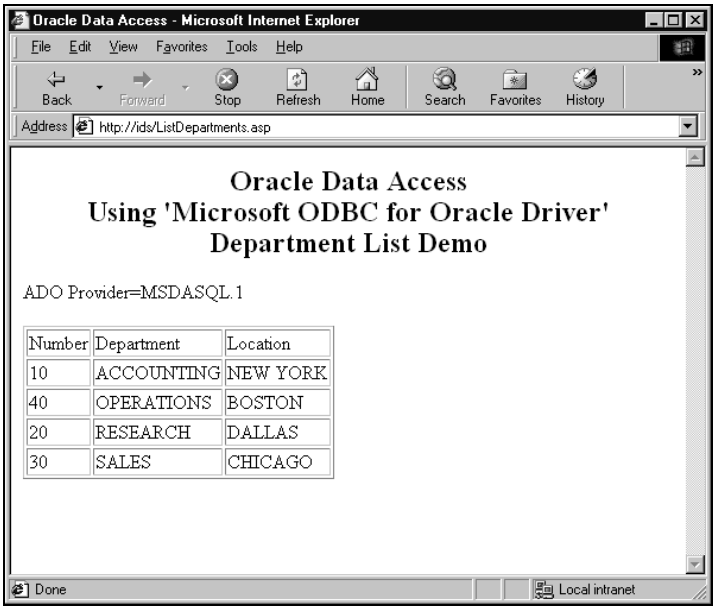
The Microsoft ODBC Driver for Oracle supports the same set of Oracle8 data types as the OLE DB Provider for Oracle. When using this driver with ADO, we are actually using the **OLE DB Provider for ODBC Drivers** (MSDASQL), which in turn uses the Microsoft ODBC for Oracle Driver.

Microsoft released the OLE DB Provider for ODBC (MSDASQL) so that all existing ODBC-based applications could use the new features found in OLE DB through ADO. When connecting to any data source using ADO, this is the default provider that is used.

ODBC connection strings use the older `DRIVER=`, `DSN=`, `UID=`, `PWD=` and `SERVER=` (optionally in the place of DSN=) parameters to connect to a data source. Don't forget that there must be a valid Data Source Name, DSN, registered through the ODBC Data Source Administrator in the Administration Tools (or Control Panel) if you are going to use the `DSN` parameter.

> *Each time you connect to a database using a DSN, ODBC must look through the Windows Registry in order to retrieve connection details for your DSN. There may be some performance improvements in your application if you use DSN-less connections, as the Windows Registry is notoriously slow to access. If you do need to use DSNs, then remember to use System DSNs rather than File DSNs as anonymous users, which your server is more than likely to use, have access to them.*

We are going to create a simple ASP script that uses some of the principles we used with the OLE DB Provider for Oracle to show a list of departments from the `scott` database's `dept` table.



The only real difference to this code is the connection string used, so we'll just show that line of code:

```
   .ConnectionString = "Provider=MSDASQL;" & _
                       "DRIVER={Microsoft ODBC for Oracle}; " & _
                       "SERVER=Oracle8_dev; UID=scott; PWD=tiger;"
```

We make use of the `DRIVER` property to tell MSDASQL to use the Microsoft ODBC Driver for Oracle, `SERVER` points to our database server, `Oracle8_dev`, and we use `UID` and `PWD` rather than the `User ID` and `Password` combination.

Notice that we didn't specify a DSN so we don't have to create one, and although it's not actually necessary in this case, we've specified the name of the `Provider` to use.

Oracle Objects for OLE (OO4O)

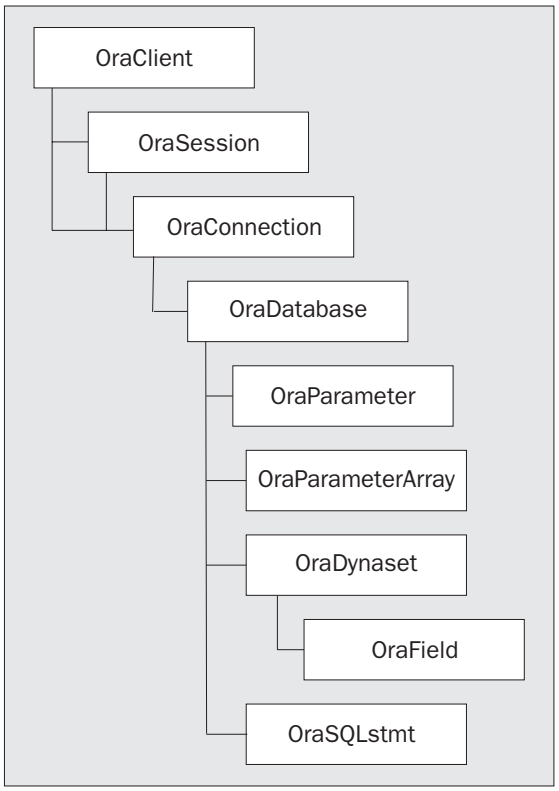
Oracle provides us with its own native client software that sits above the Oracle Call Interface, as mentioned earlier, allowing us to communicate with an Oracle database using a COM/OLE component.

Oracle Objects for OLE, or OO4O as it is usually abbreviated to, allows us to execute SQL and PL/SQL statements in a native "pass-through" format. This means we can make use of all Oracle data types as well as additional features, such as **bind variables**.

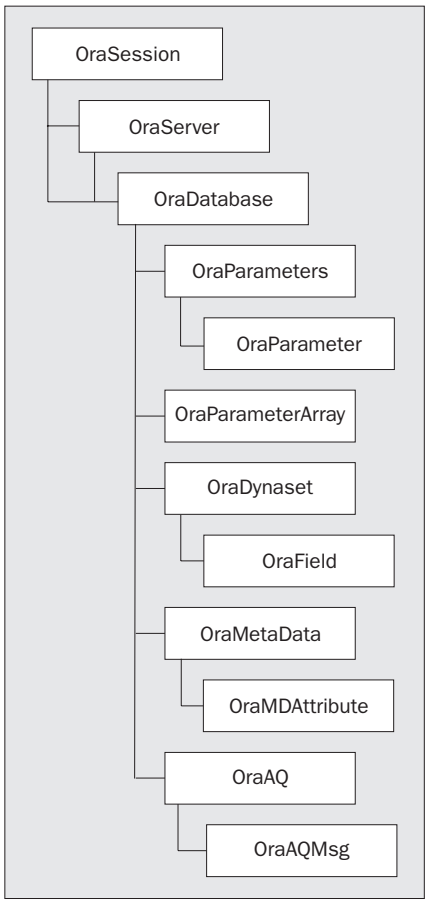
Bind variables are an efficient way to execute the same SQL statement with differing parameters without Oracle having to re-parse the statement each time. Unfortunately, it won't make that much of a performance difference to our web page, as we will only execute the statement twice and then close our database connection. However, this feature is ideal for client/server applications that maintain the database connection until the application is closed. We will be discussing bind variables in the following examples.

Unfortunately, by using OO4O we'll have to forfeit the usual methods found in ADO. However, OO4O does implement the same, or very similar methods, so the learning curve is not that steep.

Version 2.3 of OO4O,which shipped with Oracle8, has the following object model:



You'll find that the later version, 8.1, as shipped with Oracle8i, has a similar model with a number of extra objects:



The table gives a brief description of each object:

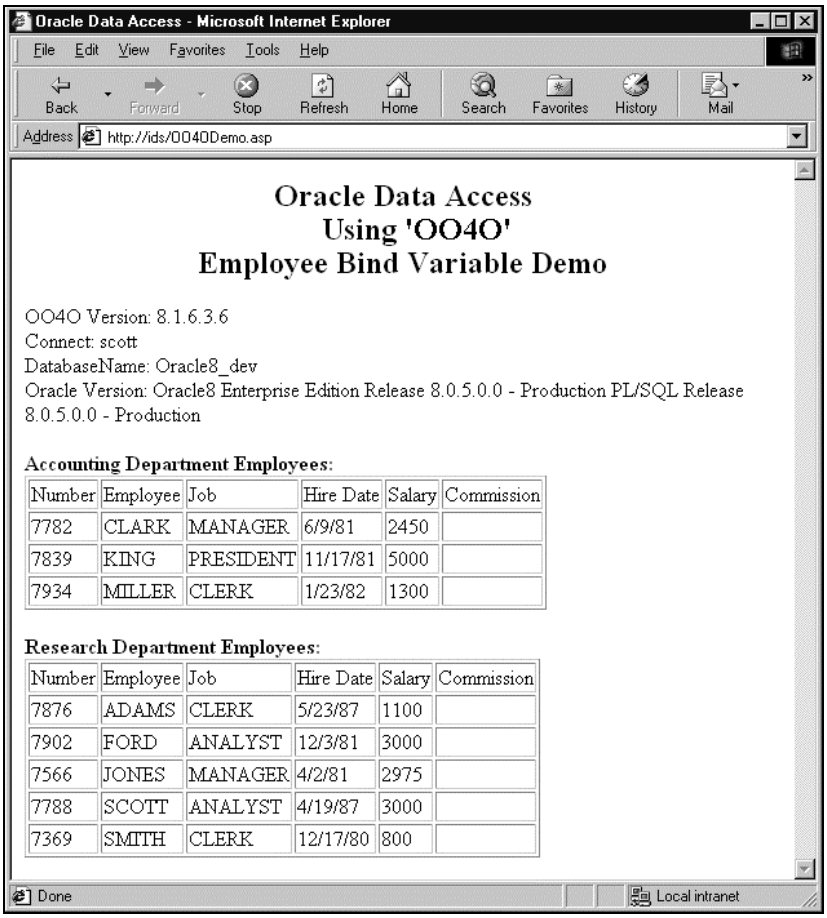
Name	OO4O Version	Description
`OraSession`	2.3 8.1	This is the first top-level object needed before we connect to an Oracle database.
`OraServer`	8.1	Represents a physical connection to an Oracle database server instance. The `OpenDatabase` function can be used to create client sessions by returning an `OraDatabase` object.
`OraConnection`	2.3	Returns various pieces of user information about the current `OraSession` object. It can be shared by many `OraDatabase` objects, but each `OraDatabase` must exist in the same `OraSession` object.

Name	OO4O Version	Description
`OraDatabase`	2.3 8.1	Represents a single login to an Oracle database. Similar to the ADO `Connection` object. `OraDatabase` objects are returned by the `OraSession.OpenDatabase` function.
`OraDynaset`	2.3 8.1	Similar to an ADO `Recordset` object. Represents the results retrieved by a call to the `OraDatabase.CreateDynaset` function.
`OraField`	2.3 8.1	Represents a column of data within an `OraDynaset` object. Similar to the ADO `Field` object of an ADO `Recordset`.
`OraClient`	2.3	Automatically created by OO4O as needed. Maintains a list of all active `OraSession` objects currently running on the workstation.
`OraParameter`	2.3 8.1	Represents a bind variable for a SQL statement or PL/SQL block to be executed using the `OraDynaset` object. Similar to the `Parameter` object in an ADO `Command` object.
`OraParamArray`	2.3 8.1	Allows **arrays** of parameters to be set for the `OraDatabase.Parameters` function.
`OraSQLStmt`	2.3 8.1	Represents a single SQL statement. Typically used with SQL statements that include bind variables to improve performance as Oracle does not have to parse the statement each time it is executed. Can be thought of as conceptually similar to the ADO `Command` object.
`OraMetaData`	8.1	Returns meta data to describe a particular schema such as column names. Similar to the SQL Server DMO object library. See the meta data example below.
`OraAQ`	8.1	The `CreateAQ` method of the `OraDatabase` returns an `OraAQ` object. This provides access to Oracle's Advanced Queuing message system that allows messages to be passed between applications, much like MSMQ.

We are going to create a sample ASP script that executes a SQL statement to return a list of employees for a specific department number using bind variables. This example, which is compatible with both the 2.3 and 8.1 versions of OO4O, will use `OraSession`, `OraDatabase`, `OraDynaset` and `OraFields` objects, as they are the most commonly used objects in OO4O.

Using Bind Variables

Our script contains simple VBScript function, `CreateEmployeeTable`, declared at the bottom of the script, to handle the refreshing of the `Parameters` collection and writing out of the HTML results table each time.




```
<% Option Explicit %>
<HTML>
<HEAD><TITLE>Oracle Data Access</TITLE>
<BODY>
<CENTER>
   <H2>
      Oracle Data Access<BR>
      Using 'OO4O'<BR>
      Employee Bind Variable Demo<BR>
   </H2>
</CENTER>
<%
Const cAccountingDeptCode = 10
Const cResearchDeptCode = 20

Dim varOraSession
Dim varOraDatabase
Dim varOraDynaset
Dim varSQL
```

We've made use of two constants to store the department code for the Accounting and Research departments. These are passed into our `objDatabase.Parameters` object for each department.

Each of the remaining variables defined stores a reference to the `OraSession`, `OraDatabase`, and `OraDynaset` objects respectively. Again we use a variable called `varSQL` to store our nicely formatted SQL statement:

```
Set varOraSession = Server.CreateObject("OracleInProcServer.XOraSession")
Set varOraDatabase = varOraSession.OpenDatabase("Oracle8_dev", _
                                                "scott/tiger", 0)
```

The only object that we have to create ourselves explicitly is the `OraSession` object as all other objects are created from other existing objects. We use the familiar `Server.CreateObject` to create an instance of the OO4O component whose internal `ProgID` is `OracleInProcServer.XOraSession`.

The `OpenDatabase` function returns an `OraDatabase` object, which in our case is the `scott` account on the `Oracle8_dev` service. `OpenDatabase` is called in the following way:

```
Set oraDatabase=oraSession.OpenDatabase(db_name, connectstring, options)
```

Where *`db_name`* is the **service name** to connect to, *`connectstring`* is the standard Oracle connection format of "*`username/password`*", and *`options`* is a collection of bit flags to indicate the mode in which the database should be opened. In our case we are passing in 0 to indicate that we want the database to be opened in the default mode, which means that any fields that we do not explicitly set a value for using the `AddNew` or `Edit` methods will be set to `Null` (this will incidentally override any server column default values!), rows will be locked as soon as the `Edit` method is called, and that non-blocking SQL functionality will **not** be used. A non-blocking call provides the same concept as an asynchronous ADO call in which the calling application does not have to wait until the server completes a request before continuing.

> **Some client installations may cause the Oracle error "Credential Retrieval Failed" whenever you try to connect. If this is the case then your client installation is trying to use a different form of client authentication to that of the server. Client authentication should be set to none, rather than native (NTS), so edit your `sqlnet.ora` file, located in the same folder as `tnsnames.ora`, and replace the line**
>
> `SQLNET.AUTHENTICATION_SERVICES=(NTS)`
>
> **with**
>
> `SQLNET.AUTHENTICATION_SERVICES=(NONE).`

```
Response.Write "OO4O Version:" & varOraSession.OIPVersionNumber & "<BR>" &_
               "Connect: " & varOraDatabase.connect & "<BR>" & _
               "DatabaseName: " & varOraDatabase.DatabaseName & "<BR>" & _
               "Oracle Version: " & varOraDatabase.RDBMSVersion & "<P>"
```

Just for our benefit we write out some information about the version of OO4O and the Oracle server that we are connecting to:

```
varSQL = "SELECT empno, ename, job, hiredate," & _
         "       sal, comm" & _
         " FROM emp" & _
         " WHERE deptno = :deptnoparam" & _
         " ORDER BY ename"
```

`varSQL` stores the SQL statement to execute using an Oracle bind variable, `:deptnoparam`. As we hinted at above, using bind variables can improve the performance of data access when you have the same SQL statement to execute, but need to alter a parameter. Each time Oracle executes a statement it has to go through the statement to understand how it should be executed. By using the bind variable, we can get Oracle to parse it only once. Remember, though, that this only exists for the life of your `OraDatabase` object – which should only be kept around for the life of the script and not in an ASP `Session` variable.

```
varOraDatabase.Parameters.Add "deptnoparam", 0, 1
```

Before we can tell Oracle about the bind variable we need to add it to the `OraDatabase` object's `Parameters` collection using the `Add` command, which is called in this way:

```
OraParameters.Add Name, Value, IOType
```

The `Name` argument is a string that represents the name of the parameter to add, and it must match that of the bind variable defined in the SQL statement, `Value` is a `variant`, `IOType` indicates the direction of this parameter:

Enumerator	Value	Description
`ORAPARM_INPUT`	`1`	Use as an input variable only
`ORAPARM_OUTPUT`	`2`	Use as output variable only
`ORAPARM_BOTH`	`3`	For variables that are both input and output

`IOType` is much like the `adDirection` enumerator used in the ADO `Command` object's `Parameters` collection to set the direction of SQL parameters.

In our example we passed in a value of `0` as a default because we don't actually have a value to use and a `1` to indicate that is for input use only.

```
Set varOraDynaset = varOraDatabase.CreateDynaset(varSQL, &H4)
```

Now it's just a case of calling the `OraDatabase.CreateDynaset` to retrieve an `OraDynaset` object. `CreateDynaset` is called in this way:

```
Set oradynaset = oradatabase.CreateDynaset(sql_statement, options)
```

Where `sql_statement` is the SQL to execute and `options` contains a bit flag of settings to define how the `OraDynaset` object behaves, such as whether it is updateable, or to cache data on the client. In our case we're passing in the hex value &H4 to indicate that it should be opened in read-only mode as we only want to display some data.

Behind the scenes, Oracle parses the statement ready for execution. It doesn't actually fetch any data until we set the `deptnoparam` parameter's `Value` property in `OraDatabase.Parameters` collection and ask OO4O to refresh the dynaset using `OraDynaset.Refresh`:

```
Response.Write "<B>Accounting Department Employees:</B><BR>"
CreateEmployeeTable cAccountingDeptCode

Response.Write "<BR><B>Research Department Employees:</B><BR>"
CreateEmployeeTable cResearchDeptCode
```

Here we can see our VBScript procedure `CreateEmployeeTable` being called to set the value for each of the department number parameters to create a nicely formatted table, as defined below:

```
Set varOraDatabase = Nothing
Set varOraDynaset = Nothing
Set varOraSession = Nothing
```

As always, we close down all of our objects as soon as possible in order to save server resources. Now to the `CreateEmployeeTable` function:

```
Sub CreateEmployeeTable(ByVal varDeptCode)

   varOraDatabase.Parameters("deptnoparam").Value = varDeptCode
   varOraDynaset.Refresh
```

`CreateEmployeeTable` is passed the required department code, which it binds to the original SQL statement's bind variable `deptnoparam`. Each time you specify a new value for a bind variable, you must call the `Refresh` method to fetch the new data.

Calling `Refresh` cancels all record edit operations that may have been pending through the `Edit` and `AddNew` methods, executes the SQL statement, and then moves to the first row of the resulting dynaset.

```
   Response.Write "<TABLE BORDER=1><TR>" & _
                  "  <TD>Number</TD>" & _
                  "  <TD>Employee</TD>" & _
                  "  <TD>Job</TD>" & _
                  "  <TD>Hire Date</TD>" & _
                  "  <TD>Salary</TD>" & _
                  "  <TD>Commission</TD>" & _
                  "</TR>"

   Do While Not varOraDynaset.EOF
```

So now we create the `TABLE` tag and loop through the records until we come to the end of file, `EOF`, exactly as we would with the ADO `Recordset.EOF` property.

```
      Response.Write "<TR>" & _
                     "   <TD>" & varOraDynaset.Fields("empno").Value & _
                     "</TD>" & _
                     "   <TD>" & varOraDynaset.Fields("ename").Value & _
                     "</TD>" & _
```

```
                        "   <TD>" & varOraDynaset.Fields("job").Value & _
                        "</TD>" & _
                        "   <TD>" & varOraDynaset.Fields("hiredate").Value & _
                        "</TD>" & _
                        "   <TD>" & varOraDynaset.Fields("sal").Value & _
                        "</TD>" & _
                        "   <TD>" & varOraDynaset.Fields("comm").Value & _
                        " </TD>" & _
                        "</TR>"
```

The `Fields` collection returns a named list of columns for the current record, which we use to create a new table row for each record. We haven't shown it, but as with an ADO `Recordset` object, the `Fields` property is the default value, and for a `Field` object, the `Value` is the default so `varOraDynaset.Fields("sal").Value` is equal to `varOraDynaset("sal")`. For better performance you should use the latter.

```
         varOraDynaset.MoveNext
      Loop
      Response.Write "</TABLE>"
   End Sub
   %>
   </BODY>
   </HTML>
```

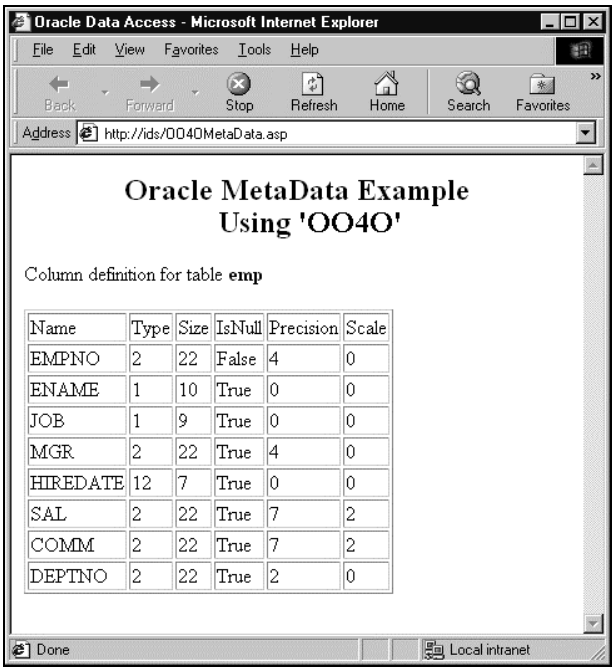
As with ADO, the `MoveNext` method moves to the next record.

Getting Meta Data

For our final look at OO4O we will use the `OraMetaData` object found in version 8.1 to retrieve a list of attributes for the `emp` table within the `scott` schema. As we said earlier, `OraMetaData` can retrieve all sorts of information about a schema, by calling the `OraDatabase` object's `Describe("schema_name")` function to return an `OraMetaData` object. The `OraMetaData` object returns a collection of `OraMDAttribute` objects that actually describe the data found and contains the following methods and properties:

Name	Description
`Count`	Returns the number of `OraMDAttribute` objects contained in the collection.
`Type`	The type of object described, for example `ORAMD_TABLE` which enumerates to the value 1 for an Oracle table.
`Attribute(pos)`	Returns an `OraMDAttribute` object at the specified position. This can be the 0 based index or a string name, such as "`ColumnList`".

To make things slightly complicated the `OraMDAttribute` object has a property called `IsMDObject` that returns `True` if the `Value` property contains yet another `OraMetaData` object. This allows you to recursively search through a hierarchy of `OraMetaData` objects. If it returns `False` then `Value` contains a string representation of the item.



The following code produces the screenshot shown above. We start off with the usual header:

```
<%Option Explicit%>
<HTML>
<HEAD><TITLE>Oracle Data Access</TITLE>
</HEAD>
<BODY>
<CENTER>
   <H2>
      Oracle MetaData Example<BR>
      Using 'OO4O'<BR>
   </H2>
</CENTER>
```

The `cTableName` constant contains the name of the table that we want to describe. As usual we are using the `OraSession` object to hold a reference to OO4O and `OraDatabase` to connect to our Oracle8i server:

```
<%
Const cTableName = "emp"

Dim objOraSession
Dim objOraDatabase
Dim objOraMetaData
Dim objOraMDAttribute
Dim objColumnList
Dim objiColCount
Dim objColumnDetails
```

`objOraMetaData` is used to store our top-level `OraMetaData` object returned by the `Describe` function and `objOraMDAttribute` stores the item name `"ColumnList"` from the `objOraMetaData` object, which represents the list of columns in the `emp` table. The actual `Value` for `objOraMDAttribute` is stored in `objColumnList`.

```
Set objOraSession = CreateObject("OracleInProcServer.XOraSession")
Set objOraDatabase = objOraSession.OpenDatabase("Oracle8i_dev", _
                                                "scott/tiger", 0)
Set objOraMetaData = objOraDatabase.Describe(cTableName)
Set objOraMDAttribute = objOraMetaData("ColumnList")
```

We connect to the `Oracle8i_dev` service and call the `OraDatabase` object's `Describe` function to return our first `OraMetaData` object to `objOraMetaData` for the `emp` table. `objOraMetaData` will contain a collection of `OraMDAttribute` items, so we pass in `ColumnList` to retrieve the list of column names.

```
If objOraMDAttribute.IsMDObject Then

   Response.Write "Column definition for table <B>" & cTableName & _
                  "</B><P>" & _
                  "<TABLE BORDER=1><TR>" & _
                  "<TD>Name</TD><TD>Type</TD><TD>Size</TD>" & _
                  "<TD>IsNull</TD><TD>Precision</TD>" & _
                  "<TD>Scale</TD>" & _
                  "</TR>"

   Set objColumnList = objOraMDAttribute.Value
```

Even though it's not strictly necessary with this example, we check the `IsMDObject` property to see if the `Value` property contains another `objMetaData` object. In our case, it will always be `True`, since we asked for the list of column names, which is another `objMetaData` object.

> **`IsMDObject` is a property so if you try to call it as a function by adding `()` to the end you'll get runtime error 'Object doesn't support this property or method'.**

To make the code easier to read and run quicker we transfer the `Value` property into a new variable `objColumnList`:

```
   For iColCount = 0 To objColumnList.Count - 1
      Set objColumnDetails = objColumnList(iColCount).Value
      Response.Write  "<TR>" & _
         "<TD>" & objColumnDetails("Name")      & "</TD>" & _
         "<TD>" & objColumnDetails("DataType")  & "</TD>" & _
         "<TD>" & objColumnDetails("DataSize")  & "</TD>" & _
         "<TD>" & objColumnDetails("IsNull")    & "</TD>" & _
         "<TD>" & objColumnDetails("Precision") & "</TD>" & _
         "<TD>" & objColumnDetails("Scale")     & "</TD>" & _
       "</TR>"
   Next
   Response.Write "</TABLE>"
End If
```

Now it's just a case of moving through the zero-based collection of column details and writing out the value for each item. We finish off by shutting down our objects:

```
Set objColumnDetails = Nothing
Set objColumnList = Nothing
Set objOraMDAttribute = Nothing
Set objOraMetaData = Nothing
Set objOraDatabase = Nothing
Set objOraSession = Nothing
%>
</BODY>
</HTML>
```

That covers our introduction into the common objects you'll come across in OO4O. OO4O offers a rather flexible approach to connecting to an Oracle database and also provides us with additional PL/SQL functionality not available through ADO, such as the use of input arrays for stored procedures.

So, which one should you use in your ASP applications? Unfortunately, there is no simple answer. Each method claims to be faster than the next whilst providing support for additional functionality. It really does pay to try each of the methods in your own environment before committing to any particular one.

An Overview of PL/SQL

We've shown you a number of techniques available to connect to an Oracle database. Now we shall provide a quick overview of Oracle's own procedural extensions to SQL.

> *This section doesn't aim to be a PL/SQL bible. Instead, we'll cover some of the main differences between PL/SQL and standard ANSI SQL.*

The "PL" in PL/SQL is short for **Procedural Language**. It is an extension to SQL that allows you to create PL/SQL *programs* that contain standard programming features such as error handling, flow-of-control structures, and variables, all allowing you to manipulate Oracle data. By itself, SQL does not support these concepts.

Block Structure

A PL/SQL program consists of any number of *blocks* or sections of code. In our ASP scripts we can create any number of chunks of code to execute on the server using the `<%...%>` tags. This is similar to PL/SQL, where a set of statements can be grouped logically together as part of a larger set of instructions:

```
DECLARE TotalSal NUMBER(5);
BEGIN
   SELECT SUM(Sal) INTO TotalSal
     FROM emp
    WHERE ename LIKE 'S%';

   dbms_output.put_line('totalSalary=' || TotalSal );
   IF TotalSal < 10000 THEN
      UPDATE emp SET
        Sal = Sal * 1.1
      WHERE ename LIKE 'S%';
   END IF;
```

```
    COMMIT;

  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      dbms_output.put_line('No records found.');
    WHEN OTHERS THEN
      dbms_output.put_line(SQLERRM);
END;
```

A PL/SQL block has three distinct sections:

- Declarations
- Statements
- Handlers

They are defined in the following way:

```
[DECLARE declarations]
BEGIN
   statements
   [EXCEPTION handlers]
END;
```

The declarations section contains any variables or constants that are going to be used within the `statements` section. You can have any number of statements to execute, but if an error occurs in any of them, processing will stop and execution will move to the exception section for trapping, if any are defined.

In above example we declare `TotalSal` as a variable in the declarations section:

```
DECLARE TotalSal NUMBER(5);
```

All of the remaining code up to the `EXCEPTION` line forms the `statements` section, followed by two exception handlers: `NO_DATA_FOUND` and `OTHERS`.

When you declare an exception handler you must tell Oracle which one of the in-built exceptions you want to trap, such as `ZERO_DIVIDE`. In our case we've trapped `NO_DATA_FOUND`, which is raised when an empty result set is retrieved, and `OTHERS`, which is a catch-all handler that will trap any other exceptions that you have not explicitly named. You can have any number of exception handlers and you can also set up your own exception types, but that is beyond the scope of this chapter.

Once an exception has been trapped you cannot issue the equivalent to a VBScript `RESUME NEXT` as the PL/SQL program will exit at the last line in the exception handler. This is somewhat different to the operation of SQL Server's T-SQL in which you can check the value of `@@Error` after any statement, provided that the error was of a trappable nature.

The `dbms_output.put_line('No records found.');` statement allows us to briefly mention PL/SQL debugging. `dbms_output` is a built-in Oracle Package (a package is a way to group together collections of stored procedures) that can be used to send messages to the console. In order to actually see these messages you must execute the `SET SERVEROUTPUT ON;` statement from within the SQL*Plus SQL editor. Each call to `dbms_output.put_line` will write out the string message passed to it.

Oracle uses the / character to mark the end of a block of SQL to execute within SQL*Plus.

Variable Declaration

At the start of a PL/SQL block you must define any variables that are to be used, after the `DECLARE` statement. You can use any of the standard Oracle data-types such as `NUMBER`, `VARCHAR2` or any PL/SQL data-type, such as `BOOLEAN`. It is just a case of defining the variable name followed by the data-type and using a semi-colon between multiple declarations:

```
DECLARE TotalBonus  NUMBER(6);
        BonusPaid   BOOLEAN;
```

For a full list of Oracle data-types check out
http://technet.us.oracle.com/doc/server.804/a58227/ch6.htm#649

Assigning Values to Variables

In ASP we use the = statement to assign a value to one of our variables. In PL/SQL it is slightly different, in that we must use `:=`.

```
SalePrice := (ProductPrice / 100)  * SalesTax;
```

If we are returning a value from a database table or system function, then we use the `INTO` statement:

```
SELECT SUM(Quantity) INTO ItemsOrdered FROM OrderBasket;
```

Conditional Flow of Control

We use the `If...Then...Else` construct to control the execution flow of our ASP scripts. PL/SQL also supports this construct in a similar format:

```
IF SaleCount > 10 AND SaleCount < 20 THEN
   UPDATE emp SET sal = sal * 0.3;
ELSIF SaleCount = 5 THEN
   UPDATE emp SET sal = sal * 0.2;
ELSE
   UPDATE emp SET sal = sal * 0.1;
END IF;
```

Surprisingly, PL/SQL doesn't yet provide support for the CASE statement.

Looping Flow Control

To loop through a section of code, PL/SQL supports a number of `LOOP` statements. The first is similar to the VBScript `For...Next` statement:

```
FOR countervar IN start..end LOOP
   statements to execute
END LOOP;
```

Where `countervar` is the counter variable, `start` is the initial starting value and `end` is the final value. For example:

```
FOR intCounter IN 1..5 LOOP
   INSERT INTO OrderLine(ID)
      VALUES(OrderLineID.NEXTVAL);
END LOOP;
```

The equivalent loop in VBScript would be:

```
FOR intCounter = 1 To 5
   Response.Write "Value=" & intCounter
NEXT
```

The `WHILE...LOOP` allows us to execute a section of code until a certain condition is true, just as we do with the `Do...Loop` structure in ASP:

```
WHILE TotalBonus < 10000 LOOP
   SELECT Bonus, EmpID INTO EmpBonus, MyEmpID
      FROM emp
      WHERE EmpID <> MyEmpID;
   Totalbonus := TotalBonus + Bonus;
   RecordCount := RecordCount + 1;
END LOOP;
```

Of course, there's a lot more to PL/SQL than that. PL/SQL is like any programming language with many constructs, statements and functions, but these are the typical building blocks that you will come across in any PL/SQL program.

Oracle Packages

We covered stored procedures a few chapters ago, so now we'll take a quick look at **Oracle Packages**. An Oracle package serves as a way to group procedures and functions into common groups typically based upon their functionality. A package has two sections: the **specification** that contains a definition of any objects that can be referenced outside of the package, and a **body** that contains the implementation of the objects. The specification section must be declared first:

```
PACKAGE package_name
IS
   {variable and type declarations }
   {cursor declarations}
   [module specifications]
END {package_name};
```

For example:

```
CREATE OR REPLACE PACKAGE Employee_pkg
AS

   PROCEDURE GetEmployeeName(i_empno    IN   NUMBER,
                             o_ename    OUT  VARCHAR2);
END Employee_Pkg;
```

This defines a package called `Employee_pkg` that contains a single stored procedure called `GetEmployeeAge`.

The package body contains the actual implementation of the procedures within the package. This effectively allows us to hide procedures inside the package by not declaring them in the package specification:

```
PACKAGE BODY package_name
IS
  {variable and type declarations}
  {cursor specifications - SELECT statements}
  [module specifications]
BEGIN
  [procedure bodies]
END {package_name};
```

The specification for our `Employee_pkg` could look like this:

```
CREATE OR REPLACE PACKAGE BODY Employee_pkg
AS

   PROCEDURE GetEmployeeName(i_empno    IN   NUMBER,
                             o_ename    OUT  VARCHAR2)
   IS
   BEGIN
      SELECT ename
         INTO o_ename
         FROM emp
      WHERE empno = i_empno;
   END GetEmployeeName;

END Employee_pkg;
```

To call the `GetEmployeeName` procedure within `Employee_pkg` from ASP we use must prefix the stored procedure name with the package name. We'll be covering the execution of stored procedures in much more detail in the next section:

```
   With objCommand
      .CommandText = "{call Employee_pkg.GetEmployeeName(?, ?)}"
      .CommandType = adCmdText
      .Parameters(0).Direction = adParamInput
      .Parameters(0).Value = varEmpNo
      .Parameters(1).Direction = adParamOutput
      .Execute
      Response.Write "Name=" & .Parameters(1).Value
   End With
```

Now that we've had a brief look at Oracle packages we can use some of their features in the final section in this chapter, when we come to retrieving ADO resultsets from an Oracle stored procedure. Before we do that, let's create a sample application that uses a number of stored procedures to perform common data-entry actions.

A Sample Oracle ASP Application

We are going to bring together all of the concepts discussed so far into a small ASP application based around the `scott` database schema. This application will show a list of employees from the `emp` table and allow the user to perform the usual data-entry procedures:

- Create a new employee
- Edit an existing employee
- Delete an employee

To implement this application we will be using four ASP script files, an include file, and the `global.asa` file. The include file is an ADO helper file that we have created ourselves called `ADOFunctions_inc.asp` used to create our database connections as needed.

> *It is often a good idea to rename your included ASP files from `.inc` to `.asp` to prevent unauthorized people from simply opening them in a browser. We've done this with `ADOFunctions_inc.asp` as it contains a username and password which we don't want people to have access to. I've kept the `_inc` suffix so that I know it's an include file.*

We will be retrieving lists of data using simple `SELECT` statements whereas the add, edit, update and delete functionality will be provided by four Oracle stored procedures. This will let us examine how we go about calling Oracle stored procedure using `INPUT` and `OUTPUT` parameters with the aid of the Microsoft OLE DB Provider for Oracle.

> *It is notoriously difficult to retrieve an ADO `Recordset` from an Oracle stored procedure. Oracle simply does not allow us to execute a `SELECT` statement from within stored procedure without assigning the returned values to a PL/SQL variable using the `INTO` keyword. There is a way to achieve this functionality with ADO using **PL/SQL tables** or by using **reference cursors**. In the next section, we will be covering the retrieval of an ADO `Recordset` from Oracle stored procedures using PL/SQL tables and then we'll look at doing the same thing using reference cursors and a PL/SQL package.*

One word of warning though, in order to concentrate on the Oracle fundamentals, we won't be using any DHTML features, so the screens do look rather bland!

global.asa

We won't use `global.asa` to handle application and session events, but we will use it to add a reference to the ADO type library to all of our ASP scripts. This will allow us to use the constants such as `adCmdText` for our ADO `Command` object. Enter the following line into `global.asa`:

```
<!-- METADATA TYPE="TypeLib"
     FILE="C:\Program Files\Common Files\System\ado\msado15.dll" -->
```

This uses the `METADATA` tag to include a `TYPELIB` file from the location specified. This is the default location into which the ADO library is located, but you should update it to reflect your own installation if it is different. By adding this line we can make use of all of the standard ADO constants and enumerators.

> *Traditionally, ASP developers would include the Microsoft ADO include file, `ADOVBS.inc`, in order to refer to the ADO constants. This would have to be done on every ASP script and is potentially difficult to support. By using the `METADATA` tag you only have to declare it once which is faster for your web server.*

ADOFunctions_inc.asp

This include file is used in all of our ASP scripts that need to connect to the database. It is much better to put commonly used code into a single include file and reference that in each of our pages, as there would be only one place in which we need to change the username and password if we ever needed to.

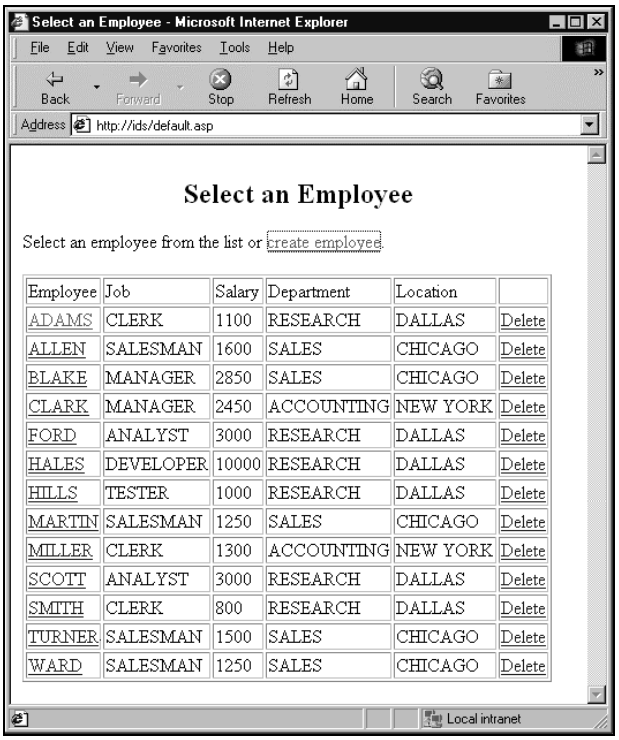
So create a new folder called `includes` and add a new file called `ADOFunctions_inc.asp` containing the following code:

```
<%
Function GetDBConnection()
   Dim objConnection

   Set objConnection = Server.CreateObject("ADODB.Connection")
   With objConnection
      .ConnectionString = "Provider=MSDAORA; " & _
                          "Data Source=Oracle8_dev; " & _
                          "User ID=scott; Password=tiger;"
      .Open
   End With

   Set GetDBConnection = objConnection
End Function
%>
```

The `GetDBConnection` function simply returns an `ADODB.Connection` object which points to our Oracle database using the `scott` account.

Default.asp

Our home page, `Default.asp`, displays a list of all employees from the `emp` table using a `SELECT` statement ordered by name. This page allows the user to create a new employee record by clicking the create employee link, delete an employee by pressing the Delete link, or edit an employee by clicking the employee's name. Both the edit and add employee link go to the `EditEmp.asp` page.



Employee	Job	Salary	Department	Location	
ADAMS	CLERK	1100	RESEARCH	DALLAS	Delete
ALLEN	SALESMAN	1600	SALES	CHICAGO	Delete
BLAKE	MANAGER	2850	SALES	CHICAGO	Delete
CLARK	MANAGER	2450	ACCOUNTING	NEW YORK	Delete
FORD	ANALYST	3000	RESEARCH	DALLAS	Delete
HALES	DEVELOPER	10000	RESEARCH	DALLAS	Delete
HILLS	TESTER	1000	RESEARCH	DALLAS	Delete
MARTIN	SALESMAN	1250	SALES	CHICAGO	Delete
MILLER	CLERK	1300	ACCOUNTING	NEW YORK	Delete
SCOTT	ANALYST	3000	RESEARCH	DALLAS	Delete
SMITH	CLERK	800	RESEARCH	DALLAS	Delete
TURNER	SALESMAN	1500	SALES	CHICAGO	Delete
WARD	SALESMAN	1250	SALES	CHICAGO	Delete

So let's have a look at the ASP code behind this page:

```
<% Option Explicit
   Response.Expires = 0 %>
<!-- #include file="includes/ADOFunctions_inc.asp" -->
<HTML>
<HEAD>
   <META HTTP-EQUIV="Pragma" CONTENT="no-cache">
   <TITLE>Select an Employee</TITLE>
</HEAD>
<BODY>
   <CENTER><H2>Select an Employee</H2></CENTER>
      Select an employee from the list or
      <A HREF="EditEmp.asp">create employee</A>.<P>
```

As usual we start off with the `Option Explicit` statement so that we must declare all variables and constants used in our code. We don't want this page to be cached by the browser so that any amended records are displayed each time the page is shown. We achieve this using `Response.Expires = 0` to tell the browser that this page expires immediately.If your site is going to be accessed by users in different time zones then it's a good idea to actually set this to a large negative number.

The line `<META>` tag is used to tell any proxy servers that they should not cache this page for the same reason.

You'll notice this is the first time that we include our `ADOFunctions.inc` using the `#include` directive.

```
<%
Dim objRecordset
Dim varSQL
Dim varEmpNo

varSQL = "SELECT emp.empno, emp.ename, emp.job, " & _
         "       emp.sal, dept.dname, dept.loc" & _
         "   FROM emp, dept" & _
         "   WHERE emp.deptno = dept.deptno" & _
         "   ORDER BY UPPER(emp.ename)"
Set objRecordset = GetDBConnection().Execute(varSQL)

Response.Write "<TABLE BORDER=1><TR>" & _
               "   <TD>Employee</TD>" & _
               "   <TD>Job</TD>" & _
               "   <TD>Salary</TD>" & _
               "   <TD>Department</TD>" & _
               "   <TD>Location</TD>" & _
               "   <TD> </TD>" & _
               "</TR>"
```

The `objRecordset` variable stores the result of our `SELECT` statement executed by calling the `GetDBConnection` function to return an ADO `Connection`.

As with SQL Server, Oracle also supports table name aliases that can be used for long or duplicated tables, such as:

```
SELECT EmpHol.Name
FROM EmployeesOnHoliday EmpHol
WHERE EmpHol.Department=1
```

Now we fill out the table with the data:

```
Do While Not objRecordset.EOF

   varEmpNo = objRecordset.Fields("empno")

   Response.Write "<TR>" & _
                  "   <TD><A HREF=EditEmp.ASP?EmpNo=" & varEmpNo & ">" & _
                              objRecordset("ename") & "</A></TD>" & _
                  "   <TD>" & objRecordset("job") & "</TD>" & _
                  "   <TD>" & objRecordset("sal") & "</TD>" & _
                  "   <TD>" & objRecordset("dname") & "</TD>" & _
                  "   <TD>" & objRecordset("loc") & "</TD>" & _
                  "   <TD><A HREF=javascript:deleteEmployee(" & _
                  varEmpNo & ");>Delete</A></TD>" & _
                  "</TR>"

   objRecordset.MoveNext
Loop
Response.Write "</TABLE>"
```

We navigate through the records contained in the `Recordset` object, creating a table row for each employee. We cache the employee number as it is used as part of the URL for the hyperlink to `EditEmp.asp`.

```
Set objRecordset = Nothing
%>

<SCRIPT>
function deleteEmployee(EmpNo) {
   if (window.confirm("Are you sure you want to delete employee?") == true)
   {
      window.location = "DeleteEmp.ASP?EmpNo=" + EmpNo;
   }
}
</SCRIPT>
</BODY>
</HTML>
```

We finish by closing off the ASP script and defining the local JavaScript function `deleteEmployee`. This function uses the `window.confirm` function to confirm whether the record should be deleted. If `Yes`, then the employee delete script, `DeleteEmp.asp` is called.

DeleteEmp.asp

This page simply calls an Oracle stored procedure, `emp_Delete`, passing in the employee number so that it can be deleted from the `emp` table.

We've covered stored procedures earlier in this book, so we'll just explain the important parts of this new procedure. This stored procedure doesn't come as part of the default database, so we are going to create it ourselves. Using SQL*Plus, or your preferred Oracle editor, you will need to connect to the `scott` account and execute the following SQL to create the new procedure:

```
CREATE OR REPLACE PROCEDURE emp_Delete
   (i_empno IN NUMBER)
AS
BEGIN
   DELETE
      FROM emp
      WHERE empno = i_empno;
END;
```

As you can see, it is a very simple procedure that takes one input parameter, `i_empno`, and deletes the record with the corresponding employee number from the `emp` table. We use the `IN` statement to tell Oracle that this parameter is for input only. You must tell Oracle if you want the value of parameters to be updated as the procedure exits, using the `OUT` statement, in exactly the same way that you should use the `ByVal` and `ByRef` statements in your own ASP procedures. I tend to prefix the name of each parameter with an `i_` or `o_` to denote the direction. You can also specify a parameter as being both `IN` and `OUT` but that's not a recommended practice.

So jumping back to `DeleteEmp.asp`, we have the following code:

```
<% Option Explicit
   Response.Buffer = True
%>
<!-- #include file="includes/ADOFunctions_inc.asp " -->
<HTML>
<%
Dim objCommand
Dim varEmpNo

varEmpNo = Request.QueryString("EmpNo")

Set objCommand = Server.CreateObject("ADODB.Command")
Set objCommand.ActiveConnection = GetDBConnection()
```

This time we are using the ADO `Command` object to execute our stored procedure because we need to get at the parameters that make up this stored procedure. This is more important when you want to retrieve the value of output parameters, as they are only accessible from the `Command` object's `Parameters` collection rather than a `Recordset`.

```
With objCommand
   .CommandText = "{call emp_Delete(?)}"
   .CommandType = adCmdText
   .Parameters(0).Value = varEmpNo
   .Execute()
End With

Set objCommand = Nothing
Response.Redirect "default.asp"
%>
</HTML>
```

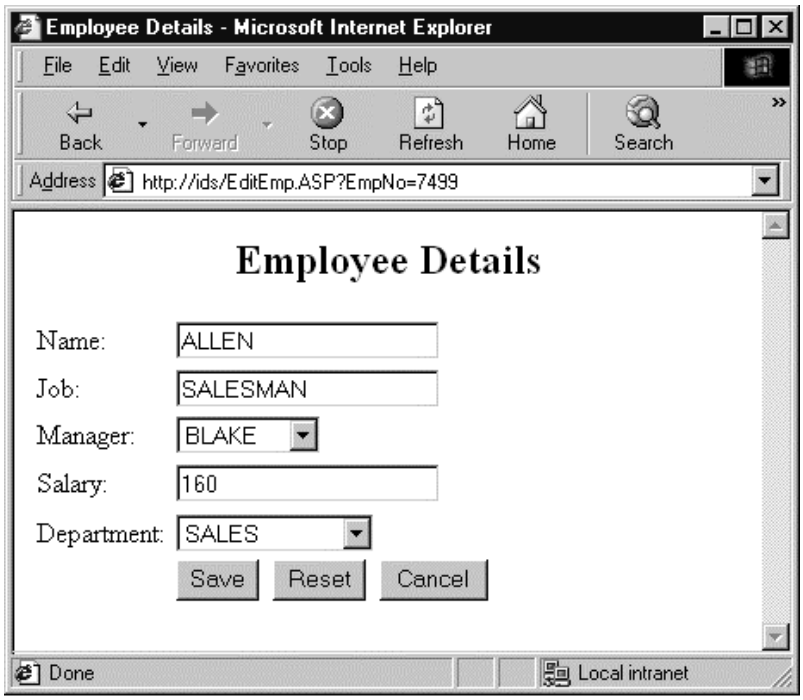
We use the `CommandText` property to tell the `Command` object the SQL statement to execute using the `{call` *`procname`*`}` syntax. Each `?` refers to a parameter to this stored procedure and can be referenced in the `Command` object's `Parameters` collection – starting from 0. We simply set the first and only parameter to that of the employee number passed in through the URL and then run the procedure using the `Execute` function.

Finally, we redirect the user back to our home page, `default.asp`.

> *Another approach could have been to open a new pop-up window to confirm the delete, which would have then refreshed* `default.asp` *using the JavaScript* `window.opener` *property, if the delete was successful. If the delete operation failed, for any reason, the pop-up window could have stayed open displaying the error message that was returned.*

EditEmp.asp

This page allows existing employee's records to be updated or new ones to be added. If this is an existing employee record, we will be passed the employee number as part of the URL. If there is no employee number, then the page assumes that the user wants to add a new employee record.



This page uses a stored procedure, `emp_GetData`, to return the fields for this employee through a number of output parameters, so you'll need to create the following stored procedure:

> *Remember that with Oracle, we cannot simply execute a* `SELECT` *statement inside a stored procedure to return some records as we can with SQL Server!*

```
CREATE OR REPLACE PROCEDURE emp_GetData
  (i_empno    IN   NUMBER,
   o_ename    OUT  VARCHAR2,
   o_job      OUT  VARCHAR2,
   o_mgr      OUT  NUMBER,
   o_sal      OUT  NUMBER,
   o_deptno   OUT  NUMBER)
AS
BEGIN
  SELECT ename, job, mgr,
         sal, deptno
    INTO o_ename, o_job, o_mgr,
         o_sal, o_deptno
    FROM emp
   WHERE empno = i_empno;
END;
```

This time we have only one input parameter and five output parameters that are used to store the employee details using the `SELECT...INTO` statement to transfer the values.

The ASP script has to do quite bit of work to display this page. It populates the list of departments and managers using a custom VBScript procedure that writes out a list of `OPTION` statements based on a `Recordset` of data, as we'll see shortly.

```
<% Option Explicit %>
<!-- #include file="includes/ADOFunctions_inc.asp " -->
<HTML>
<HEAD>
   <TITLE>Employee Details</TITLE>
</HEAD>
<BODY>
   <CENTER><H2>Employee Details</H2></CENTER>
<%
Dim objConnection
Dim objCommand
Dim objRSDepartments
Dim objRSManagers

Dim varEmpNo
Dim varEName
Dim varJob
Dim varMgr
Dim varSalary
Dim varDeptNo
```

We use a separate `Recordset` object to store the list of departments and managers so that we can populate the `SELECT` list in the correct place. I always find it easier to transfer the record values to local variables in one place.

```
On Error Goto Next
Set objConnection = GetDBConnection()

If Request.QueryString("EmpNo") = "" Then
   varEmpNo = 0
```

We create a database connection using `GetDBConnection`, and if there is no employee number passed in the URL, we set the employee number to zero. If the user clicked on an employee's name, we would have been passed the correct employee number.

> *For new employees, we use an Oracle Sequence to generate the new employee number, which we'll cover shortly.*

```
Else
   varEmpNo = Request.QueryString("EmpNo")
   Set objCommand = Server.CreateObject("ADODB.Command")
   Set objCommand.ActiveConnection = objConnection
   With objCommand
      .CommandText = "{call emp_GetData(?, ?, ?, ?, ?, ?)}"
      .CommandType = adCmdText
      .Parameters(0).Value = varEmpNo
      .Execute()
```

If we have an employee number then we need to create a `Command` object and specify the `emp_GetData` stored procedure. This time we have six parameters with the first one being the input parameter, the employee number, and the remaining five output parameters storing the employee's details.

```
        varEName    = .Parameters(1)
        varJob      = .Parameters(2)
        varMgr      = .Parameters(3)
        varSalary   = .Parameters(4)
        varDeptNo   = .Parameters(5)
     End With

  End If
```

Once we've called the `Execute` function, each of the `Parameters` items will contain our employee's fields so it's just a case of transferring them to our local variables.

```
  Set objRSDepartments = objConnection.Execute( _
                         "SELECT deptno, dname FROM dept ORDER BY dname")
  Set objRSManagers = objConnection.Execute( _
                      "SELECT empno, ename FROM emp ORDER BY ename")

  Set objCommand = Nothing
  Set objConnection = Nothing
```

We use our connection object, `objConnection`, to retrieve a list of departments and managers for our `SELECT` lists and then shut down the `Command` and `Connection` objects as soon as we've finished with them.

```
  Sub PopulateSelectOptions(ByVal objRecordset, ByVal varCurrentID)

     Dim varHTML
     Dim varSelected

     objRecordSet.MoveFirst

     Do While Not objRecordset.EOF
        If CLng(varCurrentID) = Clng(objRecordset.Fields(0)) Then
           varSelected = " SELECTED"
        Else
           varSelected = ""
        End If

        varHTML = varHTML & "<OPTION VALUE=" & objRecordset.Fields(0) & _
                  varSelected & ">" & objRecordset.Fields(1) & "</OPTION>"
        objRecordset.MoveNext
     Loop
     Response.Write varHTML
  End Sub
  %>
```

`PopulateSelectOptions` is a general-purpose procedure that is passed a `Recordset` of data and the ID of the default item to select. Its purpose is to navigate through each record and create a collection of HTML `OPTION` tags using the field at position 0 as the ID and field 1 as the text to display. If this was a full-blown application we'd probably put this function in an include file so that other pages could use its functionality, but as this is an example, we'll leave it in the ASP.

```
      <FORM ACTION="EditEmp_HND.ASP?EmpNo=<%=varEmpNo%>" METHOD="POST">
         <TABLE>
            <TR>
               <TD>Name:</TD>
               <TD><INPUT NAME="varEName" VALUE="<%=varEName%>"></TD>
            </TR>
            <TR>
               <TD>Job:</TD>
               <TD><INPUT NAME="varJob" VALUE="<%=varJob%>"></TD>
            </TR>
```

Now we can define the `FORM` that allows the user to enter the employee details. Notice that we append the employee number, which can be zero for new employee records, to the query string for the form action handler, `EditEmp_HND.asp`.

```
            <TR>
               <TD>Manager:</TD>
               <TD><SELECT NAME="varMgr" SIZE="1">
                  <%
                     Call PopulateSelectOptions(objRSManagers, varMgr)
                     Set objRSManagers = Nothing
                  %>               </SELECT></TD>
            </TR>
```

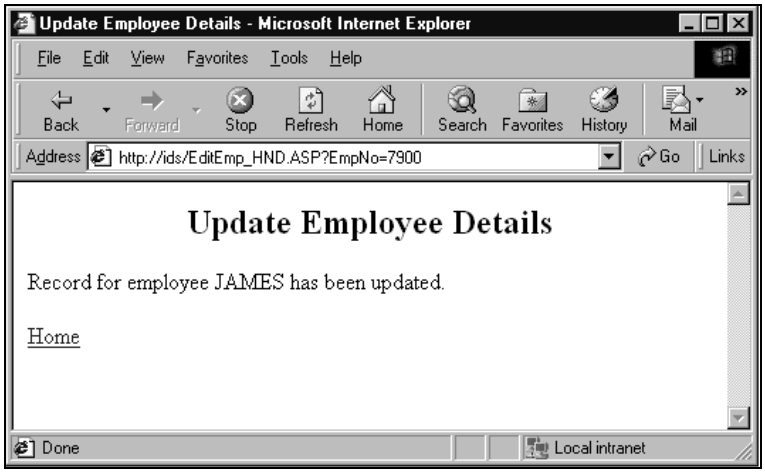
This is the first time that we call `PopulateSelectOptions` to create our list of `OPTION` tags. We already have the `<SELECT>` tag so `PopulateSelectOptions` will generate the corresponding list of `<OPTION>` tags for each record in `objRSManagers`.

```
            <TR>
               <TD>Salary:</TD>
               <TD><INPUT NAME="varSalary"
                          VALUE="<%= varSalary %>"></TD>
            </TR>
            <TR>
               <TD>Department:</TD>
               <TD><SELECT NAME="varDeptNo" SIZE="1">
                  <%
                     Call PopulateSelectOptions(objRSDepartments, varDeptNo)
                     Set objRSDepartments = Nothing
                  %>
                 </SELECT></TD>
          </TR>
            <TR>
               <TD></TD>
               <TD>
                  <INPUT TYPE="SUBMIT" VALUE="Save">  
                  <INPUT TYPE="RESET" VALUE="Reset"> 
                  <INPUT TYPE="BUTTON" VALUE="Cancel"
                       onclick="document.location.href='/';">
              </TD>
            </TR>

         </TABLE>
      </FORM>
   </BODY>
   </HTML>
```

We finish off by completing the input form, again using `PopulateSelectOptions` to show a list of departments, and adding a Submit to submit the form, a Reset button to clear any edits and a Cancel button to take the user back to the home page.

EditEmp_HND.asp

This page is the form handler that is called when the user submits the data-entry form. It calls the parameterized stored procedure `emp_Update` to update an existing record or add a new one using `emp_Add`.



Again, we are going to create these new stored procedures, so jump back to your SQL editor and execute the following lines:

```
CREATE OR REPLACE PROCEDURE emp_Update
  (i_empno    IN  NUMBER,
   i_ename    IN  VARCHAR2,
   i_job      IN  VARCHAR2,
   i_mgr      IN  NUMBER,
   i_sal      IN  NUMBER,
   i_deptno   IN  NUMBER)
AS
BEGIN
   UPDATE emp SET
      ename = i_ename, job = i_job,
      mgr = i_mgr, sal = i_sal,
     deptno = i_deptno
   WHERE empno = i_empno;
END;
```

Now that we've created the stored procedure for updates, we need to create an Oracle `Sequence` object before we create the `emp_Add` procedure. A sequence is an object that generates sequential numbers that we can use as primary keys for our employee number column. Oracle does not support the `IDENTITY` column that you would use in SQL Server so we must create a `Sequence` object to generate the numbers for us. Sequences are created separately from the table that they are created for, so if a table happens to be deleted (that is *dropped*) the sequence object will still exist. Each time you request the next number in the sequence using the `NEXTVAL` property, the sequence will automatically update itself irrespective of the table to column that it was originally created for.

So from your SQL editor execute the following statement to create the sequence:

```
CREATE SEQUENCE empno_seq START WITH 9000;
```

The sequence is called `empno_seq` and starts at 9000. The reason why I've decided to start at 9000 is because the `emp` table already contains some records and, in my case, the largest employee number was 7934, so I want to start at a number greater than 7934. A sequence has a number of properties that you can call, but `NEXTVAL` is the one we need to get the next number in the sequence.

Now that's done we can create the add stored procedure by running the following SQL:

```
CREATE OR REPLACE PROCEDURE emp_Add
   (i_ename    IN  VARCHAR2,
    i_job      IN  VARCHAR2,
    i_mgr      IN  NUMBER,
    i_sal      IN  NUMBER,
    i_deptno   IN  NUMBER)
AS
BEGIN
   INSERT INTO emp(empno,
                   ename, job, mgr,
                   sal, deptno)
           VALUES(empno_seq.NEXTVAL,
                  i_ename, i_job, i_mgr,
                  i_sal, i_deptno);
END;
```

The ASP script is relatively simple:

```
<% Option Explicit %>
<!-- #include file="includes/ADOFunctions_inc.asp " -->
<HTML>
<HEAD>
   <TITLE>Update Employee Details</TITLE>
</HEAD>
<BODY>
   <CENTER><H2>Update Employee Details</H2></CENTER>
<%
Dim objCommand

Dim varEmpNo
Dim varEName
Dim varJob
Dim varMgr
Dim varSalary
Dim varDeptNo
```

We will be using a `Command` object in order to set the stored procedure's parameters and local variables to store the value from the submitted form.

```
With Request

   varEmpNo = .QueryString("EmpNo")
   varEName    = .Form("varEName")
   varJob      = .Form("varJob")
   varMgr      = .Form("varMgr")
   varSalary   = .Form("varSalary")
   varDeptNo   = .Form("varDeptNo")

End With
```

We transfer the form fields into local variables.

```
Set objCommand = Server.CreateObject("ADODB.Command")
Set objCommand.ActiveConnection = GetDBConnection()

If varEmpNo <> 0 Then

   With objCommand
      .CommandText = "{call emp_Update(?, ?, ?, ?, ?, ?)}"
      .CommandType = adCmdText
      .Parameters(0).Value = varEmpNo
      .Parameters(1).Value = varEName
      .Parameters(2).Value = varJob
      .Parameters(3).Value = varMgr
      .Parameters(4).Value = CInt(varSalary)
      .Parameters(5).Value = varDeptNo
      .Execute()

      Response.Write "Record for employee " & varEName & _
                     " has been updated."
   End With
```

If we have employee number then it's just a case of calling the `emp_Update` stored procedure and pass in each of the values.

```
Else
   With objCommand
      .CommandText = "{call emp_Add(?, ?, ?, ?, ?)}"
      .CommandType = adCmdText
      .Parameters(0).Value = varEName
      .Parameters(1).Value = varJob
      .Parameters(2).Value = varMgr
      .Parameters(3).Value = CInt(varSalary)
      .Parameters(4).Value = varDeptNo
      .Execute()

      Response.Write "Record for employee " & varEName & " has been added."
   End With
End If

Set objCommand = Nothing
%>
   <P>
   <A HREF="default.asp">Home</A>
</BODY>
</HTML>
```

In the case of a new record, we call the `emp_add` stored procedure and pass in the new employee's details.

That concludes our brief ASP sample application based on the `scott` employee data. We've seen how it is possible to call stored procedures using the `{call` *`procname`*`?}` syntax to retrieve data for a single record and to manipulate records using the `Command.Parameters` collection. We made use of a standard include file to create our database connection and a useful function to output a list of `OPTION` tags based on a `Recordset` of data.

Retrieving ADO Recordsets from an Oracle Stored Procedure

We'll finish off with something of a holy grail. Unlike SQL Server, PL/SQL does not allow us to execute a `SELECT` statement within a stored procedure without a corresponding `INTO` statement. This means we cannot easily return a recordset back to the calling client whether it is an ASP script or another PL/SQL program.

Consider the following SQL Server stored procedure:

```
CREATE PROCEDURE sp_GetAuthors
AS
BEGIN
   SELECT au_lname, au_fname
   FROM authors
   ORDER BY au_lname, au_fname
END
```

Try creating the following very similar stored procedure in Oracle:

```
CREATE PROCEDURE sp_GetAuthors
AS
BEGIN
   SELECT ename
   FROM emp
   ORDER BY ename;
END;
```

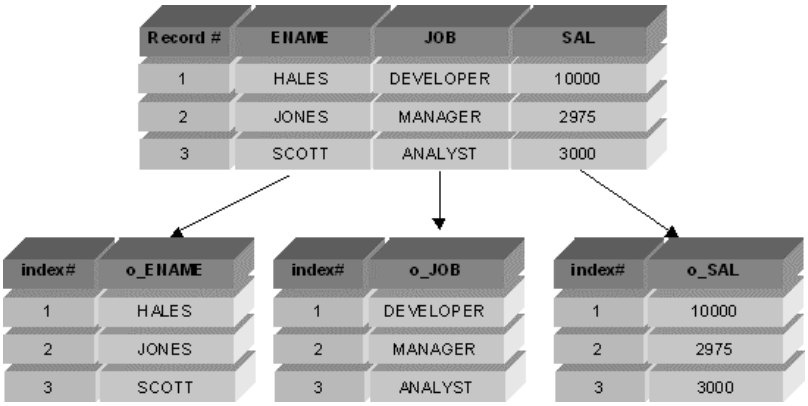
You'll receive the following error messages:

```
Errors for PROCEDURE SP_GETAUTHORS:

LINE/COL ERROR
-------- -----------------------------------------------------------------
4/3      PLS-00428: an INTO clause is expected in this SELECT statement
4/3      PL/SQL: SQL Statement ignored
```

Once upon a time, I searched Oracle's own PL/SQL documentation for an answer to this, and I got the impression that this will never be implemented. I believe the reason was, that they feel a calling program, X, should pass parameters into another program, Y, allowing Y to populate the results so that X can then deal with them. This approach doesn't really help us from an ADO point of view.

However, it can actually be achieved by using **PL/SQL tables** and the **Microsoft ODBC for Oracle**, or **reference cursors** with Oracle's **Oracle Provider for OLE DB**. We'll start off with PL/SQL tables and cover reference cursors in the next section.

PL/SQL Tables are somewhat of a misnomer as it might be easier if they were called *PL/SQL Arrays*. The following diagram shows three records from the `emp` table and how they would be represented in three PL/SQL Table variables:



We have three columns, `ENAME`, `JOB` and `SAL` in our source result set. For each column of data we have a corresponding PL/SQL table variable, `o_ENAME`, `o_JOB` and `o_SAL`, each mapping to a value of each column. The PL/SQL table variables are distinct entities in their own right. In order to populate the PL/SQL tables we need to scroll through the records in the source resultset, and add an entry for each column to the corresponding element in each PL/SQL table.

PL/SQL tables have the following characteristics:

- **One-Dimensional**: each PL/SQL table can contain only one column of data.
- **Integer-Indexed**: Each element of the array is indexed by a single integer much like a VBScript array.
- **Unbounded Dimensions**: There is no limit to the size of a PL/SQL table, as the structure will alter in size to accommodate new elements.
- **Uniform Data Type**: Only a single uniform data type can be stored in a particular PL/SQL table. So, if you start off with a `NUMBER` data-type, then all other elements must also be a `NUMBER`.

PL/SQL table types are defined using the `TYPE` statement, for example:

```
TYPE tblFirstName IS TABLE OF VARCHAR2(30) INDEX BY BINARY_INTEGER;
```

This would declare a PL/SQL table type called `tblFirstName` that could be used by a variable to store an array of strings up to 30 characters in length. A variable of this type could be declared as the parameter to a stored procedure, thus:

```
PROCEDURE GetEmployeeList(o_FirstName OUT tblFirstName)
```

Each PL/SQL table type that you want to use must be defined within the **specification** section of an Oracle **Package**.

In the case of a stored procedure that returns a list of employee names and numbers, we must create an individual parameter for both the employee name and the employee number values, both being declared using the PL/SQL table type as defined in our package specification.

In order to populate the employee number and employee name PL/SQL tables with data, we can use a **cursor** that loops through a selection of records and transfers each item of data into the corresponding PL/SQL table element.

> *A cursor allows you to programmatically step through a result set of data, performing operations based on the current row until the end of the result set is reached.*

The easiest way to implement an Oracle cursor is by declaring it outside of a program block and then opening it using a cursor `FOR...LOOP`. The cursor `FOR...LOOP` opens the cursor for you, repeatedly fetches rows of values from the result set into fields and then closes the cursor once all rows have been processed.

For example, the following cursor will calculate the total salary for all employees in the `emp` table:

```
DECLARE CURSOR emp_cur IS SELECT sal FROM emp;
               TotalSalary NUMBER;
BEGIN
   FOR emp_rec IN emp_cur LOOP
      TotalSalary := TotalSalary  + emp_rec.sal;
   END LOOP;
END;
```

We'll start our example off by creating a simple package that contains one stored procedure called `EmployeeSearch`. This will allow us to retrieve a list of employees from the `emp` table within the `scott` schema, based on their name.

Jump to your SQL editor and add the following package specification to the `scott` schema:

```
CREATE OR REPLACE PACKAGE Employee_Pkg
AS
TYPE tblEmpNo     IS TABLE OF NUMBER(4)    INDEX BY BINARY_INTEGER;
TYPE tblEName     IS TABLE OF VARCHAR2(10) INDEX BY BINARY_INTEGER;
TYPE tblJob       IS TABLE OF VARCHAR2(9)  INDEX BY BINARY_INTEGER;

PROCEDURE EmployeeSearch
   (i_EName   IN    VARCHAR2,
    o_EmpNo   OUT   tblEmpNo,
    o_EName   OUT   tblEName,
    o_Job     OUT   tblJob);
END Employee_Pkg;
```

Our package is called `Employee_Pkg`, which we will need to use when referencing the `EmployeeSearch` procedure. We will be returning three columns in our `Recordset`: employee number, name and job, so we have created a separate PL/SQL table type for each column.

Note that `EmployeeSearch` doesn't actually include any code – that's the job of the package body. If you try to define the implementation here you'll get an error from Oracle.

We've defined one input parameter, the name to search for, and a separate output parameter for each of the columns to return. Now we can create the package body – the bit that does the actual work, so execute the following SQL script:

```
CREATE OR REPLACE PACKAGE BODY Employee_Pkg
AS
PROCEDURE EmployeeSearch
   (i_EName   IN    VARCHAR2,
    o_EmpNo   OUT   tblEmpNo,
    o_EName   OUT   tblEName,
    o_Job     OUT   tblJob)
IS
```

We start off by adding the word `BODY` before the package name, dropping the PL/SQL table definitions, and adding the word `IS` to start the implementation.

```
CURSOR cur_employee (curName VARCHAR2) IS
   SELECT empno,
          ename,
          job
   FROM emp
   WHERE UPPER(ename) LIKE '%' || UPPER(curName) || '%'
   ORDER BY ename;

   RecordCount NUMBER DEFAULT 0;
```

If you recall from our overview of PL/SQL blocks, we need to declare any variables or cursors that are going to be used by our procedure. We define a cursor called `cur_employee` that has its own input parameter called `curName` and a number variable called `RecordCount` to store a count of the records processed.

Our cursor isn't that sophisticated: it uses || to add the wildcard character '%' to the beginning and the end of the required search name. In SQL Server, we would have used the + string concatenation operator. This enables the `LIKE` statement to find any employee's names that contain the specified characters. As we populate each of the PL/SQL table parameters we need to keep a track of the current element being set, so we use `RecordCount`. PL/SQL tables are 1-based so we must increment the `RecordCount` first as it starts from 0 initially.

```
   BEGIN
      FOR curRecEmployee IN cur_employee(i_EName) LOOP

         RecordCount:= RecordCount + 1;
         o_EmpNo(RecordCount):= curRecEmployee.empno;
         o_EName(RecordCount):= curRecEmployee.ename;
         o_Job(RecordCount):=   curRecEmployee.job;
      END LOOP;
   END EmployeeSearch;
END Employee_Pkg;
```

Here we have defined the actual implementation of the `EmployeeSearch` procedure. We simply open the cursor and ask it to transfer each record into a cursor variable called `curRecEmployee`. Notice that we didn't actually define the variable `curRecEmployee`, as this is simply a reference name to the record structure for the cursor. We can still refer to it within our cursor `FOR...LOOP` as though it was declared.

Then it's just a case of moving through each record, incrementing the record count, and transferring each individual field into each output parameter in the identical element position using `RecordCount`.

Now we need to call the procedure from an ASP script to populate the data. This is where you're likely to have the most problems when writing your own procedures. The following rules *must* be remembered, otherwise it simply won't work and you could spend days and days trying to work out why – as I did!

- Use the Microsoft ODBC Driver for Oracle.

  If you try to use the OLE DB Provider for Oracle you'll get an error message saying "Catastrophic Error"! You should also try to ensure that you're using at least version **2.573.4202.00** of the driver.

- Argument Naming and Positioning.

  When setting the `Command` object's `CommandText`, you must ensure that you use *exactly* the same name and same position for each parameter as you did when you declared each parameter in your stored procedure. If not, you'll get the rather misleading ODBC error message "Resultset column must be a formal argument".

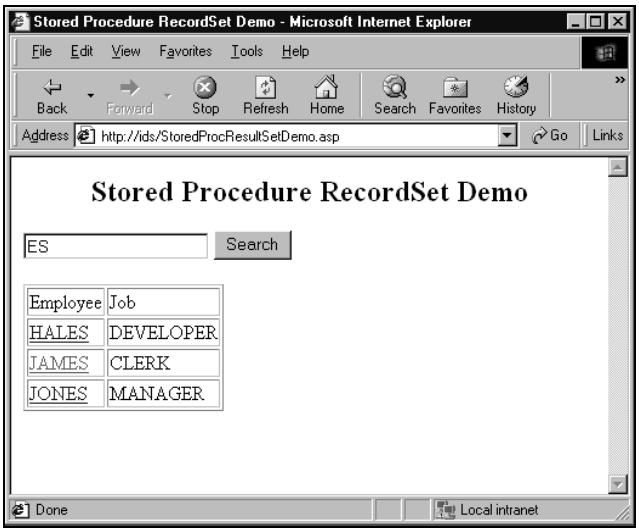
- Maximum Records Returned.

  You must use the `resultset` qualifier as part of your `CommandText` string to tell the driver which parameters are recordsets, such as:

```
"{call Employee_Pkg.EmployeeSearch("?, {resultset 100, o_EmpNo, o_EName,
o_Job})}"
```

  The number after `resultset` indicates the maximum number of records to be returned in this call. The driver actually allocates a memory cache to store this amount of data. (There appears to be no documentation that confirms what happens when the number of records is a lot less than this number.) If you exceed this number, by even one record, then you will receive Oracle error ORA-06512. It is suggested that you limit the number of records within your cursor population by passing the required value as an additional parameter to your stored procedure and limiting the cursor `FOR...LOOP`. We didn't do this in our example but it might be a nice exercise to try.

So we can now create a simple ASP script to call our procedure. I'm going to use a single ASP script that contains a form that submits to itself and writes out the search results.



```
<% Option Explicit
   Response.Expires = 0%>
<HTML>
<HEAD>
   <TITLE>Stored Procedure Recordset Demo</TITLE>
</HEAD>
<BODY>
   <CENTER><H2>Stored Procedure Recordset Demo</H2></CENTER>
<%
Dim strSearchName
Dim objConnection
Dim objCommand
Dim objRecordset
Dim varEmpNo

strSearchName = Request.Form("txtSearchName")
If strSearchName = "" Then strSearchName = "%"
```

We transfer the `txtSearchName` input field from the form into a variable. If it was empty, which it will be the first time, we set it to % so that we get all matching names.

```
Set objConnection = Server.CreateObject("ADODB.Connection")
With objConnection
   .ConnectionString = "driver={Microsoft ODBC for Oracle};" & _
                       "server=Oracle8_dev;UID=scott;PWD=tiger;"
   .CursorLocation = adUseClient
   .Open
End With
```

Here we connect to the database using the **Microsoft ODBC Driver for Oracle**.

Now for the fun part:

```
Set objCommand = Server.CreateObject("ADODB.Command")
With objCommand
   Set .ActiveConnection = objConnection

   .CommandText = "{call Employee_Pkg.EmployeeSearch(" & _
                  "?, {resultset 100, o_EmpNo, o_EName, o_Job})}"
   .CommandType = adCmdText

   .Parameters(0).Value = strSearchName
   Set objRecordset = .Execute()
End With
%>
```

We are using the standard `{call...}` and `?` syntax to define the first input parameter. Notice that we have included the `{resultset 100....}` string, as mentioned above, to define those parameters that are to be returned in the `Recordset` object and that we only want 100 records returned. We have simply pasted in the names of the parameters exactly as we declared them. The only parameter that we actually set is the first input parameter, the search name. Finally, we call the `Execute` statement to get our data.

What you do is navigate through the records in the `Recordset` and creating a nicely formatted HTML table.

```
<FORM ACTION="StoredProcResultSetDemo.asp" METHOD="POST">
<INPUT NAME="txtSearchName" VALUE="<%=strSearchName%>">
<INPUT TYPE="SUBMIT" VALUE="Search">
<P>
<TABLE BORDER=1>
  <TR><TD>Employee</TD><TD>Job</TD></TR>
<%
Do While Not objRecordset.EOF
   varEmpNo = objRecordset.Fields("o_EmpNo")
   Response.Write "<TR>" & _
                  "   <TD><A HREF=EditEmp.ASP?EmpNo=" & varEmpNo & ">" & _
                          objRecordset.Fields("o_EName") & "</A></TD>" & _
                  "   <TD>" & objRecordset.Fields("o_Job") & "</TD>" & _
                  "</TR>"

   objRecordset.MoveNext
Loop

Set objRecordset = Nothing
Set objCommand = Nothing
Set objConnection = Nothing
%>
</TABLE>
</FORM>
</BODY>
</HTML>
```

Retrieving ADO Recordsets using Reference Cursors

Oracle has released version 8.1.6 of its own provider, **Oracle Provider for OLE DB**. This provider has a class name of **OraOLEDB.Oracle** that is used when defining your ADO connection string. It supports the same set of Oracle data types as Microsoft's OLE DB Provider for Oracle with the additional support for the binary object types BLOB, CLOB, NCLOB, and BFILE, but as with Microsoft's provider, it also does not provide support for the Oracle8i object data types.

This provider gives us pretty much the same level of functionality as Microsoft's, except that it supports the use of Oracle reference cursors so that we can return back an ADO `Recordset` object from a stored procedure. A **reference cursor** is a pointer to a memory location that can be passed between different PL/SQL clients, thus allowing query result sets to be passed back and forth between clients.

A reference cursor is a variable type defined using the PL/SQL `TYPE` statement within an Oracle package, much like a PL/SQL table:

```
TYPE ref_type_name IS REF CURSOR RETURN return_type;
```

Here, `ref_type_name` is the name given to the type and `return_type` represents a record in the database. You do not have to specify the return type as this could be used as a general catch-all reference cursor. Such *non-restrictive* types are known as *weak*, whereas specifying the return type is *restrictive*, or *strong*. The following example uses `%ROWTYPE` to define a strong return type that represents the record structure of the `emp` table:

```
DECLARE TYPE EmpCurType IS REF CURSOR RETURN emp%ROWTYPE;
```

So let's jump straight to an example. We'll create a new Oracle package that contains a single procedure, `EmployeeSearch`, which returns a list of matching employee names. From your SQL editor, execute the following code to create the package specification:

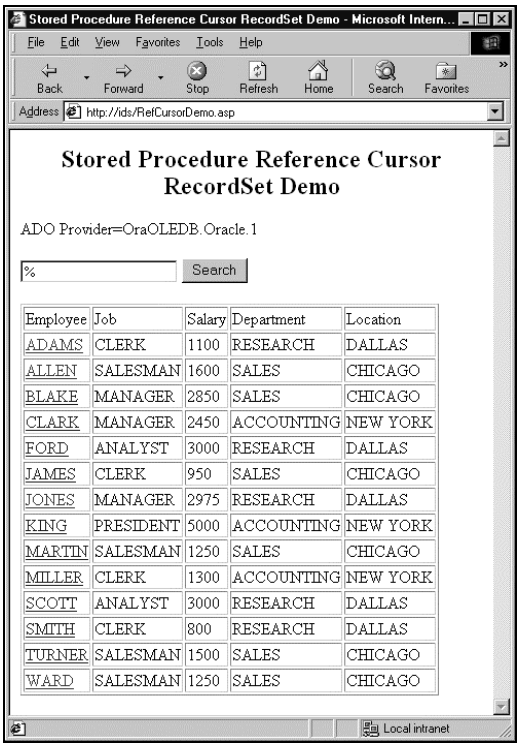
```
CREATE OR REPLACE PACKAGE Employee_RefCur_pkg
AS

   TYPE empcur IS REF CURSOR;
   PROCEDURE EmployeeSearch(i_EName     IN  VARCHAR2,
                            o_EmpCursor OUT empcur);
END Employee_RefCur_pkg;
```

We've created a new type called `empcur` that returns a weak reference cursor that we use as an output parameter to the `EmployeeSearch` procedure. Now we need the package body:

```
CREATE OR REPLACE PACKAGE BODY Employee_RefCur_pkg
AS
   PROCEDURE EmployeeSearch(i_EName     IN  VARCHAR2,
                            o_EmpCursor OUT empcur)
   IS
   BEGIN

      OPEN o_EmpCursor FOR
         SELECT emp.empno, emp.ename, emp.job,
             emp.sal, dept.dname, dept.loc
         FROM emp, dept
         WHERE ename LIKE '%' || i_EName || '%'
            AND emp.deptno = dept.deptno
         ORDER BY UPPER(emp.ename);
   END EmployeeSearch;
END Employee_RefCur_pkg;
```

This code is very similar to our previous stored procedure, except that we don't need to transfer each column in distinct PL/SQL tables, as the reference cursor, `o_EmpCursor`, is returned back to the client. The Oracle Provider for OLE DB converts any parameters that reference cursors into an ADO `Recordset` for us – but only if we add `PLSQLRSet=1` to our connection string, which we'll cover next.

Let's have a look at the results page that calls this stored procedure:



Employee	Job	Salary	Department	Location
ADAMS	CLERK	1100	RESEARCH	DALLAS
ALLEN	SALESMAN	1600	SALES	CHICAGO
BLAKE	MANAGER	2850	SALES	CHICAGO
CLARK	MANAGER	2450	ACCOUNTING	NEW YORK
FORD	ANALYST	3000	RESEARCH	DALLAS
JAMES	CLERK	950	SALES	CHICAGO
JONES	MANAGER	2975	RESEARCH	DALLAS
KING	PRESIDENT	5000	ACCOUNTING	NEW YORK
MARTIN	SALESMAN	1250	SALES	CHICAGO
MILLER	CLERK	1300	ACCOUNTING	NEW YORK
SCOTT	ANALYST	3000	RESEARCH	DALLAS
SMITH	CLERK	800	RESEARCH	DALLAS
TURNER	SALESMAN	1500	SALES	CHICAGO
WARD	SALESMAN	1250	SALES	CHICAGO

The actual ASP is very similar to our previous example so we'll just concentrate on the sections that are different:

```
<%
Dim strSearchName
Dim objConnection
Dim objCommand
Dim objRecordSet
Dim objNameParam
Dim varEmpNo

strSearchName = Request.Form("txtSearchName")
If strSearchName = "" Then strSearchName = "%"

Set objConnection = Server.CreateObject("ADODB.Connection")
```

So far it's just the same, except that we define a new variable, `objNameParam`, that we'll use as an ADO `Parameter` object to pass in the search name entered.

```
With objConnection
  .ConnectionString = "Provider=OraOLEDB.Oracle;" & _
                      "Data Source=Oracle8i_dev;" & _
                      "User ID=scott;" & _
                      "Password=tiger;" & _
                      "PLSQLRSet=1;"
  .Open
  Response.Write "ADO Provider=" & .Provider & "<P>"
End With
```

Here we tell ADO to use the Oracle Provider for OLE DB, `OraOLEDB.Oracle`, and we set the `PLSQLRSet` attribute to tell the provider that it should parse the PL/SQL stored procedures to determine if any parameters return a record set. `OraOLEDB` can only return one recordset per stored procedure. If you call a stored procedure that returns more than one recordset then `OraOLEDB` will only return the first argument of a ref cursor type.

If you omit the `PLSQLRSet` attribute, or you set it to `0`, then you'll receive the following Oracle error:

```
ORA-06550: line 1, column 7: PLS-00306: wrong number or types of arguments in call
to 'EMPLOYEESEARCH' ORA-06550: line 1, column 7: PL/SQL: Statement ignored
```

The rest of the code goes as follows:

```
Set objCommand = Server.CreateObject("ADODB.Command")
With objCommand
   Set .ActiveConnection = objConnection

   .CommandText = "{call Employee_RefCur_pkg.EmployeeSearch(?)}"
   Set objNameParam = .CreateParameter("SearchName", adBSTR, _
                                       adParamInput, , strSearchName)
   .Parameters.Append objNameParam

   Set objRecordSet = .Execute()
End With
```

Although our stored procedure has two parameters, the search name and the reference cursor that is returned, you must *not* bind the reference cursor as a parameter using the `?` attribute when using `OraOLEDB`, so we've included only one ? character to represent the `Name` input parameter.

The ADO `Parameter` object, `objNameParam`, is created using the `Command` object's `CreateParameter` function. `CreateParameter` is called in the following way:

```
Set parameter = command.CreateParameter(Name, Type, Direction, Size, Value)
```

`objNameParam` is declared as an `adBSTR` type because this maps to Oracle's `VARCHAR2` data type. Once we've created the `Parameter` we need to add it to the `Command` object's `Parameters` collection using the `Append` method.

Finally we call the `Execute` function to return a `Recordset` object that represents the result set from the `o_EmpCursor` reference cursor parameter. That's all there is to it. We can then navigate through the `Recordset` object as usual.

It's worth remembering that if you try to call the stored procedure using the `Parameters` collection directly:

```
.CommandText = "{call Employee_RefCur_pkg.EmployeeSearch(?)}"
.Parameters(0).Type = adBSTR
.Parameters(0).Direction = adParamInput
.Parameters(0).Value = strSearchName
```

you'll get the following runtime error:

```
The provider cannot derive parameter info and SetParameterInfo has not been called
```

Therefore you must use the `CreateParameter` function.

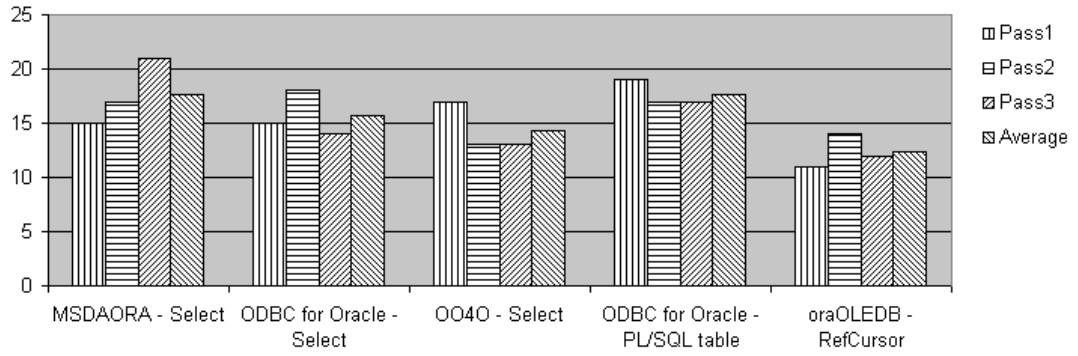
That wraps up our look at retrieving ADO `Recordset` objects from Oracle stored procedures. As you've seen, we have two choices: PL/SQL tables with the Microsoft ODBC for Oracle Driver or reference cursors with Oracle's Oracle Provider for OLE DB. On the face of it, the use of PL/SQL tables does appear rather convoluted in comparison to the ease of reference cursors. Both are relatively inefficient in terms of server performance and the Oracle Provider for OLE DB has been regarded as rather buggy. Again, it's your choice; it's difficult to define what each can and can't do. As ever, you should investigate how both methods perform in your own environment, looking at response times along with CPU and memory usage.

Summary

That just about brings us to the end of this guide to connecting to an Oracle database from an ASP application. We covered quite lot of ground here:

- Installation and configuration of the Oracle8 client software, Net8
- Using the Microsoft OLE DB Provider for Oracle
- Using the Microsoft OLE DB Provider for ODBC
- Using Oracle Objects for OLE (OO4O)
- PL/SQL fundamentals
- Creating a sample ASP application based on the `scott` account
- Showing that it is possible to retrieve an ADO `Recordset` from an Oracle stored procedure, using both PL/SQL tables and reference cursors

Before we finish this chapter, take look at the chart below comparing each of the common methods of data access for Oracle. I added an additional 7000 records to the `emp` table and then used each of the methods to retrieve these records and display them using an ASP script. Each method was executed three times and after each test I rebooted the server machine so that there would be very little chance of data being cached by either Oracle or the web server (for this test the web server also doubled as the Oracle database server to cut the time taken to shutdown and restart).



In these tests, MSDAORA was used with a standard SQL `SELECT` statement, as was the ODBC Driver for Oracle and OO4O, and finally I used the ODBC Driver for Oracle in conjunction with PL/SQL tables, and Oracle's Oracle Provider for OLE DB with a reference cursors as just described. The Y-axis shows the amount of time taken to complete each test in seconds. I also monitored the CPU and memory usage and they were all very similar for each test.

You can see that there is not that much difference between each method. When choosing which method to use, the underlying factor will always be good database design and coding practices.

> *Don't forget that you can download all of the SQL and ASP scripts for this chapter from the Wrox web site at* `http://www.wrox.com`.