Using Object Relational Features

Starting with Oracle 8, Oracle has given us the ability to use **object relational features** in the database. In short, the object relational features in Oracle allow you to extend the set of data types available in the database to go beyond simple **numbers**, **dates**, and **strings**. You can set up your own types that include:

- One or more attributes, where each attribute may be a scalar type, or a set (array) of other object/data types.
- One or more methods that operate on this type.
- One or more static methods.
- An optional comparison method used for sorting and equality/inequality comparisons.

You can then use this new type you have created in order to create database tables, specify columns of a table, create views, or as an excellent way to extend the SQL and PL/SQL languages. Once created, your data type is available to be used in the same manner as the basic data type **date**.

What I would like to do in this chapter is cover how I use the object relational features in Oracle. Just as important, will be how I do **not** use the feature. I will explain the components of this technology as I introduce them. However this should not be considered a complete overview of everything that you can possibly do with the object relational features in Oracle. This is covered in a 200 page manual from Oracle titled *Application Developer's Guide - Object-Relational Features*. The goal of this chapter is to expose why and how you would want to use these capabilities.

Oracle's object relational features are accessible using many languages. Languages such as Java via **JDBC**, Visual Basic with **OO4O (Oracle Objects for Ole)**, **OCI (Oracle Call Interface)**, **PL/SQL**, and **Pro*C** can all easily make use of this functionality. Oracle provides various tools to make using object relational features in these languages easy. For example, when using Java/JDBC, one might take
advantage of Oracle’s JPublisher, a utility that generates Java classes that represent database object types, collection types, and PL/SQL packages for you (it is a code generator that removes any complexity associated with mapping complex SQL types to Java types). OCI has a built-in client side object cache used to efficiently manipulate and work with objects. Pro*C has the OTT (Object Type Translator) tool to generate C/C++ structs for use in that language. In this book, we will not be investigating using these languages or tools specifically; each is documented in depth with the Oracle Server documentation. Rather we will focus on the implementation and creation of objects types in the database itself.

Reasons for Using These Features

The reason I use the object relational features in Oracle is predominantly as a means to extend the PL/SQL language in a natural fashion. The object type is an excellent way to extend PL/SQL with new functionality in the same way a class structure does this in C++ or Java. We will take a look at an example of doing this in the following section.

Object types can be used to enforce standardization as well. I can create a new type, say ADDRESS_TYPE, which encapsulates the definition of an address – the discrete components that make it up. I can even add convenience functions (methods) around this type, perhaps to return the address in a format suitable for printing on labels for example. Now, whenever I create a table needing a column that is an address, I can simply declare it as ADDRESS_TYPE. The attributes that constitute an address will be added to my table for me automatically. We will walk through an example of this as well.

Object types can be used to present an object relational view of strictly relational data. That is, I could take the simple EMP/DEPT example, and build an object relational view of it to expose each row of the DEPT table as if it contained a collection of EMP objects. Instead of joining EMP to DEPT, I can simply query the DEPT object view to see the DEPT and EMP information in a single row. In the next section, we’ll take a look at this example as well.

Object types may also be used to create object tables. We covered the pros and cons of object tables in Chapter 6 on Tables. Object tables have many hidden columns, side effects, and ‘magic’ happening behind them. Additionally, you usually need a strictly relational view of data for a variety of purposes (in particular, for the large number of existing utilities and report generators that do not ‘understand’ these new object types). I tend to not use them myself for this reason. I do use object views of relational data, which gives me the same effect in the end as an object table. However, I control the physical storage of everything. For this reason, we will not go into depth on object tables in this chapter.

How Object Relational Features Work

In this section we will look at using the object relational features in Oracle to perform the following goals:

- Impose standard data types on a system.
- Naturally extend the PL/SQL language.
- Present object relational views of inherently relational data.
Adding Data Types to your System

We will start with the basics here, the simple ADDRESS_TYPE. We will look at the syntax involved, what is possible, what side effects we might observe, and so on. In order to begin, we'll need a simple type to start with:

```sql
create or replace type Address_Type
  as object
  (  street_addr1   varchar2(25),
  street_addr2   varchar2(25),
  city           varchar2(30),
  state          varchar2(2),
  zip_code       number
  )
/
```

Type created.

This is the most basic sort of CREATE TYPE statement we can use. We'll add more features to it as we progress in this example. It is the most basic because it is a type composed only of other pre-existing scalar types; it has no methods, no comparison functions, nothing 'fancy'. We can immediately begin to use this type in our tables and PL/SQL code however:

```sql
create table people
  ( name           varchar2(10),
  home_address   address_type,
  work_address   address_type
  )
/
```

Table created.

```sql
declare
  l_home_address address_type;
  l_work_address address_type;
begin
  l_home_address := Address_Type( '123 Main Street', null, 'Reston', 'VA', 45678 );
  l_work_address := Address_Type( '1 Oracle Way', null, 'Redwood', 'CA', 23456 );
insert into people
  ( name, home_address, work_address )
values
  ( 'Tom Kyte', l_home_address, l_work_address );
end;
/
```

PL/SQL procedure successfully completed.

```sql
select * from people;
```
So, as you can see, using the new type in a CREATE TABLE is as easy as using the NUMBER type. Additionally, declaring variables of type ADDRESS_TYPE in PL/SQL is straightforward as well – PL/SQL is immediately aware of the new types. The new bit of functionality we see in the PL/SQL code is on lines 5 through 8. Here we are invoking the object constructor for the new type. The default object constructor in the type allows us to set all of the attributes of the object type to some value. There is, by default, only one default constructor and it must be invoked with a value for every attribute in the type. In the section on Using Types to Extend PL/SQL, we’ll see how to write our own custom constructors using static member functions.

Once we create and set variables of the type ADDRESS_TYPE, we can use them as bind variables in SQL very easily, as demonstrated above. We just insert the NAME, HOME_ADDRESS, and WORK_ADDRESS, and we are done. A simple SQL query retrieves the data. We can use SQL to not only gain access to the column HOME_ADDRESS, but to each of the components of HOME_ADDRESS as well. For example:

```
tkyte@TKYTE816> select name, home_address.state, work_address.state
2    from people
3  /
```

I've shown both the incorrect and correct method to do this. The first example is probably what most people would naturally try. It obviously does not work. To access the components of an object type, we must use a correlation name, as I did in the second query. Here I alias the table PEOPLE with P (any valid identifier could have been used, including the word PEOPLE itself). Then, when I want to reference the individual components of the addresses, I use the alias.

So, what does the physical table PEOPLE actually look like? What Oracle shows us, and what is really there, are quite different, as you might expect if you read Chapter 6 on Tables, and saw the nested table or object table example:
Oracle tells us we have three columns, the real data dictionary, however says thirteen. We can see our scalar columns hidden in there. Even though there is a little bit of magic and some hidden columns here, using scalar object types (no nested tables) in this fashion is very straightforward. This is the sort of magic we can live with. If we use the `SET DESCRIBE` option in SQL*PLUS, we can get SQL*PLUS to show us the entire hierarchy of our type:

```sql
tkyte@TKYTE816> set describe depth all

tkyte@TKYTE816> desc people

<table>
<thead>
<tr>
<th>Name</th>
<th>Null?</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td></td>
<td>VARCHAR2(10)</td>
</tr>
<tr>
<td>HOME_ADDRESS</td>
<td></td>
<td>ADDRESS_TYPE</td>
</tr>
<tr>
<td>STREET_ADDR1</td>
<td></td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>STREET_ADDR2</td>
<td></td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>CITY</td>
<td></td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>STATE</td>
<td></td>
<td>VARCHAR2(2)</td>
</tr>
<tr>
<td>ZIP_CODE</td>
<td></td>
<td>NUMBER</td>
</tr>
<tr>
<td>WORK_ADDRESS</td>
<td></td>
<td>ADDRESS_TYPE</td>
</tr>
<tr>
<td>STREET_ADDR1</td>
<td></td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>STREET_ADDR2</td>
<td></td>
<td>VARCHAR2(25)</td>
</tr>
<tr>
<td>CITY</td>
<td></td>
<td>VARCHAR2(30)</td>
</tr>
<tr>
<td>STATE</td>
<td></td>
<td>VARCHAR2(2)</td>
</tr>
<tr>
<td>ZIP_CODE</td>
<td></td>
<td>NUMBER</td>
</tr>
</tbody>
</table>
```

This is very handy for determining what attributes are available to us.
Chapter 20

Now, let's take our ADDRESS_TYPE one step further. We would like to have a convenient routine that returns a nicely formatted address for us in one field. We can do this by adding a member function to the type body:

```sql
tkyte@TKYTE816> alter type Address_Type
replace
as object
(  street_addr1  varchar2(25),
  street_addr2  varchar2(25),
  city          varchar2(30),
  state         varchar2(2),
  zip_code      number,
  member function toString return varchar2
)
/
Type altered.

tkyte@TKYTE816> create or replace type body Address_Type
as
member function toString return varchar2
is
begin
  if ( street_addr2 is not NULL )
    then
      return street_addr1 || chr(10) ||
      street_addr2 || chr(10) ||
      city || ',' || state || ' ' || zip_code;
  else
      return street_addr1 || chr(10) ||
      city || ',' || state || ' ' || zip_code;
  end if;
end;
/
Type body created.

tkyte@TKYTE816> select name, p.home_address.toString()
from people P
/
NAME
--------------------
P.HOME_ADDRESS.TOSTRING()
--------------------
Tom Kyte
123 Main Street
Reston, VA 45678
```

Here, we are looking at our first example of an object method. Each method is invoked with an implicit SELF parameter. We could have prefixed STREET_ADDR1, STREET_ADDR2, and so on with:

```sql
SELF.street_addr1 || chr(10) || SELF.street_addr2 ...
```
but it is implied for us. You might be looking at this and saying, ‘well, that’s nice but it is nothing we
couldn’t do with a relational table and a PL/SQL package.’ You would be correct. However, there are
advantages to using the object type with methods, as I have done above.

- **It serves as a better encapsulation mechanism** – The ADDRESS_TYPE encapsulates and
  enforces our concept of what an address is, what attributes it has, and what functionality is
  available.

- **It tightly binds methods to specific data** – This is a more subtle point but extremely important.
  If we implemented the above using scalar columns and a PL/SQL function to format them
  into a nice printable address, we could call this function with any data we wanted. I could pass
  in an EMPLOYEE_NUMBER as the zip code, a FIRST_NAME as the street address, and so on. By
  coupling the method with the attribute data, the TOSTRING method can only work on address
  data. The people who call this method do not have to be aware of the proper data to be
  passed – the proper data is ‘just there already’.

However, there is a disadvantage to the object type that you must be aware of. In Oracle 8i, the object

- type is not very ‘alterable’. You can add methods to the type via the ALTER statement, but you can
  neither remove nor add additional attributes once you have a table created using that type, nor can you
  remove methods once you’ve added them. The only thing you can do, pretty much, is add methods, or
  change their implementation (type body). In other words, schema evolution is not well supported using
  the type. If you discovered over time that you needed another attribute in the ADDRESS_TYPE object,
  you would have to rebuild objects that have that type embedded in them. This does not affect object
  types that are not used as columns in database tables, or as the type in a CREATE TABLE OF TYPE
  statement. That is, if you use object types solely in object views and as a method to extend PL/SQL (the
  following two sections) you can ignore this caveat.

One special set of methods associated with object types are the MAP and ORDER methods. These are

- used when sorting, comparing, or grouping instances of object types. If an object type does not have a
  MAP or ORDER function, you will find the following to be true:

```sql
tkyte@TKYTE816> select * from people order by home_address;
select * from people order by home_address
*      ERROR at line 1:
ORA-22950: cannot ORDER objects without MAP or ORDER method

select * from people where home_address > work_address;
select * from people where home_address > work_address
*      ERROR at line 1:
ORA-22950: cannot ORDER objects without MAP or ORDER method

select * from people where home_address = work_address;
no rows selected
```

You cannot order by the object type and you cannot use them in ‘greater than’ or ‘less than’ searches.

- The only thing you can do with them is to use them in direct equality comparisons. Then Oracle does
  an attribute-by-attribute compare for us to see if they are equal. The solution to the above is to add a
  MAP or an ORDER method – one or the other (an object type may only have a MAP or an ORDER method,
  but not both).
A MAP method is simply a function that works on a single instance of an object, and returns some scalar type that Oracle will use to compare to other object types. For example, if the object type in question represented a point with an X and Y co-ordinate, the MAP function might return the square root of \((X^2 + Y^2)\) – the distance from the origin. An ORDER method on the other hand receives two object instances; SELF and something to compare to SELF. The ORDER method returns 1 if SELF is greater than the other object, -1 if SELF is less than the other object, or 0 if they are equal. A MAP method is the preferred mechanism as it can be much faster and can even be invoked in a parallel query (whereas the ORDER method cannot). A MAP method has to be invoked once on an object instance and then Oracle can sort it – an ORDER function might be called hundreds or thousands of times with the same inputs to sort a large set. I’ll demonstrate both examples using the ADDRESS_TYPE from above. First the ORDER method:

```
tkyte@TKYTE816> alter type Address_Type
2  REPLACE
3 as object
4  ( street_addr1 varchar2(25),
5      street_addr2 varchar2(25),
6      city varchar2(30),
7      state varchar2(2),
8      zip_code number,
9      member function toString return varchar2,
10     order member function order_function(compare2 in Address_Type)
11     return number
12  )
13 /
Type altered.
tkyte@TKYTE816> create or replace type body Address_Type
2 as
3  member function toString return varchar2
4  is
5  begin
6    if ( street_addr2 is not NULL )
7      then
8        return street_addr1 || chr(10) ||
9            street_addr2 || chr(10) ||
10           city || ', ' || state || ' ' || zip_code;
11      else
12        return street_addr1 || chr(10) ||
13           city || ', ' || state || ' ' || zip_code;
14      end if;
15  end;
16  order member function order_function(compare2 in Address_Type)
18  return number
19  is
20  begin
21    if (nvl(self.zip_code,-99999) <> nvl(compare2.zip_code,-99999))
22      then
23        return sign(nvl(self.zip_code,-99999) - nvl(compare2.zip_code,-99999));
24      end if;
25    if (nvl(self.city,chr(0)) > nvl(compare2.city,chr(0)))
26      then
```
This would compare two addresses using the following algorithm:

1. If the ZIP_CODES differ, return -1 if SELF is less than COMPARE2, else return 1.

2. If the CITY differ, return -1 if SELF is less than COMPARE2, else return 1.

3. If the STREET_ADDR1 differ, return -1 if SELF is less than COMPARE2, else return 1.

4. If the STREET_ADDR2 differ, return -1 if SELF is less than COMPARE2, else return 1.

5. Else return 0 (they are the same).

As you can see, we have to worry about Nulls in the comparison and such. The logic is long and complex. It is definitely not efficient. Any time you are thinking of coding an ORDER member function, you should try to find a way to make it into a MAP member function instead. The above logic, I realize a better way would be to code it as a MAP member function. Note that if you have already altered the type to have the ORDER member function from above, you will have to drop the table that depends on this type, drop the type itself, and start over. Member functions cannot be removed, only added via the ALTER TYPE command and we need to get rid of the existing ORDER member function. The full example of the following would have a DROP TABLE PEOPLE, DROP TYPE ADDRESSS_TYPE, and a CREATE TYPE command preceding the ALTER TYPE:
By returning a fixed length string with the ZIP_CODE, then CITY, then STREET_ADDR fields, Oracle will do the sorting and comparisons for us.

Before we continue onto the next use of object types (my favorite use – as a means to extend PL/SQL), I would like to introduce the other collection type, VARRAYS. In Chapter 6 on Tables, we investigated nested tables and their implementation. We saw that they are nothing more than a parent/child table pair, implemented with a hidden surrogate key in the parent table and a NESTED_TABLE_ID in the child table. A VARRAY is, in many ways, similar to a nested table but is implemented very differently.
A VARRAY (or nested table) would be used to store an array of data associated with a single row. For example, if you had the need to store additional addresses with the PEOPLE table (perhaps an array of previous home addresses from oldest to newest), we could do the following:

```
tkyte@TKYTE816> create or replace type Address_Array_Type
  2  as varray(25) of Address_Type
  3  /
Type created.
tkyte@TKYTE816> alter table people add previous_addresses Address_Array_Type
  2  /
Table altered.
tkyte@TKYTE816> set describe depth all
tkyte@TKYTE816> desc people
Name                                  Null?    Type
------------------------------------- -------- --------------------
NAME                                           VARCHAR2(10)
HOME_ADDRESS                                    VARCHAR2(25)
  STREET_ADDR1                                 VARCHAR2(25)
  STREET_ADDR2                                 VARCHAR2(25)
  CITY                                         VARCHAR2(30)
  STATE                                        VARCHAR2(2)
  ZIP_CODE                                     NUMBER

METHOD
------
MEMBER FUNCTION TOSTRING RETURNS VARCHAR2

METHOD
------
MAP MEMBER FUNCTION MAPPING_FUNCTION RETURNS VARCHAR2
  WORK_ADDRESS                                    ADDRESS_TYPE
  STREET_ADDR1                                 VARCHAR2(25)
  STREET_ADDR2                                 VARCHAR2(25)
  CITY                                         VARCHAR2(30)
  STATE                                        VARCHAR2(2)
  ZIP_CODE                                     NUMBER

METHOD
------
MEMBER FUNCTION TOSTRING RETURNS VARCHAR2

METHOD
------
MAP MEMBER FUNCTION MAPPING_FUNCTION RETURNS VARCHAR2
  PREVIOUS_ADDRESSES                              ADDRESS_ARRAY_TYPE
    STREET_ADDR1                                 VARCHAR2(25)
    STREET_ADDR2                                 VARCHAR2(25)
    CITY                                         VARCHAR2(30)
    STATE                                        VARCHAR2(2)
    ZIP_CODE                                     NUMBER
```
Chapter 20

METHOD
------
MEMBER FUNCTION TOSTRING RETURNS VARCHAR2

METHOD
------
MAP MEMBER FUNCTION MAPPING_FUNCTION RETURNS VARCHAR2

So, now our table has the ability to optionally store up to 25 previous addresses. The question is; what went on behind the covers in order to facilitate this? If we query the ‘real’ data dictionary, we’ll see:

```
tkyte@TKYTE816> select name, length
2    from sys.col$
3    where obj# = ( select object
4                   from user
5                   where object
6                   name = ‘PEOPLE’ )
7  /
NAME                     LENGTH
-------------------- ----------
NAME                         10
HOME_ADDRESS                  1
SYS_NC00003$                 25
SYS_NC00004$                 25
SYS_NC00005$                 30
SYS_NC00006$                  2
SYS_NC00007$                 22
WORK_ADDRESS                  1
SYS_NC00009$                 25
SYS_NC00010$                 25
SYS_NC00011$                 30
SYS_NC00012$                  2
SYS_NC00013$                 22
PREVIOUS_ADDRESSES        2940
14 rows selected.
```

Oracle has added a 2,940 byte column to support our VARRAY implementation. The data for our VARRAY will be stored inline (in the row itself). This raises an interesting question; what will happen if our array could exceed 4,000 bytes (the largest structured column that Oracle supports)? If we drop the column and recreate it as a VARRAY(50), we can see what happens:

```
tkyte@TKYTE816> alter table people drop column previous_addresses
2  /
Table altered.
tkyte@TKYTE816> create or replace type Address_Array_Type
2  as varray(50) of Address_Type
3  /
Type created.
```
Using Object Relational Features

```
tkyte@TKYTE816> alter table people add previous_addresses Address_Array_Type
  2  /
Table altered.

/* the query that will be used to see if we have an object */
tkyte@TKYTE816> select object_type, object_name,
  2  decode(status,'INVALID','*','') status,
  3  tablespace_name
  4 from user_objects a, user_segments b
  5 where a.object_name = b.segment_name (+)
  6 order by object_type, object_name
  7  /
OBJECT_TYPE  OBJECT_NAME                    S TABLESPACE_NAME
------------ ------------------------------ - ------------------------------
LOB          SYS_LOB0000026301C00014$$        DATA
TABLE        PEOPLE                           DATA
TYPE         ADDRESS_ARRAY_TYPE
            ADDRESS_TYPE
TYPE BODY    ADDRESS_TYPE

/* the query that will be used to see if we have an object */
tkyte@TKYTE816> select name, length
  2  from sys.col$
  3  where obj# = ( select object_id
  4     from user_objects
  5       where object_name = 'PEOPLE' )
  6  /
NAME                     LENGTH
-------------------- ----------
NAME                         10
HOME_ADDRESS                  1
SYS_NC00003$                25
SYS_NC00004$                25
SYS_NC00005$                30
SYS_NC00006$                 2
SYS_NC00007$                22
WORK_ADDRESS                 1
SYS_NC00009$                25
SYS_NC00010$                25
SYS_NC00011$                30
SYS_NC00012$                 2
SYS_NC00013$                22
PREVIOUS_ADDRESSES          3852
14 rows selected.
```

What we see here now is that Oracle created a LOB for us. If the data in the VARRAY is under about 4,000 bytes, the data will be stored inline. If the data exceeds this, the VARRAY will be moved out-of-line into the LOB segment (just as any LOB would be).
VARRAYs are either stored as a RAW column inline, or as a LOB when they get too large. The overhead of a VARRAY (as compared to a nested table) is very small, making them a good choice as a method to store repeating data. VARRAYs can be searched on by un-nesting them, making them as flexible as nested tables in this respect:

```sql
tkyte@TKYTE816> update people
2    set previous_addresses = Address_Array_Type( 
3          Address_Type( '312 Johnston Dr', null, 
4              'Bethlehem', 'PA', 18017 ),
5          Address_Type( '513 Zulema St', 'Apartment #3', 
6              'Pittsburg', 'PA', 18123 ),
7          Address_Type( '840 South Frederick St', null, 
8              'Alexandria', 'VA', 20654 ));
1 row updated.

tkyte@TKYTE816> select name, prev.city, prev.state, prev.zip_code
2    from people p, table( p.previous_addresses ) prev
3   where prev.state = 'PA';

<table>
<thead>
<tr>
<th>NAME</th>
<th>CITY</th>
<th>ST</th>
<th>ZIP_CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom Kyte</td>
<td>Bethlehem</td>
<td>PA</td>
<td>18017</td>
</tr>
<tr>
<td>Tom Kyte</td>
<td>Pittsburg</td>
<td>PA</td>
<td>18123</td>
</tr>
</tbody>
</table>
```

One big difference here is that in the nested table implementation, we could have created an index on the nested table’s STATE column, and the optimizer would have been able to use that. Here, the STATE column cannot be indexed.

The main differences between nested tables and VARRAYs can be summarized as follows:

<table>
<thead>
<tr>
<th>Nested Table</th>
<th>VARRAY</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘Array’ elements have no specific order. The data in the collection may be returned in a very different order than it was in when you inserted it.</td>
<td>VARRAYs are true arrays. The data will remain inherently ordered as you left it. In our example above, the addresses are appended to the array. This implies that the oldest address is the first address, and the last previous address is the last address found in the array. A nested table implementation would need another attribute in order to identify the relative age of an address.</td>
</tr>
<tr>
<td>Nested tables are physically stored as a parent child table with surrogate keys.</td>
<td>VARRAYs are stored as a RAW column or as a LOB. There is minimal overhead introduced for the functionality.</td>
</tr>
<tr>
<td>Nested tables have no upper bound as to the number of elements that can be stored.</td>
<td>VARRAYs have an upper bound on the number of elements that can be stored. This maximum upper bound is defined at the time the type itself is created.</td>
</tr>
</tbody>
</table>
Nested Table  VARRAY

Nested tables may be modified (inserted/updated/deleted) from, using SQL.

VARRAYs must be procedurally modified. You cannot:

INSERT INTO
TABLE ( SELECT P.PREVIOUS_ADDRESSES
FROM PEOPLE P )
VALUES ... 

as you could with a nested table. To add an address, you would have to use procedural code (see example below).

Nested tables will perform a relational JOIN to bring the collection back with the row. For small collections, this may be expensive.

VARRAYs do not join. The data is accessed inline for small collections, and as a LOB segment for large collections. In general, there will be less overhead associated with accessing a VARRAY compared to a nested table. There is potentially more overhead associated with updating a VARRAY as compared to a nested table however, since the entire VARRAY must be replaced – not just an element of it.

In the table above, I mentioned that VARRAYs cannot be modified using SQL and the TABLE clause, we must procedurally process them. You will most likely write a stored procedure to facilitate the modification of VARRAY columns. The code would look something like this:

```
tkyte@TKYTE816> declare
2    l_prev_addresses   address_Array_Type;
3  begin
4          select p.previous_addresses into l_prev_addresses
5            from people p
6           where p.name = 'Tom Kyte';
7
8          l_prev_addresses.extend;
9          l_prev_addresses(l_prev_addresses.count) :=
10              Address_Type( '123 Main Street', null,
11                          'Reston', 'VA', 45678 );
12
13          update people
14             set previous_addresses = l_prev_addresses
15           where name = 'Tom Kyte';
16  end;
17  /
PL/SQL procedure successfully completed.
tkyte@TKYTE816> select name, prev.city, prev.state, prev.zip_code
2    from people p, table( p.previous_addresses ) prev
3  /
```
Adding Data Types Wrap-Up

In this section, we reviewed the advantages and disadvantages of using the Oracle type extensions in your database tables. You must decide if the ability to create new data types, with standard column definitions and methods that may operate on those types unambiguously, is overshadowed by the inability to evolve these types over time (add/remove attributes).

I would like to point out that in Oracle 9i, this changes radically as the ability to evolve the type schema over time is added.

We also reviewed the use of VARRAYs versus nested tables as a physical storage mechanism. We have seen how VARRAYs are excellent for storing a bounded set of ordered items compared to a nested table. You will find VARRAYs useful in many cases where you need to store a list of items, such as previous addresses, phone numbers, names of pets, and so on. Any time you have a list of items that do not necessitate an entire table all to themselves, VARRAYs will be useful.

Judicious use of types can add much to your system and its design. Using Oracle object types as columns in a table (as opposed to a table created as a TYPE, seen in Chapter 6 on Tables) is useful to enforce standardization, and ensure that the procedures (methods) are invoked with the correct inputs. The downside is the current lack of true schema evolution for the types, once you have created a table, which uses that type.

Using Types to Extend PL/SQL

This is where the object relational features of Oracle excel. PL/SQL is a very flexible, and capable, language as evidenced by the fact that Advanced Replication was written entirely in PL/SQL way back in Oracle 7.1.6. Oracle Applications (Human Resources, Financial Applications, CRM applications, and so on) are developed using PL/SQL as one of the predominant languages. In spite of its agility as a programming language, there will be times when you want to extend its capabilities – just as you would in Java, C++, or any language. Object types are the way to do this. They add functionality to PL/SQL similar to what the class adds to Java or C++.

In this section, I would like to create an example that demonstrates how I use object types to make PL/SQL programming easier. Here I would like to create a File Type built on top of UTL_FILE. UTL_FILE is a supplied Oracle package that allows PL/SQL to perform text I/O operations (reading and writing) on the server's file system. It is a procedural API similar to the C language's family of functions (fopen, fclose, fread, fwrite, and so on). We would like to encapsulate the functionality of UTL_FILE in an easier-to-use object type.
Creating a New PL/SQL Data Type

UTL_FILE works by returning a PL/SQL RECORD type. This will complicate our implementation slightly but can be worked around. The reason it complicates things is because a SQL object type can only contain SQL types, not PL/SQL types. Hence, we cannot create an object type that contains a PL/SQL RECORD in it, but we must if we want to encapsulate the functionality. In order to solve this, we'll use a small PL/SQL package in conjunction with our type.

I'll start with the type specification – our prototype for what we will build:

```sql
create or replace type FileType
as object
  begin
  g_file_name      varchar2(255),
  g_path           varchar2(255),
  g_file_hdl       number,

  static function open( p_path        in varchar2,
                     p_file_name    in varchar2,
                     p_mode         in varchar2 default 'r',
                     p_maxlinesize  in number default 32765 )
    return FileType,

  member function isOpen return boolean,
  member procedure close,
  member function get_line return varchar2,
  member procedure put( p_text in varchar2 ),
  member procedure new_line( p_lines in number default 1 ),
  member procedure put_line( p_text in varchar2 ),
  member procedure putf( p_fmt  in varchar2,
                      p_arg1 in varchar2 default null,
                      p_arg2 in varchar2 default null,
                      p_arg3 in varchar2 default null,
                      p_arg4 in varchar2 default null,
                      p_arg5 in varchar2 default null ),
  member procedure flush,

  static procedure write_io( p_file      in number,
                           p_operation in varchar2,
                           p_parm1     in varchar2 default null,
                           p_parm2     in varchar2 default null,
                           p_parm3     in varchar2 default null,
                           p_parm4     in varchar2 default null,
                           p_parm5     in varchar2 default null,
                           p_parm6     in varchar2 default null )
  end;
/
Type created.
```

It looks a lot like the UTL_FILE package itself (if you are not familiar with UTL_FILE, you might want to read up on it in Appendix A on Necessary Supplied Packages). It provides almost the same functionality as the package, just more intuitively as you'll see (in my opinion). Now, if you remember from the ADDRESS_TYPE example above I said each object type has one default constructor, and that you must set each of the attributes of the object type to some value in that constructor. No user-defined code may
execute as a side effect of this default constructor. In other words, this default constructor can be used only to set every attribute of the object type. This is not very useful. The static function OPEN in the above type will be used to demonstrate how we can create our own, infinitely more useful [and complex], constructors for our types. Notice how OPEN, part of the FILETYPE object type, returns a FILETYPE itself. It will do the necessary setup work, and then return a fully instantiated object type for us. This is the main use of static member functions in object types – they provide the ability to create your own complex object constructors. Static functions and procedures in an object type differ from member procedures and functions in that they do not get an implicit SELF parameter. These functions are very similar in nature to a packaged procedure or function. They are useful for coding common utility routines that many of the other member functions would invoke, but do not themselves need access to the instance data (the object attributes). WRITE_IO in the above object type will be an example of just such a routine. I use this one routine to do all of the UTL_FILE calls that write to a file so as to not have to repeat the same 14-line exception block every time.

Now, you will notice that the UTL_FILE.FILE_TYPE data type is not referenced in this object type, and in fact, it cannot be. The object type attributes can only be SQL types. We must save this record elsewhere. In order to do this, I am going to use a PL/SQL package as follows:

```sql
create or replace package FileType_pkg
as
  type utl_fileArrayType is table of utl_file.file_type
    index by binary_integer;

  g_files utl_fileArrayType;

  g_invalid_path_msg constant varchar2(131) default 'INVALID_PATH: File location or filename was invalid.';

  g_invalid_mode_msg constant varchar2(131) default 'INVALID_MODE: The open_mode parameter %s in FOPEN was invalid.';

  g_invalid_filehandle_msg constant varchar2(131) default 'INVALID_FILEHANDLE: The file handle was invalid.';

  g_invalid_operation_msg constant varchar2(131) default 'INVALID_OPERATION: The file could not be opened or operated as requested.';

  g_read_error_msg constant varchar2(131) default 'READ_ERROR: An operating system error occurred during the read operation.';

  g_write_error_msg constant varchar2(131) default 'WRITE_ERROR: An operating system error occurred during the write operation.';

  g_internal_error_msg constant varchar2(131) default 'INTERNAL_ERROR: An unspecified error in PL/SQL.';

  g_invalid_maxlinesize_msg constant varchar2(131) default 'INVALID_MAXLINESIZE: Specified max linesize %d is too large or too small';
end;
/
```

Package created.
This package will be used at run-time to hold any, and all, UTL.FILE.FILE_TYPE records for us. Each object type instance (variable) we declare of FILE_TYPE will allocate themselves an empty ‘slot’ in the G_FILES array above. This shows a method to implement ‘private’ data in Oracle object types. We will store the real runtime data in this packaged array variable G_FILES and storing only the handle (the index into the array) in the object type. In Oracle's current object implementation, all data in an object type is PUBLIC data. There is no way to hide an attribute in the type, to make it inaccessible to the users of that type. For example, given the FILE_TYPE above, it will be legitimate for us to access the G_FILE_NAME instance variable directly. If that were not desirable, we would ‘hide’ this variable in a PL/SQL package in the same fashion that we are going to ‘hide’ the PL/SQL record type there. No one can access the data in the PL/SQL package unless we grant EXECUTE on that package to them, therefore it is protected data.

This package is also used to hold some constants for us. Object types do not support constant data, hence the package becomes a nice holding place for that as well.

I tend to name a package that supports a type like this, after the type itself. So, since we created the type FILETYPE, I have a FILETYPE_PKG to go along with it. Now we are ready for the FILETYPE type body. This will contain the implementation of all of our member and static functions and procedures from above. I'll present the code with comments about what it is doing interspersed throughout:

```sql
create or replace type body FileType
as
  static function open( p_path        in varchar2,
                        p_file_name   in varchar2,
                        p_mode        in varchar2 default 'r',
                        p_maxlinesize in number default 32765 )
    return FileType
  is
    l_file_hdl number;
    l_utl_file_dir varchar2(1024);
  begin
    l_file_hdl := nvl( fileType_pkg.g_files.last, 0 )+1;
    fileType_pkg.g_files(l_file_hdl) :=
      utl_file.fopen( p_path, p_file_name, p_mode, p_maxlinesize );
    return fileType( p_file_name, p_path, l_file_hdl );
  exception
    when utl_file.invalid_path then
      begin
        execute immediate 'select value
                                   from v$parameter
                                   where name = :1';
      end;
  end;
```

The above portion of the static member function OPEN is responsible for finding an empty slot in our private data (hidden in the filetype_pkg package). It does this by adding one to the LAST attribute of the table. If the table is empty, LAST returns NULL so we NVL this value – the first entry we will allocate will be the array index 1. The next one will be 2 and so on. Our CLOSE function deletes the entries as we close the file so we will reuse space in this array over time as we open and close files. The remainder of the function is very straightforward; it opens the requested file and returns a fully instantiated instance of a FILETYPE object for us to use. The rest of the FILETYPE.OPEN method is an exception block to catch and handle all of the errors that UTL_FILE.FOPEN might raise:

```sql
exception
  when utl_file.invalid_path then
    begin
      execute immediate 'select value
                                   from v$parameter';
    end;
```

885
This exception block is designed to catch and re-raise all UTL_FILE exceptions in a better way than UTL_FILE does natively. Instead of receiving the SQLERRM of USER DEFINED EXCEPTION in the invoking routine as you would normally, we’ll receive something meaningful like THE OPEN MODE PARAMETER WAS INVALID. Additionally, for the INVALID_PATH exception, which is raised if the file could not be opened due to an invalid path or filename, we go out of our way to provide a meaningful error message. If the OWNER of this type has been granted SELECT on SYS.V_$PARAMETER, we will retrieve the entire UTL_FILE_DIR INIT.ORA parameter and verify the path we are attempting to use is in fact set up to be used. If it is not we return an error message stating that this is the case. Of all of the errors raised by UTL_FILE, this one exception is by far the most ‘popular’. Having an error message this meaningful will save many hours of ‘debugging’ for the novice UTL_FILE user.

Continuing on, we have the is Open method:

It is simply a layer on top of the existing UTL_FILE.IS_OPEN. Since this UTL_FILE function never raises any errors, it is a very easy routine to implement. The next method is GET_LINE, its logic is a little more involved:
Here we use a local variable of type VARCHAR2(32765), which is the largest PL/SQL variable you can have and is the largest line UTL_FILE can actually read. Again, much like the OPEN method above, we catch and handle each and every error that UTL_FILE.GET_LINE can raise and convert it into a RAISE_APPLICATION_ERROR call. This allows us to get meaningful error messages from the GET_LINE function now (and makes GET_LINE a function, not a procedure, which is done for convenience).

Now for another static procedure; WRITE_IO. The sole purpose of WRITE_IO is to avoid having to code the same exception handler in six times for each of the WRITE oriented routines, because they all throw the same exceptions. This function is here purely for convenience and simply calls one of six UTL_FILE functions and handles the errors generically:
Chapter 20

888

p_parm3,p_parm4,p_parm5,
p_parm6);
else raise program_error;
end if;
exception
when utl_file.invalid_filehandle then
  raise_application_error
(-20002,fileType_pkg.g_invalid_filehandle_msg);
when utl_file.invalid_operation then
  raise_application_error
(-20003,fileType_pkg.g_invalid_operation_msg);
when utl_file.write_error then
  raise_application_error
(-20005,fileType_pkg.g_write_error_msg);
when utl_file.internal_error then
  raise_application_error
(-20006,fileType_pkg.g_internal_error_msg);
end;

The six remaining methods simply call the WRITE_IO method to do their work:

member procedure close
is
begin
  fileType.write_io(g_file_hdl, 'close');
  filetype_pkg.g_files.delete(g_file_hdl);
end;

member procedure put( p_text in varchar2 )
is
begin
  fileType.write_io(g_file_hdl, 'put',p_text );
end;

member procedure new_line( p_lines in number default 1 )
is
begin
  fileType.write_io(g_file_hdl, 'new_line',p_lines );
end;

member procedure put_line( p_text in varchar2 )
is
begin
  fileType.write_io(g_file_hdl, 'put_line',p_text );
end;

member procedure putf
( p_fmt  in varchar2, p_arg1 in varchar2 default null,
p_arg2 in varchar2 default null, p_arg3 in varchar2 default null,
p_arg4 in varchar2 default null, p_arg5 in varchar2 default null )
is
begin
  fileType.write_io
  (g_file_hdl, 'putf', p_fmt, p_arg1,
p_arg2, p_arg3, p_arg4, p_arg5);
Now, you'll notice in the above that I catch each, and every UTL_FILE exception, and raise another error using RAISE_APPLICATION_ERROR. This is the main reason I decided to encapsulate UTL_FILE in the first place. UTL_FILE uses 'USER-DEFINED EXCEPTIONS' to raise errors. These exceptions are defined by the developers of UTL_FILE and when they raise these exceptions, the error message Oracle associates with that is simply 'USER-DEFINED EXCEPTION'. This is not very meaningful and doesn't help us figure out what went wrong. I prefer to use RAISE_APPLICATION_ERROR, which allows me to set the SQLCODE and SQLERRM returned to the client. To see the effect this can have on us, we just need to look at the following small example which demonstrates the types of error messages we will receive from UTL_FILE and FILETYPE:
It is not hard to tell which one is easier to figure out what exactly the error is. The second error message, since the owner of the type had access to V$PARAMETER, is very precise as to the exact cause of the error; the directory I used was invalid, it was not in the UTL_DIR init.ora parameter. Even if the owner didn’t have access to V$PARAMETER, the error would have been:

```sql
* ERROR at line 1:
ORA-20001: INVALID_PATH: File location or filename was invalid.
ORA-06512: at "TKYTE.FILETYPE", line 59
ORA-06512: at line 2
```

which is still much more useful than user-defined exception.

Another thing to notice in this type is that I am able to set up my preferred defaults for routines. For example, prior to Oracle 8.0.5, UTL_FILE was limited to a maximum line size of 1,023 bytes per line. If you tried to print out a line of text that exceeded this, UTL_FILE would raise an exception. By default, this behavior persists in Oracle 8i. Unless you call UTL_FILE.FOPEN and tell it to use a particular line size, it will still default to 1,023. I myself would rather have this default to the maximum size of 32 KB by default. I also implemented a default open mode at the start of the code of ‘R’ for read. Since 90 percent of the time I use UTL_FILE I’m using it to read a file, this made the most sense for me.

Now to exercise the package and show how each function/procedure works. The first example will create a file (assumes that we are running on Windows NT, that the c:\temp directory exists, and that the UTL_FILE_DIR init.ora parameter contains c:\temp) and write some known data to it. It will then close that file, saving the data. This demonstrates the WRITE functionality of the FILETYPE type:

```sql
tkyte@TKYTE816> declare
  2   f fileType := fileType.open('c:\temp', 'foo.txt', 'w');
  3   begin
  4     if ( f.isOpen )
  5        then
  6            dbms_output.put_line('File is OPEN');
  7        end if;
  8     for i in 1 .. 10 loop
  9        f.put( i || ',' );
 10      end loop;
 11      f.put_line(11);
 12     f.new_line(5 );
 13     for i in 1 .. 5
 14       loop
 15         f.put_line('line ' || i );
 16       end loop;
 17      f.puts( '%s %s', 'Hello', 'World' );
 18      f.flush;
 19      f.close;
 20   end;
 21 File is OPEN
PL/SQL procedure successfully completed.
```

...
The second half of the example demonstrates reading a file using FILETYPE. It will open the file we just wrote and verify the data we read in is exactly the data we expect:

tkyte@TKYTE816> declare
2      f fileType := fileType.open( 'c:\temp', 'foo.txt' );
3  begin
4      if ( f.isOpen )
5      then
6          dbms_output.put_line( 'File is OPEN' );
7      end if;
8
dbms_output.put_line
9      ( 'line 1: (should be 1,2,...,11)' || f.getLine );
10     for i in 2 .. 6
11       loop
12           dbms_output.put_line
13           ( 'line ' || i || ': (should be blank)' || f.getLine );
14       end loop;
15
16     for i in 7 .. 11
17       loop
18           dbms_output.put_line
19           ( 'line ' || to_char(i+1) || ': (should be line N)' || f.getLine );
20       end loop;
21
22     dbms_output.put_line
23      ( 'line 12: (should be Hello World)' || f.getLine );
24
25     begin
26          dbms_output.put_line( f.getLine );
27          dbms_output.put_line( 'the above is an error' );
28     exception
29      when NO_DATA_FOUND then
30          dbms_output.put_line( 'got a no data found as expected' );
31      end;
32      end;
33      f.close;
34  end;
35  /
File is OPEN
line 1: (should be 1,2,...,11)1,2,3,4,5,6,7,8,9,10,11
line 2: (should be blank)
line 3: (should be blank)
line 4: (should be blank)
line 5: (should be blank)
line 6: (should be blank)
line 8: (should be line N)line 1
line 9: (should be line N)line 2
line 10: (should be line N)line 3
line 11: (should be line N)line 4
line 12: (should be line N)line 5
line 12: (should be Hello World)Hello World
got a no data found as expected

PL/SQL procedure successfully completed.
We have encapsulated the UTL_FILE type using an Oracle object type. We have a nice layer on top of the supplied package that works the way we want it to exactly. In ‘object programming’ terms we have just extended the UTL_FILE class – implementing it with methods that work as we prefer, instead of exactly the way the Oracle developers set it up. We haven’t reinvented UTL_FILE, simply repackaged it. This is a good overall programming technique; if the implementation of UTL_FILE changes, or a bug is introduced during an upgrade, the odds are you can work around it in your type body, avoiding having to change hundreds or thousands of dependent routines. For example, in one release of UTL_FILE, opening a non-existent file in A mode didn’t work; it would not create the file as it should have. The work around was to code:

```sql
begin
  file_stat := utl_file.fopen(file_dir, file_name, 'a');
exception
  -- if file does not exist, fopen will fail with
  -- mode 'a' - bug:371510
  when utl_file.invalid_operation then
    -- let any other exception propagate
    -- to the outer block as normal
    file_stat := utl_file.fopen(file_dir, file_name, 'w');
end;
```

Now, if you opened a file in APPEND mode in 100 routines, you would have a lot of fixing to do. If on the other hand, you had this nice layer, you would be able to fix it in one location and be done with it.

**Unique Uses for Collections**

Another use of object types in PL/SQL is the use of collection types, and their potential to interact with SQL and PL/SQL easily. There are three things collection types can do in SQL and PL/SQL that people frequently ask how to do. These are:

- The ability to `SELECT *` from PLSQL_FUNCTION – You can write a PL/SQL routine and query from it, instead of querying from a database table.
- The ability to array fetch into a table of records – PL/SQL gives us the ability to `BULK COLLECT` (fetch more than one row at a time) into PL/SQL tables natively. Unfortunately, this works only with PL/SQL tables of SCALARs. I cannot do the following:

  ```sql
  select c1, c2 BULK COLLECT INTO record_type from T
  ```

  so I must code:

  ```sql
  select c1, c2 BULK COLLECT INTO table1, table2 from T
  ```

  Using collection types, I can array fetch into a ‘record type’.
- The ability to insert a record – Instead of inserting column by column, I can insert a single record.

**SELECT * from PLSQL_FUNCTION**

In order to demonstrate this ability, we will revisit a bind variable issue (I cannot leave this topic alone). A common requirement I see frequently is the stated need to issue a query such as:

```sql
select * from t where c in ( :bind_variable )
```
Using Object Relational Features

where the \texttt{BIND\_VARIABLE} is a list of values. That is, the value of \texttt{BIND\_VARIABLE} is ‘1, 2, 3’ perhaps, and you want the above query to be executed as if it were:

\begin{verbatim}
select * from t where c in ( 1, 2, 3 )
\end{verbatim}

This would return rows where \( c = 1 \) or \( 2 \) or \( 3 \), but it will really be executed as:

\begin{verbatim}
select * from t where c in ( '1,2,3' )
\end{verbatim}

This will return rows where \( c = '1,2,3' \) – a single string. This usually arises from a user interface where the end user is supplied with a list box of values, and may select one or more (any number) items from the list. In order to avoid having to create unique queries for each request (we know how bad that would be) we need a method of binding a varying number of elements in an in list. Well, since we can \texttt{SELECT*FROM PLSQL\_FUNCTION}, we are in business. This demonstrates how:

\begin{verbatim}
tkyte@TKYTE816> create or replace type myTableType
2  as table of number;
3  /
Type created.
\end{verbatim}

The type we created is the one our PL/SQL function will return. This type \textit{must} be defined at the SQL level via the \texttt{CREATE\_TYPE} statement. It cannot be a type inside of a PL/SQL package, the goal is to retrieve this data via SQL, hence we need a SQL type. This also proves that I am not totally biased against nested tables. In fact, they are the collection of choice when it comes to programming in PL/SQL, in this fashion. A \texttt{VARRAY} would limit us to some artificial upper bound in the array – the nested table implementation is limited only by the available memory on your system.

\begin{verbatim}
tkyte@TKYTE816> create or replace
2  function str2tbl( p_str in varchar2 ) return myTableType
3  as
4      l_str   long default p_str || ',
5      l_n      number;
6      l_data    myTableType := myTabletype();
7  begin
8      loop
9          l_n := instr( l_str, ',' );
10          exit when (nvl(l_n,0) = 0);
11          l_data.extend;
12          l_data( l_data.count ) :=
13              ltrim(rtrim(substr(l_str,1,l_n-1)));
14          l_str := substr( l_str, l_n+1);
15      end loop;
16      return l_data;
17  end;
18  /
Function created.
\end{verbatim}

So, we have a PL/SQL function that will take a comma-delimited string of values, and parse it out into a SQL Type \texttt{MYTABLETYPE}. All we need to do now is find a way to retrieve this, using SQL. Using the \texttt{TABLE} operator and a \texttt{CAST}, we can do this easily:
Now, using this as an IN subquery becomes trivial:

You can use this functionality in many places now. You can now take a PL/SQL variable and apply an ORDER BY to it, you can return sets of data that was generated by the PL/SQL routine back to the client easily, you can apply WHERE clauses to PL/SQL variables, and so on.

Going one step further here, we can return full multi-column result-sets this way as well. For example:

```
tkyte@TKYTE816> create type myRecordType as object
2  ( seq int,
3    a int,
4    b varchar2(10),
5    c date
6  )
7  /
Type created.
tkyte@TKYTE816> create table t ( x int, y varchar2(10), z date );
Table created.
```
tkyte@TKYTE816> create or replace type myTableType
2   as table of myRecordType
3  /
Type created.

tkyte@TKYTE816> create or replace function my_function return myTableType
2   is
3      l_data myTableType;
4   begin
5      l_data := myTableType();
6      for i in 1..5
7         loop
8             l_data.extend;
9             l_data(i) := myRecordType( i, i, 'row ' || i, sysdate+i );
10        end loop;
11      return l_data;
12  end;
13  /
Function created.

tkyte@TKYTE816> select *
2   from TABLE ( cast( my_function() as mytableType ) )
3   where c > sysdate+1
4   order by seq desc
5  /
<table>
<thead>
<tr>
<th>SEQ</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td>row 5</td>
<td>29-MAR-01</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>row 4</td>
<td>28-MAR-01</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>row 3</td>
<td>27-MAR-01</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>row 2</td>
<td>26-MAR-01</td>
</tr>
</tbody>
</table>

**Bulk Fetching into RECORD Types**

So, we’ve seen how we can use the collection type to `SELECT * FROM PLSQL_FUNCTION`, now we’ll see that we can use it to do a bulk fetch into the equivalent of a PL/SQL record type. We cannot actually do a bulk fetch into a true PL/SQL record type, but we can fetch into a SQL nested table type easily. For this, we will need two object types, a scalar type that represents our record, and a table of that type. For example:

```sql
tkyte@TKYTE816> create type myScalarType
2   as object
3   ( username varchar2(30),
4       user_id number,
5       created date
6   )
7  /
Type created.
```
Chapter 20

```
tkyte@TKYTE816> create type myTableType as table of myScalarType
  2  /
Type created.
```

Now we are ready to select into a variable of MYTABLETYPE as follows:

```
tkyte@TKYTE816> declare
  2   l_users    myTableType;
  3  begin
  4      select cast( multiset(select username, user_id, created
  5                          from all_users
  6                          order by username )
  7                as myTableType )
  8            into l_users
  9            from dual;
10  end;
11  /
12  PL/SQL procedure successfully completed.
```

We can substitute the query against ALL_USERS with any query that fetches a VARCHAR2(30), number and a date. The query can be arbitrarily complex, involve joins, and so on. The trick is to cast the results of that subquery as being of our object type. We can then fetch that entire result set into our local variable using the standard SELECT ... INTO syntax.

**Inserting Using a RECORD Type**

Given that we can SELECT * FROM COLLECTION_VARIABLE, where the collection variable is either a local variable or a PL/SQL function that returns a nested table type, it is not too hard to figure out how to INSERT using this method. We simply define a variable of our nested table type, and fill it up with as many records as we would like to insert. The following example demonstrates what a single row insert would look like:

```
tkyte@TKYTE816> create table t as select * from all_users where 1=0;
Table created.

  tkyte@TKYTE816> declare
  2     l_users   myTableType :=
  3          myTableType( myScalarType( 'tom', 1, sysdate ) );
  4  begin
  5    insert into t
  6        select * from TABLE ( cast( l_users as myTableType ) );
  7  end;
  8  /

tkyte@TKYTE816> select * from t;
USERNAME    USER_ID    CREATED
----------    ---------    --------
tom          1          24-MAR-01
```


When dealing with a table that has many columns, this little trick can come in handy.

**Using Types to Extend PL/SQL Wrap-Up**

In this section, we have seen how we can effectively use Oracle object types, not as a storage mechanism, but rather as a way to extend PL/SQL in the same way classes are used in Java or C++ to provide generic functionality.

We have also seen some interesting uses for the nested table type. The ability to `SELECT * from a PLSQL_FUNCTION` raises some interesting opportunities. Varying sized IN lists is just the beginning. The opportunities are limitless here. You could write a small routine that uses `UTL_FILE` to read an OS file, parse each line around commas, and return a result set that is the contents of the flat file to be inserted into some other table, or joined to a table for example.

Using object types in this fashion breathes new life into an established language. Once you’ve created a type or two for yourself, you’ll begin to find applications for this technique in much of your code. It is a logical way to tightly couple data and functionality together – one of the primary goals of object-oriented programming. To not offend the purists, I won’t call this *pure* object-oriented programming in PL/SQL, but it certainly is something very close to that.

**Object Relational Views**

This is a fairly powerful feature for those of you who want to work with the object relational features, but still must present a relational view of the data to many applications. This allows you to use the standard `VIEW` mechanism to synthesize objects from relational tables. You don’t have to create tables of a `TYPE`, with all of the mysterious columns and such – you can create a view of standard tables you have created (and probably already have). These views will behave just like an object table of that type would – without much of the overhead of hidden keys, surrogate keys, and other nuances.

In this section, we will use the `EMP` and `DEPT` tables to present a department-oriented view of the data. This is similar to the example of the nested table we used in Chapter 6 on Tables, where by we had the `EMP_TAB_TYPE` as a nested table of `EMP_TYPE`, and the `DEPT` table had a column of this nested table type. Here, we will model the `EMP_TYPE` and the `EMP_TAB_TYPE` once again, but we will also create a `DEPT_TYPE` object type as well and a view of that type.

It is interesting to note that this approach of using object views allows us to have the best of both worlds (relational and object relational). For example, we might have an application that needs a department-oriented view of the data. Their view starts at the department and the employees in the department are naturally modeled as a collection inside the department. Another application however needs a different perspective. For example, when you walk up to a security guard and identify yourself as an employee, they will need to have an employee-oriented view of the data. Department in this case is inferred by the employee, not the other way around where the view was that department infers employees. This is the power of the relational model – many different views can efficiently be supported simultaneously. The object model does not support many different views of the same data as easily (if at all) or efficiently. By using many different object views of the relational data, we can satisfy everyone.
Chapter 20

The Types

The types used in this example are borrowed from Chapter 6 on Tables, with the addition of the DEPT_TYPE. They are:

```sql
scott@TKYTE816> create or replace type emp_type
2  as object
3  (empno       number(4),
4   ename       varchar2(10),
5   job         varchar2(9),
6   mgr         number(4),
7   hiredate    date,
8   sal         number(7, 2),
9   comm        number(7, 2)
10  );
11  /
Type created.

scott@TKYTE816> create or replace type emp_tab_type
2  as table of emp_type
3  /
Type created.

scott@TKYTE816> create or replace type dept_type
2  as object
3  ( deptno number(2),
4    dname  varchar2(14),
5    loc       varchar2(13),
6    emps      emp_tab_type
7  )
8  /
Type created.
```

Once again, a department is modeled as being an object with a department number, a name, a location, and employees.

The O-R View

It is easy from the above type definitions, to synthesize our data for this view from the existing relational data. It would look like:

```sql
scott@TKYTE816> create or replace view dept_or
2  of dept_type
3  with object identifier(deptno)
4  as
5  select deptno, dname, loc,
6       cast ( multiset ( select empno, ename, job, mgr, hiredate, sal, comm
7       from emp
8       where emp.deptno = dept.deptno )
9  )
```

898
Using Object Relational Features

```sql
10 as emp_tab_type )
11 from dept
12 /

View created.

We are already familiar with the role of the CAST and the MULTISET – we are just turning a correlated subquery into a nested table collection here. For each row in DEPT, we will query out all of the employees. We’ve told Oracle which column(s) identify a row uniquely in the view using the WITH OBJECT IDENTIFIER clause. This allows Oracle to synthesize an object reference for us, giving us the ability to treat this view as if it were an object table.

As soon as we have the view, we can start using it:

```sql
scott@TKYTE816> select dname, d.emps
2    from dept_or d
3  /
DNAME EMPS(EMPNO, ENAME, JOB, MGR, HIREDATE, S
-------------- ----------------------------------------
OPERATIONS EMP_TAB_TYPE()

4 rows selected.
```
So, we are on our way. We have the relational tables and the object-relational view. Externally, it is difficult to tell which is the view, and which are the tables. The functionality of an object table is available to us – we have object references on this table, the nested table is set up, and so on. The advantage here is that we specify how to join EMP to DEPT using the existing natural parent/child relationship.

So, we have created an object-relational view that exposes the data for querying. It does not however work when it comes to modifications yet:

```
scott@TKYTE816> update TABLE ( select p.emps
2                   from dept
3                  where deptno = 20 )
4     set ename = lower(ename)
5  /
set ename = lower(ename)
*  
ERROR at line 4: 
ORA-25015: cannot perform DML on this nested table view column
```

```
scott@TKYTE816> declare
2     l_emps  emp_tab_type;
3  begin
4      select p.emps into l_emps
5           from dept_or p
6       where deptno = 10;
7    for i in 1 .. l_emps.count
8        loop
9          l_emps(i).ename := lower(l_emps(i).ename);
10       end loop;
11    end;
12  end;
13  / 
declare
*  
ERROR at line 1: 
ORA-01733: virtual column not allowed here
ORA-06512: at line 13
```
We need to ‘train’ our view how to update itself. We have a somewhat, complex mapping of relational data to object-relational – it can be arbitrarily complex in fact. So, how can we ‘train’ our view to update itself? Oracle provides a mechanism called an INSTEAD OF trigger for this purpose. We can code the logic that should execute INSTEAD OF Oracle’s logic when we modify the contents of the view. For illustrative purposes, we will train the above view to allow it to update itself.

Oracle allows us to place INSTEAD OF triggers on the view DEPT.OR as well as any nested table type included in the view. If we place a trigger on the nested table columns, it will allow us to process the first update from above – the UPDATE of the nested table column as if it were a table. The trigger for this would look like:

```
scott@TKYTE816> create or replace trigger EMPS_IO
  2  instead of UPDATE on nested table emps of dept_or
  3  begin
  4    if ( :new.empno = :old.empno )
  5      then
  6        update emp
  7         set ename = :new.ename, job = :new.job, mgr = :new.mgr,
  8           hiredate = :new.hiredate, sal = :new.sal, comm = :new.comm
  9         where empno = :old.empno;
 10     else
 11       raise_application_error(-20001,'Empno cannot be updated' );
 12   end if;
 13 end;
14 /
Trigger created.
```

As you can see, this trigger will fire INSTEAD OF UPDATE on the nested table column EMPS of the DEPT.OR view. It will be called for each and every row modified in the nested table and has access to the :OLD and :NEW values – just like a ‘normal’ trigger would. In this case, it is clear what we need to do. We need to update the existing EMP row by EMPNO, setting the columns to their new values. One thing I enforce in this trigger is that an UPDATE to the primary key is not allowed (hey, we might be using object-relational features, but that doesn’t mean we should violate the basic tenets of relational database design!).

Now if we execute:

```
scott@TKYTE816> update TABLE ( select p.emps
  2   from dept_or p
  3     where deptno = 20 )
  4   set ename = lower(ename)
  5 /
5 rows updated.

scott@TKYTE816> select ename from emp where deptno = 20;
ENAME
--------
smith
jones
scott
```
We see that the update of the nested table successfully translates into the relational table updates as expected. Coding the relevant INSERT and DELETE triggers are equally as easy, the UPDATE is the most complex case, and so we'll stop there.

At this point, it we execute:

```sql
DECLARE
    l_emps EMP_TAB_TYPE;
BEGIN
    SELECT p.emps INTO l_emps
    FROM dept_or p
    WHERE deptno = 10;
    FOR I IN 1..l_emps.COUNT LOOP
        l_emps(I).ENAME := LOWER(l_emps(I).ENAME);
    END LOOP;
    UPDATE dept_or SET EMPS = l_emps WHERE DEPTNO = 10;
END;
/
```

We find that it still does not work. This might seem surprising. Should it not fire our trigger from above? In fact, no it will not. Only updates to the nested table that use the collection un-nesting will fire this trigger – only when we treat the nested table as a table will that fire the trigger. Here, we are simply updating a column in the DEPT_OR table – we are not doing a set operation on the nested table itself. In order to support this style of code (and to support updates to the other scalar attributes of the DEPT_OR view itself), we need an INSTEAD OF trigger on the DEPT_OR view. The logic of this trigger will be to process the :OLD.EMPS and :NEW.EMPS as sets and:
1. Remove all records from EMP such that their EMPNO was in the :OLD set but is not seen in the :NEW set. The SQL operator MINUS is excellent for this.

2. Update every EMP record such that their EMPNO is in the set of EMPNOs such that their :NEW record differs from their :OLD record. Again MINUS is excellent for finding that set easily.

3. INSERT into EMP all :NEW records where the EMPNO of the :NEW record was not in the :OLD set of values.

This implementation is:

```sql
scott@TKYTE816> create or replace trigger DEPT_OR_IO_UPDATE
  2 instead of update on dept_or
  3 begin
  4   if ( :new.deptno = :old.deptno )
  5     then
  6       if updating('DNAME') or updating('LOC')
  7         then
  8           update dept
  9             set dname = :new.dname, loc = :new.loc
 10             where deptno = :new.deptno;
 11           end if;
 12
 13     end if;
 14
 15     if ( updating('EMPS') )
 16     then
 17       delete from emp
 18           where empno in
 19           ( select empno
 20             from TABLE(cast(:old.emps as emp_tab_type))
 21             MINUS
 22             select empno
 23               from TABLE(cast(:new.emps as emp_tab_type))
 24             );
 25       dbms_output.put_line( 'deleted ' || sql%rowcount );
 26   end if;
```

This first MINUS gives us the set of EMPNOs that were in the :OLD but are not seen in the :NEW set, we need to remove those records from the EMP table as they no longer exist in the collection. Next, we'll modify the changed collection records:

```sql
24  update emp E
25     set ( deptno, ename, job, mgr,
26           hiredate, sal, comm ) =
27         ( select :new.deptno, ename, job, mgr,
28           hiredate, sal, comm
29           from TABLE(cast(:new.emps as emp_tab_type)) T
30             where T.empno = E.empno
31       )
32     where empno in
33     ( select empno
34       from (select *
35         from TABLE(cast(:new.emps as emp_tab_type)))
```
That MINUS returned everything in :NEW minus anything in :OLD; which is the set of modified records. We used this in a subquery to get the set of EMPNOs we need to update in the EMP table and then used a correlated subquery to actually set those values. Lastly, we will add all new records:

```
insert into emp
   ( deptno, empno, ename, job, mgr, hiredate, sal, comm )
select :new.deptno,empno,ename,job,mgr,hiredate,sal,comm
from ( select *
      from TABLE(cast(:new.emps as emp_tab_type))
     where empno in
      ( select empno
        from TABLE(cast(:new.emps as emp_tab_type))
        MINUS
        select empno
        from TABLE(cast(:old.emps as emp_tab_type))
      )
    );
```

That MINUS generated the set of EMPNOs in the :NEW collection that were not present in the :OLD collection; this presents a list of rows to add to EMP.

This looks like a monster trigger, but is actually straightforward. To recap, it begins by seeing if the scalar columns of DEPT_OR were modified. If so, it applies the changes to the DEPT table. Next, if the nested table column was updated (all of its values replaced), it reflects those changes to the EMP table. What we need to do to reflect those changes is:

1. DELETE any records in EMP that were removed from the EMPS nested table column.
2. UPDATE any records in EMP that had their values modified in the EMPS nested table column.
3. INSERT any records into EMP that were added to EMP’s nested table column.
Using Object Relational Features

Fortunately, the SQL MINUS operator and the ability to TABLE the nested column variable make this easy for us. Now we can process:

```sql
scott@TKYTE816> declare
2    l_emps  emp_tab_type;
3  begin
4      select p.emps into l_emps
5        from dept_or p
6        where deptno = 10;
7    for i in 1 .. l_emps.count
8      loop
9          l_emps(i).ename := lower(l_emps(i).ename);
10     end loop;
11     update dept_or
12        set emps = l_emps
13        where deptno = 10;
14  end;
15  /
16 deleted 0
17 updated 3
18 inserted 0

PL/SQL procedure successfully completed.
```

```sql
scott@TKYTE816> declare
2    l_emps  emp_tab_type;
3  begin
4      select p.emps into l_emps
5        from dept_or p
6        where deptno = 10;
7    for i in 1 .. l_emps.count
8      loop
9          if ( l_emps(i).ename = 'miller' )
10             then
11                l_emps.delete(i);
12          else
13                l_emps(i).ename := initcap( l_emps(i).ename );
14          end if;
15     end loop;
16     l_emps.extend;
17     l_emps(l_emps.count) :=
18           emp_type(1234, 'Tom', 'Boss',
19           null, sysdate, 1000, 500 );
20     update dept_or
21        set emps = l_emps
22        where deptno = 10;
23  end;
24  /
25 deleted 1
26 updated 2
27 inserted 1

PL/SQL procedure successfully completed.
```
The trigger translates our operations on the object instance into the equivalent modifications against the base relational tables.

This capability, to expose our relational data as object-relational views, allows us to maximize the benefits of both the relational model and the object relational model.

The relational model shows its strength in its ability to answer almost any question you might have of the underlying data, easily and efficiently. Whether your view of the data is a departmental view (query a department and related employees) or employee-oriented (you specify an employee number and need to see departmental information), we can support you. You can use the relational tables directly or we can generate an object type model that exposes your view of the data, pulls all of the requisite data together, and gives it to you easily. Consider the results of these two queries:

```
scott@TKYTE816> select * from dept or where deptno = 10;
DEPTNO DNAME       LOC           EMPS(EMPNO, ENAME, JOB, MGR, HIREDATE, S
--------- ----------- ------------- ----------------------------------------
10 Accounting  NEW YORK      EMP
    TAB
    TYPE(EMP
    TYPE(7782, 'Clark',
        'MANAGER', 7839, '09-JUN-81', 2450,
        NULL), EMP
    TYPE(7839, 'King',
        'PRESIDENT', NULL, '17-NOV-81', 5000,
        NULL), EMP
    TYPE(1234, 'Tom', 'Boss',
        NULL, '25-MAR-01', 1000, 500))
```

```
scott@TKYTE816> select dept.*, empno, ename, job, mgr, hiredate, sal, comm
2  from emp, dept
3  where emp.deptno = dept.deptno
4  and dept.deptno = 10
5  /
DEPTNO DNAME       LOC EMPNO ENAME JOB  MGR  HIREDATE    SAL  COMM
------ ----------- -------- ----- ----- --------- ---- --------- ----- ----
10 Accounting  NEW YORK  7782 Clark MANAGER  7839 09-JUN-81  2450
10 Accounting  NEW YORK  7839 King  PRESIDENT 17-NOV-81  5000
10 Accounting  NEW YORK  1234 Tom   Boss      25-MAR-01  1000  500
```

They return similar data. The first one concisely gives you all of the information about a department in a single row. It could come back with many nested table types, which in SQL would take many queries.
It can do a lot of work in the server assembling the answer for you, and shipping it back in a single row. If you are in an environment where network round trips are to be avoided when possible (long latency times) this could be extremely beneficial. Not to mention that a single SELECT * FROM T can do the work of many SQL statements. Also, notice that repeating columns of data don’t occur with the object view. The DEPTNO, DNAME, and LOC columns are not repeated for each employee; they are returned only once, which may be more intuitive for many applications.

The second query requires the developer to have more knowledge of the data (that’s not a bad thing mind you, just something to consider). They must know how to join the data together and, if there were many other tables to join to, they might very well need many separate queries they would have to assemble back together themselves to get the same answer. As a matter of example by what I mean by that, suppose in your model that a department has a fiscal year budget. It is stored relationally as:

```sql
create table dept_fy_budget (
    deptno number(2) references dept,
    fy date,
    amount number,
    constraint dept_fy_budget_pk primary key(deptno,fy)
) /
```

Table created.

You have some data in there representing this year and the last couple of year’s FY budgets by department. Your application needs the department view that has all of the scalar data surrounding the department (name, location). It also needs the employee related information (the EMP_TAB_TYPE). It also however, needs the FY budget information as well. In order to get that relationally, the application programmer will have to code:

```sql
select dept.*, empno, ename, job, mgr, hiredate, sal, comm
from emp, dept
where emp.deptno = dept.deptno
and dept.deptno = 10
/
```

DEPTNO DNAME       LOC      EMPNO ENAME JOB       MGR  HIREDATE    SAL  COMM
------ ----------- -------- ----- ----- --------- ---- --------- ----- ----
10 Accounting  NEW YORK  7782 Clark MANAGER   7839 09-JUN-81  2450
10 Accounting  NEW YORK  7839 King  PRESIDENT      17-NOV-81  5000
10 Accounting  NEW YORK  1234 Tom   Boss           25-MAR-01  1000  500
3 rows selected.

```sql
select fy, amount
from dept_fy_budget
where deptno = 10
/
```

FY          AMOUNT
--------- ----------
01-JAN-99    500
01-JAN-00    750
01-JAN-01    1000
3 rows selected.
Chapter 20

It is not possible to write a single relational query that retrieves this data in one call. We can use some Oracle extensions (the CURSOR function in SQL) to return rows that return result sets themselves:

```
scott@TKYTE816> select
    2 dept.deptno, dept.dname,
    3 cursor(select empno from emp where deptno = dept.deptno),
    4 cursor(select fy, amount from dept_fy_budget where deptno = dept.deptno)
    5  from dept
    6  where deptno = 10
    7  /
```

```
<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DNAME</th>
<th>CURSOR(SELECTEMPNO)</th>
<th>CURSOR(SELECTFY,AMOUNT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>CURSOR STATEMENT : 3</td>
<td>CURSOR STATEMENT : 4</td>
</tr>
</tbody>
</table>
```

```
scott@TKYTE816> select
    2 dept.deptno, dept.dname,
    3 cursor(select empno from emp where deptno = dept.deptno),
    4 cursor(select fy, amount from dept_fy_budget where deptno = dept.deptno)
    5  from dept
    6  where deptno = 10
    7  /
```

```
<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DNAME</th>
<th>CURSOR(SELECTEMPNO)</th>
<th>CURSOR(SELECTFY,AMOUNT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>ACCOUNTING</td>
<td>CURSOR STATEMENT : 3</td>
<td>CURSOR STATEMENT : 4</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>EMPNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7782</td>
</tr>
<tr>
<td>7839</td>
</tr>
<tr>
<td>7934</td>
</tr>
</tbody>
</table>
```

```
CURSOR STATEMENT : 3

<table>
<thead>
<tr>
<th>EMPNO</th>
</tr>
</thead>
<tbody>
<tr>
<td>7782</td>
</tr>
<tr>
<td>7839</td>
</tr>
<tr>
<td>7934</td>
</tr>
</tbody>
</table>
```

```
CURSOR STATEMENT : 4

<table>
<thead>
<tr>
<th>FY</th>
<th>AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-JAN-99</td>
<td>500</td>
</tr>
<tr>
<td>01-JAN-00</td>
<td>750</td>
</tr>
<tr>
<td>01-JAN-01</td>
<td>1000</td>
</tr>
</tbody>
</table>
```

```
CURSOR STATEMENT : 4

<table>
<thead>
<tr>
<th>FY</th>
<th>AMOUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>01-JAN-99</td>
<td>500</td>
</tr>
<tr>
<td>01-JAN-00</td>
<td>750</td>
</tr>
<tr>
<td>01-JAN-01</td>
<td>1000</td>
</tr>
</tbody>
</table>
```

In this case, 1 row was selected and this row returned two more cursors to the client. The client fetched data from each cursor and displayed the results. This works nicely, but requires a knowledge of the underlying data and how to put it together (how to write the correlated subqueries to generate the cursors). We can instead model this data using the object-relational extensions and recreate our view as follows:

```
scott@TKYTE816> create or replace type dept_budget_type
    2  as object
    3  ( fy       date,
    4       amount number
    5  )
    6  /
```

```
Type created.
```

```
scott@TKYTE816> create or replace type dept_budget_tab_type
    2  as table of dept_budget_type
    3  /
```

```
Type created.
```
Using Object Relational Features

```sql
scott@TKYTE816> create or replace type dept_type
2  as object
3  ( deptno number(2),
4    dname varchar2(14),
5    loc varchar2(13),
6    emps emp_tab_type,
7    budget dept_budget_tab_type
8  )
9  /
Type created.

scott@TKYTE816> create or replace view dept_or
2  of dept_type
3  with object identifier(deptno)
4  as
5  select deptno, dname, loc,
6    cast ( multiset (
7      select empno, ename, job, mgr, hiredate, sal, comm
8        from emp
9        where emp.deptno = dept.deptno )
10      as emp_tab_type ) emps,
11    cast ( multiset (
12      select fy, amount
13        from dept_fy_budget
14        where dept_fy_budget.deptno = dept.deptno )
15      as dept_budget_tab_type ) budget
16  from dept
17  /
View created.

Now remember, the above is work we do once, the complexity is hidden from the application. The application will simply code:

```sql
scott@TKYTE816> select * from dept_or where deptno = 10
2  /

<table>
<thead>
<tr>
<th>DEPTNO</th>
<th>DNAME</th>
<th>LOC</th>
<th>EMPS(EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM)</th>
<th>BUDGET(FY, AMOUNT)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Accounting</td>
<td>NEW YORK</td>
<td>EMP_TAB_TYPE(EMP_TYP(E(7782, 'Clark', 'MANAGER', 7839, '09-JUN-81', 2450, NULL), 0, 0, 0, 0, 0, 0, 0), DEPT_BUDGET_TAB_TYPE(DEPT_BUDGET_TYPE(0, -JAN-00', 750), '01', 7839, DEPT_BUDGET_TYPE('01', -JAN-01', 1000)))</td>
<td>DEPT_BUDGET_TAB_TYPE(DEPT_BUDGET_TYPE('01', -JAN-01', 1000))</td>
</tr>
</tbody>
</table>

1 row selected.

```
Again, they get back one row, one object instance, which represents their view of the data. This can be quite handy indeed. The complexity of the underlying physical model is removed and it is easy to see how you might populate a GUI screen with this data. Languages such as Java JDBC, Visual Basic with OO4O (Oracle Objects for Ole), OCI (Oracle Call Interface), PL/SQL, and Pro*C can all make use of this functionality easily. Using the relational model, it is a little more cumbersome as you get more and more complex one to many relationships. With the object-relational model, it is a little more natural. Of course, we would have to modify our INSTEAD OF triggers to support modifications to the underlying relational data as well, so it is not totally completed but the gist is there.

Summary

In this chapter, we reviewed the major uses of Oracle object types and extensibility. There are four ways to use them, three of which we looked at in depth.

We covered its use as a method to impose standard data types on a system. Using the ADDRESS_TYPE we were able to not only enforce a common naming convention and data type usage, but to provide for data specific methods and functionality.

We covered its use as a method to naturally extend the PL/SQL language. Here, we took a supplied package and wrapped it in an object type layer. This protects us from future changes in the implementation of the supplied package, as well as giving us a more 'object' way to programming in PL/SQL, similar to the class structure in C++ or Java. Additionally, we saw how using collection types gives us the extremely interesting ability to SELECT * FROM PLSQL_FUNCTION. This ability in itself is worth looking into the object features for.

Finally, we investigated how to use this feature as a method to present object relational views of inherently relational data. As we saw, this allows us to easily present an application-specific object view of the relational data to as many different applications as we like. The main benefit of this approach is that the SQL required in the client becomes almost 'trivial' to code. There are no joins, no multiple queries to assemble the answer together. A simple single row FETCH should return everything we need in one call.

The fourth option, to create tables of a type, was covered in the Chapter 6 on Tables. As they behave just like object views (or is that the other way around?) their use has been covered as well. I tend to not use object tables. I lean towards object views of relation tables for many of the reasons of the covered above. The most prevalent reason being that you almost always need the relational view at the end of the day in order to provide the many specific application views of the data. Object-relational views are excellent for modeling application specific views of the data.