Name: Quote Minder
Connection: SCOTT/TIGER
DatabaseName: ExampleDb
Database Table(s): ANY
Files: quote.bas, quote.exe, quoabout.frm, quoerror.frm, quologin.frm, quoerror.frm, quoerror.frx, quote.mak, oradc.vbx
Purpose: This example demonstrates the use of multiple tables. It creates tables, sequences and database triggers (which are stored procedures). It drops tables, sequences and triggers. It filters the query from the database - constructing the SQL query at runtime. It adds records. It deletes records. It changes the string in the data control to indicate the current mode (reading or editing). It slices, it dices...

Usage:
After logging in if the needed tables don't exist the main form will only allow you to create the tables. If the tables do exist the main form will allow you to drop the tables, look at quotes, filter quotes, etc.

Deleting a quote: Press the DELETE button to delete the current quote.
Adding a new quote category: Type a new category under the category filter list and next to the ADD button and press ADD.
Adding a new quote person: Just like adding a category.
Filtering the quote display: Click on a category, or a person. "any" means no restriction, otherwise only the quotes for that category/person will be displayed. If there are none the quote box will be made invisible.
Adding a new quote: A new quote can be added when both a category and a person are selected. The new quote will be for that category and person. Press the ADD QUOTE button, then type the quote into the quote box. When you move away from that quote, either by navigating or by choosing a new filter criterion, the new quote will be placed in the database.

Technical notes:
The major demonstration of this example is relational table access. The most interesting table (quotes) references the other tables (qcats, qpersons) via foreign keys. For instance, each record in qcats has a unique number in the catnum column. That number is the primary key of qcats. These numbers are also found in the catnum column of quotes. There they are references to a another table - hence "foreign" key. In this way the quotes table does not have to include all of the category information in each record.

Of course, the end users don't want to see these keys, they want to see names. In a query only application it would be simplest to use a recordsource with a query like this:

```
select quote, pname, category from quotes,qpersons,qcats where
   quote.pnum = qpersons.pnum and quote.catnum = qcats.catnum
```

This "join" select statement would simultaneously access data from several tables. Unfortunately, such a joined select statement is not updatable. So we wouldn't be able to directly add to and delete from the dynaset of a data control with such a recordsource.

To make sure we have an updatable dynaset we need to select only from columns in a single table. So the recordsource on the data control is always "select * from quotes" with, perhaps, a where clause.

Now, how do we construct the where clause? If we knew the key numbers of what we wanted to filter on it would be easy. Knowing that shakespeare is pnum 1 we can get shakespeare quotes with

```
select * from quotes where pnum = 1
```

But all we actually have is the name "shakespeare" not the key number "1". The list boxes are constructed at form load time by getting all the names (or categories) from the appropriate tables. At that time we could remember the matching key values, somewhere. But we don't need to. After all, the database contains the information that "shakespeare" corresponds to key 1. It just requires a select from the person table. We can get that information with a subquery:

```
select * from quotes where
   pnum = (select pnum from qpersons where pname = 'shakespeare')
```

Queries like this are easily constructed programatically. There are just some syntactical details that have to be watched (like putting "and"s between clauses). The procedure QueryQuotes handles this in query.bas.

How do we get these unique key values for the tables? The basic program could generate them, but then if a couple of different client machines are accessing a single server some coordination would have to be done to assign unique numbers, etc. Messy. It is much easier to let the server assign the numbers. Oracle7 has objects called sequences that will hand out unique numbers. So at the time we create the tables we also create a sequence for use by all three tables. And we
have the server get numbers from the sequence and assign them to the key field at the time that records are inserted. This is done with a database trigger. The database trigger is a block of PL/SQL code that is fired on a particular event. We created blocks of code that are fired before the insert into the database. This gives us the opportunity to modify column values before the server attempts to put the record into the database.

Note that because we are using the same sequence for all three tables, it isn't possible to assume that any of the table will have a continuous set of index numbers. But even if we had created a sequence for each table, deletions would remove numbers anyway.

When we are adding a record the user interface is actually interacting in a different way with the data control's dynaset. The mode is different. It is useful to the user to have some way of knowing what mode they are in. There is a little code in the data control's validate method to change the string appearing in the control so that users can tell whether they're reading or editing data.

When a record is deleted we do a little navigation on the data control's dynaset to change the current record to something that is valid.