8

Data Storage, Retrieval, and Manipulation

To retrieve data from the server and store it in your application, you configure a set of data-storage objects (instances of DBRecordList) and tell them to fetch. This is a fairly simple process that's complicated, primarily, by the extent to which you refine your DBRecordList objects.

Once you've retrieved and stored data, you use instances of the DBValue class to examine and modify individual data values in a DBRecordList. The modified data isn't seen by the server immediately; to save your modifications, you must invoke a method that tells the DBRecordList to send the modified data back to the server.

This chapter describes the principal techniques for configuring and using DBRecordList and DBValue objects to fetch, manipulate, and save data. These descriptions are far from the entire tale; the next two chapters (Chapters 9 and 10) tell you how to refine DBRecordList objects, and how to exhibit greater control over data transactions. Although the star of this show is DBRecordList, the story is easier put if DBValue is introduced first.

The DBValue Class

The DBValue class defines general-purpose, atomic-data storage objects. DBValue objects are an important part of the Database Kit's data-storage system, but their use isn't confined to that arenaÐyou can use a DBValue to represent almost any single-valued data item.

There are two parts to a DBValue: a value and a data type. The value is represented directly; the data type is represented by a DBTypes-conforming object (the DBTypes protocol was described in Chapter 6). For example, a DBValue that represents the floating-point number 3.1416 might be depicted as:

Figure_75. A DBValue Object

Setting a DBValue

You set a DBValue's value and data type at the same time, through one of the **set*Type*Value:** methods:

- **setObjectValue:**(id)*anObject*

- **setStringValue:**(const char *)*aString*
- **setIntValue:**(int)*anInteger*
- **setFloatValue:**(float)*aFloat*
- **setDoubleValue:**(double)*aDouble*

The argument to each of these methods supplies the object's value; the individual methods set the DBTypes object to represent the appropriate (named) data type. The DBValue shown in Figure 75, for instance, would have been set through the **setFloatValue:** message, as shown below:

```
DBValue *aValue = [[DBValue alloc] init];
[aValue setFloatValue:3.1416];
```

The **setObjectValue:** and **setStringValue:** methods copy the value of their arguments and then set the DBValue's value to the copies. To set an object or string value without copying, use one of these methods:

- **setObjectValueNoCopy:**(id)*anObject*
- **setStringValueNoCopy:**(const char *)*aString*

Null Values

In deference to the significance of NULL values in a database (in that they represent ªmissingº values, as explained in Chapter 2), DBValue provides the **setNull** method. This method sets a DBValue's value to NULL.

You should ask a DBValue if it has a NULL value before asking for the value itself; determining whether a DBValue holds NULL is done through the **isNull** method (the value-retrieval methods are described in the next section). A DBValue will hold NULL if it has received a **setNull** message or if it hasn't yet had its value set.

Getting Values

To retrieve a DBValue's value, you invoke one of the value-retrieval methods:

- (id)**objectValue**
- (const char *)**stringValue**
- (int)**intValue**
- (float)**floatValue**
- (double)**doubleValue**

These methods convert the object's value, if necessary, before returning it, thus you don't have to use the ªcorrectº method when asking a DBValue for its value. In other words, the data type by which you ask for the value needn't match that by which it was set. Of course, the requested conversion must be reasonable:

- You can convert between any of the numeric types (with a possible loss of precision), from a number to a string, and from a string to a number (if the string represents a number)
- You can never convert a non-object to an object.
- If you send an object-laden DBValue a **stringValue** message, the message will be forwarded to the embedded object. This is provided as a convenience; however, the courtesy doesn't extend to the number-retrieving methods (in other words, **intValue**, **floatValue**, and **doubleValue** are not forwarded to the embedded object).
- If an unsupported conversion is requested, the value-getting methods return the following values:

Method	Value

objectValue	nil
stringValue	ªoº
intValue	0
floatValue	0.0
doubleValue	0.0

Asking a DBValue for its value in an ªunnaturalº data type, and so prompting a data conversion, doesn't change the DBValue itself. For example, if you set a DBValue thus:

```
[aValue setFloatValue:2.6];
```

and then send it the following message:

```
int anInt = [aValue intValue];
```

the **anInt** variable will properly hold the value 2. However, the **aValue** object's value and data type aren't changedÐthe object still holds the floating-point value 2.6

Getting Data Types

To retrieve a DBValue's **DBTypes** object, you send it a **valueType** message. To the DBTypes object you send, primarily, the **objcType** message. As described in Chapter 6, the method returns a string that represents the data types according to the following convention:

Data type	DBTypes objcType value
object	ª@º
string	ª*º
integer	ªiº
float	ªfº
double	ªdº

The DBRecordList Class

DBRecordList is the Database Kit's pre-eminent data-storage class. It provides a random access storage system for records that are drawn from a database server.

A DBRecordList stores records in its *record table*. Each column in this two-dimensional table is represented by a property, and each row holds a single record. A record is identified by an index that gives its ordinal position in the table (starting with record 0). The intersection of a column and row is called a ªfield.º Each field holds a single atomic value that's cast as the data type of the property that represents the field's column.

Figure 76, below, illustrates the components of a DBRecordList's record table. The entity to which the DBRecordList corresponds is Customer; the properties that are shown here are assumed to be directly contained by this entity.

Figure_76. A Record Table

Properties, Records, and Fields as Programming Elements

The properties, records, and fields that make up a record table are not only architecturally distinct, they're programmatically different things. Each component demands its own techniques for its

identification and manipulation. Most of the rest of this chapter explains these techniques in detail. But before proceeding to these explanations, you should be aware of the programmatic differences between records, properties, and fields:

- As mentioned in the previous section, the properties that are used to represent a record table's columns are the familiar DBProperties objects (described in Chapter 6). The important point here is that a record table's properties are *objects*. You set them into a DBRecordList and retrieve them as objects, you can send them messages, and, perhaps most important, a property has a meaning independent of its use in a record table.

- Records, on the other hand, are *not* objects. Moreover, what they *are* is unspecified; as stated above, you can identify a record (within a record table) by its index, but that's as close as you get to the record itself. All the methods that manipulate recordsÐmethods that add, delete, and reorder recordsÐdo so on the basis of record indices. And these methods are performed within the context of a particular record table; records have no meaning outside a record table. This means, for example, that you can't move a record directly from one table to another.

- Fields are also identified by position: To identify a particular field, you specify the property object that represents its column and the index of the record in which it lies. To examine or change the value that lies in a particular field, you must provide a DBValue object. The DBValue takes on the value (and data type) of the identified field. Similarly, to alter a field's value, you provide a DBValue from which you want the value copied, and specify the field that you want the value copied into. But at all times, the DBValue is independent of the field (and the record table in general).

One more element deserves consideration: the record table itself. A DBRecordList object represents a single record table; in object-oriented terms, the object *is* the table.

DBRecordStream

The DBRecordList class inherits from the DBRecordStream class. This class isn't abstractÐyou can use instances of DBRecordStream directly in your application. However, DBRecordStream objects aren't as flexible as DBRecordLists: A DBRecordStream provides serial, forward-only, one-record-at-a-time access to a set of records that lie on the server.

Although DBRecordStreams are sometimes more efficient than DBRecordLists, the increased efficiency is seldom very compelling. In practice, you rarely create instances of DBRecordStreamÐyou almost always use DBRecordList objects to store database data in your application. Accordingly, the sections below employ DBRecordList objects exclusively to demonstrate and explain the concepts of data storage. Exceptional differences between DBRecordStream and DBRecordList are noted.

Setting the Properties

Having minted a DBRecordList object (through the usual nesting of **alloc** and **init** methods), one of the first things you do with it is tell it what sort of data you want it to provide storage for. This is done by setting the object's property list, through the **setProperties:ofSource:** method:

- The method's first argument is a list of property objects (more technically, a list of DBProperties-conforming objects).

- The second argument is the ªsourceº of the properties; in other words, it's an object that represents the entity to which the properties in the property list belong. In the simplest case, as exemplified below, this argument is a DBEntities object. An example of a different class of source object is given in the next chapter.

The single source argument enforces a rule that's stated thus: All the properties that are set in a DBRecordList must be ªrooted atº the same entity. As a convenience, you can pass **nil** as the source; this sets the DBRecordList's source to the entity of the first object in the property list.

The following example shows a DBRecordList-creating function called **makeRecordList()**. The function creates and returns a DBRecordList that's configured to store data for all the properties in its **entity** argument:

```
/* Create a DBRecordList that contains all the properties found in
   the given entity.*/
DBRecordList *makeRecordList(id entity)
{
   DBRecordList *recordList = [[DBRecordList alloc] init];
   List *props = [[List alloc] init];

   /* Get the entity's property list. */
   [entity getProperties:props];

   /* Make sure the DBDatabase is connected to the server (this
      is explained in the Note below). */
   if (![[entity database] isConnected])
      if (![[entity database] connect])
          return nil; // failure

   /* Set the property list into the DBRecordList object. */
   [recordList setProperties:props ofSource:nil];
   [props free];
   return recordList;
}
```

Notice that the List object that's passed as the first argument to **setProperties:ofSource:** is freed after the method is invoked. DBRecordList maintains its own property List object into which it copies the contents of the List that you pass, leaving you to free the argument yourself.

Important: If you're using the Oracle adaptor, then you *must* make sure that your DBDatabase object is connected to the server before you invoke **setProperties:ofSource:**. For other adaptors this isn't a requirement, but it can't hurt.

Property List Discrimination

The function shown in the example above doesn't discriminate between attributes and relationships: It sets the entity's gamut of properties into the DBRecordList. Even though relationships don't represent actual data, they're allowed in a DBRecordList's property list. Furthermore, and as will be shown in the next chapter, this is a necessary practice when constructing master-detail tables. For the present, it's enough to understand that a DBRecordList's list of properties isn't restricted to attributes.

As a corollary to this, you don't *have* to use all the properties in an entity when setting a DBRecordList's property list. In the following example, only string-valued properties are used:

```
/* Create a DBRecordList that contains all the string-typed
   properties found in the given entity.*/
DBRecordList *makeStringRecordList(id entity)
{
   DBRecordList *recordList = [[DBRecordList alloc] init];
   List *props = [[List alloc] init];
   int n;
   id thisProp;

   /* Get the entity's property list. */
   [entity getProperties:props];

   /* Remove the non-string properties from the list. */
```

```
    for ( n = [props count]; n > 0; n--) {
       thisProp = [props objectAt:n-1];
       if ( *[[thisProp propertyType] objcType] != '*')
           [props removeObject:thisProp];
    }
    [recordList setProperties:props ofSource:nil];
    [props free];
    return recordList;
}
```

Resetting a Property List

Typically, the lifetime of a DBRecordList object is defined by the desirability of its property list. Accordingly, you usually send the **setProperties:ofSource:** message only once to each DBRecordList that you use. You can then fetch data, modify the values, save the modifications, re-fetch, re-modify, re-save, and so on, but you rarely send **setProperties:ofSource:** a second time. When you're no longer interested in the DBRecordList's set of properties, you should free the entire object rather than reuse it.

However, if you find that you absolutely *must* reset a DBRecordList's property list, then be aware that you must clear the object first, by sending it a **clear** message. Each time a DBRecordList object receives a **setProperties:ofSource:** message it empties its current property list, but it doesn't fully reset itself to the virgin state that's attained through **clear**.

Retrieving a DBRecordList's Properties

You retrieve a DBRecordList's properties through the **getProperties:** method. The method copies (by reference) the DBRecordList's properties into the List object that you supply as an argument. If the object isn't able to fulfill the request, it returns **nil**, otherwise it returns the argument List that you passed. The method should only fail if the object's property list hasn't been set (or has been cleared).

The list of properties that you retrieve through the **getProperties:** method may contain objects that you didn't explicitly set. Specifically, the list may contain primary key attributes that were added automatically, as explained in the ªKey Propertiesº section, below.

In the following example function, which demonstrates a use of the **getProperties:** method, a DBRecordList's property list is searched for a particular (named) property:

```
<DBProperties *>findRecordListPropertyByName(DBRecordList *rList,
                                                const char *propName)
{
   List *propList = [[List alloc] init];
   id prop;
   int n;

   /* Get the property list (and check for success).*/
   if ([rList getProperties:propList]) {

      /* Look for the property by name. */
      for (n=0; n<[propList count]; prop=[propList objectAt:n++])
          if (strcmp([prop name], propName) == 0)
              break;
          else
              prop = nil;

   /* Free the (local) property list. */
   [propList free];

   /* Return the property. */
```

```
    return prop;
}
```

Key Properties

In order to function properly and completely, a DBRecordList must know the primary keys for its source. One of the features of the **setProperties:ofSource:** method is that it tries to make this determination for you: It gets the properties from the source entity and asks each one if it's a primary key. If it is, the method adds the property to the DBRecordList's property list, even if you didn't ask for it to be included.

You can retrieve a DBRecordList's primary key properties through the **getKeyProperties:** method.

Setting the Key Properties Directly

Although the automatic key-searching mechanism is usually sufficient, it's possible to set the primary keys yourself, through the **setKeyProperties:** method. However, setting the key properties of a DBRecordList subverts the design of the model that you're using, so you should rarely have need of this method. Also, if you set the key properties yourself, you should be aware of some subtleties of the **setKeyProperties:** method:

- The property objects that you include in the list that you pass as the argument to **setKeyProperties:** aren't themselves marked as being key; they're only key with respect to the DBRecordList that receives the message. The following code example demonstrates this; in it, the **name** property responds NO to both invocations of **isKey**:

```
/*    By the definition of this example, this message returns NO. */
BOOL nameIsKey = [name isKey];

/*    Add name to the (existing) keyList List object and set it as a
   key property of the (existing) recordList DBRecordList object. */
[keyList addObject:name];
[recordList setKeyProperties:keyList];

/*    This will still return NO, even though the name property is used
   as a primary key for the recordList object. */
nameIsKey = [name isKey];
```

- An invocation of **setKeyProperties:** erases the DBRecordList's current notion of its key properties, whether this notion was gotten from a previous invocation of the method, or through the automatic key-searching mechanism included with **setProperties:ofSource:**.

- If you invoke **setProperties:ofSource:** before invoking **setKeyProperties:**, the primary keys that were found and added to the DBRecordList through the former method's key-searching mechanism will still be present in the object's property list (although, as stated above, they will no longer function as primary keys). If you reverse the sequence of these messagesÐif you set the key properties before setting the ªregularº propertiesÐthe key-searching mechanism is turned off, thus the attributes that are marked as primary keys (in the entity) won't automatically appear in the property list.

Fetching Data

To tell a DBRecordList to retrieve data from the server, you send it a **fetchUsingQualifier:** message. The argument is a DBQualifier object that limits the range of records that are returned.

The DBQualifier class is discussed in the next chapter; as a default, you can pass **nil** as the argument, thereby returning all the records (for a given entity) that lie on the server. The following example demonstrates this:

```
/*    For the purposes of the example, we "hard-wire" two properties
   from an entity in the OracleDemo model into the DBRecordList. */
DBDatabase *db = [DBDatabase findDatabaseNamed:"OracleDemo"
                  connect:NO];
DBRecordList *customerList = [[DBRecordList alloc] init];
List *props = [[List alloc] init];
id customer = [db entityNamed:"Customer"];

[props addObject:[customer propertyNamed:"name"]];
[props addObject:[customer propertyNamed:"creditLimit"]];
[customerList setProperties:props ofSource:nil];

/* Remember to connect first. */
[db connect]

/* Test the connection and fetch. */
if (![db isConnected])
   fprintf(stderr, "The DBDatabase isn't connected\n");
else if ([customerList fetchUsingQualifier:nil])
      fprintf( stderr, "Fetch succeeded\n");
   else
      fprintf( stderr, "Fetch failed\n");
```

As implied by this demonstration, the **fetchUsingQualifier:** method returns **nil** if the fetch fails.

Figure 76, earlier in this chapter, depicts the record table that's constructed as a result of this fetch. Notice that the property named customerID has been added to the DBRecordList's properties. This is a result of the automatic key-searching mechanism described earlier.

The finer points of fetching are described in Chapter 10.

The Record Table

DBRecordList, through its conformance to the DBContainers protocol, provides a couple of methods that let you examine and manipulate its record table. These are:

- **count** returns the number of records that are in the table.
- **empty** removes all the records from the table (it sets the record count to 0).

Of these two methods, **count** is by far the more important. You use it whenever you need to perform an operation on all the records in a record table. The trivial example below shows a typical loop construction that uses **count** for such a purpose:

```
/* n, a counting variable, and recordList, a DBRecordList, are
   assumed to exist. */
for (n = 0; n < [recordList count]; n++)
{
   printf("Got a record.\n");
}
```

The **empty** method is invoked automatically when you set a DBRecordList's properties, when you tell the object to fetch (this can be suppressed, as explained in Chapter 10), and when you send it a **clear** message. Because of this automatism, you rarely have to invoke **empty** directly.

If you do find yourself tempted by the **empty** method, be aware that emptying a record table is *not* the same as deleting all its records. When you delete a record (and this is explained in greater detail in a section below) and then save the DBRecordList, the data-saving mechanism will attempt to delete, from the server, the data that corresponds to the deleted record. Emptying a record table

doesn't mark the removed records for such deletion. You can think of the **empty** method as setting a record table to its pre-fetch state.

Warning: The DBContainers protocol declares other methods as well, but **count** and **empty** are the only DBContainers messages that you should send to a DBRecordList object.

Warning: DBRecordList also conforms to the DBCursorPositioning protocol. This protocol declares methods that let you point to records in the table by moving a ªcursor.º Although these methods will work with a DBRecordList object, they don't let you do anything that you can't otherwise do through methods declared directly by the DBRecordList class. Furthermore, it's strongly advised that you not use the cursor positioning methods as they can cause some confusion, particularly if you're using Database Kit interface layer objects to control your DBRecordList objects.

Finding a Record

Finding a record in a record table seems like a fairly simple task: As described earlier in this chapter, you identify a record by the index that gives its position in the table. More specifically, the DBRecordList methods that act on particular records take record index arguments. The challenge, here, is in supplying the proper index value.

If you're simply walking down the record table and looking at records one-by-one, then you shouldn't have much of a problem supplying an index. In the following example function (which jumps ahead a bit with its use of the **getValue:forProperty:at:** method), the value for each record's name property is printed:

```
void printAllNames(DBRecordList *rList)
{
   int n;
   id nameProp;
   DBValue *nameVal = [[DBValue alloc] init];

   /* Get the "name" property by calling the
      findRecordListPropertyByName() function (as defined
      in a previous example).. */
   if (!(nameProp = findRecordListPropertyByName(rList, "name")))
   {
      printf("Property not found.\n");
      [nameVal free];
      return;
   }

   /* Loop over the record table, printing the name value of each
      record. */
   for (n = 0; n < [rList count]; n++)
   {
      [rList getValue:nameVal forProperty:nameProp at:n]
      printf("%s\n", [nameVal stringValue]);
   }
   [nameVal free];
}
```

Finding a Specific Record

If you want a specific record, then you have to supply a particular index. But because records can be reordered (as described in a later section), you can't rely on index values completely. Instead, the correct way to find a record is to ask for the index of the record with a given value as its primary key. This is done through the **positionForRecordKey:** method. The method takes a DBValue object that's been set to hold the primary key value of the record that you're interested in.

In the following example function, the value of the name property of a particular record is printed.

The function takes, as arguments, a DBRecordList object and an integer that's assumed to characterize the value of the record's primary key attribute:

```
void printOneName(DBRecordList *rList, int keyVal)
{
   int index;
   id nameProp;
   DBValue *indexVal = [[DBValue alloc] init];
   DBValue *nameVal = [[DBValue alloc] init];


   /* Get the "name" property. */
   if (!(nameProp = findRecordListPropertyByName(rList, "name")))
   {
      printf("Property not found.\n");
      [indexVal free];
      [nameVal free];
      return;
   }

   /* Set the indexVal object to hold the primary key value. */
   [indexVal setIntValue:keyVal];

   /* Get the index of the requested record. */
   if ((index = [rlist positionForRecordKey:keyVal]) == DB_NoIndex)
   {
      printf("No record with %d as a primary key.\n", keyVal);
      return;
   }
   /* Get the name value and print it. */
   [rList getValue:nameVal forProperty:nameProp at:index];
   printf("%s\n", [nameVal stringValue]);
   [indexVal free];
   [nameVal free];
}
```

As shown in the example, the **positionForRecordKey:** method returns DB_NoIndex if the record table doesn't hold a record with the designated primary key value.

The result of the **positionForRecordKey:** method should be applied immediately and then forgotten. Put a better way, if you're searching for a specific record, you should invoke this method immediately before you need to use the index. The method's return value shouldn't be stored for later use since this defeats the problem that it's trying to solve, namely the mutability of record position within a record table.

DBRecordList doesn't provide a method that lets you quickly find a record that has a compound primary key. To do this, you must ask the DBRecordList for its list of primary key attributes, ask for the values of these attributes from each record in the table (through the **getValue:forProperty:at:** method, which was demonstrated in the examples, and will be explained in greater detail later in this chapter), and compare these values to the target values.

Manipulating Records

You can change the contents of the record table by manipulating entire records. Specifically, you can add new records to the table, delete records, and move a record from one position to another (but only within the same table).

Adding Records

DBRecordList provides two methods that let you add records to its record table:

- **insertRecordAt:** adds a new record at the location specified by the index argument. The previous occupant of the given index and all subsequent records are moved down to make room for the new record.

- **appendRecord** adds a new record to the end of the record table.

Both of these methods return **nil** if the addition is unsuccessfulÐthis should only happen if you pass an out-of-bounds index to **insertRecordAt:** or if the DBRecordList's property list hasn't been set.

To determine if the record at a particular index is new (as opposed to having been fetched from the server), you invoke the boolean method **isNewRecordAt:**.

All the fields in a new record are set to NULL.

New Records and Primary Keys

A newly-added record is in a curious state: Since all its fields are NULL, it has no primary key value. Whenever you add a new record to a record table, one of your first tasks is to supply a value for its primary key (the method by which a field's value is set is described later). However, the uniqueness of a primary key is determined by the server, not by the Database Kit.

Most servers provide routines that generate new primary key values, or that check candidate values. You should consult your server's API for more information on how to guarantee the unique-ness of primary key values.

Adding Records Without Fetching

Although the sequence of actions described so far places the act of fetching as a precursor to other record manipulations, this order isn't inviolable: You don't have to fetch before you add a new record. For example, consider an application that doesn't care about existing data, all it wants to do is create new records, fill them with data, and then save them to the server. To create a table to hold these records, you would create a DBRecordList, set its property list, and then send it a series of record-adding messages.

Deleting Records

To delete a record, you pass its index to the **deleteRecordAt:** method.

Moving Records

You can move records within the record table through these methods:

- **swapRecordAt:withRecordAt:** switches the locations of the two records identified by the index arguments.

- **moveRecordAt:to:** moves the record at the index given by the first argument to the index given by the second argument.

In general, there's little reason to programmatically move records in a record table. The DBRecordList class provides a means for sorting records as they're fetched (as described in the next chapter). You should try to use the sorting techniques rather than move records after they've been fetched.

Examining and Modifying Fields

You can't examine the values of a record's fields directly; to determine the value of a field, you must use a DBValue object. A DBValue object takes on a value stored in a DBRecordList object through DBRecordList's **getValue:forProperty:at:** method. The method takes, as arguments, a DBValue object, a property object, and a record index. When the method returns, the DBValue object that you passed will hold the value that was found at the field designated by the record and property arguments. An example of this method was given in the ªFinding a Recordº section, earlier in this chapter.

To write a value back into a record table field, you use DBRecordList's **setValue:forProperty:at:** method. The order and types of arguments are the same as in the **getValue:forProperty:at:** method: The method takes a DBValue object, a property object, and a record index.

Record Field Data Type Immutability

The **setValue:forProperty:at:** method, you'll notice, doesn't let you proclaim the data type of the field that's being set. This is because you can't change the data type of a field. For example, the creditLimit property, shown in Figure 76 earlier in this chapter, represents floating-point data. All fields that are created for this property hold floating-point data. If you try to set an integer-typed DBValue object into a ªcreditLimitº field, the value that's set in the record will be a floating-point number that's converted from the DBValue's integer value.

Altering Primary Key Values

In general, you shouldn't alter primary key values. However, there are two cases in which you may have to:

- *You've added a new record to the DBRecordList*. The records that you add through DBRecordList methods such as **newRecord** are created with NULL values for every property, including the primary key attributes. Obviously, you'll have to set the primary key values (at least) of any new records before they can be sent to the server.
- *The primary key holds significant value*. As described in Chapter 2, the actual values that are held by most primary keys are arbitrary numbers that are meaningful only to the database (and then only to discriminate between records). Some primary keys, however, may hold ªreal worldº values; for example, a table of ªpeopleº data might use social security numbers as primary key values. If such a value changes in the real world, then it will probably have to be changed in the database.

As a general rule, if you must change the primary key value(s) of an existing record, you should create a new record with the new primary key, copy all the other values that you need from the old record, and then delete the old record.

Saving Modified Records

You've fetched records from the server into a DBRecordList, examined the records' values, and modified some of them; perhaps you've added some new records or deleted some existing ones. Now you want to write your changes back to the server. To do so, you send a **saveModifications** message to the modified DBRecordList. If the attempt at saving is successful, the method returns zero.

Warning: DBRecordStream objects, which also respond to **saveModifications**, return 1 if the save is successful.

There are a number of reasons why an attempt at saving modified data may fail; these are examined and their antidotes described in Chapter 10.