8

Retrieving Records

The Enterprise Objects Framework offers two ways to fetch, or retrieve, persistent data. The first is by fetching the entire data set with a single fetch command. You fetch in this manner when you're working with a data source (or at the controller level). The second way to retrieve persistent data is to select the desired data and then fetch the selected records one-by-one in a *fetch loop*, which you do at the database and adaptor levels.

To reflect these two ways of working, two sections make up the bulk of this chapter:

- ªFetching All Objects at Onceº talks about how you retrieve records into enterprise objects at or above the data source level.
- ªRetrieving Data with a Fetch Loopº discusses the technique you use when retrieving records at the database and adaptor levels. Although the technique is the same at either of these levels, the outcome is different. At the database level, you fetch records into enterprise objects. At the adaptor level, you fetch records into NSDictionary objects.

Each of these sections discusses the indicated fetch technique, how to retrieve records that contain relationships, how to control various aspects of the transaction, and how to handle any errors that might arise.

This chapter also contains two sections that discuss techniques common to both retrieval methods:

- ªQualifying the Resultsº discusses how you can limit the set of records that are retrieved from the database.
- ªSpecifying the Retrieval Orderº shows you how to indicate that you want the records returned in a particular order.

This chapter only discusses fetching. For information on altering the contents of a database, see Chapter 9, ªInserting, Deleting, and Updating Records.º This chapter assumes that you already know how to establish a connection to the database; see Chapter 7, ªConnecting to the Databaseº if you need more information.

Fetching All Objects at Once

The EODatabaseDataSource and EOController classes each provide a method that obtains all of the selected enterprise objects for their associated entity in a single operation (**fetchObjects** and **fetch**, respectively). At this level, the Enterprise Objects Framework does the bulk of the work for you. You don't have to concern yourself with fetch loops, fetching across relationships, and so on.

Convenience has its trade-offs, however. Fetching all objects at once can consume a great deal of memory. For most applications, though, this isn't significant enough to outweigh the convenience of being able to issue a single fetch command and immediately have all objects at hand.

The basic procedure to use when fetching objects with a data source is:

1. Apply any relevant qualifiers to the data source object. See ªQualifying the Resultsº for instructions on constructing and using qualifiers.
2. If the order of the objects in the resulting NSArray is important, specify your desired retrieval order. ªSpecifying the Retrieval Orderº discusses how you do this.
3. Fetch the selected set of objects.

The following code excerpt illustrates these steps:

```
EODatabaseDataSource *myDbDataSource;    /* Assume this exists. */
NSArray *myObjects;
NSArray *sortOrder;
EOQualifier *salaryQualifier;
EOAttribute *primaryAttribute;
EOAttributeOrdering *primaryOrdering;

/* Qualify the fetch: all employees who make more than $5000/month. */
salaryQualifier = [[[EOQualifier alloc]
    initWithEntity:[myDbDataSource entity]
    qualifierFormat:@"%A > %f", @"salary", 5000.0] autorelease];
[myDbDataSource setQualifier:salaryQualifier];

/* Specify the fetch order - sort by last name. */
primaryAttribute = [[myDbDataSource entity]
    attributeNamed:@"lastName"];
primaryOrdering = [EOAttributeOrdering
    attributeOrderingWithAttribute:primaryAttribute
    ordering:EOAscendingOrder];
sortOrder = [NSArray arrayWithObject:primaryOrdering];
[myDbDataSource setFetchOrder:sortOrder];

/* Get the entire set of objects. */
myObjects = [myDbDataSource fetchObjects];

/* Process the set of objects in myObjects. */
```

Fetching Across a Relationship

When working with enterprise objects, the Framework effectively fetches related objects as well. For to-one relationships, the value for the relationship contains the related object. For to-many relationships, the relationship's value contains an array object that in turn contains the related set of zero or more objects. (In reality, the Framework constructs fault objects to stand in for related objects. For more information, see Chapter 2, ªFrom Database to Objects,º or the EOFault class specification in the *Enterprise Objects Framework Reference*.)

As an illustration of this simplicity, the following code extract prints (using Foundation's **NSLog()** function) an employee's last name and the name of the department to which the employee belongs, given an employee object (myObject) and a related department object:

```
NSLog(@"%@\n", [myObject lastName]);
NSLog(@"%@\n", [[myObject department] departmentName]);
```

The enterprise object classes in the above example implement convenience methods for directly accessing the given attributes: **lastName** and **department** for employee enterprise objects, and **departmentName** for department enterprise objects.

Controlling Transactions with a Data Source

An Enterprise Objects Framework application needs to define a transaction scope, even if it's only fetching data. By default, all EODatabaseDataSources begin transactions automatically. You can verify this for a particular data source by sending it **beginsTransactionsAutomatically** and checking the returned value.

If your data source doesn't automatically start a transaction for you, your application has to explicitly start one before performing any database operations. The following line of code shows how you do this, given a data source object named **myDataSource**:

```
transactionStarted = [[[myDataSource databaseChannel]
    databaseContext] beginTransaction];
```

Starting and ending a transaction does more than just send the appropriate messages to the database. The database level takes advantage of this information when uniquing objects and making snapshots. For more information on how objects are uniqued, see Chapter 2, ªFrom Database to Objects.º

Your application can also gain a measure of control over database transactions by providing the database channel with a delegate. In particular, by implementing **databaseChannel:willFetchObjectOfClass:withZone:**, the delegate can approve each fetch before it's performed.

For a complete list of all database channel delegate methods, see the EODatabaseChannel class specification in the *Enterprise Objects Framework Reference*.

Handling Errors at the Data Source Level

EODatabaseDataSources don't send delegate messages when an error occurs. Because a data source relies on methods in the access layer, however, you can take advantage of the access layer's error-reporting methods to catch errors as they occur. For a discussion of error handling in the access layer, see ªHandling Errors in the Access Layer.º

Handling Exceptions

Certain errors cause exceptions to be raised. These errors typically occur in the Foundation Kit or in the Enterprise Object Framework's access layer. NEXTSTEP's exception-handling capabilities can be used to catch and handle these exceptions.

One common situation in which exceptions are raised is when the database doesn't have a destination row for a to-one relationship. Because to-one relationships must have exactly one destination object (no more, no less), the Framework assumes that there is a real object there. When the source object is fetched, a fault object is constructed to stand in for the destination. When your code sends a message that requires that the real object be fetched and substituted for the stand-in fault object, an exception is raised when the Framework discovers that no such object actually exists. The correct way to deal with this problem is either to ensure that the data in the database is correct, or to make the relationship to-many. If neither of these options will work in your application, you'll have to place an exception handler around your code at the point where it attempts to resolve the relationship, as shown in this example (which assumes that a set of employee objectsÐ**myObjects**Ðhas already been fetched, and that a related set of department objects are currently represented by fault objects):

```
for (i = [myObjects count] - 1; i >= 0; i--) {
    NSLog(@"%@\n", [[myObjects objectAtIndex:i] lastName]);
NS_DURING
    NSLog(@"%@\n", [[[myObjects objectAtIndex:i] department]
        departmentName]);
NS_HANDLER
```

```
    NSLog(@"<none>");
NS_ENDHANDLER
}
```

The enterprise object classes in the above example implement convenience methods for directly accessing the given attributes: **lastName** and **department** for employee enterprise objects, and **departmentName** for department enterprise objects.

Note that the above technique isn't possible when these values are displayed in the user interface by an EOController, since in that case the controller or an association triggers the fault.

Retrieving Data with a Fetch Loop

When you don't want the overhead of bringing all of your objects into memory at once, and are willing to live with the additional programming involved, you can retrieve data using a fetch loop. You can use a fetch loop to fetch data either as enterprise objects (at the database level) or as dictionary rows (at the adaptor level). Whichever form you prefer, the basic procedure is the same.

The Basic Fetch Loop

Assuming that you have already opened a channel to a database, the basic steps in a fetch loop are:

1. Get an entity for the database table from which you want to retrieve data.
2. Build a qualifier to select only the desired records.
3. Specify the order in which the records are to be retrieved.
4. Begin a transaction.
5. Select the records to be fetched.
6. Fetch a record from the database.
7. Process the record.
8. Repeat steps 6 and 7 until done.
9. Commit or roll back the transaction.

Fetching Enterprise Objects

The following code illustrates the basic fetch loop used when retrieving records as enterprise objects:

```
EODatabaseChannel *myDbChannel;    /* Assume this exists. */
EOEntity *myEntity;                /* Assume this exists. */
EOQualifier *myQualifier;
id myObject;
BOOL ok;

/* Channel should be open at this point. */
myQualifier = [myEntity qualifier];
[[myDbChannel databaseContext] beginTransaction];

/* Select the records to be fetched. */
ok = [myDbChannel selectObjectsDescribedByQualifier:myQualifier
    fetchOrder:nil];
```

```
if (ok) {
    /* This is the fetch loop. */
    while (myObject = [myDbChannel fetchWithZone:NULL]) {
        /* Process the object. */
    }
}

[[myDbChannel databaseContext] commitTransaction];
```

For information on qualifying the records to be fetched, see ªQualifying the Results.º For instructions on specifying the order in which records are fetched, see ªSpecifying the Retrieval Order.º

Resolving Relationships When Fetching Objects

When you fetch an enterprise object using database level methods, resolving a relationship can be as simple as accessing the value for the enterprise object key that represents the relationship. For a to-one relationship, the relationship's value contains the related object. For a to-many relationship, the value for the relationship contains an array object that in turn contains the related set of zero or more objects.

As an illustration of this simplicity, when used outside of a fetch loop the following code excerpt prints an employee's last name and the name of the department to which the employee belongs, given an employee object (myObject) and a related department object:

```
NSLog(@"%@\n", [myObject lastName]);
NSLog(@"%@\n", [[myObject department] departmentName]);
```

The enterprise object classes in the above example implement convenience methods for directly accessing the given attributes: **lastName** and **department** for employee enterprise objects, and **departmentName** for department enterprise objects.

Because the Framework actually uses fault objects to represent relationships, however, this code won't work as written within a fetch loop. When you try to resolve a relationship within a fetch loop using code such as that shown above, the fault is triggered, causing the Framework to try to fetch the related objects. Since your code is already in the middle of a fetch, it won't be able to start another in order to resolve the relationship.

There are two ways around this problem:

· Fetch all objects first, issue a **cancelFetch** if the fetch isn't finished, and then evaluate the relationship. Of course, this implies that you fetch all objects into an array, cancel the fetch, and then work with the objects in the array. While you can write your own code to do this, the methods provided by EODatabaseDataSource can do this for you.

· Open a second channel, and use that channel to resolve the relationship. You can take advantage of EOFault's **qualifierForFault:** class method; given a fault object, it returns the qualifier used to fetch the object that the fault represents. Once you have this qualifier, you can then use it in a **selectObjectsDescribedByQualifier:fetchOrder:** message on the second channel to obtain the destination objects.

For an extensive discussion of fault objects, see Chapter 2, ªFrom Database to Objects,º or the EOFault class specification in the *Enterprise Objects Framework Reference*.

Fetching Row Data

When working at the adaptor level, rows are returned as NSDictionary objects. The following code illustrates the basic fetch loop used when retrieving records from the adaptor. It assumes that you have already opened a channel at the adaptor level.

```
EOAdaptorChannel *adaptorChannel;    /* Assume this exists. */
EOAdaptorContext *adaptorContext;    /* Assume this exists. */
EOModel *myModel;                    /* Assume this exists. */
EOEntity *entity;
EOQualifier *myQualifier;
NSArray *allAttributes;
id myRow;
BOOL ok;

entity = [myModel entityNamed:@"Employee"];

myQualifier = [entity qualifier];
allAttributes = [entity attributes];

/* Channel should be open at this point. */
[adaptorContext beginTransaction];

/* Select the records to be fetched. */
ok = [adaptorChannel selectAttributes:allAttributes
    describedByQualifier:myQualifier
    fetchOrder:nil lock:NO];

if (ok) {
    /* This is the fetch loop. */
    while (myRow = [adaptorChannel fetchAttributes:allAttributes
        withZone:NULL]) {
        /* Process the dictionary object. */
    }
}

[adaptorContext commitTransaction];
```

For information on qualifying the records to be fetched, see ªQualifying the Results.º For instructions on specifying the order in which the records are to be fetched, see ªSpecifying the Retrieval Order.º

Resolving Relationships When Fetching Rows

At the adaptor level, the Framework doesn't do any special processing to deal with relationships. Using the value for the relationship from the source row, you must construct a qualifier and set up a second fetch (with a separate channel, if you need to resolve the relationship within the fetch loop) to obtain the destination row or rows. For example, the following code excerpt displays the last name and department for each employee in the database, given that the department information is stored in department objects that are related to the employee objects. A separate context and channel are used to fetch the department information, so that the relationship can be resolved for each employee record as it's fetched.

```
EOEntity *empEntity;                 /* Assume this exists. */
EOEntity *deptEntity;                /* Assume this exists. */
EOAdaptorContext *empContext;        /* Assume this exists. */
EOAdaptorChannel *empChannel;        /* Assume this exists. */
EOAdaptorChannel *deptChannel;
BOOL ok;
NSDictionary *empRow;
NSDictionary *deptRow;
EOQualifier *deptQualifier;

/* empChannel should be open at this point. */

/* Both channels share a single context. */
deptChannel = [[empContext createAdaptorChannel] retain];
[deptChannel openChannel];
```

```
if([empContext beginTransaction]){
    ok = [empChannel selectAttributes:[empEntity attributes]
        describedByQualifier:[empEntity qualifier]
        fetchOrder:nil lock:NO];

   if (ok) {
        while(empRow = [empChannel
            fetchAttributes:[empEntity attributes] withZone:NULL]){

           NSLog(@"%@\n", [empRow objectForKey:@"lastName"]);

           /* Construct a qualifier for the department channel. */
           deptQualifier = [[EOQualifier alloc]
                initWithEntity:deptEntity
                qualifierFormat:@"deptID = %@",
                [empRow objectForKey:@"deptID"]];

           /* Select and fetch the department record. */
            [deptChannel selectAttributes:[deptEntity attributes]
                describedByQualifier:deptQualifier
                fetchOrder:nil lock:NO];
            deptRow = [deptChannel fetchAttributes:
                [deptEntity attributes] withZone:NULL];
            [deptChannel cancelFetch];

           if (deptRow) {
                NSLog(@"%@\n",
                    [deptRow objectForKey:@"departmentName"]);
            }
        }
    }
    [empContext commitTransaction];
}
```

In the above example, the qualifier could have been constructed using information obtained from the model. For more information, see Chapter 11, ªExploring and Constructing Models.º

Controlling Transactions in the Access Layer

When interacting with the database, an Enterprise Objects Framework application needs to define a transaction scope, even if it's only fetching data. Starting and ending a transaction does more than just send the appropriate messages to the database. The Enterprise Objects Framework takes advantage of this information when uniquing objects and taking snapshots. (For more information on how objects are uniqued, see Chapter 2, ªFrom Database to Objects.º)

As shown in the previous example, you begin a transaction by sending a **beginTransaction** message to your channel. The value returned from **beginTransaction** can be checked to ensure that the transaction was started successfully. When you're done with the channel, but before you close it, send it a **commitTransaction** message (or **rollbackTransaction**, if appropriate).

Your application can also gain a measure of control over database transactions by providing the channel with a delegate. In particular, by implementing **databaseChannel:willFetchObjectOfClass:withZone:** (or **adaptorChannel:willFetchAttributes:withZone:**, if you are working strictly at the adaptor level), the delegate can approve each fetch before it is performed.

For a complete list of all channel delegate methods, see the EODatabaseChannel and EOAdaptorChannel class specifications in the *Enterprise Objects Framework Reference*.

Handling Errors in the Access Layer

EODatabaseChannel, EOAdaptorChannel, and EOAdaptorContext all notify their respective delegates before performing most database operations, using ªwillº messages (such as **adaptor:willReportError:**). Delegates can then check for possible error conditions before they occur, and then allow or disallow the operation, or alter an operation's result. All a delegate need do to receive a notification from its channel or context is implement the appropriate methods. The specific action available to the delegate depends on the message received. If the delegate doesn't implement a particular method, the method isn't invoked.

The delegate's response to most of these ªwillº messages indicates how the operation should proceed:

Return Value	**Interpretation**
EODelegateRejects	Abort the requested operation and return a failure result.
EODelegateApproves	Continue the requested operation and return an appropriate result.
EODelegateOverrides	Do nothing, but return a success result.

When your delegate overrides an operation, it's responsible for seeing that the result of that operation is performed successfully. For example, if the delegate returns **EODelegateOverrides** after being sent a **databaseChannel:willFetchObjectOfClass:withZone:** message, it must have itself successfully fetched an object of the named class.

One of these ªwillº messages, EOAdaptor's **adaptor:willReportError:**, is sent to the EOAdaptor's delegate when an error is generated by the database. These errors are passed to the adaptor's delegate as NSStrings. Because these errors are generated by the database itself, the strings are not guaranteed to be easily parsed by your application. Note that adaptors often have additional database-specific error-handling methods; see the documentation supplied with your adaptor for more information.

You may also want to consider implementing one or both of the following database channel delegate methods:

· Since an EOEntity can be associated with an enterprise object class that isn't linked into the application, it's possible that the database channel won't find the class when it fetches the first object for that entity. In this case, it sends **databaseChannel:failedToLookupClassNamed:** to its delegate, which is then responsible for finding and loading that class (probably from a bundle).

· **databaseChannel:willRefetchConflictingObject:withSnapshot:** is invoked whenever an object is fetched in one transaction while an update on the object is pending in another transaction. This situation frequently arises as the result of a programming error. If this situation occurs, a delegate *must* respond to this method or the Enterprise Objects Framework will raise an exception.

See the EODatabaseChannel, EOAdaptorChannel, and EOAdaptorContext class specifications in the *Enterprise Objects Framework Reference* for more information on these delegate methods.

For information on handling exceptions, see ªHandling Exceptions.º

Qualifying the Results

Qualifiers limit the set of objects selected for a fetch. For instance, you can select all employees with a salary greater than $5000 per month. Or, you could select just those employees that work in a particular department. You can even combine the two, so that you select all employees in a particular department who earn more than $5000 per month.

When fetching all objects at once, you set the qualifier before fetching your objects with EODatabaseDataSource's **setQualifier:** method. When retrieving data in a fetch loop, you specify the qualifier to the select method appropriate to the level at which you are working.

Unrestricted Qualifiers

You use an unrestricted qualifier to indicate that all objects for an entity should be fetched. When you create a data source object, its qualifier is unrestricted by default. When working at the database or adaptor levels, however, you must explicitly specify a qualifier to the **selectObjectsDescribedByQualifier:fetchOrder:** or **selectAttributes:describedByQualifier:fetchOrder:lock:** method. To obtain an unrestricted qualifier from your entity that you can then pass to the select method, send it a **qualifier** message:

```
unrestrictedQualifier = [myEntity qualifier];
```

Qualifier Format

Qualifiers are based on the attributes of the entity being qualified. The Framework uses your qualifier in a SQL WHERE clause; thus, your qualifier can be any valid SQL expression using the internal names of properties belonging to the qualifier's entity. You're responsible for ensuring that the SQL you use in your qualifiers is valid for your database server.

The following are examples of typical qualifier strings:

lastName = \`Smith'
salary > 5000
department.departmentName = \`Sales and Marketing'

An EOQualifier can embed strings, numbers, and other objects into its format string using the conversion specifications listed in the EOQualifier class specification (see the *Enterprise Objects Framework Reference*). This allows qualifiers to be built dynamically, based on the return values of methods.

The following code excerpts build qualifiers similar to the qualifier strings mentioned above, but take the specific values from an already-fetched enterprise object:

```
myQualifier = [[EOQualifier alloc] initWithEntity:empEntity
    qualifierFormat:@"%A = '%@'", @"lastName",
    [anEmpObject lastName]];
myQualifier = [[EOQualifier alloc] initWithEntity:empEntity
    qualifierFormat:@"%A > %f", @"salary", [anEmpObject salary]];
myQualifier = [[EOQualifier alloc] initWithEntity:peopleEntity
    qualifierFormat:@"%A = '%@'", @"department.departmentName",
        [aDepartmentObject departmentName]];
```

The enterprise object classes here implement convenience methods for directly accessing the given attributes: **lastName** and **salary** for employee enterprise objects, and **departmentName** for department enterprise objects.

Specifying the Retrieval Order

The order in which the set of enterprise objects or dictionary rows are retrieved can be specified by an array of EOAttributeOrdering objects. This array is passed in the select method of a database or adaptor channel, or in a **setFetchOrder:** method on an EODatabaseDataSource.

You don't have to specify a retrieval order. To indicate that the retrieval order isn't important when working at the database or adaptor level, specify **nil** as the fetch order in the select message you send to the database or adaptor channel.

No matter which level you are working at (data source, database, or adaptor), the object you use to specify the fetch order is constructed in the same manner. Because you might want to sort on multiple attributes, the fetch order is specified as an array of EOAttributeOrdering objects. The position of each EOAttributeOrdering object within the array determines the attribute's priority within the sort; the first object in the array indicates the primary sort order, the second object in the array (if there is one) indicates the secondary sort order, and so forth.

EOAttributeOrdering objects encompass two pieces of information: the attribute to be used for sorting, and the orderÐascending or descendingÐthat's used to arrange objects with those attributes. The following table lists the ways in which the objects can be ordered:

Parameter	**Sort Order**
EOAnyOrder	unspecified
EOAscendingOrder	lowest-valued first
EODescendingOrder	highest-valued first

The basic steps when specifying the retrieval order are:

1. Identify the EOAttribute object for each attribute needed to order the fetched data.
2. Create an EOAttributeOrdering object for each EOAttribute object, specifying the sort order.
3. Create an NSArray whose elements are the EOAttributeOrdering objects created in the previous step, placing the primary EOAttributeOrdering object first, followed in order of decreasing importance by the remaining EOAttributeOrdering objects.

The following excerpt illustrates how to specify that objects or rows be ordered first by last name, then by first name:

```
EOAttribute *primaryAttribute;
EOAttribute *secondaryAttribute;
EOAttributeOrdering *primaryOrdering;
EOAttributeOrdering *secondaryOrdering;
NSArray *sortOrder;

/* Identify the attribute objects. */
primaryAttribute = [entity attributeNamed:@"lastName"];
secondaryAttribute = [entity attributeNamed:@"firstName"];

/* Create the attribute ordering objects, specifying whether each */
/* attribute should be sorted in ascending or descending order.   */
primaryOrdering = [EOAttributeOrdering
   attributeOrderingWithAttribute:primaryAttribute
   ordering:EOAscendingOrder];
secondaryOrdering = [EOAttributeOrdering
   attributeOrderingWithAttribute:secondaryAttribute
   ordering:EOAscendingOrder];

/* Finally, create the array of attribute ordering objects. */
sortOrder = [NSArray arrayWithObjects:primaryOrdering,
secondaryOrdering, nil];
```