WebObjects Enterprise Objects Programming Guide

2005-08-11

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Figures, Tables, and Listings

Introduction to WebObjects Enterprise Objects Programming Guide

Note: This document was previously titled *Enterprise Objects*.

The Enterprise Object technology brings the benefits of object-oriented programming to database application development. You can use the Enterprise Objects frameworks to build feature-rich database applications that encapsulate your business logic, yet are independent of any particular data source. Enterprise Objects allows you to focus on writing business logic that brings your stored data to life rather than on writing SQL or other database code to access that data.

There are many solutions on the market that provide technology to feed today's information-hungry applications from vast amounts of data in various data sources. However, almost all these solutions are vendor-specific and force the applications you write to depend on a particular data source.

As your application and business needs evolve, these restraints will likely prevent your applications from reaching their full potential. With Enterprise Objects, however, you can build data-driven applications that are independent of any particular data source or data access mechanism. An application built with Enterprise Objects can more easily adapt to new requirements and changing business needs.

Who Should Read This Document

This document is appropriate for Enterprise Objects developers of all levels, but much of the material is more relevant to intermediate and advanced users who are working with fairly complex applications. However, the appendix "Principal Methods" (page 117) and many of the tasks sections throughout the document provide introductory information to help new users get started with the Enterprise Objects frameworks.

New Enterprise Objects developers should consider the document *EOModeler User Guide* a prerequisite to this document. As well as teaching you how to use EOModeler, it is the best way to familiarize yourself with general database concepts and with the core concepts in the Enterprise Objects frameworks such as data modeling. It also discusses the most important characteristics of EOEnterpriseObjects, rather than discussing the mechanics of the Enterprise Objects frameworks, as this book does.

"Road Map to Concepts" (page 11) and "Road Map to Tasks" (page 11) provide road maps to help you decide which sections of this document are most relevant to you.

Terminology

This section provides a key to some of the overloaded terms in the Enterprise Objects frameworks:

- the Enterprise Object technology is also referred to as "Enterprise Objects" (note capitalization) or as the "Enterprise Objects frameworks"
- classes or instances of classes that implement the EOEnterpriseObject interface are referred to as "enterprise objects" (note capitalization)

The terms "Enterprise Objects," "the Enterprise Objects frameworks," and "the Enterprise Object technology" refer to all of the classes in WebObjects that begin with the prefix "EO." This document speaks specifically to the core Enterprise Objects frameworks: classes in the package `com.webobjects.eocontrol` (referred to as the **control layer**) and classes in the package `com.webobjects.eoaccess` (referred to as the **access layer**).

How to Learn About Enterprise Objects

The Enterprise Objects frameworks provide a great deal of flexibility through an extensive API—most of which you'll never need to use or know about. You can go far in building an Enterprise Objects application if you're familiar with a handful of classes and a few dozen methods. The best way to learn about these particular classes and methods is by performing small, focused tasks to gradually get acquainted with the commonly used ones. By learning the technology through these tasks, you build up a set of mental tools that you can later use to build real, data-driven applications. This document provides a number of these tasks to help you get started.

While learning the technology, don't be tempted to use classes and methods that you either don't need or don't understand. You'll probably ever use only 10% of the public API throughout the technology, so don't be too adventurous in the beginning. An axiom of Enterprise Objects development is that "he who writes the least code wins." That is, if in dealing with a particular problem, you end up writing a lot of complicated, custom code, you've probably overlooked functionality that the frameworks provide.

When learning this technology, keep in mind that the learning curve is high but the productivity that results allows you to rapidly build sophisticated, elegant business solutions in a fraction of the time that other database application development systems require. To help tackle the high learning curve, don't try to understand the whole system at once. Fortunately, the layered design of Enterprise Objects facilitates this approach. By working with one layer at a time, the whole system gradually comes into focus.

Part of the difficultly in learning Enterprise Objects is its abstract architecture. It is designed to allow you to work with data sources at an abstract level; it allows you to work with objects that are abstractions of data sources. The objects within the frameworks provide abstract representations of databases, of database transactions, and of the data retrieved from databases.

Although the abstractions within Enterprise Objects contribute to its high learning curve, once you understand the architecture, you'll see that its abstract design provides huge benefits in terms of portability to different data sources and to various client-server application configurations. So if all the abstraction at first seems confusing, be patient—you'll eventually come to appreciate it.

Road Map to Concepts

If you've read *Using EOModeler* or have a little experience building Enterprise Objects applications, you should read these chapters first:

- "Enterprise Objects Overview" (page 13)
- "Business Objects" (page 23)
- "Business Logic" (page 39)

If you are an intermediate developer who wants to know more about the mechanics of Enterprise Objects, read these chapters:

- "Fetching Data" (page 55)
- "Working With the Object Graph" (page 77)
- "Saving Data" (page 83)

If you are an advanced developer, you may be interested in these chapters, which discuss advanced topics:

- "Core Framework Stack" (page 47)
- "Update Strategies" (page 91)
- "Connecting to a Database" (page 101)
- "Concurrency" (page 109)

Road Map to Tasks

If you've read *Using EOModeler* or have a little experience building Enterprise Objects applications, you are probably interested in the tasks described in these sections:

- "Synchronizing Model Changes to Class Files" (page 37)
- "Instantiating Enterprise Objects" (page 37)
- "Providing Initial Values" (page 40)
- "Adding Validation" (page 41)
- "Writing Business Methods" (page 43)
- "Manipulating Relationships" (page 45)
- "Building a Reusable Framework" (page 45)
- "Constructing Fetch Specifications" (page 72)
- "Filtering Fetch Results in Memory" (page 74)
- "Sorting Fetch Results in Memory" (page 74)
- "Accessing WebObjects in Enterprise Objects" (page 113)

Intermediate and advanced developers are probably interested in the tasks in these sections:

- "Providing Separate Stacks" (page 52)
- "Accessing Lower-Level Objects" (page 53)
- "Generating Custom Primary Keys" (page 88)
- "Using Compound Primary Keys" (page 90)
- "Recovering and Refaulting" (page 95)
- "Recovering and Last Write Wins" (page 98)
- "Providing a Connection Dictionary in Code" (page 106)
- "Providing Multiple Database Channels" (page 106)
- "Closing Database Channels" (page 107)
- "Accessing Database Keys" (page 75)

See Also

You can find further documentation for WebObjects and Java Client in three places:

- Project Builder's Developer Help Center, accessible through the Help menu
- Apple's WebObjects documentation site:
  http://developer.apple.com/documentation/WebObjects/index.html
- The WebObjects CD-ROM, which contains the WebObjects API reference, various documents in HTML and PDF, examples, what's new, and legacy documentation

Enterprise Objects Overview

The Enterprise Object technology provides a set of flexible, modular, and mature object-oriented frameworks that allow you to build data-driven applications without needing to worry about an application's data sources. It allows you to work with objects rather than directly with the data source, which reduces development time, reduces the amount of code you need to write, and allows you to build reusable, portable business objects that can be shared between many different applications.

Enterprise Objects specializes in providing mechanisms to retrieve data from various data sources, such as relational databases via JDBC and JNDI directories, and mechanisms to commit data back to those data sources. These mechanisms are designed in a layered, abstract approach that allows you to think about data retrieval and commitment at a higher level than a specific data source or data source vendor.

In marrying relational databases to object-oriented programming, the fundamental problem is how to define the mapping between each world. To solve this problem, Enterprise Objects provides tools and frameworks for **object-relational mapping** to allow you to work with data objects, rather than directly with database tables. These data objects are referred to as **enterprise objects**.

The core element to this mapping is a model file that you build with a tool called EOModeler. This tool is documented in *EOModeler User Guide*, which you should read before reading this document. By understanding how to build models that Enterprise Objects can use, you'll be better equipped to understand the concepts in this book.

This chapter introduces the major concepts in Enterprise Objects. It is divided into the following sections:

- "What Does Enterprise Objects Do?" (page 14) describes the core purposes of Enterprise Objects.
- "Why Enterprise Objects?" (page 15) discusses the Enterprise Objects advantage.
- "What Is an Enterprise Object?" (page 16) introduces the key element in Enterprise Objects applications: instances of classes that implement the EOEnterpriseObject interface.
- "Core Layers" (page 18) introduces the core layers in the Enterprise Objects frameworks.
- "Key Concepts of Object-Oriented Programming" (page 19) discusses some important object-oriented programming principles that you'll encounter when using Enterprise Objects.

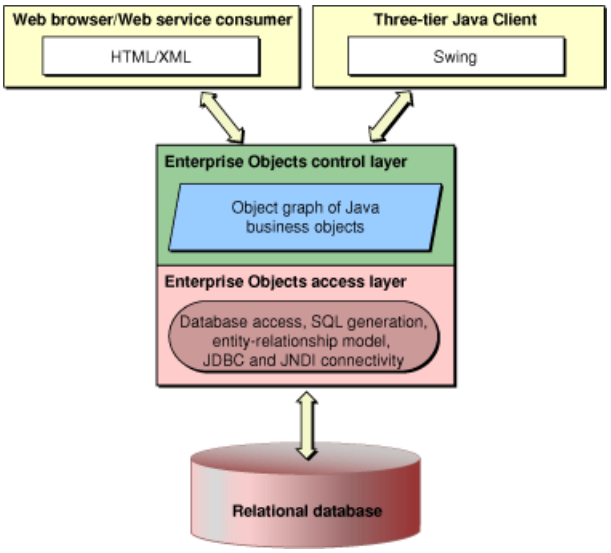
What Does Enterprise Objects Do?

Enterprise Objects is a complex system that does much more than just communicate with data sources. It also includes objects that participate in user interface-to-data source synchronization, it helps you build business logic into Java objects, and it provides an object-oriented framework with which you develop data-driven Web and desktop applications. Enterprise Objects does the following:

- manages transactions for JDBC, JNDI, and other types of data sources
- maps database contents to Java objects
- allows you to focus on writing business logic rather than on writing SQL to talk to a data store
- helps you build object-oriented database applications
- helps you to maintain platform independence while providing the flexibility to take advantage of platform-specific features
- lets you build robust database applications that respect the Model-View-Controller paradigm and object-oriented design principles
- provides a distribution layer between client and server in Java client desktop applications
- provides objects that allow heterogeneous user interface kits to communicate with Java business objects
- provides a change tracking mechanism, as well as automatic undo and redo

"Figure 2-1" gives a high-level view of the core layers of Enterprise Objects.

Figure 1-1 High-level view of Enterprise Objects



Why Enterprise Objects?

There are two common design patterns in database application development, both of which suffer from the same fault. The first design pattern is to implement business logic in user interface code. The second design pattern is to mix business logic with the logic for accessing business data. A variation of the second pattern is to put business logic rules in the data store using triggers, stored procedures, and the like.

In the first case, Swing user interface code or HTML is mixed in the same class with Java business logic. In the second case, Java business logic and SQL code are mixed within the same class. In both cases, this practice of mixing presentation code with business logic code greatly reduces the reusability of those objects. Aside from violating good object-oriented programming practices, both approaches distract developers from their primary task: building applications that use and profit from custom business logic.

In short, mixing business logic with application logic, user interface logic, or with data access logic results in these consequences:

- **Business objects are not reusable.** Putting business logic in user interface code requires you to reimplement business logic in each application and within multiple user interface components in an application. Putting business logic in the data store locks you into that data store, forces you to work in raw SQL, and in general makes your business logic less maintainable.
- **Data integrity is compromised.** If you have to constantly reimplement your business logic in each application you build, you rely on each reimplementation to be correct, which is very risky. If you leave tasks like validation to the data store, you give up earlier and more effective interception points and require a round trip to the data store for tasks that can occur without that kind of network overhead.
- **Business objects are not maintainable.** When you hard-code database artifacts like attribute and entity names within user interface code, any changes to your database schema require you to find and update this related information in each user interface in which business logic is implemented. When you put business logic in the data store, you are forced to put database-specific SQL within your business objects to use the business logic in the data store.
- **Business logic becomes dirty.** The last thing you should think about when developing business logic classes are database artifacts like primary and foreign keys. However, most database application development environments force you to do just this! Needing to think about these things distracts you from the important task of writing great business logic.
- **You are forced to mix languages.** Although many database application development environments force you to mix SQL, Swing, HTML, and Java code within components and classes, this is terrible design and doesn't reflect the expertise within development organizations. Database administrators are SQL experts; application developers are Java experts; Web designers are HTML experts. Combining languages within components devalues the talents within development organizations as it prevents the clean separation of business logic code, user interface code, and data access code.

Fortunately, there is a third approach to database application development that allows you to build clean, reusable, maintainable, and robust business objects without needing to mix SQL, HTML, or Swing code within Java business classes. This is the approach the Enterprise Object technology offers.

It allows you to put business rules in business objects called enterprise objects. Enterprise objects have no knowledge of the data store or data access mechanism from which they derive their data, nor do they have knowledge of the user interfaces that use the data they provide. Enterprise object classes are made up exclusively of Java foundation classes—they contain no SQL, no HTML, and no Swing code. Rather, they work within a hierarchy of other classes that perform specific functions.

This clean separation of business logic, user interface logic, and data access logic allows you to focus on the task of writing business logic and building applications that benefit from that logic.

What Is an Enterprise Object?

An enterprise object is an instance of a class that implements the `com.webobjects.eocontrol.EOEnterpriseObject` interface. You can think of an enterprise object as a business object. It contains your business rules and represents the data on which your business relies.

An enterprise object is like any other object in that it couples data with the methods for operating on that data. However, an enterprise object has certain characteristics that distinguish it from other classes:

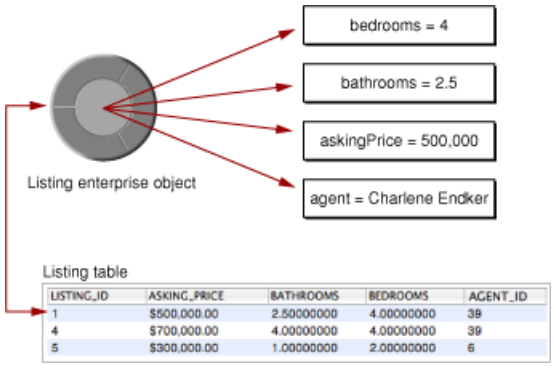
- It has properties that map to stored data; an enterprise object instance typically corresponds to a single row or record in a database.
- It knows how to interact with other parts of the Enterprise Object technology to give and receive values for its properties.

An enterprise object is an instance of a class (such as `com.webobjects.eocontrol.EOGenericRecord`) that has attributes that correspond to the data values from the database row or record from which the object is instantiated. An enterprise object has a corresponding model that defines the mapping between the class's object model and the database schema. However, an enterprise object doesn't explicitly know about its model.

This model (an EOModel) allows Enterprise Objects to map database tables to enterprise objects (Java objects) so that a database row maps to an instance of a particular enterprise object class. For instance, a database table named "LISTING" maps to an enterprise object class, which is usually an instance of or subclass of `com.webobjects.eocontrol.EOGenericRecord`, which implements the `com.webobjects.eocontrol.EOEnterpriseObject` interface.

You control what kind of enterprise object class a given database table maps to. "Figure 2-2" illustrates the mapping between a database table and an enterprise object.

Figure 1-2 Mapping between a database table and an enterprise object



You have a good deal of control over how this mapping happens. For example:

- You can map an enterprise object class to a single database table, a subset of a table, or to more than one table (via relationships, flattening, and other mechanisms). For example, a Listing enterprise object can get its price, square footage, and number of bedrooms from a LISTING table, its street address, city, state, and zip code information from an ADDRESS table, and its property tax information from a LISTING_TAX table.
- Generally, an enterprise object property (a datum in an enterprise object) maps to a single column in a table. For example, an employee's salary can be defined strictly by the row value of the SALARY column in an EMPLOYEE table. But an employee's salary can also be mapped to a derived attribute such as "SALARY + BONUS" or "SALARY * 12".
- You can map an enterprise object inheritance hierarchy to one or more database tables. Inheritance is a powerful feature that lets you think of your database schema in terms of an object hierarchy. It is discussed in detail in *EOModeler User Guide*.

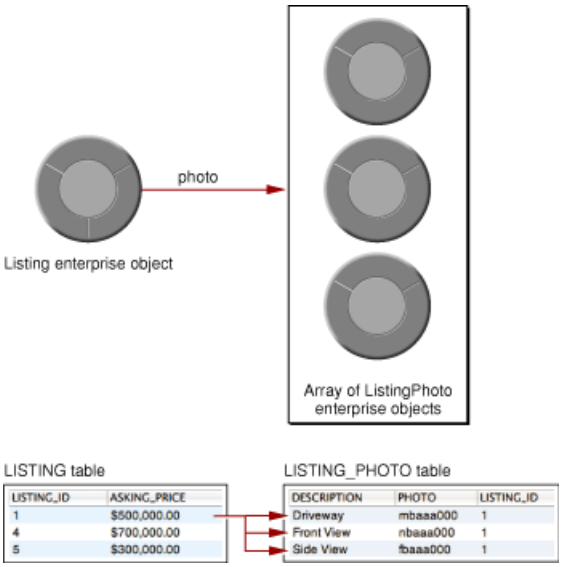
One of the most powerful features of Enterprise Objects is that it maps database joins to relationships between objects. Again, you have a great deal of flexibility with regard to how this mapping occurs. You can map to-one, to-many, and many-to-many relationships, and you can control the business rules that govern those relationships, such as delete rule, optionality, and ownership.

"Figure 2-3" illustrates the mapping between a database table and an enterprise object. A single row in the LISTING table is related to one or more rows in the LISTING_PHOTO table: there is a to-many relationship between the LISTING table and the LISTING_PHOTO table.

When an enterprise object is instantiated from the LISTING table, placeholder objects (called **faults**) for related rows in the LISTING_PHOTO table are also instantiated. When a Listing's `photos` relationship is accessed, those faults are turned into fully formed enterprise objects, which allows the Listing enterprise object to access data in the ListingPhoto enterprise objects. All the SQL you would normally need to write to resolve the joins is handled for you automatically.

Faulting is explained in more detail in "Faulting" (page 33).

Figure 1-3 Mapping between two joined tables and enterprise object instances



Core Layers

Now that you understand the basic ideas behind enterprise object instances, you need to understand how those instances work within the Enterprise Objects frameworks.

Enterprise Objects is composed of many layers, each of which is responsible for a particular set of tasks in an application. A layer generally corresponds to a Java package so that the `com.webobjects.eoaccess` package is referred to as the access layer and the `com.webobjects.eointerface` package is referred to as the interface layer.

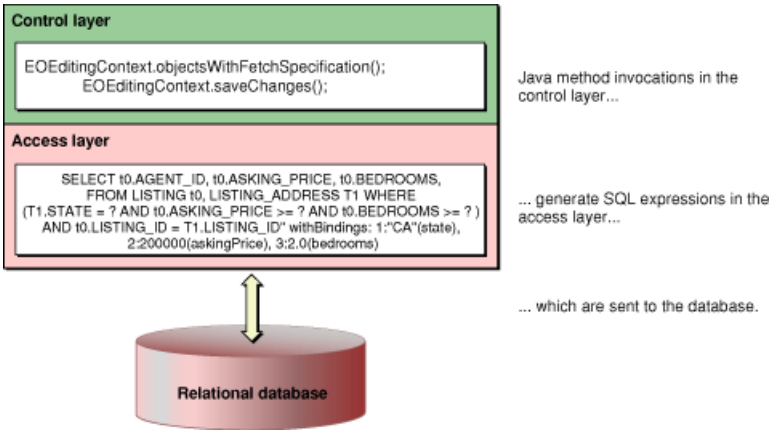
The first layer you work with is referred to throughout the documentation as the control layer and is composed of the classes in the `com.webobjects.eocontrol` package. Many things happen at the control layer, but the most important thing to remember is that the control layer is where the graph of enterprise objects lives. This layer is not concerned with and has no direct knowledge of how data is collected into it.

The **object graph** is the organizing principle of Enterprise Objects. The object graph refers to a graph, or structure, of business objects and the objects that support those business objects, such as editing contexts. In an Enterprise Objects application, the object graph is made up of editing contexts, which in turn contain enterprise objects—objects instantiated from classes that implement the EOEnterpriseObject interface. Editing contexts allow enterprise objects to know about each other and also to know when data from other layers arrives at the control layer and affects the data within enterprise object instances.

The advantage of the object graph is that it allows you to work with database content as Java objects, rather than as SQL or another more primitive language. For example, rather than managing a particular data source's primary keys, you can rely on Enterprise Objects to manage primary keys based on the operations you make within a given object graph. Likewise, rather than managing join tables for a particular database, you simply manipulate related enterprise objects in the object graph and then rely on Enterprise Objects to generate the appropriate SQL to propagate changes to the data source.

The control layer is where you insert new enterprise objects, delete enterprise objects, invoke fetches and commits, manipulate relationships, and most importantly, it is where you implement business logic. The operations you perform at the control layer are eventually transformed into SQL and vendor-specific or vendor-optimized calls at lower levels of the technology, as illustrated in "Figure 2-4".

Figure 1-4 Java method invocations map to SQL expressions



Two of the most common challenges new developers face when coming to the Enterprise Object technology from other database development environments is to think at a level higher than SQL and to not think of any particular data source vendor. So, once you begin to trust that Enterprise Objects produces SQL for you and that you can invoke methods on business objects that result in database operations, you are better prepared to understand the technology.

Key Concepts of Object-Oriented Programming

There are a few key concepts you should understand when learning the Enterprise Objects frameworks. They are all rooted in good object-oriented design principles and many of them share ancestry with Apple's **Cocoa** object-oriented application framework. Although understanding these concepts isn't necessary to develop Enterprise Objects applications, familiarity with them helps you gain a better appreciation of how the technology works.

That said, there is a lot of information to digest when learning Enterprise Objects and you won't need to put these concepts into use until after you know the basics. So you may want to skip this section and come back to it later on.

Key-Value Coding

Key-value coding is a data access mechanism in which the properties of an object are accessed indirectly by key or name, rather than directly as fields or by invocation of accessor methods. It is used throughout Enterprise Objects but is perhaps most useful to you when accessing data in relationships between enterprise objects.

Key-value coding enables the use of **keypaths** to traverse relationships. For example, if a Person entity has a relationship called `toPhoto` whose destination entity (called PersonPhoto) contains an attribute called `photo`, you could access that data by traversing the keypath `toPhoto.photo` from within a Person object.

Keypaths are just one way key-value coding is an invaluable feature of Enterprise Objects. In general, though, it is most useful in providing a consistent way to access an object's data. Rather than needing to know if an object's data members have accessor methods, what the names of those accessor methods are, or if the data is accessible through fields, all you need to know are the keys that represent an object's data. Key-value coding automatically finds the data, regardless of how the object provides its data. In this context, key-value coding satisfies the classic design pattern principle to "encapsulate the things that varies."

Using the previous example, if you're working in a class that has access to a PersonPhoto object, you can get the photo data in the PersonPhoto object by invoking `valueForKey("photo")`. You don't need to know if the PersonPhoto object returns that data by providing an accessor method called `photo` or `getPhoto`, or through a field. Key-value coding is another feature of Enterprise Objects that reduces the amount of code you need to write and helps you write reusable business objects.

Advanced key-value coding concepts are discussed in:

- "Relationships With Other Enterprise Objects" (page 27) discusses how you can use key-value coding to access data across relationships between enterprise object instances.
- "Accessing an Enterprise Object's Data" (page 34) discusses the lookup order in key-value coding, the "stored" version of key-value coding methods, and other key-value coding concepts.
- "Error Handling for Accessors" (page 36) discusses what happens when key-value coding lookup fails for a particular key and how to manage the failure.

In addition, the API reference for `com.webobjects.eocontrol.EOKeyValueCoding` and for `com.webobjects.foundation.NSKeyValueCoding` provide more detailed information on key-value coding.

Delegation

Delegation is a way of extending a class in an object-oriented framework. Whereas subclassing means that you write a new class based on an existing class, delegation means that you make use of hooks that the class provides to change its behavior. In object-oriented design methodology, delegation is a form of class composition.

Delegation does just what it suggests: When you implement an object's delegate, you are telling the object to use the behavior that your delegate specifies at specific points in an application's execution. So for example, in the case of an editing context object, if you write a delegate for this object, the editing context object checks your delegate at certain points in an application's execution and uses the logic your delegate specifies instead of or in addition to its own. The pseudocode in "Listing 2-1" illustrates delegation.

Listing 1-1 Pseudocode illustrating delegation

```
if (delegateIsImplemented) {
    useDelegateImplementation();
} else {
```

```
    useDefaultImplementation();
}
```

Compared to subclassing, delegation is usually a more elegant solution to extend or customize the behavior of a class. A class's delegate provides hooks for those parts of a class that the class's designer wants to be customizable. Furthermore, subclassing can be a tricky, tedious, and undesirable task for large classes that have many dependencies on other classes. Delegation reduces the amount of code you have to write and better ensures that your custom code will work, even when the delegate's class is updated.

Notification

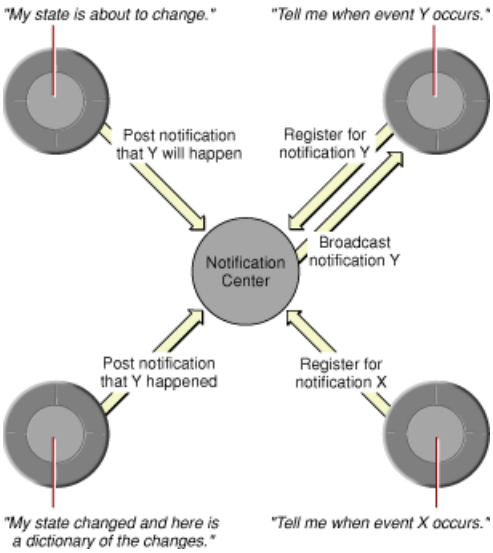
The objects in an application are no good on their own. It is the relationships and communications between objects that breathe life into an application. In a complex system like Enterprise Objects, it's common that a single object needs to communicate with multiple objects simultaneously.

There are any number of bad approaches to this problem and one remarkably good one: **notifications**. Notification allows for a loose coupling of objects—objects that need to communicate with one another don't need to know about one another in advance. Rather, when objects need to communicate with one another, they rely on a notification center to manage their communications.

Enterprise Objects uses an object called a notification center to broadcast messages to objects in an application. Objects register with the notification center for specific notifications they want to receive. They also post notifications with the notification center when they want to notify other objects that a particular event occurred.

"Figure 2-5" illustrates how objects and the notification center interact. A given object tells the notification center which notifications it's interested in; it posts notifications to the notification center when its state changes; it receives notifications that it is registered to receive.

Figure 1-5 Notification center and its clients



Notifications are particularly important in the control layer. Enterprise objects often have relationships to other enterprise objects in the same editing context. When a value in one enterprise object changes, other enterprise objects in that editing context (and perhaps in other editing contexts, too) need to be told of the change. So, when a value in an enterprise object changes, that enterprise object's editing context posts a notification to let the other enterprise objects in that editing context update their data accordingly.

Notification creates a kind of dynamic nervous system within the object graph with the notification center as the brain of the system. You can think of the object graph as an organic system with many built-in feedback loops. As you begin to work with object graphs, you'll find them to be a very natural way of thinking about data.

You may wonder about the differences between delegation and notification. With delegation, an object (the delegator) asks another object (the delegate) for a response before it performs a particular action (communication between objects can be forced and synchronous). Notification differs from delegation in that objects that post notifications don't directly communicate with the objects that receive those notifications (communication between objects can never be forced and synchronous; object communication through notification is always asynchronous). With delegation, the delegate object has the opportunity to affect the tasks of its delegator, whereas in notification, objects don't directly affect the execution of other objects.

Principal Methods of Enterprise Objects

As discussed in "How to Learn About Enterprise Objects" (page 10), Enterprise Objects includes an extensive API, most of which you never need to know about or use. To help you get started with the API, the appendix "Principal Methods" (page 117) describes the 30 most common methods that all Enterprise Objects applications are likely to use.

Business Objects

The most important part of applications that use the Enterprise Object technology are the application's enterprise object instances—your application's business objects. Designing these objects is one of the primary tasks in developing an Enterprise Objects application. This chapter explains the structure of enterprise objects, how they interact with other parts of the technology, and how you can leverage the power of the technology in your custom enterprise object classes.

Designing enterprise objects entails three major steps:

- Designing a database schema
- Modeling enterprise objects
- Implementing enterprise object classes

This chapter briefly discusses the first two steps but leaves the details to *EOModeler User Guide*. This chapter focuses on the third step, implementing enterprise object classes. The chapter is divided into these sections:

- "Reference Entity" (page 24) introduces a simple entity object that is referred to throughout the chapter.
- "Designing a Schema" (page 24) discusses how design decisions in database schemas and in their corresponding enterprise object models affect one another.
- "Defining the Model" (page 25) discusses the earliest design decisions you make about enterprise objects classes, which are reflected in your application's data model.
- "Which Enterprise Object Class?" (page 25) discusses the various concrete implementations of the EOEnterpriseObject interface.
- "Choosing Class Properties" (page 26) discusses how to choose class properties.
- "Relationships With Other Enterprise Objects" (page 27) discusses relationships between enterprise objects.
- "Referential Integrity" (page 28) discusses the referential integrity rules you can build into enterprise objects.
- "Implementing an Enterprise Object" (page 29) discusses how to implement an enterprise object class.
- "Accessing an Enterprise Object's Data" (page 34) discusses how to use key-value coding to access an enterprise object's data.

- "Common Programming Errors" (page 36) discusses some common mistakes in implementing enterprise object classes.
- "Synchronizing Model Changes to Class Files" (page 37) discusses how to merge model changes with existing enterprise object class files.
- "Instantiating Enterprise Objects" (page 37) discusses common ways to instantiate enterprise objects programmatically.

Reference Entity

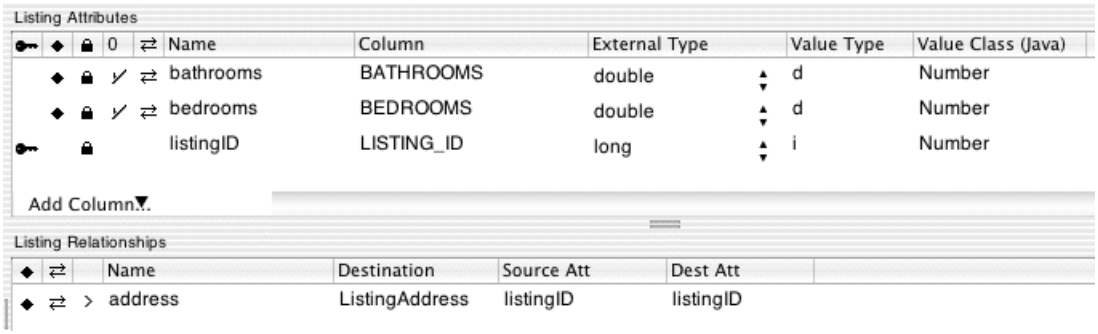
To help you better understand the concepts in this chapter, it helps to refer to a simple entity that includes characteristics that illustrate those concepts. This section describes that entity.

The entity is a simplified version of the Listing entity in the Real Estate model. (This example model is included with WebObjects 5.2. It is installed in `/Developer/Examples/JavaWebObjects/Frameworks/JavaRealEstate`.) It includes three attributes and one relationship:

- a primary key called `listingID`
- an attribute called `bedrooms`, which stores the number of bedrooms
- an attribute called `bathrooms`, which stores the number of bathrooms
- a relationship called `address` to a ListingAddress entity

This entity appears in EOModeler as shown in "Figure 3-1".

Figure 2-1 A simplified version of the Listing entity



Designing a Schema

If the application you're building with Enterprise Objects accesses an existing database, its schema dictates many of the design decisions you'll make when modeling enterprise objects. However, if you're designing a database schema while designing enterprise objects, you have more flexibility and you can design a database schema that takes advantage of features of enterprise objects, such as faulting.

If you're designing the database schema while you design enterprise objects, be sure to keep both designs in mind as you work; decisions you make about either design affect the other.

This chapter doesn't directly address issues of database design, but the information presented here helps you create a design that works effectively with the Enterprise Objects frameworks.

Defining the Model

The work of writing enterprise objects typically begins in EOModeler. An individual enterprise object class maps to an entity in the data model. It then follows that an entity's characteristics map to an enterprise object's characteristics and behavior. You make the following decisions for a particular enterprise object in EOModeler:

- Should an enterprise object map to EOGenericRecord or a custom class? ("Which Enterprise Object Class?" (page 25))
- What entity attributes should be class properties? ("Choosing Class Properties" (page 26))
- What relationships does an enterprise object have with other enterprise objects? ("Relationships With Other Enterprise Objects" (page 27))
- What referential integrity rules should the relationships in an enterprise object have? ("Referential Integrity" (page 28))

Which Enterprise Object Class?

An enterprise object is an instance of a class that implements the `com.webobjects.eocontrol.EOEnterpriseObject` interface. This interface provides enterprise objects with the infrastructure they need to communicate with other enterprise objects, to set and return values for enterprise object attributes, and to provide a framework for validation and other business logic.

The Enterprise Objects frameworks provide for you two classes that each provide a complete implementation of the EOEnterpriseObject interface. These classes are `com.webobjects.eocontrol.EOCustomObject` and `com.webobjects.eocontrol.EOGenericRecord`. By default, enterprise object instances map to the implementation of the EOEnterpriseObject interface in the class EOGenericRecord. That is, entities you add to an EOModel are mapped to EOGenericRecord.

EOGenericRecord is sufficient for enterprise objects in which you don't need to add custom business logic. But when you need to add custom business logic to a particular enterprise object class, you need to create a custom subclass of EOGenericRecord. To do this, you assign to an entity the name of a custom subclass and then generate source files for that custom class. This process is described in "Generating Source Files" (page 29).

When you assign an entity to a custom enterprise object class, the class file EOModeler generates for the custom class inherits from EOGenericRecord, but you can change the superclass to EOCustomObject. Most people prefer to work with enterprise object classes that inherit from EOGenericRecord because they are a bit easier to work with. There are at least a few reasons:

- EOGenericRecords by default don't include fields for their properties—they use key-value coding accessors and store the values in a dictionary. This makes EOGenericRecord subclasses easier to maintain, especially if you make changes to the entity to which a generic record maps.
- EOGenericRecord subclasses automatically use deferred faulting, which increases performance and decreases memory usage. See "Advanced Faulting" (page 66) for more information on deferred faulting.
- EOGenericRecord subclasses automatically take care of certain faulting requirements like invoking `willRead` and `willChange` when necessary. See "Faulting" (page 33) for more information about these two methods.

That said, there are a few reasons why you might want custom enterprise object classes to inherit directly from EOCustomObject rather than from EOGenericRecord. They include:

- EOCustomObject subclasses tend to be more transparent than generic record subclasses; there is less "magic" going on behind the scenes.
- You access the fields of EOCustomObject subclasses with `set` and `get` accessor methods, rather than through key-value coding, which some people prefer.
- If you are making an existing Java class persistent, it's easier to do so by changing its superclass to EOCustomObject rather than EOGenericRecord as you can then use the class's existing fields and accessor methods.
- EOCustomObject subclasses allow you to strongly type attributes, which provides compile-time type checking and saves you from having to type cast the results of key-value coding accessors.

In short, the choice of which class to use is really a matter of preference. Both types of classes are first-class citizens within the Enterprise Objects frameworks, so you don't lose any power or flexibility with either choice. If you're new to the technology, you'll probably find that EOGenericRecords are easier to work with because they encapsulate some of the framework's complexities.

"Implementing an Enterprise Object" (page 29) compares concrete examples of both types of classes.

Note: Although the enterprise object classes you write are of one of these types, when you program to those classes in your application, when possible, *always program to the interface rather than to the implementation*. That is, declare enterprise objects you use as EOEnterpriseObject rather than as EOCustomObject or as EOGenericRecord, regardless of that object's concrete class type. This makes your applications easier to maintain and allows you to change how enterprise object classes are implemented without affecting the code that uses them.

Choosing Class Properties

When you add attributes to entities in EOModeler, they are marked as class properties (denoted by the diamond in an attribute's row). When an attribute is marked as a class property, accessor methods for that attribute are included in that class definition when you generate source files for the class using EOModeler. When instances of an enterprise object are created, the data that maps to an attribute is fetched from the database and stored in the enterprise object instance, so you should mark attributes only as class properties if their data is needed in an enterprise object instance.

As a general rule, you should mark attributes as class properties only if their values are meaningful to a particular enterprise object's business logic or if their values are displayed in the user interface. An attribute that won't be used in business logic or that users don't need to see or manipulate shouldn't be marked as class properties.

Database artifacts such as primary and foreign keys shouldn't be marked as class properties. There are exceptions to this rule, such as when you provide custom primary keys, but they are rare. Enterprise Objects handles key generation for you, so these keys aren't necessary in your business logic classes. Also, it is rare that a primary key would contain data valuable to a user. (This generally reflects poor schema design, such as when a driver license number is used as a table's primary key.)

In the Listing entity described in "Reference Entity" (page 24), the `bedrooms`, `bathrooms`, and `address` properties are selected as class properties, while the entity's primary key, `listingID`, is not marked as a class property. The `bedrooms`, `bathrooms`, and `address` properties contain data that is meaningful to the application's users; the `listingID` property does not.

A consideration, especially for advanced users, is to ensure that a given entity doesn't have any redundant attributes that are marked as class properties. A common mistake when building data models that use entity inheritance is to mark multiple attributes as class properties when the attributes in fact represent the same data, such as can occur when flattening attributes and relationships. This issue is discussed in more detail in *EOModeler User Guide*.

Relationships With Other Enterprise Objects

One of the most important parts of a data model is the relationships it expresses between entities. You use relationships to access data across multiple objects in an object graph. Relationships are similar to attributes in that they can be marked as class properties. When they are marked as class properties, accessor methods are included in the generated class file for them, as is the case for attributes that are marked as class properties.

Unlike attributes, however, there are additional factors to consider when marking a relationship as a class property. Relationships in Enterprise Objects are not used just to specify the affinity between two tables. Relationships are also used when configuring entity inheritance and when flattening relationships (either explicitly or in a many-to-many relationship). Follow these guidelines when deciding whether to mark a relationship as a class property:

- Relationships that represent an entity's relationship to an intermediate join table (such as in a many-to-many relationship) shouldn't be marked as class properties (unless they contain data that's meaningful in your application, which is rare).
- When modeling vertical inheritance mapping, the to-one relationship from a child entity to its parent shouldn't be marked as a class property. This relationship is used to allow the parent entity's attributes to be flattened into the child entity and in most cases isn't of any use as a class property. See *Using EOModeler* for more information on entity inheritance.

In the Listing entity described in "Reference Entity" (page 24), the `address` property, which represents the entity's relationship to a ListingAddress entity, is marked as a class property. The relationship is a to-one relationship and joins on the source entity's primary key `listingID` and the destination entity's primary key `listingID`.

When you mark relationships as class properties, to-one relationships are represented as references to other objects and to-many relationships are represented as NSArray objects. You usually access the data in other objects by using relationship properties to traverse the object graph in running applications. For example, the following code uses Listing's `address` relationship to access the `street` property in the Address object:

```
String streetAddress = (String)listing.valueForKeyPath("address.street");
```

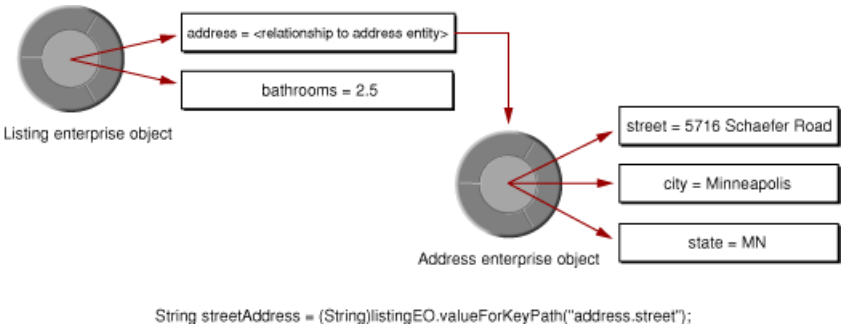
"Figure 3-2" illustrates this example.

The Listing enterprise object has many properties—many attributes and relationships. To access an attribute in an enterprise object, you invoke on it the method `valueForKey("attributeName")`. To access an attribute in one of an enterprise object's relationships, you invoke on it the method `valueForKeyPath("relationship.relationshipAttribute")`. The methods `valueForKey` and `valueForKeyPath` return a `java.lang.Object`, which you usually cast to the data type of the attribute being accessed (such as `java.lang.Number` and `java.lang.String`, or NSArray if the attribute represents a to-many relationship in the enterprise object).

To access a Listing's features, you use this code to get an array of a particular listing's features:

```
NSArray features = (NSArray)listing.valueForKeyPath("features");
```

Figure 2-2 Key value-coding and relationships as key paths



Enterprise Objects handles the resolution of relationships automatically, which may involve fetching from the database. You don't have to worry about explicitly generating SQL statements or resolving joins; Enterprise Objects takes care of all this for you. All you need to supply is a well-designed data model and Enterprise Objects takes care of the rest.

Referential Integrity

Referential integrity rules are part of the definition of relationships between enterprise objects, just like joins between database tables. These rules work to maintain the integrity and consistency of the data that gets committed to the database. Enterprise Objects supports these referential integrity rules:

- optionality
- delete rule
- owns destination

- propagate primary key

These rules are described in more detail in *EOModeler User Guide*. In most cases, you configure referential integrity rules in EOModeler. Your enterprise object classes have no direct knowledge of the referential integrity rules of their relationships. These rules are characteristics of EORelationship objects of which individual enterprise objects have no direct knowledge. This means that you need to be concerned with referential integrity rules when designing data models but not when adding custom logic to enterprise object classes.

Implementing an Enterprise Object

As discussed in "Which Enterprise Object Class?" (page 25), one of the first decisions you need to make about an enterprise object is what class it maps to. In many cases, an enterprise object should map to EOGenericRecord. Enterprise object classes that include custom business logic should map to custom subclasses of EOGenericRecord. In some cases, particularly for advanced developers, an enterprise object can map to a custom subclass of EOCustomObject.

From the perspective of Enterprise Objects, these three choices are identical: they all interact with the technology in the same way because they all implement the EOEnterpriseObject interface. That interface specifies a complete set of methods that support common operations for all types of enterprise objects. It includes methods for initializing enterprise object instances, announcing changes to other enterprise objects, setting and retrieving property values in enterprise objects, and performing validation of enterprise object values.

One of the three types of enterprise object class mappings should be sufficient for all circumstances. *You should never need to implement the EOEnterpriseObject interface from scratch.* Most of the methods in EOEnterpriseObject are meant to be used internally, and besides, the default implementations of the interface provide the default behavior you want.

Generating Source Files

The first step in creating a custom enterprise object is with EOModeler's Generate Java Files and Generate Client Java Files commands. These commands take the entities, attributes, and relationships you've defined in a model to generate a corresponding enterprise object class. Before generating a class file for a particular entity, you must assign a class name to that entity. You assign custom class names to entities in EOModeler's table mode. While in table mode, select the root of the tree view to display the model's entities in table mode, as "Figure 3-3" illustrates.

In this table, you specify the class name to use for an entity. The class name is used when you generate source files for that entity.

Figure 2-3 Custom classes for entities

Name	Table	Class Name	Client-Side Class Name
ContactType	CONTACT_TYPE	EOGenericRecord	EOGenericRecord
Customer	USER	webobjectsexamples.realestate.server.Customer	webobjectsexamples.realestate.
Feature	FEATURE	EOGenericRecord	EOGenericRecord
Listing	LISTING	webobjectsexamples.realestate.common.Listing	webobjectsexamples.realestate.
ListingAddress	LISTING_ADDRESS	EOGenericRecord	EOGenericRecord

EOModeler warns you if you try to generate a source file for an entity that hasn't been assigned a custom class name. When you specify a custom class name for an entity, in most cases EOModeler generates a class file whose class member inherits from EOGenericRecord. However, if you're using inheritance, a child entity's class definition is generated to inherit from its parent.

The class that EOModeler generates for the entity described in "Reference Entity" (page 24) appears in "Listing 3-1". It provides public accessor methods for the entity's class properties, but not for the entity's primary key, `listingID`, which is not selected as a class property.

Listing 2-1 Generated class for simplified Listing entity (EOGenericRecord)

```
import com.webobjects.foundation.*;
import com.webobjects.eocontrol.*;
import java.math.BigDecimal;
import java.util.*;

public class Listing extends EOGenericRecord {

    public Listing() {
        super();
    }

    public Number bedrooms() {
        return (Number)storedValueForKey("bedrooms");
    }

    public void setBedrooms(Number value) {
        takeStoredValueForKey(value, "bedrooms");
    }

    public Number bathrooms() {
        return (Number)storedValueForKey("bathrooms");
    }

    public void setBathrooms(Number value) {
        takeStoredValueForKey(value, "bathrooms");
    }

    public EOEnterpriseObject address() {
        return (EOEnterpriseObject)storedValueForKey("address");
    }

    public void setAddress(EOEnterpriseObject value) {
        takeStoredValueForKey(value, "address");
    }
```

```
    public NSArray features() {
        return (NSArray)storedValueForKey("features");
    }

    public void setFeatures(NSArray value) {
        takeStoredValueForKey(value, "features");
    }

    public void addToFeatures(EOEnterpriseObject object) {
        includeObjectIntoPropertyWithKey(object, "features");
    }

    public void removeFromFeatures(EOEnterpriseObject object) {
        excludeObjectFromPropertyWithKey(object, "features");
    }
}
```

If you prefer to use EOCustomObject subclasses, you can modify the class in "Listing 3-1" as shown in "Listing 3-2".

Listing 2-2 Generated class for simplified Listing entity (EOCustomObject)

```
import com.webobjects.foundation.*;
import com.webobjects.eocontrol.*;
import java.math.BigDecimal;
import java.util.*;

public class Listing extends EOCustomObject {

    private Double _bathrooms = null;
    private Double _bedrooms = null;
    private EOEnterpriseObject _address = null;
    private NSArray _features = null;

    public Listing() {
        super();
    }
    public static boolean usesDeferredFaultCreation() {
        return true;
    }
    public Double bathrooms() {
        willRead();
        return _bathrooms;
    }

    public void setBathrooms(Double value) {
        willChange();
        _bathrooms = value;
    }

    public Double bedrooms() {
        willRead();
        return _bedrooms;
    }

    public void setBedrooms(Double value) {
        willChange();
        _bedrooms = value;
```

```
    }
    public EOEnterpriseObject address() {
        willRead();
        willReadRelationship(_address);
        return _address;
    }

    public void setAddress(EOEnterpriseObject value) {
        willChange();
        _address = value;
    }
    public NSArray features() {
        willRead();
        willReadRelationship(_features);
        return _features;
    }

    public void setFeatures(NSArray value) {
        willChange();
        _features = value;
    }

    public void addToFeatures(EOEnterpriseObject object) {
        includeObjectIntoPropertyWithKey(object, "features");
    }

    public void removeFromFeatures(EOEnterpriseObject object) {
        excludeObjectFromPropertyWithKey(object, "features");
    }
}
```

These two code listings demonstrate the primary differences between EOGenericRecord subclasses and EOCustomObject subclasses. "Listing 3-1" (page 30) contains no fields for the object's attributes; they are stored in a dictionary that the class maintains internally. Contrast this with "Listing 3-2" (page 31), which includes fields for the object's attributes: `_bathrooms`, `_bedrooms`, `_address`, and `_features`.

"Listing 3-2" includes accessor methods that explicitly act on the object's fields, whereas the accessor methods in "Listing 3-1" use key-value coding to access the object's data in its internal dictionary.

"Listing 3-2", being a class that inherits from EOCustomObject, explicitly calls `willChange` immediately before a property is mutated and calls `willRead` immediately before a property is accessed. These invocations are required in EOCustomObject subclasses to support faulting and change notification; they are invoked automatically in EOGenericRecord subclasses, so they aren't present in the class in "Listing 3-1".

Finally, "Listing 3-2" implements a feature of Enterprise Objects called deferred faulting, which is discussed in "Deferred Faulting" (page 66). It implements this feature by overriding `usesDeferredFaultCreation` and by invoking `willReadRelationship` in accessor methods that return values from the object's relationships. "Listing 3-1" uses deferred faulting automatically because it is an EOGenericRecord subclass.

Fields

As discussed in "Generating Source Files" (page 29), EOModeler generates class files whose class members inherit from EOGenericRecord. By default, EOGenericRecords do not include fields for their properties. Rather, they store their properties in a dictionary that can be accessed using key-value coding accessors. This both reduces the amount of code you need to write and makes your custom enterprise object classes easier to maintain.

That said, you can add fields for an enterprise object's properties to an EOGenericRecord subclass if you want to, or you can use an EOCustomObject in which you are required to use fields, as "Listing 3-2" (page 31) illustrates.

Change Notification

In Enterprise Objects, objects that need to know about changes to an enterprise object register as observers for particular change notifications. When an enterprise object is about to change, it is responsible for posting a notification so that registered observers are notified that it's about to change. To do this, enterprise object instances invoke `willChange` prior to altering their state. Whenever you add your own methods that change an object's state, you need to include this invocation.

When invoked in an EOGenericRecord subclass, the stored value accessor (`takeStoredValueForKey`), automatically invokes`willChange` for you, as a convenience. However, in enterprise object class that includes fields for its properties, this isn't the case, so you must invoke `willChange` yourself, as the class in "Listing 3-2" (page 31) does.

The fact that change notification is handled for you is another reason why some people prefer to use EOGenericRecord subclasses.

Faulting

When an enterprise object is about to retrieve data for one or more of its properties, it is required to notify other objects that it's about to take action. It does this by invoking `willRead`. An EOGenericRecord class using `takeStoredValueForKey` to access data automatically invokes `willRead`.

This method is a part of Enterprise Object's **faulting** mechanism. Faulting is the mechanism by which Enterprise Objects delays the full initialization of an enterprise object instance until that object's data is actually required. You can think of faulting as creating a shell of an enterprise object that includes just some (or perhaps none) of its data. See "Figure 6-3" (page 61) for an illustration.

Faulting reduces memory consumption and provides a performance improvement to applications by delaying fetches to the database until data is actually needed. Database fetches are expensive, especially during the resolution of relationships. Often, an enterprise object needs a reference to a particular relationship but doesn't necessarily need the data in that relationship. By providing that enterprise object with a reference to the relationship using a fault, you save the expense of performing a fetch if it's not necessary.

The default implementation of `willRead` checks to see if its receiver has already been fully initialized (that is, if its receiver is a full-formed enterprise object). If it hasn't been fully initialized, it fills the object with values fetched from the database. Before an application attempts to invoke a method on a particular enterprise object, you must ensure that object has already fetched its data. To ensure that

an enterprise object is in the correct state before its data is accessed, you need to invoke `willRead`, typically in "get" methods. (Enterprise objects don't have to invoke `willRead` in "set" methods because the default implementation of `willChange` invokes `willRead` internally.)

Again, if you use EOGenericRecord subclasses that don't access their properties with fields, `willRead` is handled for you by `storedValueForKey`.

Accessing an Enterprise Object's Data

In implementing your enterprise object classes, you want to focus on the code that's unique to your application, not on the code that deals with fitting your objects into the framework of Enterprise Objects. To help with this, Enterprise Objects provides key-value coding, a standard interface for accessing an enterprise object's properties (key-value coding was introduced in "Key-Value Coding" (page 19)).

Key-value coding specifies that an object's properties are accessed indirectly by name or key rather than directly through invocation of an accessor method or as fields. This provides a consistent way to access an object's data, regardless if the object provides accessor methods or fields to its data. The EOEnterpriseObject interface implements the EOKeyValueCoding and NSKeyValueCoding interfaces, so custom enterprise object classes automatically inherit key-value coding behavior.

The basic methods for accessing an object's values are `takeValueForKey` and `valueForKey`, which set or return the value for the specified key, respectively. Key-value coding uses the first accessor it finds when both setting and getting the value for a given key. The following lists describe the lookup order of accessors.

The order of lookup when retrieving a value is:

- `get`*Keyname*`()` method
- *keyname*`()` method
- `_get`*Keyname*`()` method
- *_keyname*`()` method
- *keyname* field
- *_keyname* field
- `handleQueryWithUnboundKey("`*keyname*`")` method, if none of the above accessors are found

In the Listing entity described in "Reference Entity" (page 24), an invocation of `valueForKey("`*bedrooms*`")` on a Listing enterprise object looks up the value of the `bedrooms` property by invoking accessor methods and fields in the following order:

- `get`*Bedrooms*`()` method
- *bedrooms*`()` method
- `_get`*bedrooms*`()` method
- *_bedrooms*`()` method
- *bedrooms* field

- *_bedrooms* field
- `handleQueryWithUnboundKey("`*bedrooms*`")` method, if none of the above accessors are found

The order of lookup when setting a value is:

- `set`*Keyname*`(`*value*`)` method
- `_set`*Keyname*`(`*value*`)` method
- *keyname* field
- *_keyname* field
- `handleTakeValueForUnboundKey(`*value*`, "`*keyname*`")` method, if none of the above accessors are found

In the Listing entity described in "Reference Entity" (page 24), an invocation of `takeValueForKey(`*new Integer(4)*`, "`*bedrooms*`")` on a Listing enterprise object attempts to set the value of the `bedrooms` property by invoking accessor methods and setting fields in the following order:

- `set`*Bedrooms*`(new Integer(4))`
- `_set`*Bedrooms*`(new Integer(4))`
- `bedrooms = new Integer(4)`
- `_bedrooms = new Integer(4)`
- `handleTakeValueForUnboundKey(new Integer(4), "`*bedrooms*`")` method, if none of the above accessors are found

There is another set of methods defined by the EOKeyValueCoding interface, `takeStoredValueForKey` and `storedValueForKey`. You never explicitly invoke these methods, but you may implement them in enterprise object classes. The stored value methods are used internally to transport data to and from trusted sources. For example, `takeStoredValueForKey` is used to initialize an object's properties with values fetched from the database, whereas `takeValueForKey` is used to modify an object's properties from values provided by a user.

The default lookup order for the stored value methods for retrieving the value of a property is:

- `_get`*Keyname*`()` and *_keyname*`()` methods
- *_keyname* and *keyname* fields
- get*keyname*`()` and *keyname*`()` methods

The default lookup order for the stored value methods for setting the value of a property is:

- `_set`*Keyname*`()` method
- *_keyname*, and *keyname* fields
- `set`*Keyname*`()` method

Refer to the API reference for `com.webobjects.eocontrol.EOKeyValueCoding` and for `com.webobjects.foundation.NSKeyValueCoding` for complete information about accessor lookup order.

EOGenericRecord adds some additional behavior to key-value coding. When `takeStoredValueForKey` or `storedValueForKey` is invoked in an EOGenericRecord object, a corresponding `willChange` or `willRead` invocation is sent automatically, before the accessor is looked up. This supports the requirement that enterprise object instances notify other instances when they're about to change the state of their data. This requirement is described in "Change Notification" (page 33).

Error Handling for Accessors

As introduced in "Accessing an Enterprise Object's Data" (page 34), the public key-value coding accessors `valueForKey` and `takeValueForKey` have a default lookup order of accessors. The final lookup key for `valueForKey` is `handleQueryWithUnboundKey("`*keyName*`")` and for `takeValueForKey` is `handleTakeValueForUnboundKey(`*value*`, "`*keyName*`")`. That is, the default implementation of key-value coding invokes these methods when they receive a key for which they can find no accessor methods or fields. The default implementations throw exceptions, but you can override them to handle the error more gracefully.

Common Programming Errors

This section discusses some of the more subtle issues you need to be aware of when building enterprise object classes.

Overriding equals or hashcode

Don't override `equals` or `hashcode`. Enterprise Objects uses the methods `equals` and `hashcode` in enterprise object classes to perform object comparisons and other functions. You should not implement either method in your enterprise object classes.

A common mistake is to use `equals` to compare enterprise objects. Avoid doing this and instead compare the global IDs of enterprise objects using `equals`. The code in "Listing 3-3" provides an example.

Listing 2-3 Comparing two enterprise objects using `equals` on their global IDs

```
if ((editingContext.globalIDForObject(enterpriseObject1)).
equals(editingContext.globalIDForObject(enterpriseObject2)));
```

Immutable Primary Keys

Enterprise Objects does not support changeable primary keys. That is, you cannot change the value of a row's primary key. A common, though poor, design pattern in database application development is to store meaningful business data as primary keys such as driver license numbers.

What if a value like this changes? If you try to change the value of a row's primary key in Enterprise Objects, the enterprise object that represents that row will never be allowed to save in the Enterprise Objects application. The optimistic locking mechanism in Enterprise Objects always throws an exception if a row's primary key changes. So, don't mix columns that include meaningful business data with columns that are used to define database structure.

If you encounter a situation in which you must change an object's primary key, the recommended procedure is to create a new enterprise object of the same type, copy the data from the original object into the new object, and delete the original object.

Primary Keys and Allows Null

In some cases, you need to set the allows null characteristic of a primary key attribute to `true`. This situation often occurs in master-detail relationships in which a detail object is added to the relationship but is not immediately saved to the database. You usually set a primary key's allows null characteristic in the advanced attribute inspector in EOModeler.

What a primary key's allows null characteristic means, however, may be unclear. No primary key is ever allowed to be `null` in a database that Enterprise Objects accesses. When you allow `null` for a primary key, validation within Enterprise Objects doesn't throw an exception when a new enterprise object is created and is used within the application before being inserted into the database; on the way to the database, Enterprise Objects generates a value for the primary key regardless of its allows `null` characteristic.

Synchronizing Model Changes to Class Files

In the course of building an Enterprise Objects application, you'll likely make changes to a model that affect enterprise object class files that you've generated from EOModeler, as described in "Generating Source Files" (page 29). Since you usually add custom logic to those enterprise object class files, you must make sure not to overwrite those changes when generating class files from an updated EOModel. EOModeler is careful to not blindly overwrite generated class files and instead presents a dialog allowing you to merge changes.

Clicking Merge in this dialog opens the FileMerge application and allows you to merge the updated entity definition class with the old entity definition class that contains custom business logic.

Instantiating Enterprise Objects

There are at least a few ways to create enterprise objects programmatically. The approach you use largely depends on the configuration of the entity from which you want to create the object.

In EOModeler, if the entity's class is assigned to EOGenericRecord, you must first retrieve the entity's class description before instantiating an enterprise object. An entity's class description holds meta-information about an entity that describes an entity's various characteristics, such as its attributes and relationships. Once you have an entity's class description, you can instantiate an enterprise object of that class.

For example, consider the Listing entity described in "Reference Entity" (page 24). You can use this code to instantiate an enterprise object of type Listing:

```
EOClassDescription listingCD =
EOClassDescription.classDescriptionForEntityName("Listing");
EOEnterpriseObject listing =
         listingCD.createInstanceWithEditingContext(null, null);
editingContext.insertObject(listing);
```

Note that immediately after the enterprise object is created, it is inserted into an editing context. As a cardinal rule, all enterprise objects reside in an editing context. This is necessary in order for enterprise objects to send and receive the notifications necessary for change tracking and other mechanisms within Enterprise Objects. So, for every enterprise object you create, you must immediately insert it into an editing context.

You can simplify the above code example by supplying an editing context object as the first argument to the `createInstanceWithEditingContext` invocation.

If an entity's class is assigned to a custom subclass of EOGenericRecord (so that the Listing entity's class name assignment in EOModeler is "Listing" or "com.myapp.Listing"), you can also create an enterprise object with this code by using the subclasses's constructor:

```
Listing image = new Listing();
editingContext.insertObject(image);
```

You can also use the method in the EOUtilities class called `createAndInsertInstance` to instantiate an enterprise object:

```
EOUtilities.createdAndInsertInstance(editingContext(), "Listing");
```

Business Logic

After you design and build enterprise object classes as discussed in "Business Objects" (page 23), you need to add **business logic** to some of those classes. The applications you build with Enterprise Objects derive most of their value from the business logic—or business rules—you build into them. Business rules make data relevant and the implementation of business logic is perhaps the most important task when building data-driven applications.

Many database application development environments force you to implement business logic in all the wrong places, such as in the data source itself or intermixed within user interface code. Implementing business logic in these places makes it less reusable, harder to maintain, and makes it less adaptable to changing business needs. These and other consequences of implementing business logic in the wrong places are described in "Why Enterprise Objects?" (page 15).

This chapter describes the best place to implement business logic—in reusable, portable, data source–independent Java objects called enterprise objects. It consists mostly of specific tasks that you commonly perform when adding business logic to Enterprise Objects applications. It includes these sections:

- "Custom Classes" (page 39) discusses why you need to generate class files for enterprise objects to which you want to add custom business logic.
- "Providing Initial Values" (page 40) teaches you how to provide values to enterprise object instances immediately upon creation.
- "Adding Validation" (page 41) teaches you how to validate properties and other state in enterprise object classes.
- "Writing Business Methods" (page 43) teaches you how to write business methods that return values based on an enterprise object's data.
- "Manipulating Relationships" (page 45) teaches you how to use enterprise objects in relationships.
- "Building a Reusable Framework" (page 45) teaches you how to build a reusable framework of business logic that can be shared among applications.

Custom Classes

The first step in implementing business logic in an enterprise object is assigning a custom class to an entity and generating a Java class file for that entity that will contain the custom business logic. As described in "Implementing an Enterprise Object" (page 29), by default all enterprise objects are

assigned to the class `com.webobjects.eocontrol.EOGenericRecord`. For many enterprise objects, this is sufficient. However, if you want to implement business logic in an enterprise object, you must assign that enterprise object to a custom subclass of EOGenericRecord or EOCustomObject.

The assignment of classes to enterprise objects happens in EOModeler. You assign entities class names by selecting the root of the entity tree while in table mode and editing class names in the table.

It's recommended that you supply fully qualified class names but this isn't strictly required. So while you could specify Document as the class name for the Document entity, it's best to specify a fully-qualified name such as `com.mycompany.Document` instead.

Once you assign a class to an entity, you need to generate a Java class file for that entity. To do this, simply select the entity in the tree view and choose Generate Java Files from the Property menu. Class file generation is discussed in more detail in "Generating Source Files" (page 29). In the save dialog that appears, choose your project directory or the directory of a business logic framework. See "Building a Reusable Framework" (page 45) for more information on building a framework that you can use to share the same business logic classes among many applications.

Providing Initial Values

One of the most common business rules is to assign initial values to newly inserted records. For example, when a user creates a new record, you may want to immediately populate a field in that record called `creationDate`.

To do this, simply override the method `awakeFromInsertion` in an enterprise object class:

```
public void awakeFromInsertion(EOEditingContext context) {
    super.awakeFromInsertion(context);
    if (creationDate() == null) {
        setCreationDate(new NSTimestamp());
    }
}
```

This method sets the `creationDate` attribute to the current time. (Determining if the attribute is `null` is not usually necessary except in three-tier Java Client applications.)

Important: Always invoke `super` when overriding methods in Enterprise Objects.

`awakeFromInsertion` is automatically invoked immediately after an enterprise object class is instantiated and inserted into an editing context. You use this method to set default values in enterprise objects that represent new data.

For enterprise objects that represent existing data (data that's already stored in a data source), you can use the method `awakeFromFetch` to provide additional initialization. This method is sent to an enterprise object that has just been created from a database row and initialized with database values.

You may wonder why it's not recommended to initialize an enterprise object's values in an object's constructor. An enterprise object's constructor represents the earliest state in the life of a particular object. *The state of an enterprise object at the point of construction is not complete; the object is not fully*

initialized. It may not yet have been inserted into an editing context and it might not even have been assigned a global ID. You don't ever want to manipulate an enterprise object that isn't in an editing context or that doesn't have a global ID.

Therefore, any logic you add in an enterprise object's constructor may fail or be invalidated while the object finishes initializing. By providing custom logic in `awakeFromInsertion` or `awakeFromFetch`, however, you are guaranteed that an enterprise object is fully initialized.

Adding Validation

Another common requirement of business logic is that it validates all data that users enter, both to ensure that the data will work with other business rules and also to ensure the integrity of stored data. Enterprise Objects provides many different hooks to validate data. When you provide certain validation methods in your business logic classes, Enterprise Objects automatically invokes them to validate your data.

The EOValidation interface in the control layer defines these validation methods:

- `validateClientUpdate`
- `validateForDelete`
- `validateForInsert`
- `validateForSave`
- `validateForUpdate`

Validating Before Certain Operations

Some of these methods are invoked during certain Enterprise Objects operations, and by overriding them in your business logic classes, you can implement custom validation rules. When an application performs one of these operations on an editing context (save, delete, insert, or update objects), the editing context in which the operation is invoked sends a validation message to its enterprise objects. Based on the result returned from those enterprise objects, the operation is either allowed to continue or is refused.

For example, if you implement `validateForSave` in a particular enterprise object class, when a user attempts to save changes to a particular enterprise object (thereby invoking `saveChanges` on the object's editing context), that object's editing context first asks the enterprise object if it's in a consistent state to save. It does this by invoking `validateForSave`. If `validateForSave` doesn't throw an NSValidation.ValidationException, the operation is allowed to continue. However, if `validateForSave` throws an NSValidation.ValidationException, the save operation is not allowed to continue.

It's up to you to decide what constitutes a valid enterprise object. In this example, you could write custom business logic in `validateForSave` that makes sure the enterprise object's data is valid based on your application's business rules.

All classes that implement the EOEnterpriseObject interface include a default implementation of the `validateFor`*Operation* methods. The default implementation of `validateForDelete`, for example, validates that an enterprise object's relationships conform to their specified referential integrity rules.

Important: Because all enterprise object classes inherit default implementations of the `validateFor`*Operation* methods, you must invoke super's implementation before performing custom validation logic.

The default implementations of `validateForSave`, `validateForInsert`, and `validateForUpdate` invoke `validateValueForKey` for each of an object's class properties. See "Validating Individual Properties" (page 42) for more information on `validateValueForKey`.

Validating Individual Properties

In addition to the validation that occurs before certain operations are allowed to proceed, you can also validate the individual properties of an enterprise object. The EOValidation interface defines another validation method, `validateValueForKey`. The default implementation of `validateValueForKey` searches for and invokes methods of the form `validate`*Key* in enterprise object classes. You implement a `validate`*Key* method for each property of an enterprise object you want to validate.

For example, it's common to want to validate a zip code field to make sure that a user entered five numbers (for zip codes in the United States). In an enterprise object class in which the `zipcode` field exists, you add this method:

```
public Object validateZipcode(Object zipcode) throws
NSValidation.ValidationException {
    if ((Number)zipcode.length() > 5) {
        throw new NSValidation.ValidationException("Invalid zipcode.");
}
    else return zipcode;
}
```

When a user tries to assign a value to the zipcode property of an enterprise object, the default implementation of `validateValueForKey` invokes `validateZipcode`. Based on the result of that method, the user-entered value is either allowed to be assigned to the property or refused. Immediately before invoking `validate`*Key* methods, the default implementation of `validateValueForKey` validates the property against constraints specified in the data model, such as `null` constraints and referential integrity rules.

Another common use of `validateValueForKey` is to coerce user-entered data and return the coerced value. For example, users may enter a phone number in a text field in a form. That text field is bound to the `phoneNumber` attribute of an enterprise object. Rather than force users to enter the phone number in a particular format, such as with dash or period separators, you can let them enter a number in any format. Then, in the `validatePhoneNumber` method, you can coerce that number into the format you want and return the coerced string rather than the user-entered string.

Writing Business Methods

The properties of a given enterprise object (its attributes and relationships) do not usually provide all the values or data an application needs to be useful. To be of any value to your business, you usually need to add custom business logic in an application. The data in database tables usually stores raw business data. In order to make meaningful results from that data, you need to write business logic, usually in the form of business methods.

A business method in an enterprise object class returns a value based on data in the enterprise object's properties. In the Real Estate model, you could write a business method to return the number of listings above a certain selling price that were sold by a particular agent.

This business method seeks information regarding a particular agent, so it should be implemented in the Agent business logic class. The method requires an Agent enterprise object and uses an Agent's `listings` relationship and two attributes of a Listing enterprise object, `isSold` and `sellingPrice`, to determine the desired business information. Examples of this method appear in "Listing 4-1", which assumes Listing is an EOGenericRecord subclass and in "Listing 4-2", which assumes Listing is an EOCustomObject subclass.

Listing 3-1 Business method to determine information about the properties sold by a particular agent (assuming EOGenericRecord)

```
public int listingsSoldAbovePrice(String targetSellingPrice) {
     int hitCount = 0;

     NSArray listings = listings();
     java.util.Enumeration enum = listings.objectEnumerator();

     while (enum.hasMoreElements()) {
       webobjectsexamples.realestate.common.Listing listing =
(webobjectsexamples.realestate.common.Listing)enum.nextElement();
       int isSold = ((Integer)listing.valueForKey("isSold")).intValue();
       int sellingPrice =
((Integer)listing.valueForKey("sellingPrice")).intValue();
       if ((isSold == 1) && (targetSellingPrice >= sellingPrice)) {
        hitCount++;
      }
    }
    return hitCount;
}
```

Listing 3-2 Business method to determine information about the properties sold by a particular agent (assuming EOCustomObject)

```
public int listingsSoldAbovePrice(BigDecimal targetSellingPrice) {
     int hitCount = 0;

     NSArray listings = listings();
     java.util.Enumeration enum = listings.objectEnumerator();

     while (enum.hasMoreElements()) {
       webobjectsexamples.realestate.common.Listing listing =
(webobjectsexamples.realestate.common.Listing)enum.nextElement();
       Boolean isSold = listing.isSold();
       int sellingPrice = (listing.sellingPrice()).intValue();
```

```
        if ((isSold) && (targetSellingPrice.intValue() >= sellingPrice)) {
         hitCount++;
        }
      }
    return hitCount;
}
```

Within the Enterprise Objects frameworks, there are a number of other mechanisms you can use to derive business data. At the model level, you can specify custom read and write formats for particular attributes to coerce their values when they are read from the database and written back to the database. Also at the model level, you can define derived attributes that use custom SQL you write to derive business data. Both of these techniques are discussed in *EOModeler User Guide*.

The Foundation framework also provides mechanisms that help you write business logic. The class NSArray.Operator provides a number of operators that provide common information about a set of business data. These operators are listed in "Table 4-1".

Table 3-1 NSArray operators

Operator	Operator description
`count`	Returns the number of elements in an array.
`max`	Returns the element in the array with the highest value.
`min`	Returns the element in the array with the lowest value.
`avg`	Returns the average of the values in the array.
`sum`	Returns the sum of the values in the array.

In the Real Estate business logic framework (located in `/Developer/Examples/JavaWebObjects/Frameworks/JavaRealEstate`), the `webobjectsexamples.realestate.server.Agent` class includes a method that uses the `@avg` operator. It is shown in "Listing 4-3".

Listing 3-3 A business method using an NSArray operator

```
public Number averageRating() {
    if (_averageRating == null) {
    _averageRating = (Number)(ratings().valueForKey("@avg.rating.rating"));
    }
    return _averageRating;
}
```

Each of the operators requires an array of objects whose data type is a `java.lang.Number` (which includes the concrete classes `java.lang.Integer` and `java.lang.BigDecimal`). As shown in "Listing 4-3", you use the operators within an invocation of the key-value coding method `valueForKey`. All the operators except `@count` require you to specify both an array of objects and the element within the array on which to apply the operator.

Manipulating Relationships

Enterprise Objects makes working with relationships rather simple. All you need is two enterprise objects in the same editing context to manipulate relationships programmatically. Were you to perform the same kind of task in other database development environments, you'd likely have to write many lines of SQL to relate a record in one table with a record in another table. With Enterprise Objects, however, a single method invocation does this for you.

Say you have an enterprise object representing a Document entity and a relationship in that entity called `writers`, which represents the authors of the document. To associate a new Writer record with the Document record, you simply create an enterprise object for the writer and then add it to the relationship with this code:

```
document.addObjectToBothSidesOfRelationshipWithKey(writer, "writers");
```

The first argument in the method invocation represents the Writer object. The second argument corresponds to the name of the relationship in the Document entity. If the relationship (in this case `writers`) is modeled with an inverse relationship, this method also adds the object to the other side of the relationship.

In addition to adding records in a relationship, you'll likely also need to remove them. Fortunately, Enterprise Objects provides another method that does all the work for you:

```
document.removeObjectFromBothSidesOfRelationshipWithKey(writer, "writers");
```

Building a Reusable Framework

Well-designed enterprise objects are reusable. That is, if you do not use SQL, access layer classes (besides EOUtilities), or WebObjects classes (`com.webobjects.appserver`) in enterprise object classes, your enterprise objects will be reusable.

The best way for multiple applications to share the same business logic is to build a framework. This framework holds custom Java classes generated by EOModeler and `.eomodeld` files. By using a framework, changes to enterprise object classes in that framework and to the model are picked up by all applications that use the framework.

To build a framework, make a new project of type WebObjects Framework (see the document *Project Builder for WebObjects Developers* for more information on project types). Then, add business logic classes and other resources to it.

The only tricky part is assigning classes and resources to the correct targets. Model files and server-side business logic classes should be assigned to the Application Server target, while client-side business logic classes should be assigned to the Web Server target. If you're not building a three-tier desktop application, you'll only have server-side business logic classes. See the document *Project Builder for WebObjects Developers* for more information on targets in WebObjects projects.

Core Framework Stack

This chapter introduces the objects that form the core of the Enterprise Objects frameworks in WebObjects 5.2. You need to be familiar with these core objects to understand the concepts in the rest of the book.

The core stack is composed of the classes in these packages:

- `com.webobjects.eocontrol` (the control layer)
- `com.webobjects.eoaccess` (the access layer)

The access layer also contains the classes that make up the **adaptor layer**. This chapter discusses the key objects in each layer, how the two core layers interact, and how to customize the core Enterprise Objects stack to meet certain application requirements. It is divided into these sections:

Roles

The core Enterprise Objects stack plays many roles. The most important roles are:

- connecting to data sources

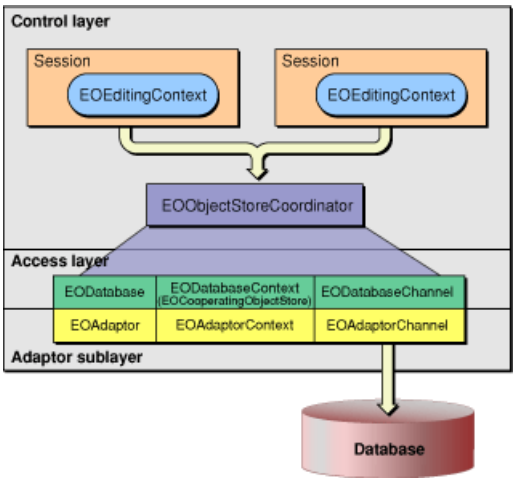
- managing transactions with data sources
- storing snapshots of database rows
- forming enterprise objects from raw row snapshots
- creating and managing the object graph

The core stack provides a default configuration that should be appropriate for most kinds of applications. Some applications, however, may need to customize the configuration for performance or other reasons. This chapter discusses the default configuration and its common variations.

Important Objects

To understand the key components of the Enterprise Objects stack is to first understand the most important objects in the core stack. They are illustrated in "Figure 5-1" (page 48). A brief description of those objects follows.

Figure 4-1 Default core objects interaction



EOObjectStore (control layer)

An object store is an "intelligent" repository of objects. It is responsible for constructing enterprise objects, registering enterprise objects, servicing enterprise object faults, and saving changes made to objects. You can think of an object store as a workspace for enterprise objects. EOObjectStore is an abstract class that has a number of specific subclasses in the core Enterprise Objects stack.

EOObjectStoreCoordinator (control layer)

A concrete subclass of EOObjectStore, an object store coordinator mediates between multiple object stores both in the control layer and in the access layer. It mediates between any number of editing contexts in the control layer and their associated database contexts (or other object store subclasses) in the access layer.

When multiple cooperating object stores needs to communicate with one another, the object store coordinator mediates this communication. It provides an abstraction to multiple data

sources and multiple workspaces that contain data from those sources, allowing objects made up of data from different data sources to work together.

EOCooperatingObjectStore (control layer)
: Another subclass of EOObjectStore, the cooperating object store defines the mechanics for object stores that work together to manage data from several distinct data repositories.

  When multiple cooperating object stores need to communicate with one another, the object store coordinator mediates this communication. If a cooperating object store determines that certain changes need to be handled by another object store, it asks the object store coordinator to handle the changes (usually by handing them off to another cooperating object store that can handle them). This is an abstract class whose most commonly used concrete subclass is EODatabaseContext.

EOEditingContext (control layer)
: Another concrete subclass of EOObjectStore, an editing context is the core object with which you most often interact. Enterprise object instances live within editing contexts, and editing contexts provide the infrastructure that allows enterprise objects to communicate with one another. An editing context is the highest-level object in the core framework stack and it communicates with a database through its lower-level objects.

EODatabaseContext (access layer)
: Another concrete subclass of EOObjectStore, a database context is responsible for analyzing the changes made to enterprise objects in editing contexts and determining exactly the changes that need to be made in the database based on the changes in editing contexts. It hands off this list of changes that need to be made in the database (this list is referred to as adaptor operations) to an adaptor channel by way of a database channel. You rarely need to explicitly interact with a database context.

EODatabaseChannel (adaptor sublayer of access layer)
: A database channel is used by a database context to communicate with an adaptor channel.

EOAdaptorChannel (adaptor sublayer of access layer)
: An adaptor channel is responsible for actually communicating with a data source and sending it commands. However, the operations an adaptor channel performs are always made within the context of a transaction that is managed by an adaptor context object.

How It Stacks Up

How the objects in "Important Objects" (page 48) work together is a bit complicated. It gets even more complicated if you customize the core stack. "Figure 5-1" shows a simplified overview of the core stack. The interactions between the objects in the core stack are discussed throughout this chapter.

The most important part of "Figure 5-1", perhaps, is that all an application's sessions use the same object store coordinator. This means that multiple users using the same application instance share a connection to the database and share row-level snapshots of data retrieved from each database (because they share the same database context). This has advantages and disadvantages. See "Multiple Coordinators and Optimistic Locking" (page 93) for more details.

The Access-Control Divide

One of the most important things to understand when considering the Enterprise Objects core is the relationship between objects in the access layer and objects in the control layer. Fortunately, this part of the Enterprise Objects core is easy to understand: EOObjectStoreCoordinator mediates between the control layer and the access layer. More specifically, it mediates between editing contexts in the control layer and database contexts in the access layer. EOObjectStoreCoordinator is discussed in "Object Store Coordinator" (page 50).

Object Store Coordinator

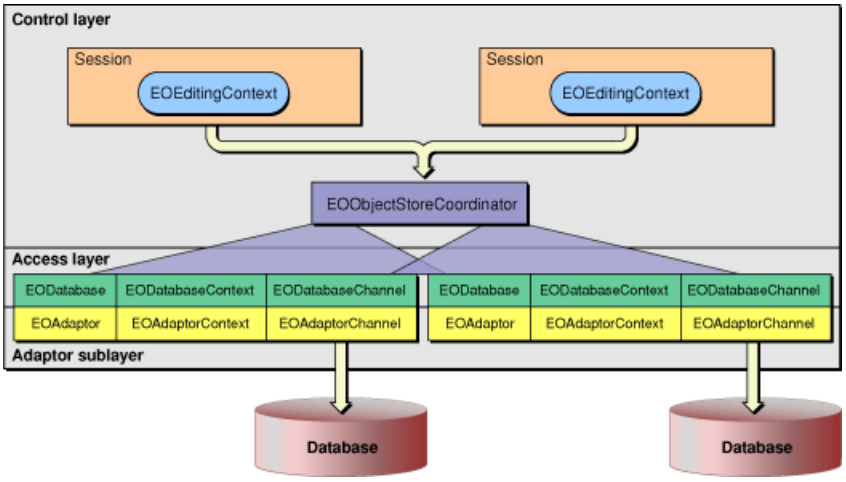
By default, an application has exactly one object store coordinator, as illustrated in "Figure 5-1". An object store coordinator manages possibly many cooperating object stores (usually database contexts, each of which represents a data source), as "Figure 5-2" illustrates. This allows individual instances of enterprise objects to include data from multiple data sources and it allows enterprise objects constituted from data in different repositories to interact with one another.

In "Figure 5-2", the object store coordinator manages the interactions between enterprise objects that need to communicate both with a MySQL database and an Oracle database. An object store coordinator is by default associated with as many database contexts (cooperating object stores) as the number of distinct data sources the application connects to.

So as illustrated in "Figure 5-2", an application that uses two models, each model connecting to a different data source, has one object store coordinator that mediates between two separate cooperating object stores (database contexts). By default, each cooperating object store has its own database channel and its own adaptor channel.

The object store coordinator allows the application to use multiple data sources and also allows the application's enterprise objects to consist of data from multiple data sources. The object store coordinator manages the logic that is necessary to support these features.

Figure 4-2 One database context per data source



When to Use Multiple Object Store Coordinators

When you want to provide each session (each user) with its own access layer stack, you configure your application to use multiple object store coordinators. Specifically, you give each session its own object store coordinator. Doing this also instruments concurrency in your application, so it should be done with caution. See "Providing Separate Stacks" (page 52) to learn how to provide each session with its own object store coordinator.

If possible, all users of an application should share an object store coordinator. This is the default behavior (that all editing contexts in a particular application instance share the same object store coordinator) and it helps you avoid and deal with update conflicts caused by multiple users editing the same row or rows of data concurrently.

There are reasons, however, why you would want to provide each user with a separate access layer stack. One of the more common reasons is to provide an audit trail in an application. A crucial part of any audit trail is tracking access to data sources. When you provide each user with a separate access layer stack, you can then use different credentials to establish the database connection, provided that you make the connection programmatically. This is discussed in "Connecting to a Database" (page 101) in the section "Storing in Code" (page 104).

Note: Applications that use multiple object store coordinators consume more memory than applications that use only one because each coordinator contains its own set of row-level snapshots and other data about the enterprise objects they fetch and manage.

Why So Many Object Stores?

After taking a look at the default core Enterprise Objects stack, you're probably wondering why there are so many different kinds of object stores. These objects are all subclasses of EOObjectStore: EOObjectStoreCoordinator, EOCooperatingObjectStore, EODatabaseContext, EOEditingContext. The short answer as to why there are so many kinds of object stores is that the different object stores know different things about enterprise object instances.

An editing context, for example, has no direct knowledge of how to initiate a transaction with a database. A database context, however, does. An editing context, also, has no knowledge that the enterprise objects it holds may be constituted with data from multiple distinct data repositories. An editing context's object store coordinator, however, knows this and knows how service requests that require managing data from disparate sources.

While you don't need to understand the complexities of all these types of object stores or how they interact, just know that they each play an important role within the frameworks.

Providing Separate Stacks

To provide each user of an application with an independent access layer stack, you provide each user with a separate object store coordinator. As described in "The Access-Control Divide" (page 50) and "Object Store Coordinator" (page 50), an EOObjectStoreCoordinator is the highest-level object in the access layer stack. Each object store coordinator instance, then, represents a single database connection to each of the cooperating object stores registered with it.

The editing contexts that share an object store coordinator share a single database connection. Editing contexts that use different object store coordinators can perform concurrent operations to the same database. So if user A performs a fetch, user B can perform a commit concurrently.

To provide each user with an individual object store coordinator, you instantiate a new EOObjectStoreCoordinator, instantiate a new EOEditingContext whose parent is the new EOObjectStoreCoordinator, and set a session's default editing context to be the new editing context whose parent is the new object store coordinator. This can be done programmatically in a Session class using the code in "Listing 5-1".

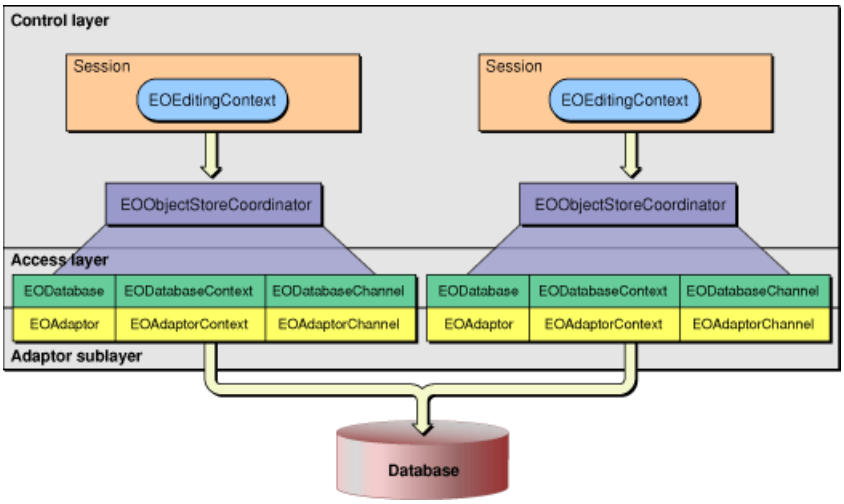
Listing 4-1 Providing separate object store coordinators

```
public class Session extends WOSession {

    public Session() {
        super();

        /* Provide each session with its own access layer stack. */
        EOObjectStoreCoordinator parentObjectStore = new
EOObjectStoreCoordinator();
       EOEditingContext editingContext = new EOEditingContext(parentObjectStore);
        setDefaultEditingContext(editingContext);
    }
}
```

A common reason to provide each user with a separate stack is to provide each user with a separate connection to the database. If you need to do this, you also need to set values in the connection dictionary for each user. This is discussed in "Providing a Connection Dictionary in Code" (page 106).

The objects in an Enterprise Objects application using a separate object store coordinator per session are illustrated in "Figure 5-3".

Figure 4-3 Per-session object store coordinators



When you provide separate object store coordinators to each session, you must adopt a different set of assumptions about how the object graph works. When you provide each session with its own access layer stack, you provide each session with a separate EODatabaseContext, which means that sessions no longer share row-level snapshots—they have entirely separate row-level snapshots. Since row-level snapshots are an integral part of Enterprise Object's optimistic locking mechanism, you must reconsider the locking strategies you implement. This and other related issues are discussed in "Multiple Coordinators and Optimistic Locking" (page 93).

Accessing Lower-Level Objects

Advanced users may need to access an editing context's access layer or adaptor layer objects. To access an editing context's access layer objects, you get an editing context's object store coordinator and the object store coordinator's database context object. From the database context object, you can get the database context's adaptor-level objects.

An editing context can have more than one set of access layer and adaptor layer objects, so you need to identify which set you're looking for. You can do this a number of ways: with a fetch specification, with a global ID, or with an enterprise object. The code in "Listing 5-2" uses a fetch specification to help the root object store determine which of its cooperating object stores can service your request.

Listing 4-2 Accessing lower-level objects

```
EOEditingContext editingContext; //Assume this exists.
EOFetchSpecification fetchSpec;
EOObjectStoreCoordinator rootObjectStore;
EODatabaseContext databaseContext;
EOAdaptor adaptor;
EOAdaptorContext adaptorContext;

fetchSpec = new EOFetchSpecification(entityName, null, null);
rootObjectStore = (EOObjectStoreCoordinator)editingContext.rootObjectStore();
```

```
databaseContext =
(EODatabaseContext)rootObjectStore.objectStoreForFetchSpecification(fetchSpec);
adaptor = databaseContext.database().adaptor();
adaptorContext = databaseContext.adaptorContext();
```

Fetching Data

This chapter discusses the mechanics of retrieving data using Enterprise Objects.

It is divided into the following sections:

- "Objects Involved in Fetching" (page 56) describes the objects involved in retrieving data.
- "Flow of Data During a Fetch" (page 56) discusses the flow of data during a fetch.
- "Enterprise Object Initialization" (page 59) discusses how enterprise objects instances are initialized.
- "Faulting and Relationship Resolution" (page 60) describes how Enterprise Objects resolves relationships and uses faulting to improve performance.
- "Data Integrity Mechanisms" (page 62) discusses some of the data integrity mechanisms Enterprise Objects uses while fetching data, including uniquing, faulting, and snapshotting.
- "Ensuring Fresh Data" (page 64) discusses how Enterprise Objects caches fetched data, when it uses cached data, how to refetch data, how to clear the cache, and in general how to ensure that the enterprise object instances in your application contain fresh data.
- "Advanced Faulting" (page 66) discusses advanced faulting topics such as deferred faulting and batch faulting.
- "Advanced Fetching" (page 69) discusses advanced fetching topics such as raw row fetching and prefetching.
- "Common Delegate Usage" (page 71) discusses delegates that are commonly implemented to customize fetching.
- "Constructing Fetch Specifications" (page 72) teaches you how to build fetch specifications and how to construct qualifiers.
- "Filtering Fetch Results in Memory" (page 74) teaches you how to filter the results of a fetch in memory.
- "Sorting Fetch Results in Memory" (page 74) teaches you how to sort the results of a fetch in memory.
- "Accessing Database Keys" (page 75) discusses how to access an entity's primary and foreign keys when they are not class properties.

Objects Involved in Fetching

There are many objects involved in retrieving data in an Enterprise Objects application. The ones you'll most commonly work with are introduced here.

EOFetchSpecification
: A fetch specification provides a description of what data to retrieve from a data source. A fetch specification always includes the name of an entity—in Enterprise Objects, a single database fetch operation is always done from the perspective of a particular entity. A fetch specification usually includes a qualifier—specific criteria to look for when searching the database. A fetch specification can also include a sort ordering, which specifies that the result set should be sorted in a particular way.

EOQualifier
: A qualifier is often included in a fetch specification to provide criteria for a particular database fetch. There are a number of different kinds of qualifiers, some of which map to a SQL expression such as AND or OR. A qualifier is commonly compound—that is, a qualifier often consists of multiple qualifiers.

EOSortOrdering
: A sort ordering is often included in a fetch specification to specify that the fetch's result set should be sorted in a particular way.

EOEditingContext
: In Enterprise Objects, a fetch almost always takes place within an object workspace called an editing context.

Other objects are involved in a fetch specification, such as EODatabaseContext and EOAdaptorChannel, but you rarely need to interact with these objects programmatically.

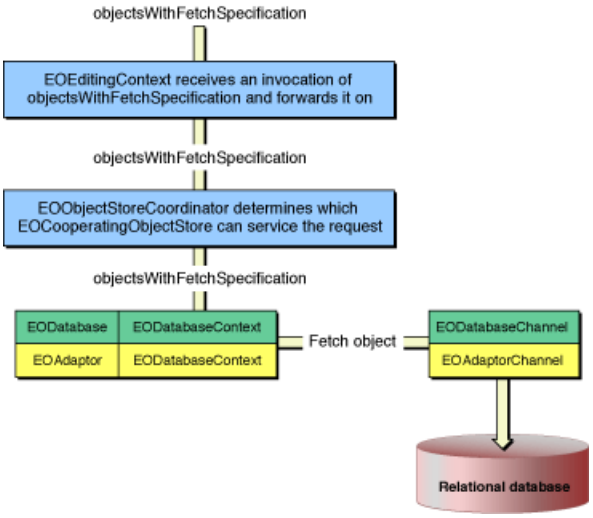
Flow of Data During a Fetch

A fetch begins with the construction of a fetch specification. You can create fetch specifications programmatically, but they are also created by various components within a WebObjects application such as display groups. You can also use EOModeler to build fetch specifications.

Once a fetch specification is created, the fetch must be initiated. Again, you commonly do this programmatically by invoking `objectsWithFetchSpecification` on an EOEditingContext, but it is also often done automatically by objects such as display groups.

"Figure 6-1" illustrates the flow of data during a fetch.

Figure 5-1 Flow of data during a fetch



Once a fetch is initiated, the following sequence occurs to retrieve data from a data source:

1. When `objectsWithFetchSpecification` is invoked on an EOEditingContext, that editing context forwards the invocation on to its parent object store. The parent object store again forwards the invocation on to its parent object store until the root object store is reached (the root object store is usually an instance of EOObjectStoreCoordinator).

2. The root object store (EOObjectStoreCoordinator) determines which of its EOCooperatingObjectStores should service the fetch specification. It forwards the `objectsWithFetchSpecification` invocation to the determined cooperating object store to ask it to retrieve data from the data source.

How does an EOObjectStoreCoordinator determine which of its EOCooperatingObjectStores should service a particular fetch specification? Remember that within an EOModelGroup, entity names must be unique. Also remember that fetch specifications are entity-centric—every fetch specification is specified on the basis of a particular entity. So an object store coordinator simply looks for the list of entities registered within its cooperating object stores to match an entity name to particular cooperating object store.

When an EOCooperatingObjectStore receives a request to fetch data from a data source, it invokes `objectsWithFetchSpecification` on its EODatabaseContext object to do the work. When a database context receives this fetch request, it fetches a number of rows from the database, transforms them into enterprise objects (in most cases), and registers them as needed with the EOEditingContext that initiated the chain of `objectsWithFetchSpecification` invocations.

A database context uses an EODatabaseChannel to do all this. That object in turn uses an EOAdaptorChannel object to communicate directly with data sources and model-level objects—EOEntity, EOAttribute, EORelationship—that are necessary to perform the fetch.

Within EODatabaseContext, fetching occurs in two major steps:

1. A database context uses a database channel to select the rows in the database for which objects are being fetched. It does this using the EODatabaseChannel method `selectObjectsWithFetchSpecification`, which takes as an argument the fetch specification that originated in the editing context.

2. The database channel fetches each enterprise object, one at a time, as the database context repeatedly invokes on it the method `fetchObject`. This method uses state built up in the first step to get data from the object, create an enterprise object instance if necessary, and register the new instance with the fetch's editing context. The database channel uses the entity name specified in the fetch specification to know which enterprise object class to instantiate for every fetched object.

When an EODatabaseChannel receives an invocation of `fetchObject` from an EODatabaseContext, the following sequence of events occurs:

1. The database channel uses an EOAdaptorChannel to retrieve a record for the requested entity. The record retrieved includes the record's primary key, class properties and client-side class properties, attributes used for locking, and any foreign keys used by the entity's relationships.

2. The database channel then assigns an EOGlobalID to the row by invoking `globalIDForRow`.

3. The database channel records a snapshot for the fetched row. A global ID may already have a recorded snapshot, but if this is not the case, the method `recordSnapshotForGlobalID` is invoked on EODatabase. However, if a snapshot is already recorded for the given global ID, the database context delegate method `databaseContextShouldUpdateCurrentSnapshot` is instead invoked. The default behavior does not update the already recorded snapshot with the new one, but you can change this by implementing the delegate method.

   At this point in the fetch, if the fetch specification is set to refresh refetched objects, an ObjectsChangedInStoreNotification is posted to invalidate (refault) any existing enterprise object instances that correspond to this global ID.

4. The database channel records whether the object was locked when it was selected. This would be the case only if you enable pessimistic locking (row-level locking) in your application.

5. The database channel then checks with the editing context in which the fetch originated to see whether a copy of the object already exists in that editing context. It uses the EOEditingContext method `objectForGlobalID` to do this.

6. If the editing context contains an enterprise object for the global ID and if that enterprise object is not a fault, the editing context returns the enterprise object. Otherwise, the enterprise object returns `null`.

7. If the editing context doesn't return an enterprise object for the global ID, the database channel invokes the EOEntityClassDescription method `createInstanceWithEditingContext`, which determines the object's class based on the fetch specification's entity and instantiates an object of that class.

8. The database channel invokes the method `recordObject` on the editing context to unique the newly created object. This is discussed in more detail in "Uniquing" (page 62).

9. If the editing context has a fault for the global ID, the fault is cleared and initialization proceeds just as if an empty enterprise object had been created and registered.

10. To initialize the object, the database channel invokes the method `initializeObject` on the editing context, which is passed down the object store hierarchy. If the editing context is nested, it passes the message to its parent editing context. If the parent editing context contains an object with a matching global ID, that object is used to initialize the object in the child editing context.

    Otherwise, the `initializeObject` invocation is forwarded down to the editing context's EODatabaseContext, which initializes the new instance from the appropriate snapshot and creates faults for its relationships. `initializeObject` in EODatabaseContext sets the values of the newly instantiated enterprise object's properties using `takeStoredValueForKey`. This is described in more detail in "Enterprise Object Initialization" (page 59).

11. The database channel invokes `awakeFromFetch` on the new enterprise object. Custom enterprise object classes can override this method to perform additional initialization after an object has been created from a database row and initialized with database values, as described in "Providing Initial Values" (page 40).

Enterprise Object Initialization

The following sequence of events occurs when an object is fetched from the database:

1. A database row is fetched as raw binary data.

2. The values retrieved from that row are converted from their database-specific types to instances of standard value classes. A sample mapping of this conversion appears in "Table 6-1". An application's EOModel specifies the mapping from external data types (database type) to internal data types (Java value type).

   Table 5-1 Database type to Java value type mapping

Database type	Java value type
`char`	String
`date`	NSTimestamp
`blob`	NSData
`int`	Integer

   `null` values in the database are mapped to an instance of NSKeyValueCoding.NullValue.

3. Once the data has been converted to objects, these objects are put in an NSDictionary. The elements of the dictionary correspond to columns in the database table: Their names are the names of the attributes they map to in the EOModel and their values are the values retrieved from the database.

   The dictionary provides a snapshot of the database row and is eventually used to initialize an enterprise object. This snapshot also participates in optimistic locking.

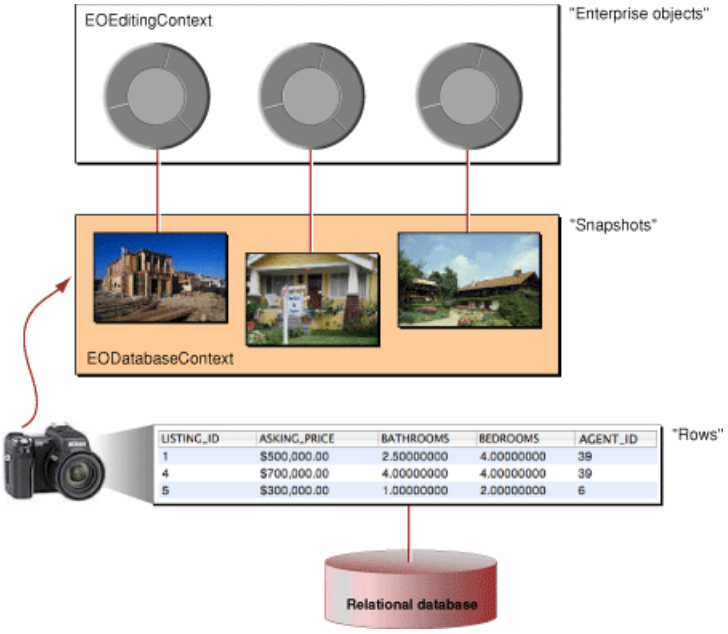
   The dictionary contains an entry for all of a row's columns, but an enterprise object initialized from the dictionary contains only the attributes that are defined as class properties or client-side class properties in the entity's EOModel.

4. A new enterprise object is instantiated by an EOEntityClassDescription object.

5. The enterprise object is initialized from a row snapshot. Only objects that are class properties or client-side class properties are included. Faults are created for any references to relationships defined in the EOModel.

"Figure 6-2" illustrates the relationship between database rows, database context snapshots, and enterprise object instances.

Figure 5-2 Relationship between rows, snapshots, and enterprise objects

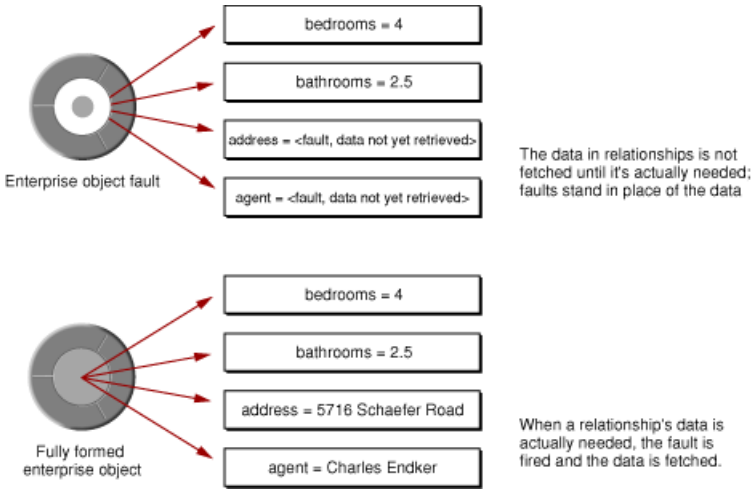


Faulting and Relationship Resolution

One of the most powerful and useful features of Enterprise Objects is that it automatically resolves the relationships defined in a model. It does this in part by delaying the actual retrieval of data—and delaying communicating with the database—until the data is needed, a feature of Enterprise Objects called faulting. Faulting happens in two phases: the creation of a placeholder object (a fault) for the data to be fetched, and fetching the data when it's needed (firing a fault).

When Enterprise Objects fetches an object, it examines the object's relationships as defined in the EOModel in which the object (entity) is defined. It then creates objects (faults) representing the destinations of the fetched object's relationships. For example, if you fetch a Listing object that has an `agent` relationship and an `address` relationship, faults are created for the destination of those relationships, which are an Agent object and an Address object. The Agent and Address objects are not fetched (their rows in the database are not accessed) until their data is actually needed. "Figure 6-3" illustrates this example.

Figure 5-3 Enterprise object as a fault and as fully formed



Fetching is resource-intensive and often recursive—fetching the destination object of one enterprise object may require fetching that destination object's destination objects, and so on until all of the interrelated rows in the database have been retrieved. To avoid this waste of time and resources, the destination objects are created as stand-ins, which are referred to as faults.

There are two kinds of faults: single-object faults for to-one relationships and array faults for to-many relationships. A single-object fault is an enterprise object instance that is associated with a particular editing context, class description, and global ID. However, the enterprise object's data hasn't yet been fetched from the database—you can think of a single-object fault as a shell of an enterprise object.

Array faults are instances of NSMutableArray and are triggered to fire their faults by any request for a member object or for the number of objects in the array (the number of objects in a to-many relationship can't be determined without actually fetching them all). More specifically, array faults may start out as deferred faults, which are very small and cheap and contain little information. They may then become NSMutableArrays, which have more information about their state and contents. If an object in the relationship is then directly accessed (if an element in the array is accessed), the array fault fully fires, filling the array with enterprise objects.

You can find more information about faults in these places:

- "Change Notification" (page 33)
- "Faulting" (page 33)
- "Deferred Faulting" (page 66)
- "Batch Faulting" (page 67)

Data Integrity Mechanisms

When you work with an object graph rather than directly with data in a database, you are working with copies of that data. While working with those copies, the integrity of the data within an object graph is crucial. Enterprise Objects uses several mechanisms to ensure the integrity of the data it fetches and manages in its object graphs. These mechanisms are:

- **Uniquing**—Enterprise Objects maintains the mapping of each enterprise object to its corresponding database row and uses this information to ensure that an object graph does not have multiple objects representing the same database row—that each enterprise object is unique within a given object graph.
- **Snapshotting**—When Enterprise Objects fetches data, it records the state of the fetched database row in a snapshot. The information in a snapshot is used to support Enterprise Object's optimistic locking mechanism. It is also used when changes are committed back to a data source to update only the attributes that were changed since the last fetch.
- **Faults**—The data in the objects at the destination of a fetched object's relationships doesn't need to be fetched until it's actually needed. Until that point, however, a reference to those destination objects may be necessary. These references that don't contain data are called faults.

These topics are discussed in more detail in the following sections.

Uniquing

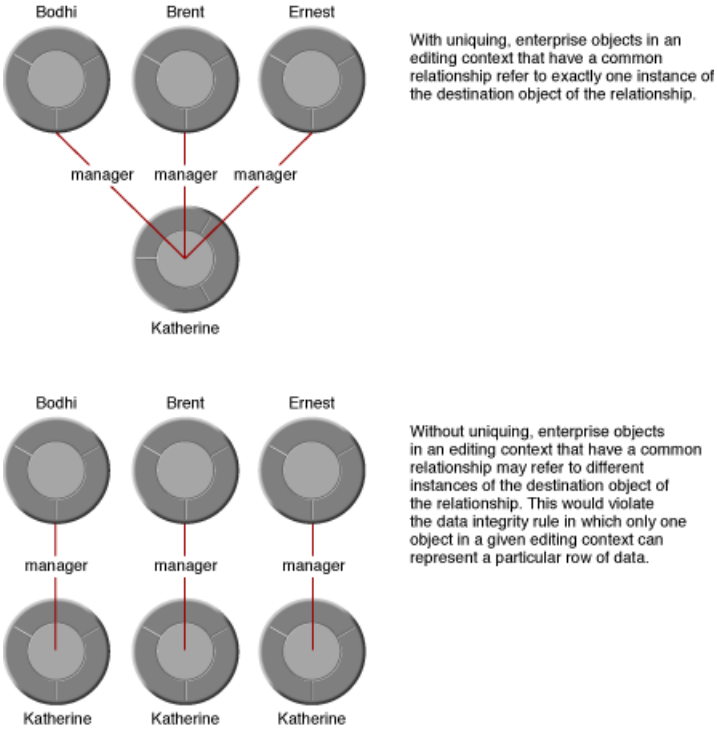
Uniquing is the mechanism in which Enterprise Objects ensures that a row in a database is associated with only one enterprise object in a given editing context in an application. The uniquing of enterprise objects limits memory usage and guarantees that the enterprise objects you work with represent the state of their associated database rows as they were last fetched into the object graph.

Without uniquing, a new enterprise object would be created every time you fetch its corresponding row, whether explicitly or through the resolution of relationships. For example, consider the case of a simple relationship between employees and a manager. Bodhi, Brent, and Ernest are represented by employee enterprise objects and Katherine is represented by a manager enterprise object that is the destination of the employee's `manager` relationship.

Without uniquing, when the database row representing Bodhi is fetched, an object representing Bodhi's manager, Katherine, is created to resolve his `manager` relationship. Then, when the database row representing Ernest is fetched, another object representing Katherine is created to resolve his `manager` relationship. If the row representing Katherine is itself explicitly fetched, yet another enterprise object representing Katherine is created. In this scenario, Katherine's row in the database can be altered by multiple enterprise object instances, resulting in objects that represent the same row but that may contain different and conflicting data.

With uniquing, however, in a given editing context, only one object representing Katherine is ever created. All the enterprise objects in a given editing context that refer to Katherine's enterprise object refer to the same instance—they have a single view of Katherine's data. So within a given editing context, there is no ambiguity with regard to the data in Katherine's enterprise object. These two scenarios are illustrated in "Figure 6-4".

Figure 5-4 Unique enterprise objects in an editing context



How does uniquing work? Objects are uniqued based on their global ID. A global ID (`com.webobjects.eocontrol.EOGlobalID`) is formed from an object's primary key and its associated entity. When a row is fetched to create an object in a particular editing context, its global ID is checked against the objects already in the editing context. If a match is found, the newly fetched object isn't added to the context.

A single enterprise object instance exists in one and only one EOEditingContext, but multiple copies of an object can exist in different editing contexts. In other words, the scope of object uniquing is a particular editing context.

Snapshotting

When an EODatabaseContext fetches objects from a database, a snapshot is recorded of the state of the fetched database row. A snapshot is a dictionary of a row's primary keys, class properties, foreign keys used in relationships that are class properties, and the attributes of an entity that participate in optimistic locking. To learn how snapshots participate in optimistic locking, see "Inside Optimistic Locking" (page 92).

You can imagine that an application that fetches hundreds of rows of data builds up a large cache of snapshots. Theoretically, if enough fetches are performed, an Enterprise Objects application can contain all the contents of a database in memory. Clearly, snapshots must be managed in order to prevent this situation.

So how are snapshots cleaned up? This is the responsibility of a mechanism called snapshot reference counting. This mechanism keeps track of the enterprise objects that are associated with a particular snapshot—enterprise objects that contain data from a particular snapshot. When there are no remaining enterprise object instances associated with a particular snapshot (which Enterprise Objects determines by maintaining a list of these references), that snapshot is released.

Snapshot reference counting is handled automatically by the framework, so you don't need to think about it.

Uniquing and Faulting

When a fault is constructed for a to-one relationship, the global ID for that fault is checked to see if the fault or its fully initialized enterprise object counterpart already exists in a given editing context. If so, that object is used to immediately resolve the relationship. This preserves the uniqueness requirement for enterprise objects by ensuring that there's never more than one global ID representing the same row in the database. Whether that global ID represents an actual enterprise object or a fault doesn't matter, since the data is fetched when it's needed.

If Enterprise Objects fetches data for an object that's already been created as a fault, that fault is fired and the enterprise object finishes initializing.

Ensuring Fresh Data

When developing Enterprise Objects applications, one of the most common challenges is providing users with the freshest possible data while maintaining reasonable application performance. In a multiuser database environment, there is a risk of update conflicts occurring in which multiple users access and attempt to change the same set of data simultaneously. Providing fresher data to users can help alleviate update conflicts. The philosophy of update conflicts is discussed in "Update Strategies" (page 91).

This section helps you understand when and how Enterprise Objects uses cached data and how you can influence the caching architecture.

When Does Database Fetching Occur?

The first thing to understand when dealing with the issue of data freshness is to understand when Enterprise Objects uses cached data and when it fetches data from a database. In most cases, if an editing context asks an enterprise object for its data, it receives cached data unless:

- the timestamp of the snapshots of enterprise objects are older than the editing context's timestamp
- the enterprise object has been invalidated
- the enterprise object is a fault (its data hasn't yet been fetched)

When multiple users access the same data source by sharing an application instance, they most often share data caches. *This means that one user's data query may not actually invoke a fetch from the data source if the data requested has already been fetched by another user and so exists in the cache.* A common design pattern for this situation is to provide each user with a separate snapshot cache, as discussed in

"Providing Separate Stacks" (page 52). But even in this scenario, a user's fetch request may not actually trigger a fetch from the database if the requested data has already been retrieved by an earlier fetch made by that user.

Distributed Change Notification?

Often, the issue of fresh data occurs when multiple users are using different application instances that all access the same data source. You want to ensure that changes one user makes are reflected in other user's applications. Unfortunately, Enterprise Objects does not include a distributed change notification mechanism to help you with this problem (though third parties have developed solutions).

But a distributed change notification mechanism isn't necessarily the solution. In any multiwriter database environment in which multiple users have concurrent write access to the same database, it is fundamentally impossible to guarantee fresh data. In this scenario, recovery is a better mechanism than prevention.

That is, instrumenting your applications to be resilient when update conflicts occur is a more reasonable approach than trying to prevent update conflicts altogether. Strategies for implementing resiliency are discussed in "Update Strategies" (page 91). That said, a mix of prevention and recovery is probably the best solution for most cases. The following sections discuss the built-in prevention mechanisms provided by the Enterprise Object frameworks.

Fetch Timestamp

Each editing context in an application includes a fetch timestamp that it uses to tell its parent object store that it wants cached data or fresh data from the database. An editing context prefers data that was fetched on or after an absolute time that is tracked by an editing context's **fetch timestamp**. (Ultimately, an editing context's parent object store decides when to perform database fetches. In the default case, an editing context's parent object store does honor its editing context's fetch timestamps, but this may not be the case for all object stores).

When enterprise objects are requested from an editing context, the editing context sends this request along with a fetch timestamp to its parent object store. If the requested enterprise objects have already been fetched, the parent object store finds the snapshots of those enterprise objects and compares their fetch timestamps with the fetch timestamp sent by the editing context that requested the objects.

If the timestamp of the snapshots from which the requested enterprise objects were formed is older than the editing context's fetch timestamp, the snapshots are considered stale and fresh values for those enterprise objects are requested from the database. Otherwise, cached enterprise object values are used (these cached values are in the database context's snapshots).

Timestamp Lag

An editing context's fetch timestamp is set to the time of a fetch minus the default timestamp lag. The default lag is sixty minutes so the default fetch timestamp on an application's editing contexts is one hour before a fetch occurred. So, if the timestamp of an enterprise object's snapshots in the database context are within an hour of the fetch timestamp of the object's editing context, a fetch returns the

cached data in those snapshots rather than refetching from the database. However, if the timestamp of the snapshots in the database context are older than an hour (or older than the editing context's fetch timestamp), the snapshots are discarded and data is refetched from the database.

A common design pattern is to set the default timestamp lag to a smaller number to encourage more refetching from the database. You can change the default timestamp lag for all the editing contexts in an application using the static method on EOEditingContext called `setDefaultFetchTimestampLag`. In some cases, you may want to explicitly set the fetch timestamp of a particular editing context to encourage refetching of its data. You can do this by invoking `setFetchTimestamp` on an editing context.

Nested editing contexts use the fetch timestamp of their parent, so applications that make heavy use of nested editing contexts may have to take additional measures to ensure fresh data.

Other Mechanisms to Ensure Freshness

Enterprise Objects provides other mechanisms to ensure the freshness of data in enterprise objects. By using the method `refreshesRefetchedObjects`, you force data values to be updated with fresh values from the data source when those objects are refetched. Using `refreshesRefetchedObjects` is described in "Refreshing Cached Data" (page 80).

Another, more severe mechanism to ensure fresh data is the method `invalidateObjectsWithGlobalIDs` on EOEditingContext, which flushes all the snapshots corresponding to the given global IDs and refetches those snapshot's rows. It has an even more severe counterpart, `invalidateAllObjects`. These are rather drastic measures that you should use cautiously. See "Discarding Cached Objects" (page 79) for more information.

As an alternative to both, you should instead consider using the method `setFetchTimestampLag` along with `refaultAllObjects` or `refreshAllObjects` on an editing context to update the data in enterprise objects in a given editing context.

Advanced Faulting

This section discusses the advanced faulting techniques available in Enterprise Objects.

Deferred Faulting

As described in "Faulting and Relationship Resolution" (page 60), Enterprise Objects uses faults to improve application performance. Fault creation is much faster than enterprise object creation and faults consume fewer resources than whole enterprise object instances. However, fault instantiation still takes time. To improve performance even further, Enterprise Objects uses **deferred faults**.

In an enterprise object class that can use deferred faulting, the object's relationships are initialized as deferred faults. For a particular relationship, a single deferred fault is shared between all instances of an enterprise object class. This sharing can significantly reduce the number of faults that need to be created and usually reduces the overhead of fault creation during a fetch.

For example, consider a Listing entity that has an `agent` relationship. Assuming the worst case in which each Listing enterprise object has a different Agent enterprise object, without deferred faulting, a fetch of twenty Listing objects results in the creation of twenty faults for the `agent` relationship—a fault for each Listing. With deferred faulting, only one fault is created—a deferred fault that is shared by all the Listing objects.

If you use EOGenericRecord subclasses for your custom enterprise object classes as described in "Which Enterprise Object Class?" (page 25), those classes automatically use deferred faulting. If your custom enterprise object classes instead inherit from EOCustomObject and you want to enable deferred faulting in them, override the method `usesDeferredFaultCreation` to return `true` in those classes, as "Listing 3-2" (page 31) does. In those classes, you must also invoke `willReadRelationship` before accessing a relationship that might be a deferred fault.

Note: If your custom classes inhert from EOCustomObject and those classes use entity inheritance, they *must* use deferred faulting. A relationship whose destination entity is an inherited entity must be deferred.

Batch Faulting

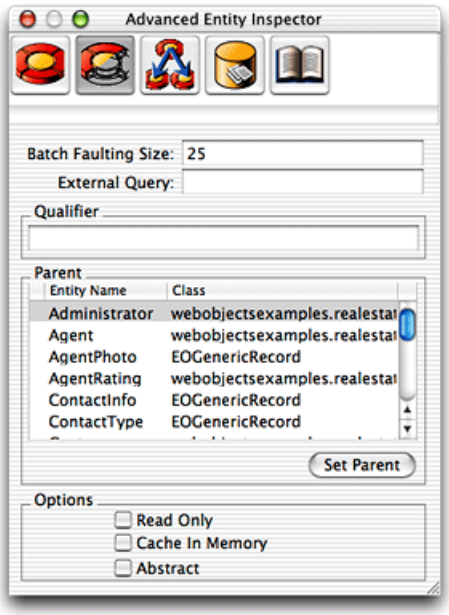
Another advanced faulting feature is **batch faulting**. When a fault is fired, its data is fetched from the database. However, firing one fault has no effect on other faults—firing one fault just fetches the object or objects for the one fault. By batching fault firings together, you can more efficiently use the round trip to the database that is necessary when a single fault is fired.

For example, given an array of Employee enterprise objects, you can fetch all of the objects that are the destination of their `department` relationship with one round trip to the server. Without batch faulting, a round trip to the database is made to resolve each Employee's `department` relationship.

There are a number of ways to implement batch faulting. You can configure batch faulting in three contexts: on entities, on relationships, and on relationships under certain circumstances.

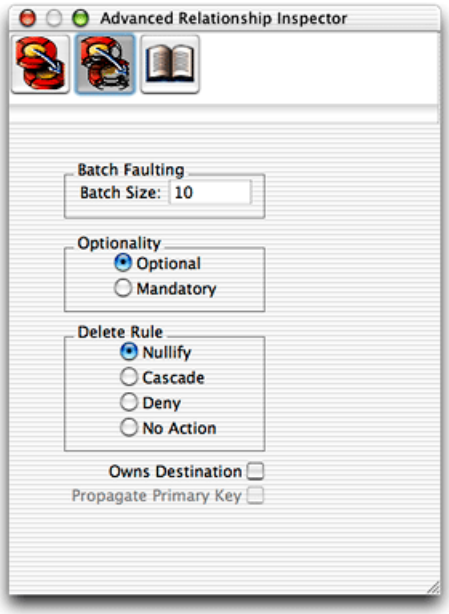
You configure batch faulting for an entity in EOModeler in an entity's advanced inspector, as shown in "Figure 6-5". The integer you specify in the Batch Faulting Size field specifies the number of faults to fire the first time a fault is fired for any relationship in that entity. You can set this size programmatically using the method in EOEntity called `setMaxNumberOfInstancesToBatchFetch`.

Figure 5-5 Configure batch faulting for an entity



You can also specify a batch faulting size for a particular relationship. The easiest and most common way to do this is in EOModeler using the advanced relationship inspector, which is shown in "Figure 6-6". The batch size specifies the number of faults to fire when the first fault in the relationship is fired.

Figure 5-6 Configure batch faulting for a relationship



Finally, you can take more precise control of batch faulting by explicitly batching together faults for particular objects. When you specify the batch size in EOModeler for all of an entity's relationships or for particular relationships, you don't actually control which faults are fired. The method `batchFetchRelationship` in EODatabaseContext allows you to batch fetch all of the faults in a particular relationship. The method `databaseContextShouldFetchArrayFault` in EODatabaseContext.Delegate allows you to turn batch faulting on and off arbitrarily. See the API reference for EODatabaseContext and EODatabaseContext.Delegate for more details.

Advanced Fetching

Enterprise Objects supports a number of advanced fetching techniques.

Prefetching

As described in "Faulting and Relationship Resolution" (page 60), when Enterprise Objects fetches an enterprise object, it creates faults for the object's relationships. Each time a fault is fired, a round trip is made to the database to retrieve the fault's data. You can batch together fault firing as described in "Batch Faulting" (page 67) to reduce the number of round trips to the database. However, you can go even further in reducing the number of round trips to the database by **prefetching** all the objects in a particular relationship. Prefetching allows you to anticipate that some of an enterprise object's relationships will be fetched and provides a mechanism to preload them; it provides a performance opportunity.

For example, consider a Listing entity that has an `agent` relationship. When you fetch twenty Listing objects, faults are created for each Listing's `agent` relationship. When the data in the `agent` relationship is accessed for a particular Listing object, a fault is fired to retrieve the relationship's data, which invokes a round trip to the database. Implementing batch faulting reduces the number of round trips but you can further reduce the number of round trips by simply prefetching *all* of the agent data in the database. With prefetching, when a fetch is performed for a particular entity, the objects in the relationships specified by the prefetching key paths (`agent` in this case) are immediately fetched.

You instrument your application for prefetching by configuring certain fetch specifications for prefetching. You can use the Prefetching pane of EOModeler's fetch specification builder to configure it for a particular fetch specification or you can invoke `setPrefetchingRelationshipKeyPaths` on a fetch specification, which takes an array of strings representing the relationships to prefetch.

There are a few guidelines to consider when using prefetching. First, if memory usage is an issue for your application rather than database performance, don't use prefetching as it consumes more memory. In fact, prefetching can consume an inordinate amount of memory depending on the size of the data set, so it's probably more appropriate to prefetch only those relationships that have a small number of destination objects.

Second, don't use prefetching on a fetch specification that uses a fetch limit. The prefetching hint ignores the fetch limit.

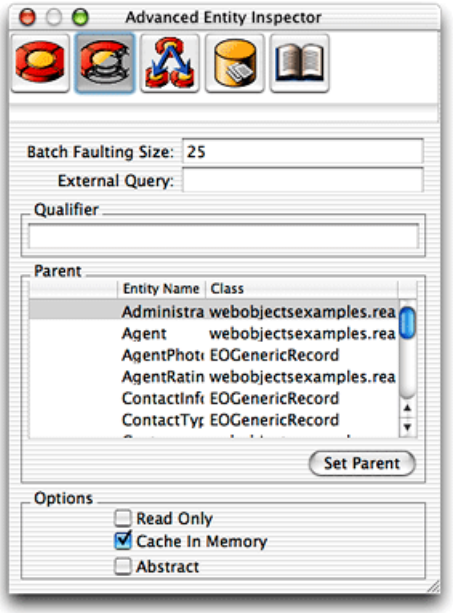
Third, don't use prefetching when performing multiple queries that return the same records. The performance benefits of prefetching are negated by the overhead of re-creating enterprise objects of the same rows of data multiple times.

Entity Caching

Many applications have read-only entities that contain static data such as lists of states and countries, building names, or department names. Since many users of an application use the data in these entities, it makes sense to cache the data in memory to reduce the number of fetches to the database. In Enterprise Objects, you can cache an entire table in memory, thereby eliminating unnecessary fetches for the same static data by multiple users.

To enable entity caching for an entity, select the Cache In Memory option in EOModeler's advanced entity inspector for a particular entity, as shown in "Figure 6-7". You can enable this option programmatically using the method `setCachesObjects` on EOEntity.

Figure 5-7 Enable entity caching



When entity caching is enabled for a particular entity, the first fetch of that entity's table causes the whole table to be fetched into memory. Clearly, this option is appropriate only for tables with a small number of rows.

An entity's cache of objects is maintained by an EODatabaseContext. If you provide a separate access layer stack for each user as described in "Providing Separate Stacks" (page 52), each session has its own EODatabaseContext, so bear in mind that entity caching in this scenario may consume a lot of memory.

Raw Row Fetching

Fetch specifications provide an option to fetch raw rows. When you use raw row fetching, database rows that are fetched are not automatically transformed into enterprise object instances. There are a number of reasons why you'd want to specify raw row fetching for a particular fetch specification. These include:

- reducing memory usage when fetching large data sets
- improving application performance when fetching large data sets
- reducing the general overhead of an application instance

You can specify raw row fetching for a particular fetch specification either in EOModeler's fetch specification builder or by invoking the method `setFetchesRawRows` on a fetch specification. You can more closely control which rows are fetched as raw rows using the method `setRawRowKeyPaths` on a fetch specification.

When you fetch raw rows, you lose many of the benefits of using full-fledged enterprise object instances such as the object graph, change notifications, and so forth. But many of the cases in which you need to fetch raw rows involve fetching large data sets that don't need the benefits of the object graph, so this is an acceptable trade-off in light of the performance benefits of raw row fetching.

Plus, you can always instantiate an enterprise object of that row using the method `faultForRawRow` on an EOEditingContext.

Raw SQL Fetching

Although fetch specifications are the most common type of objects used to fetch data in Enterprise Objects applications, a lighter-weight mechanism is also provided that fetches raw rows based on an SQL expression you provide. This mechanism is provided as a method called `rawRowsForSQL` on the EOUtilities class. You pass to the method as arguments an editing context, a String representing the model that contains the entities on which to perform to the fetch, and a valid SQL expression. The results are returned as raw rows rather than as full-fledged enterprise objects.

Common Delegate Usage

Note: This section is intended for experts.

A number of control points are provided that let you customize fetch operations in Enterprise Objects applications. "Table 6-2" lists the delegate methods in EODatabaseContext that you can use to customize fetch operations.

You can set the delegate of EODatabaseContext by invoking the class method `EODatabaseContext.setDefaultDelegate`.

Table 5-2 EODatabaseContext delegate methods

Method	Description
`databaseContext-ShouldSelectObjects`	Invoked just before a `SELECT` operation occurs. Use this delegate method to return `false` to tell the adaptor channel to skip the `SELECT`. You might want to do this to issue your own custom SQL to the adaptor.

Method	Description
`databaseContext-ShouldUsePessimisticLock`	Use this delegate method to selectively turn off the locking of rows when you're using a pessimistic locking strategy.
`databaseContextDid-SelectObjects`	Invoked immediately after a `SELECT` operation occurs. You can use it to log diagnostic information or initialize internal state for subsequent fetches.
`databaseContext-ShouldFetchObjects`	Use this method to satisfy an editing context's fetch request from its snapshot cache.
`databaseContextDid-FetchObjects`	Invoked after an EODatabaseContext fetches objects. Use it to record in a local cache the results of a fetch.
`databaseContext-FailedToFetchObject`	Invoked when a to-one fault cannot find its data in the database. Use this method to immediately throw an exception.
`databaseContext-ShouldLockObjectWithGlobalID`	Invoked from `lockObjectWithGlobalID`. Use it to implement custom locking.
`databaseContext-ShouldRaiseException-ForLockFailure`	Use this method to suppress an exception that occurs when EODatabaseContext attempts to lock an object.
`databaseContext-ShouldUpdateCurrentSnapshot`	Invoked when an EODatabaseContext already has a snapshot for a row fetched from the database. Use it to compare the snapshots and possibly resolve conflicts.

Constructing Fetch Specifications

You commonly create and configure fetch specifications using EOModeler's fetch specification builder. This is described in the chapter "Working With Fetch Specifications" in *EOModeler User Guide*. However, you also commonly create and configure fetch specifications programmatically, as this section describes.

As discussed in "Objects Involved in Fetching" (page 56), a fetch specification includes an entity name, a qualifier (optional), and a sort ordering (optional). The trickiest part of building a fetch specification programmatically is building qualifiers. The code samples here assume that you're using the Real Estate model and database.

Qualifiers

There are many ways to programmatically create a qualifier. One of the most common ways is to provide a format string to a qualifier. A format string is a logical expression that specifies parameters for performing a comparison. In a format string, you specify a data attribute to compare and a value with which to compare it. Enterprise Objects supports comparisons of equality, greater than, less than, greater than or equal to, less than or equal to, not equal, like, and case–insensitive like.

A format string also includes a conversion character, which specifies the data type of the value in the comparison. “Table 6-3” lists the available conversion characters.

Table 5-3 Format string conversion characters

Conversion character	Expected value or result
%s	A String object, or the result of the method `toString`.
%d	An Integer object or something that can be converted to an Integer.
%f	A Double object or something that can be converted to a Double.
%@	An arbitrary object (like an EOEnterpriseObject). No conversions are performed.
%K	Similar to %@ except the argument is coerced into a String with the method `toString` and is treated as a key that can determine whether the resulting qualifier is an EOKeyValueQualifier or an EOKeyComparisonQualifier.
%%	Results in a literal % character.

The following are examples of qualifier strings:

- `agent.firstName caseInsensitiveLike %@`
- `bedrooms >= %d`
- `bathrooms <= %d AND bedrooms = %d`

The following sections provide code examples that teach you how to build format strings. Many types of format strings are possible; only a few are presented here. See the API reference for `com.webobjects.eocontrol.EOQualifier` for more information. The following sections provide concrete code examples that show you how to build different kinds of qualifiers, after a section introducing qualifiers.

Simple String Qualifier

The code below constructs a qualifier to find the listings associated with a particular agent, based on the agent’s last name.

```
EOQualifier.qualifierWithQualifierFormat("agent.lastName = %s", new
NSArray(new Object[] {"Basset", "Travers"}));
```

Simple Integer Qualifier

The code below constructs a qualifier to find the listings with at least 3 bedrooms.

```
EOQualifier.qualifierWithQualifierFormat("bedrooms >= %d", new NSArray(new
Object[] {3}));
```

Wildcard Qualifiers

The code below constructs a qualifier to find the Listings associated with Agents whose first names begins with the letter b.

```
EOQualifier.qualifierWithQualifierFormat("agent.lastName caseInsensitiveLike
%@", new           NSArray(new Object[] {"B*"}));
```

Compound Qualifiers

Providing a fetch specification with a single qualifier often doesn't provide the precision you need for search criteria. Fortunately, you can easily form boolean combinations of qualifiers. These types of qualifiers, qualifiers composed of other qualifiers, are called compound qualifiers. Enterprise Objects supports AND, OR, and NOT boolean combinations of qualifiers.

This code builds a compound qualifier that combines the qualifiers constructed in "Simple String Qualifier" (page 73) and in "Wildcard Qualifiers" (page 74):

```
EOQualifier compoundQualifier = new EOAndQualifier(new NSArray(new Object[]
     {EOQualifier.qualifierWithQualifierFormat("agent.lastName = %s", new
NSArray(new
       Object[] {"Basset", "Travers"})),
     EOQualifier.qualifierWithQualifierFormat("agent.lastName caseInsensitiveLike
 %s",
        new NSArray(new Object[] {"B*"}))})));
```

This creates a qualifier that searches for the criteria specified in both qualifiers.

Filtering Fetch Results in Memory

A common task related to fetching data is filtering fetch results in memory. Given an array of objects to filter and a qualifier that specifies how to filter them, you can use static methods on EOQualifier to sort enterprise objects in memory. The qualifier you use to filter objects in memory is the same type of qualifier you use when fetching data. The method invocation to filter fetch results in memory is:

```
EOQualifier.filteredArrayWithQualifier(objects, qualifier);
```

If you write custom EOQualifier subclasses, they must implement the EOQualifierEvaluation interface if you want them to participate in in-memory filtering.

Sorting Fetch Results in Memory

Another common task related to fetching data is sorting fetch results in memory. Given an array of objects to filter and a sort ordering that specifies how to sort the array, you can use static methods in EOSortOrdering to sort enterprise objects in memory. The qualifier you use to sort objects in memory is the same type of qualifier you use when fetching data.

For example, to sort an array of enterprise objects in ascending order based on a `sellingPrice` property, you can use the code in "Listing 6-1".

Listing 5-1 Sort fetch results in memory

```
NSArray sortedObjects =
EOSortOrdering.sortedArrayUsingKeyOrderArray(objectsToSort, new
 NSArray(new Object[] {EOSortOrdering.sortOrderingWithKey("sellingPrice",
 EOSortOrdering.CompareAscending)}));
```

"Listing 6-1" illustrates the four objects required for sorting: an array of enterprise objects to sort (`objectsToSort`), a String representing the property of the enterprise object to sort on (`sellingPrice`), an NSSelector object representing how to sort the array (`EOSortOrdering.CompareAscending`), and an EOSortOrdering object that is made up of the `sellingPrice` String and the `EOSortOrdering.CompareAscending` NSSelector. You then invoke `EOSortOrdering.sortedArrayUsingKeyOrderArray` and pass in the array of objects to sort and the EOSortOrdering object. That method returns an array that is sorted with the specified criteria.

Accessing Database Keys

One of the great benefits of the Enterprise Objects frameworks is that they insulate you from the complexities of relational databases. It does this in part by managing things like primary and foreign keys for you, automatically.

However, while developing applications, you may need to access a particular entity's primary and foreign keys for debugging and other purposes. There are a number of facilities within the frameworks that help you do this. The recommended way is to use a fetch specification to fetch certain rows of data and set that fetch specification to fetch raw rows rather than enterprise objects. The results of the fetch return an array of dictionaries; each dictionary represents one row of data and includes keys for all of an entity's attributes, whether or not they are class properties.

For example, consider the Listing entity in the Real Estate model. Perhaps you need to determine the primary key (the `listingID` attribute) of all the listings in the database and the foreign key for the agent who is responsible for each listing (the `agentID` attribute). Neither of these properties are class properties in the Listing entity, so invoking the key-value coding method `valueForKey("listingID")` or `valueForKey("agentID")` on a Listing enterprise object results in an exception. However, if you construct a fetch specification and set it to fetch raw rows, you can easily retrieve the values for each of these keys.

The code in "Listing 6-2" provides an example of fetching all Listing records as raw rows and extracting the `listingID` primary key and the `agentID` foreign key.

Listing 5-2 Fetch Listing records as raw rows

```
EOFetchSpecification fs = new EOFetchSpecification("Listing", null, null);
    fs.setFetchesRawRows(true);
    NSArray rawRows = editingContext.objectsWithFetchSpecification(fs,
 editingContext);

    java.util.Enumeration enum = rawRows.objectEnumerator();
    while (enum.hasMoreElements()) {
      NSDictionary row = (NSDictionary)enum.nextElement();
```

```
    if (row != null) {
      NSLog.out.appendln("\nlistingID pk: " + row.valueForKey("listingID"));
      NSLog.out.appendln("\nagentID fk: " + row.valueForKey("agentID"));
    }
  }
```

Working With the Object Graph

One of the primary benefits of using Enterprise Objects is that it insulates you from database details. Once you define the mapping between the database and enterprise objects in a model, you generally do not need to think about issues such as key propagation and generation, how object deletions are handled, how operations in the object graph are reflected in the database, and so forth. The frameworks let you focus on object-oriented programming rather than on database programming.

You tell the frameworks what you want to do by performing operations on the object graph. So, it's important that you understand a bit about how the object graph works and how to manage it. The object graph represents an internally consistent view of an application's data. The operations you perform on it can affect its consistency, so knowing how to correctly work with it helps you build better applications that take full advantage of the frameworks.

This chapter discusses how to work with and manage the object graph. It tells you what you need to do in an application after data is fetched and before it is saved.

It is divided into the following sections:

- "Objects Involved in Managing the Object Graph" (page 78) introduces the objects involved in graph management.
- "Getting Information About Changed Objects" (page 78) describes how to get information about objects that have been changed in the object graph.
- "Undoing Changes" (page 78) discusses how to undo changes to enterprise objects.
- "Discarding Changes" (page 79) discusses how to discard changes to enterprise objects while maintaining their cached snapshots.
- "Discarding Cached Objects" (page 79) discusses how to discard changes to enterprise objects and how to discard their cached data.
- "Refreshing Cached Data" (page 80) discusses how to refresh the data in enterprise objects from fresh data in a data store.
- "Working With Objects in Multiple Editing Contexts" (page 81) describes how to work with the same object in multiple editing contexts.
- "Memory Management" (page 81) discusses memory management issues in the object graph.

Objects Involved in Managing the Object Graph

There are many objects involved in managing the object graph in an Enterprise Objects application, but you most commonly work with these two:

EOEditingContext
: An editing context is the most important object in the object graph. Enterprise object instances always live within an editing context. Most of the operations you perform on the object graph you perform from the perspective of an editing context.

NSUndoManager
: An undo manager provides a flexible mechanism to undo and redo changes to enterprise objects in the object graph. It provides a convenience both to your application's users and to you the developer. You usually don't need to think about the undo manager, but you may encounter resource issues in your applications that require you to customize its behavior.

Getting Information About Changed Objects

An EOEditingContext maintains information about three different kinds of changes to objects in its object graph: insertions, deletions, and updates. After changes have been made to enterprise objects in an editing context and before the changes are committed to the database, you can determine which objects have changed using the methods `insertedObjects`, `deletedObjects`, and `updatedObjects`. These methods return an array of the objects that have been inserted, deleted, and updated, respectively. Before invoking these methods, you should check to see if any of the objects in an editing context have changed by invoking the method `hasChanges` on the editing context.

Undoing Changes

Once you get information about the changes made to enterprise objects in an editing context as described in "Getting Information About Changed Objects" (page 78), you may want to undo those changes. By invoking `undo` on an editing context, the latest set of changes to enterprise objects in that editing context are reversed. You can invoke the method `redo` to reverse the latest undo operation. You can invoke the method `revert` to discard all changes in an editing context, including all insertions and deletions. This restores changes to updated objects to their last committed values (their values in the database). An even more severe undo operation is possible using the `reset` method, which clears all enterprise objects in the editing context.

You may wonder what the scope of an undo operation is. When you invoke `undo` on an editing context, how much is undone? The scope of an undo in Enterprise Objects is event-based. In a WebObjects application, a single request in the request-response loop constitutes an event. In a Java Client application, a single user event such as a mouse click or menu choice that invokes an operation constitutes an event. The change of a single value in an enterprise object doesn't constitute an event, unless the change is triggered by an atomic event of either of these types. This undo scope ensures that anytime `undo` is invoked, the object graph returns to a stable state.

The undo support in EOEditingContext is arbitrarily deep for editing contexts that you create programmatically; you can undo an object repeatedly until you restore it to the state it was in when it was first created or fetched into its editing context. Starting with WebObjects 5.2, the default editing context in WOSession objects limits the number of undo operations to 10.

Even after an editing context's changes are committed to a database, you can undo a change. To support this feature, the undo manager (an instance of NSUndoManager) can consume a lot of memory.

For example, whenever an object is removed from a relationship, the corresponding editing context creates a snapshot of the modified source object. The snapshot, which references the removed object, is referenced by both the editing context and the undo manager. The editing context releases the reference to the snapshot when the change is saved, but the undo manager doesn't. The undo manager continues holding the snapshot so it can undo the deletion upon request.

The usage patterns of your applications may cause the undo manager to consume an unreasonable amount of memory. A common design pattern to limit an undo manager's memory use is to clear an undo manager's stack when an editing context saves. The most severe consequence of this it that it prevents undo beyond the point of saving. To do this, invoke the method `removeAllActions` on an editing context's undo manager after invoking `saveChanges`. You can also set maximum levels of undo using the method `setLevelsOfUndo`. Or, if an application or editing context doesn't require an undo manager, you can set an editing context's undo manager to `null` with the method `setUndoManager`.

Discarding Changes

There are a few ways to discard changes to enterprise objects in an editing context. They include:

- Invoking `undo` on an editing context, which reverses the latest uncommitted changes to enterprise objects in the editing context.
- Invoking `revert` on an editing context, which clears the editing context's undo stack, discards all inserted objects, restores all deleted objects, and refaults all updated enterprise objects (which often results in immediately updating their data).
- Invoking `reset` on an editing context, which clears the editing context's undo stack and removes all enterprise objects from the editing context. (It does not cause an editing context to refault enterprise objects but instead to forget them altogether.)

There are other ways to discard changes to enterprise objects, including those that refetch from the database. They are described in "Discarding Cached Objects" (page 79).

Discarding Cached Objects

As described in "Undoing Changes" (page 78), you can use the methods `undo` and `revert` on an editing context to discard changes made to the enterprise objects registered with that editing context. Invocation of those methods results in the refreshing of the data in the editing context's enterprise objects from cached data (data that is cached either in the undo manager or in the access layer). This is often the behavior you want, but if you're concerned about the freshness of data in your application

as discussed in "Ensuring Fresh Data" (page 64), you may want to force a refetch from the data source. The methods `invalidateAllObjects` and `invalidateObjectsWithGlobalIDs` on EOEditingContext provide this behavior.

The effect of these two methods depends on their use. If you invoke `invalidateAllObjects` on an EOEditingContext directly, it invokes `invalidateObjectsWithGlobalIDs` on its parent object store, passing as the *`globalIDs`* argument all the global IDs registered in the editing context. This message is propagated down the object store hierarchy to the EOObjectStoreCoordinator.

The object store coordinator determines which of its cooperating object stores can service the request and propagates the message appropriately. The cooperating object store that receives the `invalidateAllObjects` invocation discards the row-level snapshots for the specified global IDs and sends the notification `ObjectsChangedInStoreNotification`, which results in refaulting all the enterprise objects in the object graph. The next time those objects are accessed, their data is refetched from the database.

Invoking `invalidateAllObjects` on an editing context affects the enterprise objects in that editing context and all the enterprise objects in all the other editing contexts that share an object store coordinator, too.

If you invoke `invalidateAllObjects` directly on the EOObjectStoreCoordinator, it invokes `invalidateAllObjects` on all of its cooperating object stores, which then discard all of the row-level snapshots in the application and refault every enterprise object in all of the application's editing contexts. So, if your application uses the default configuration of the Enterprise Objects core stack as described in "How It Stacks Up" (page 49), all the enterprise objects in all the application's sessions are affected.

Be careful when using `invalidateAllObjects` and `invalidateObjectsWithGlobalIDs`: They are rather heavy-weight and can have severe consequences. A more controlled way of refreshing data in an application is described in "Refreshing Cached Data" (page 80).

Refreshing Cached Data

As discussed throughout this book, Enterprise Objects is essentially a big cache of objects. There are mechanisms within Enterprise Objects to preserve the integrity of its objects and mechanisms to improve performance when dealing with potentially large caches of objects. When you fetch data in an Enterprise Objects application, Enterprise Objects prefers to use data that's already been fetched (cached data) rather than perform a round trip to the database. The degree to which Enterprise Objects prefers cached data is controlled by many things including an editing context's fetch timestamp. This is discussed in "Ensuring Fresh Data" (page 64).

In the context of managing the object graph, there are scenarios in which you want to refresh the data in the object graph. You can refresh an entire object graph using the methods `invalidateAllObjects` and `invalidateObjectsWithGlobalIDs` on EOEditingContext as described in "Discarding Cached Objects" (page 79), but you often want to refresh the data in a specific object or objects.

By default, when you refetch data, Enterprise Objects does not update the data in enterprise object instances with the refetched data from the database. This is the default behavior that helps maintain an internally consistent view of the application's data. However, this is not always the behavior you want, especially if you are more concerned with ensuring the freshness of data than anything else (which is especially true in read-only applications).

To override the default behavior, invoke the method `setRefreshesRefetchedObjects` on an EOFetchSpecification. Then, when you refetch data, the data in enterprise object instances is refreshed with the refreshed data from the database. (You can refetch the data for a particular enterprise object by invoking on it the method `refreshObject`. If you need to refresh a set of enterprise objects, use a fetch specification.)

By default `setRefreshesRefetchedObjects` refreshes only the objects you are refetching. For example, if you refetch Employee objects, you don't also refetch the Employees' departments. However, by invoking `setPrefetchingRelationshipKeyPaths` on a fetch specification, the refetch is also propagated for all of the fetched object's relationships that you specify in that invocation.

Working With Objects in Multiple Editing Contexts

It's common to need access to an enterprise object across multiple editing contexts. A poor approach to this would be for the first editing context to just get a reference to the enterprise object in the second editing context and modify it directly. But this would violate the rule of keeping each editing context's object graph internally consistent. Instead of modifying the second editing context's object, the first editing context needs its own copy of the object. It can then modify its copy without affecting the original. When that editing context saves changes, the changes are propagated to the original object.

There are two methods you can use to get a copy of an enterprise object in an editing context from an original object in another editing context. The first method is `EOEditingContext.faultForGlobalID(EOGlobalID`*`gid`*`, EOEditingContext`*`editingContext`*`)`, which is invoked on the editing context in which you want the copy of the object to exist. It requires the global ID of the original object, which you can retrieve by invoking `globalIDForObject(EOEnterpriseObject`*`enterpriseObject`*`)` on an editing context in an application.

The other method is `EOUtilities.localInstanceOfObject(EOEditingContext`*`editingContext`*`, EOEnterpriseObject`*`enterpriseObject`*`)`, which returns a copy of the enterprise object specified by the *`enterpriseObject`* argument. You then must manually insert the enterprise object that is returned into an editing context.

Memory Management

You usually do not need to worry about managing memory in an Enterprise Objects application, but this section may be interesting for advanced users.

EOEditingContexts use weak references to the EOEnterpriseObjects registered with them. EOEnterpriseObjects hold a strong reference to the EOEditingContext in which they are registered. These types of references prevent an EOEditingContext from being garbage collected while an EOEnterpriseObject still requires it. There are several exceptions:

- EOEditingContexts hold all inserted, deleted, or updated objects by strong references. These strong references are cleared by the EOEditingContext methods `saveChanges`, `revert`, `invalidateAllObjects`, and `reset`. They may also be cleared by the EOEditingContext methods `invalidateObjectsWithGlobalIDs`, `refaultObject`, `refreshObject`, `undo`, and `redo`, depending on whether the changed state is or is not forcefully discarded.

- EOEnterpriseObjects registered with an EOSharedEditingContext are always held by strong references.
- You can force an EOEditingContext to hold strong references to all of its EOEnterpriseObjects by invoking either `setInstancesRetainRegisteredObjects` or `setRetainsRegisteredObjects` on an editing context in which no enterprise objects are registered.

Saving Data

This chapter discusses the mechanics of saving data within the Enterprise Objects frameworks.

It is divided into the following sections:

- "Objects Involved in Saving" (page 83) introduces the objects involved in saving data in an Enterprise Objects application.
- "Flow of Data During a Save" (page 84) discusses the flow of data during a save.
- "Phases of Saving" (page 84) discusses the mechanisms Enterprise Objects uses to ensure data integrity while saving.
- "Key Generation" (page 86) discusses how key generation works in Enterprise Objects.
- "Common Delegate Usage" (page 88) discusses common delegates used in the save process.
- "Generating Custom Primary Keys" (page 88) teaches you how to generate custom primary keys.
- "Using Compound Primary Keys" (page 90) discusses what you need to do to use compound primary keys.

Objects Involved in Saving

There are many objects involved in saving data in an Enterprise Objects application. The objects you'll most commonly work with are introduced here.

EOEditingContext
: In Enterprise Objects, saves to the database most often originate with a `saveChanges` invocation on an editing context.

EOObjectStoreCoordinator
: This object is responsible for figuring out which data source corresponds to the changes in a particular editing context. A given editing context can contain enterprise objects that are constituted from various data sources, so the object store coordinator manages the logistics of identifying these data sources when saving.

EOCooperatingObjectStore
: Each cooperating object store in an application is responsible for transmitting to its data source the changes handed to it by the object store coordinator. A cooperating object store (which is most often an instance of EODatabaseContext) is an abstract representation of a data source.

Other objects are involved in saving data, such as EODatabaseContext and EOAdaptorChannel, but you rarely need to interact with these objects programmatically.

Phases of Saving

The following phases are involved in pushing data back to a database:

- **Validation.** The first phase of committing data to a database is validating the data. A large part of your application's business logic involves validation—verifying that customers don't exceed their credit limits, that test scores fall within an allowable range, that phone numbers contain only numbers, and so on. In your enterprise object classes, you implement methods that perform validation at different points in an application, such as when users attempt to save, delete, update, or perform other operations on a record.

  By validating data in your enterprise object classes, you have an early opportunity to catch invalid data before it is committed to the database. Validation in enterprise object classes, however, is only a piece of the data integrity puzzle.

- **Referential integrity enforcement.** Another part of the data integrity puzzle is relationship-level referential integrity rules that specify the parameters of relationships between entities. Like validation rules, referential integrity rules often reflect your custom business logic. In the context of the Enterprise Objects stack, referential integrity enforcement occurs at a lower level in the access layer. You define referential integrity rules in an application's data model.

  Enterprise Objects supports the following referential integrity rules: ownership, optionality, delete rule, and propagate primary key.

- **Key generation.** When you use Enterprise Objects, you usually don't have to worry about database artifacts such as primary and foreign key values. Primary and foreign keys are rarely meaningful parts of a business model; rather they're used within a database to express relationships between tables. For example, the primary key of a LISTING table (LISTING_ID) doesn't have any meaning to users as they identify property listings by address.

  Most database application-development environments force you to worry about primary and foreign key generation. Part of the Enterprise Objects philosophy is that this is a task that the technology should take care of for you—your time as a developer is better spent focusing on business logic and building great applications, not on managing database key generation.

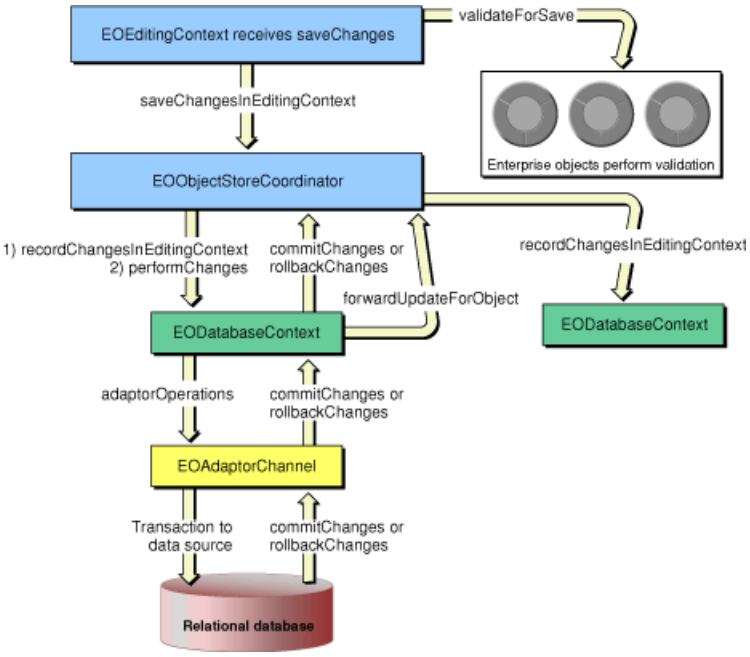
  Enterprise Objects manages primary and foreign key data for you. Your enterprise object classes don't have knowledge of primary and foreign keys and you don't have to worry about generating or propagating primary key values when adding records to a database.

Flow of Data During a Save

A save operation begins with the invocation of `saveChanges` on an editing context. If you programmatically create and manage editing contexts, you explicitly invoke this method. However, `saveChanges` is commonly invoked automatically by elements of an Enterprise Objects application such as display groups.

"Figure 8-1" provides a high-level overview of the save process.

Figure 7-1 Flow of data during a save



Once a save is initiated, the following sequence occurs to retrieve data from a data source:

1. When `saveChanges` is invoked on an editing context, that editing context invokes the method `editingContextWillSaveChanges` on its editors and delegates.

2. The editing context processes, propagates, and validates changes for deleting.

3. The editing context processes and validates changes for saving.

4. The editing context commits the changes made to its objects to its parent object store by invoking the method `saveChangesInEditingContext` on its parent. For nonnested editing contexts, the parent is typically an instance of EOObjectStoreCoordinator. When an object store coordinator receives this invocation, it guides its cooperating object stores through a multipass save protocol in which each cooperating store saves its own changes and forwards remaining changes to other cooperating stores.

5. After it receives the invocation of `saveChangesInEditingContext`, the object store coordinator invokes the method `prepareForSaveWithCoordinator` on each of its cooperating object stores. This informs each object store that a multipass save operation is beginning. When the object store is an EODatabaseContext (which is the most common case), it initiates the process of generating primary keys for any new objects in the editing context on which `saveChanges` was invoked.

6. The object store coordinator then invokes the method `recordChangesInEditingContext` on each of its cooperating object stores. This prompts each of those stores to examine the changed objects in the editing context, record any operations that need to be performed, and notify the coordinator of any changes that need to be forwarded to other cooperating stores.

For example, if in its `recordChangesInEditingContext` method, a cooperating object store notices the removal of an object from an owning relationship but that object belongs to another cooperating store, it informs the other store by invoking `forwardUpdateForObject` on the object store coordinator.

7. The object store coordinator invokes the method `performChanges` on each of its cooperating object stores. This tells the stores to transmit their changes to their underlying databases. When the cooperating store is an EODatabaseContext, it responds to this invocation by constructing a list of adaptor operations based on the list of database operations that were determined in the last step. It then forwards this list of adaptor operations on to an available EOAdaptorChannel to be executed.

8. If `performChanges` fails for any of the cooperating object stores, the method `rollbackChanges` is invoked on all the cooperating object stores in that access-layer stack.

9. If `performChanges` succeeds for all of the cooperating object stores, the method `commitChanges` is invoked on all of them, which tells the adaptor to commit its changes.

10. If `commitChanges` fails for a particular cooperating store, the method `rollbackChanges` is invoked on that store and on all subsequent stores. However, the stores that have already committed their changes do not roll back. *In other words, the object store coordinator doesn't perform the two-phase commit necessary to guarantee consistent distributed updates.*

11. If the save operation is successful, the database contexts of the editing context from which the save operation originated update their snapshots.

12. Finally, if the save operation is successful, the editing context posts the notification EditingContextDidSaveChangesNotification.

Key Generation

One of the best features of Enterprise Objects is its ability to manage primary and foreign keys for you. It does this in part by providing a mechanism to automatically generate primary keys without requiring you to write any code. To take advantage of Enterprise Object's automatic primary-key generation, an entity's primary key must satisfy these guidelines:

- It must be a simple integer; the adaptor cannot generate keys for noninteger data types.
- It cannot be compound.

Automatic primary-key generation in Enterprise Objects works differently depending on the database, but you usually don't need to be concerned with the details. With some databases such as OpenBase and MySQL, Enterprise Objects maintains a table in the database (`EO_PK_TABLE`) to help it maintain a linear sequence of nonrepeating primary keys. With an Oracle database, Enterprise Objects doesn't need this table and instead uses Oracle sequences to generate primary keys.

The advantage of using Enterprise Object's automatic primary-key generation is that it is easy to use and makes your life as a developer much simpler. However, there are some disadvantages and limitations, including these:

- It is database-specific.

- It requires a roundtrip to the database; however, key generation is batched, so even when inserting hundreds of objects at once, key generation is inexpensive.
- It cannot generate compound primary keys.
- It doesn't work if the database includes rows that are inserted by non-Enterprise Objects applications.

Fortunately, Enterprise Objects provides a number of hooks so you can provide primary key values. If an enterprise object or its snapshot already has a value for its primary key (which is the case for enterprise objects formed from preexisting data or for enterprise objects for which you set the primary key value explicitly in the enterprise object class), none of the key generation mechanisms are invoked. However, if this is not the case, the key generation mechanisms are given the opportunity to provide keys in this order:

1. If you implement a delegate for EODatabaseContext and the delegate implements the method `databaseContextNewPrimaryKey` and if that method returns a value other than `null`, the keys it generates are used. An implementation of this feature is discussed in "Generating Custom Primary Keys" (page 88).

2. If you provide to an entity the name of a stored procedure to use during Get PK operations (the operation type `EOEntity.NextPrimaryKeyProcedureOperation`), that stored procedure is invoked to generate a primary key. See the chapter "Working With Entities" in *EOModeler User Guide* to learn about automatic stored procedure invocation.

3. If an entity's primary key's data type is binary and it conforms to `EOTemporaryGlobalID.UniqueBinaryKeyLength`, Enterprise Objects automatically generates a primary key. This is discussed in "Using Binary Keys" (page 89).

4. The adaptor automatically generates a primary key for an object whose entity has a noncompound, simple integer primary key.

If the first mechanism fails to provide a primary key, the second mechanism is used. This continues through the last mechanism; if it fails to provide a primary key, an exception is thrown.

In the case of an entity that is the destination of a relationship that propagates primary key, the destination entity is assigned a primary key immediately after its source entity is assigned a key. So for destination entities in a relationship that propagates primary key, the key generation mechanisms in the list above are not invoked for that entity (though they may be invoked for the source entity).

See "Generating Custom Primary Keys" (page 88) and "Using Compound Primary Keys" (page 90) for sample implementations of custom key generation.

Common Delegate Usage

Note: This section is intended for experts.

A number of control points are provided that let you customize save operations in Enterprise Objects applications. "Table 8-1" lists the delegate methods in EOEditingContext that you can use to customize save operations in the control layer, and "Table 8-2" lists the delegate methods in EODatabaseContext that you can use to customize save operations in the access layer.

You can set the delegate for each object by invoking the class method `setDefaultDelegate`.

Table 7-1 EOEditingContext delegate methods

Method	Description
`editingContextShould-ValidateChanges`	This method is invoked when an EOEditingContext receives a `saveChanges` invocation. If this delegate method returns `false`, changes are saved without first performing validation. You can use this method to provide your own validation mechanism.
`editingContextWill-SaveChanges`	This method is invoked when an EOEditingContext receives a `saveChanges` invocation. You can use this delegate method to perform custom validation before saving.

Table 7-2 EODatabaseContext delegate methods

Method	Description
`databaseContext-WillOrderAdaptor-Operations`	This method is invoked when an EODatabaseContext receives a `performChanges` invocation. You can use this delegate method to construct custom adaptor operations, such as transforming a delete operation into an update operation or into a stored procedure operation.
`databaseContext-WillPerformAdaptor-Operations`	This method is invoked within the `performChanges` method in EODatabaseContext. You can use this delegate method to provide custom ordering of adaptor operations to, for example, avoid violating any referential integrity constraints that are enforced by the database.

Generating Custom Primary Keys

There are many reasons why you may want to or need to generate custom primary keys in an Enterprise Objects application. They are discussed in "Key Generation" (page 86). The following sections provide sample implementations of two of the custom-key generation mechanisms, using the EODatabaseContext delegate and using the automatic key generation for binary primary keys.

Using a Delegate

The EODatabaseContext class provides a delegate in which you can generate custom primary keys. This is especially useful if an entity has a compound primary key, but it is also useful if your application can't use Enterprise Object's automatic key generation for simple integer primary keys.

You can set the delegate by invoking `EODatabaseContext.setDefaultDelegate(this)` in the class in which you implement the delegate method. An implementation of `databaseContextNewPrimaryKey` is shown in "Listing 8-1".

Listing 7-1 `databaseContextNewPrimaryKey` implementation

```
public NSDictionary databaseContextNewPrimaryKey(EODatabaseContext dbCtxt, Object
 object, EOEntity entity) {
        NSArray rawRows = EOUtilities.rawRowsForSQL(new EOEditingContext(),
"PKTester",          "SELECT MAX(FOO_PK) FROM FOO");
        NSDictionary rowWithPK = (NSDictionary)rawRows.objectAtIndex(0);
        Object maxPK = rowWithPK.objectForKey("FOO_PK");
        int pk = (new Integer(maxPK.toString())).intValue();

        NSMutableDictionary newPrimaryKey = new NSMutableDictionary();
        NSArray entityPrimaryKeys = entity.primaryKeyAttributeNames();

       Enumeration primaryKeyEnumerator = entityPrimaryKeys.objectEnumerator();
        while (primaryKeyEnumerator.hasMoreElements()) {
            String pkName = (String)primaryKeyEnumerator.nextElement();
            newPrimaryKey.takeValueForKey(new Integer(++pk), pkName);
        }
        return newPrimaryKey;
 }
```

The method in "Listing 8-1" returns a dictionary of key-value pairs; the keys are the names of the entity's primary key attributes (which are returned by the method `EOEntity.primaryKeyAttributeNames`) and the values are the values you generate in the method for each of those attributes.

In "Listing 8-1", a SQL expression is sent to the database to determine the highest value for the entity's primary key, `FOO_PK`. That value is stored in the `pk` variable, which is incremented in the `while` loop to generate a unique primary key.

Using Binary Keys

Enterprise Objects provides another useful mechanism to generate primary keys. It generates a binary primary key if a primary key meets these criteria:

- Its external data type is a "raw bytes" data type, such as Oracle `RAW` and OpenBase `binary`.
- Its internal data type is NSData.
- Its internal data type width is 24 bytes.

The binary primary key generated in this case is a globally unique key based on an enterprise object's EOTemporaryGlobalID. Generating primary keys this way has these advantages:

- It doesn't require a round trip to the database.
- It is not database dependent.
- The generated keys are globally unique.

Binary primary keys have the following characteristics, which can be considered disadvantages:

- The generated keys are quite large.
- Comparing keys requires more computing resources.
- Binary keys can't be part of a compound primary key.

Using Compound Primary Keys

Enterprise Objects supports tables that have compound primary keys. In EOModeler, you identify the attributes that are a part of the compound primary key by making those attributes primary keys (so the key icon is present in an attribute's row). All of Enterprise Objects internal mechanisms that rely on primary keys (such as global ID creation) work just as well with compound primary keys as with simple primary keys. However, you cannot use a compound primary key and Enterprise Object's automatic primary-key generation. You have to provide a primary key another way.

The easiest way to generate primary keys for compound primary keys is with the delegate method `databaseContextNewPrimaryKey`. The implementation of this method in "Generating Custom Primary Keys" (page 88) supports a compound primary key. It retrieves all of an entity's primary keys and generates a unique value for each one.

Note that methods in Enterprise Objects that return primary key values usually return an NSDictionary object. This is to support the case in which a primary key is compound. Each key of the dictionary is the name of an attribute that is part of the compound key.

Update Strategies

The applications you build with Enterprise Objects will likely be used simultaneously by many different users. These users usually can access the same set of data and probably have the rights to update all or part of that data set. But what happens when multiple users try to update the same data simultaneously? How do you prevent users from overwriting other user's work? This realm of database application development is referred to as **update conflicts**.

This chapter discusses the update strategies available in Enterprise Objects. It is divided into the following sections:

Choosing a Strategy

An **update strategy** determines how update conflicts should be handled. The most common update strategy in database application development is **locking**. A locking strategy represents a preventative approach to managing update conflicts. There are a few different locking strategies, the most common of which is pessimistic locking. Using the pessimistic locking approach, a row of data in the database is locked when it is fetched to prevent other users from accessing that row of data.

It is generally a poor approach in three-tier database applications. Of the many reasons that make it a poor approach, perhaps the most important is that of all the data a given user fetches, they are likely to edit only a small amount of that data. When you implement a pessimistic locking strategy, however, you prevent other users from even viewing the data that one user has accessed. So the pessimistic approach severely impacts the usability of your application and compromises the value of the application's data (since users aren't guaranteed that they can see all the data).

Although a pessimistic locking strategy largely guarantees that update conflicts won't occur, pessimistic locking has these undesirable effects:

- Not all databases support pessimistic locking.
- Databases support pessimistic locking in different ways.
- Pessimistic locking prevents other users from reading locked data.
- Pessimistic locking can cause deadlocks and excessive locking.

Consistent with Enterprise Object's abstract character, it provides a locking strategy that works at a higher level than the database. This approach is called **optimistic locking**. With optimistic locking, database rows are never actually locked. This strategy doesn't detect update conflicts until an application attempts to save changes to the database. Optimistic locking provides these advantages:

- It is supported by all databases.
- It is database independent.
- It is easy to use.
- It doesn't use any extra database resources.

The recommended update strategy in an Enterprise Objects application is optimistic locking. In addition to pessimistic locking, Enterprise Objects supports other locking strategies but they are not generally appropriate for three-tier application environments.

Inside Optimistic Locking

By default, Enterprise Objects uses optimistic locking to manage update conflicts. The idea behind optimistic locking is rather simple: When a user attempts to save changes made to an enterprise object, the framework compares the data in that object's snapshots with the current data in the database. If the comparison yields no differences, the save is allowed to execute.

In more detail, when data is fetched from a data source, Enterprise Objects records a snapshot for each row of data that is fetched. It stores these snapshots in an EODatabaseContext object. From these snapshots, enterprise object instances are created. When users make changes to the data in enterprise object instances and attempt to commit those changes back to the data source, the framework finds the object's corresponding row-level snapshots and identifies the locking attributes in those snapshots. In the `UPDATE` statement, it uses these values in a `WHERE` clause to make sure the row that is being updated hasn't changed in the database since it was last fetched.

If the update operation returns zero rows, it means that the values in the columns referenced in the update's `WHERE` clause changed, which means that the data in the database that corresponds to the enterprise object instance that is trying to save has been changed by another user or process, and an optimistic locking exception is thrown.

Consider this example. Using the entity described in "Reference Entity" (page 24), suppose a Listing enterprise object is fetched and has the following data values:

- `bathrooms`, 2
- `bedrooms`, 4

Then suppose that a user changes the value of the `bathrooms` property to 2.5. When the user attempts to save that change, the following SQL is generated: `UPDATE LISTING SET BATHROOMS = 2.5 WHERE (BATHROOMS = 2 AND BEDROOMS = 4)`.

The columns specified in the `WHERE` clause include only the attributes that are marked for locking in the application's EOModel. If this update returns zero rows, it means that the condition of the `WHERE` clause isn't satisfied, which means that the enterprise object's data in the database changed from the time it was last fetched—an optimistic locking failure.

When a user makes changes to data and attempts to save those changes, they must be reasonably guaranteed that the data they edited represents the freshest state of the data in the database. Simply committing a user's changes back to the data source without determining if the data in the data source has changed compromises data consistency and integrity.

Multiple Coordinators and Optimistic Locking

In some ways, when you provide each session with its own access layer stack, you complicate the optimistic locking equation. When multiple users share the same row-level snapshots, in contrast, you minimize the opportunities for multiple users to cause an optimistic locking conflict.

Consider the case when user A and user B share row-level snapshots and both users open a record that contains the same row of data. When user A make changes to that row and commits those changes back to the data source, the snapshot of that row reflects user A's changes. When user B requests the same record after user A changes it, user B is guaranteed to see the data with user A's changes. In this scenario, both users share the same set of cached data.

Then consider the case when user A and user B don't share row-level snapshots. When user A makes changes to a row and commits those changes back to the data source, user B doesn't see those changes unless they explicitly request fresh data from the database. When user B changes the same record and attempts to commit those changes, an optimistic locking exception is thrown since user B was editing stale data. (The exception usually occurs in the method `updateValuesInRowDescribedByQualifier` in a EOAdaptorChannel subclass).

If you want to provide each session with an independent access layer stack, there are a number of workarounds to deal with the optimistic locking issues that result from that configuration. As discussed in "Ensuring Fresh Data" (page 64), you can explicitly set the fetch timestamp on an editing context to encourage refetching from the data source rather than from the access layer's row-level snapshots. While this causes an inordinate amount of fetching, depending on the average size of an application's data sets, it may be a viable option.

Another workaround is to perform raw row operations—operations that don't automatically result in the creation of enterprise object instances from fetched data. The results of raw row operations—such as raw row fetching, raw SQL operations, or fetch specifications that fetch raw rows—are not cached

so you don't need to worry about optimistic locking. The optimistic locking mechanism is effectively bypassed for raw row operations. This has other significant consequences as discussed in "Raw Row Fetching" (page 70).

Using Optimistic Locking

Although optimistic locking is enabled by default in an Enterprise Objects application, you still need to make decisions that affect how it works. At minimum, you need to select which attributes in your application's entities participate in optimistic locking. You identify an attribute as a participant in optimistic locking by selecting its locking characteristic in EOModeler. Attributes that are selected for locking appear with a lock icon in their row.

By default, all attributes you add to entities other than primary keys are selected for optimistic locking. However, selecting all types of attributes for optimistic locking is not optimum and can result in serious performance implications. Consider these guidelines when choosing which attributes to select for locking:

- Binary data types (such as `BLOB`, `RAW`, `object`, `binary`) must not be selected for locking since they are not easily comparable to one another.
- Avoid locking on nonbinary data types that contain a lot of data as snapshot comparisons consume resources in proportion to their size. For example, a `varchar` column of width 2048 should not be selected for locking.

Prevention

Before determining how to instrument your application to deal with update conflicts, you should understand the mechanisms Enterprise Objects uses to prevent update conflicts, and especially how you can utilize these mechanisms in your applications.

Within Enterprise Objects, there are a number of contexts in which the preventative mechanisms for avoiding update conflicts work. These contexts include application instances, individual sessions, and an application's data sources.

Within a given application instance, an Enterprise Objects application that uses a single access layer stack (as described in "Core Framework Stack" (page 47) provides a single database context (per data source) for all the application's sessions. When sessions share a database context, they share row-level snapshots that participate in the update strategy called optimistic locking. By sharing row-level snapshots, multiple sessions are less likely to encounter optimistic locking exceptions caused by other sessions since the shared snapshots contribute to fresher data in each session than if the snapshots aren't shared.

Within a given session, Enterprise Objects provides a change notification infrastructure that updates in-memory enterprise object instances when the data in other enterprise object instances changes. This helps to ensure that within a given session, a user sees the freshest data throughout, especially when they've changed data. This helps minimize the possibility of a given session from triggering an optimistic locking failure based on data that is edited within that session. For example, if a session

edits a Listing enterprise object early in the session and then later edits it again, notifications ensure that the second time the Listing object is edited, it reflects the changes made in the first edit. Otherwise, sessions would likely overwrite their own data, causing optimistic locking failures.

Finally, within a database that an Enterprise Objects application uses, optimistic locking ensures that one user's changes aren't overwritten by another user's changes. How this works is discussed in "Inside Optimistic Locking" (page 92).

Mechanisms within each of these three realms contribute to the prevention of update conflicts. In most applications, you'll need to take some control over the mechanisms in each realm. Perhaps the most common type of intervention is adjusting an editing context's fetch timestamp to encourage more fetching from the database, which ensures that an application's user interface reflects fresh data. It is discussed in "Ensuring Fresh Data" (page 64).

Recovery

Instrumenting your applications to prevent update conflicts may not be enough to deal with the problem. It's also prudent to instrument your applications to recover gracefully when an update conflict occurs. This section discusses what you need to think about to instrument recovery and provides code samples that can help you implement a recovery strategy.

When an optimistic locking conflict is detected, an EOGeneralAdaptorException is thrown. You can do a number of things to deal with this exception. By default, Enterprise Objects doesn't do anything when the exception is thrown. A common design pattern is to wrap an invocation of `EOEditingContext.saveChanges()` in a try-catch block. In the catch block, you can choose to do a number of things.

You can choose to do nothing, which simply hides the exception from the user. You can choose to identify the affected enterprise objects, refault them, tell the user to make their changes and try saving again, as discussed in "Recovering and Refaulting" (page 95). You can choose to identify the affected enterprise objects, identify the changes that failed to save, and choose to save those changes again, disregarding the data in the database, as discussed in "Recovering and Last Write Wins" (page 98).

Recovering and Refaulting

The following example catches an optimistic locking failure, identifies the enterprise objects involved in the failure, and refaults those objects.

To catch an optimistic locking failure, you typically add a try-catch block around an invocation of `EOEditingContext.saveChanges()`, as shown in "Listing 9-1".

Listing 8-1 Adding a try-catch block around `saveChanges`

```
public void save() {
        EOEditingContext editingContext = session().defaultEditingContext();
        try {
            editingContext.saveChanges();
            //Thrown for each eo that fails to save.
        } catch (EOGeneralAdaptorException saveException) {                    // 1
```

```
            //Determine if the exception is an optimistic locking exception.
            if (isOptimisticLockingFailure(saveException)) {                    // 2
                //Deal with the optimistic locking exception.
                handleOptimisticLockingFailure(saveException);                  // 3
            } else {
                //Don't know what went wrong so revert editing context to a
stable state.
                editingContext.revert();                                        // 4
            }
        }

    }
```

Code line 1 catches the exception that is thrown when a save fails, which is usually an EOGeneralAdaptorException. The method invoked in code line 2 determines if the exception is an optimistic locking failure. The method invoked in code line 3 deals with the failure. If another kind of exception is thrown during the save, code line 4 simply invokes `revert` on the editing context, which returns the editing context to a stable state.

The code sample in "Listing 9-2" determines if the exception thrown is an optimistic locking failure. It is invoked from code line 2 in "Listing 9-1".

Listing 8-2 Determining if the exception is an optimistic locking failure

```
//Determine if the exception thrown during a save is an optimistic locking
exception.
    public boolean isOptimisticLockingFailure(EOGeneralAdaptorException
                                                    exceptionWhileSaving) {
        //Get the info dictionary that is created when the exception is thrown.
        NSDictionary exceptionInfo = exceptionWhileSaving.userInfo();           // 1
        //Determine the type of the failure.
        Object failureType = (exceptionInfo != null) ?                          // 2
exceptionInfo.objectForKey(EOAdaptorChannel.AdaptorFailureKey) : null;
        //Return depending on the type of failure.
        if ((failureType != null) &&                                            // 3
(failureType.equals(EOAdaptorChannel.AdaptorOptimisticLockingFailure))) {
            return true;
        } else {
            return false;
        }
    }
```

Throughout Enterprise Objects, many of the possible exceptions that are thrown include an info dictionary that provides details about the causes of an exception. Code line 1 simply retrieves the info dictionary from the exception thrown during the optimistic locking failure. Code line 2 uses that dictionary to determine the type of failure. Code line 3 returns `true` if the failure type is an optimistic locking failure and `false` otherwise.

After the method in "Listing 9-1" determines if the exception resulted from an optimistic locking failure, the code sample in "Listing 9-3" manages the failure. The method in "Listing 9-3", `handleOptimisticLockingFailureByRefaulting`, is invoked in code line 3 of "Listing 9-1".

Listing 8-3 Managing an optimistic locking failure by refaulting

```
//Deal with an optimistic locking failure.
```

```
    public void
handleOptimisticLockingFailureByRefaulting(EOGeneralAdaptorException
lockingException) {
        //Get the info dictionary that is created when the exception is thrown.
        NSDictionary info = lockingException.userInfo();                        // 1
        //Determine the adaptor operation that triggered the optimistic locking
 failure.
        EOAdaptorOperation adaptorOperation =                                   // 2
(EOAdaptorOperation)info.objectForKey(EOAdaptorChannel.FailedAdaptorOperationKey);
        int operationType = adaptorOperation.adaptorOperator();                 // 3
        //Determine the database operation that triggered the failure.
        EODatabaseOperation dbOperation =                                       // 4
(EODatabaseOperation)info.objectForKey(EODatabaseContext.FailedDatabaseOperationKey);
        //Retrieve the enterprise object that triggered the failure.
       EOEnterpriseObject failedEO = (EOEnterpriseObject)dbOperation.object();// 5
        //Retrieve the dictionary of values involved in the failure.
        //Take action based on the type of adaptor operation that triggered the
 optimistic          locking failure.
        if (operationType == EODatabaseOperation.AdaptorUpdateOperator) {       // 6
           //Recover by refaulting the enterprise object involved in the failure.
            //This refreshes the eo's data and allows the user to enter changes
 again               and resave.
            session().defaultEditingContext().refaultObject(failedEO);          // 7
            }
        } else { //The optimistic locking failure was caused by another type of
 adaptor             operation, not an update.
            throw new NSForwardException(lockingException, "Unknown
adaptorOperator " +               operationType + " in optimistic locking
exception.");
        }
        session().defaultEditingContext().saveChanges();
    }
```

Code line 1 retrieves the info dictionary that contains detailed information about the optimistic locking failure. Code line 2 determines the adaptor operation that triggered the failure.

Code line 3 determines the type of the adaptor operation that triggered the failure. There are a number of adaptor operations that include AdaptorUpdateOperator and AdaptorDeleteOperator. See the API reference for the class `com.webobjects.eoaccess.EOAdaptorOperation` for a list of all the operations. The adaptor operation type is used in code line 6.

Code line 4 retrieves the database operation in which the optimistic locking failure originated. From the database operation, code line 5 retrieves the enterprise object that was involved in the locking failure. An optimistic locking exception is thrown when an individual enterprise object instance fails to save. When even a single enterprise object fails to save, the entire `saveChanges` operation fails.

From the information retrieved by the code in code line 3, code line 6 takes action based on the type of adaptor operation. You need to determine which adaptor operation triggered the optimistic locking failure to know how to manage the failure. You'd manage a failure resulting from an AdaptorDeleteOperator differently than you'd manage a failure resulting from an AdaptorUpdateOperator.

After determining that the exception thrown during `saveChanges` resulted from an optimistic locking failure, code line 7 attempts to refault the enterprise object that was involved in the failure. Refaulting clears in-memory changes in an enterprise object and populates its data with values from the database. Users would then need to reenter their changes and attempt to save again, so you should display a message in the user interface with those instructions.

Recovering and Last Write Wins

The following example catches an optimistic locking failure, identifies the enterprise objects involved in the failure, identifies the values of the enterprise objects involved in the failure, and attempts to commit those values to the database.

Listing 8-4 Managing an optimistic locking failure by last write wins

```
//Deal with an optimistic locking failure.
    public void
handleOptimisticLockingFailureByLastWriteWins(EOGeneralAdaptorException
lockingException) {
        //Get the info dictionary that is created when the exception is thrown.
        NSDictionary info = lockingException.userInfo();
        //Determine the adaptor operation that triggered the optimistic locking
 failure.
        EOAdaptorOperation adaptorOperation =
(EOAdaptorOperation)info.objectForKey(EOAdaptorChannel.FailedAdaptorOperationKey);
        int operationType = adaptorOperation.adaptorOperator();
        //Determine the database operation that triggered the failure.
        EODatabaseOperation dbOperation =
(EODatabaseOperation)info.objectForKey(EODatabaseContext.FailedDatabaseOperationKey);
        //Retrieve the enterprise object that triggered the failure.
        EOEnterpriseObject failedEO = (EOEnterpriseObject)dbOperation.object();
        //Retrieve the dictionary of values involved in the failure.
        NSDictionary valuesInFailedSave = adaptorOperation.changedValues();    // 1
        NSLog.out.appendln("valuesInFailedSave: " + valuesInFailedSave);

        //Take action based on the type of adaptor operation that triggered the
 optimistic locking failure.
        if (operationType == EODatabaseOperation.AdaptorUpdateOperator) {
            //Recover by essentially ignoring the optimistic locking failure
and committing the
            //changes that originally failed. This is a last write wins policy.
            //Overwrite any changes in the database with the eo's values.
            failedEO.reapplyChangesFromDictionary(valuesInFailedSave);        // 2
        } else { //The optimistic locking failure was causes by another type of
 adaptor operation, not an update.
            throw new NSForwardException(lockingException, "Unknown
adaptorOperator " + operationType + " in optimistic locking exception.");
        }
        session().defaultEditingContext().saveChanges();
    }
```

"Listing 9-4" differs from "Listing 9-3" only in code line 1 and code line 2. Code line 1 of "Listing 9-4" retrieves the values of the enterprise object that were involved in the optimistic locking failure. For example, if the enterprise object represents the Listing entity in the Real Estate model and the `bedrooms`

attribute of that enterprise object had changes, the dictionary of changes retrieved in code line 1 contains a key called `bedrooms` and its changed in-memory value (not the attribute's value in the database).

This dictionary is used in code line 2, which attempts to reapply the changes that failed to be committed. This approach is referred to as a "last write wins" approach because code line 2 commits the changes regardless of the values in the database. So if the optimistic locking failure was caused because another user or process changed the data in the database that corresponds to the edited enterprise object, code line 2 disregards those changes and writes the changes that failed in their place. This may or may not be a reasonable approach for your application.

Connecting to a Database

This chapter describes how an Enterprise Objects application connects to a database and how you can take control of this process. You may want to manually connect to a database for a few reasons:

- Your application has an audit tracking requirement.
- Different users have different levels of access to the database.
- You need to better secure database login information.
- You need to limit the number of open connections to a database.

This chapter is divided into the following sections:

- "Objects Involved in a Database Connection" (page 102) describes the objects involved in connecting to a data source.
- "When Database Connections Are Opened" (page 103) discusses when database connections are opened and how they are reused.
- "When Database Connections Are Closed" (page 103) discusses when database connections are closed and how to take control of this mechanism.
- "Connection Dictionary" (page 103) describes the connection dictionary that Enterprise Objects uses to connect to a data source and the different ways to provide it.
- "Database Channels" (page 106) discusses the default configuration of database channels in Enterprise Objects and how to customize it.
- "Connecting to Multiple Data Stores" (page 105) discusses Enterprise Object's support for accessing multiple data sources in an application.
- "Providing a Connection Dictionary in Code" (page 106) tells you how to provide a connection dictionary programmatically.
- "Providing Multiple Database Channels" (page 106) tells you how to provide multiple database channels in an application.
- "Closing Database Channels" (page 107) tells you how to monitor the number of open database channels and close them as needed.

Objects Involved in a Database Connection

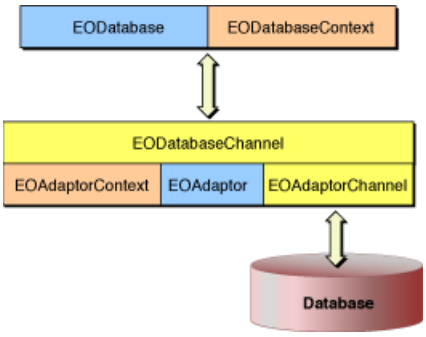
This section provides a high-level overview of the objects involved in an Enterprise Objects database connection. For more specific information, see the API reference for the classes mentioned here.

The highest-level object in an Enterprise Objects database transaction is an EODatabase object. This object represents a single database server so your application has as many EODatabase objects as the databases to which it connects. An EODatabase object communicates with an EOAdaptor object that is capable of communicating with a database server.

Each EODatabase contains at least one EODatabaseContext object. Each EODatabaseContext forms a separate transaction scope to a database using an EODatabaseChannel object (which itself uses an EOAdaptorChannel object). Each EODatabaseContext uses one or more EODatabaseChannel objects.

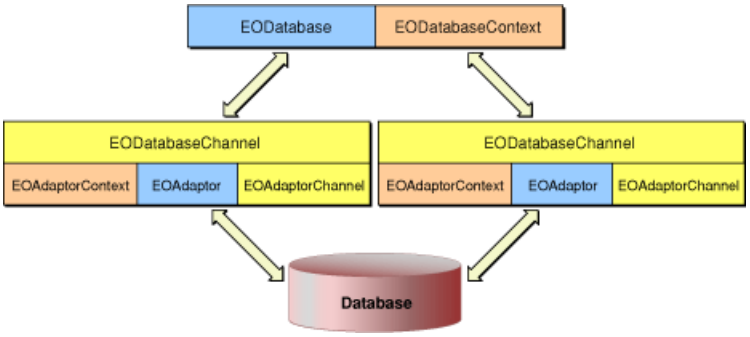
"Figure 10-1" illustrates the scenario in which an EODatabase manages a single EODatabaseContext object and the database context uses a single database channel.

Figure 9-1 EODatabaseContext managing a single database channel



By default, an EODatabaseContext uses a single EODatabaseChannel but in "Figure 10-2", the database context is shown using two database channels. Each database channel is always associated with exactly one set of adaptor objects (an EOAdaptor, an EOAdaptorContext, and an EOAdaptorChannel), as "Figure 10-2" illustrates.

Figure 9-2 One database context using two database channels



If the database server an EODatabase object represents supports multiple concurrent transaction sessions, that EODatabase may have several EODatabaseContexts. If the adaptor allows multiple channels per context, then an EODatabaseContext may in turn have several EODatabaseChannels, which handle actual access to the database.

The rest of this chapter describes how these objects are used by Enterprise Objects to connect to a database and how you can take control of the connection.

When Database Connections Are Opened

You usually never have to worry about opening database connections programmatically—Enterprise Objects takes care of it for you. The first time any editing context in your application initiates a database operation, a connection to the database is opened by the access layer. Enterprise Objects reuses that same connection for all subsequent database operations to that particular database. If your application accesses multiple databases, a separate database connection is opened for each database.

When Database Connections Are Closed

Enterprise Objects doesn't close its connections to databases until the application terminates. It reuses open connections so by default it uses a minimum number of connections and you don't need to limit the number of connections. However, there are situations in which you may want to close connections when they aren't in use. These situations are rare and usually result when many copies of an application connect to the same database or if you add more database channels as described in "Database Channels" (page 106) and the database can't handle more than a certain number of database connections.

If you're concerned about the number of open database channels in a particular application, it's probably best to set a timer in your application and attempt to close open database connections at a specified time. Before closing an open database channel, you should make sure that channel isn't performing an operation. See "Closing Database Channels" (page 107) to learn how to close database channels on demand.

Connection Dictionary

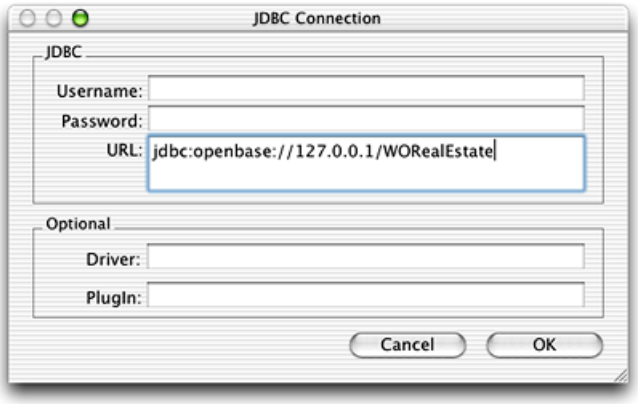
Enterprise Objects uses an EOAdaptor object to connect and log in to a database. WebObjects 5.2 provide two adaptors: one for JDBC data sources and one for JNDI data sources. An adaptor uses a **connection dictionary** to identify the address of the database and to provide login credentials.

When you're developing an Enterprise Objects application, you usually supply the connection dictionary information in the EOModel files the application uses. While this is convenient during development, it may not be adequate for deployed applications. Instead, you may need to provide the connection dictionary programmatically. The following sections describe the two ways to provide connection dictionary information to an application.

Storing in a Model File

When you create a new EOModel file, you are prompted to provide login information for the data source to which that model connects. The window that asks for this information is shown in "Figure 10-3".

Figure 9-3 Connection dictionary window in EOModeler



The information you provide in this window becomes the model file's connection dictionary. The connection dictionary is stored in the model file and doesn't require you to write any code to use it. This approach is useful when all users of the application log in to the database with the same connection information and when the login information is not regarded as sensitive.

However, if you need to provide users with different database connection information or if you still want to allow all users to share the same credentials but you want to secure those credentials, you need to provide some of the connection dictionary information programmatically. You can still provide nonsensitive connection information in the model, such as the database URL, driver, and plug-in, and just provide login credentials programmatically.

Providing the connection dictionary programmatically (or even just part of the connection dictionary) requires more code on your part, so you should do it only if you really need to. "Storing in Code" (page 104) tells you how to provide the connection dictionary programmatically.

All EOAdaptor objects that Enterprise Objects creates automatically use the connection dictionary information specified in the model. If you instantiate an adaptor programmatically, however, it doesn't automatically use the model's connection dictionary information, so you have to provide it programmatically.

Storing in Code

For any number of reasons, you may need to provide all or part of an EOAdaptor's connection dictionary programmatically. To do this, you first need to identify the dictionary's keys so you can set their values.

The following code prints the connection dictionary keys of the Real Estate model:

```
EOModel model = EOModelGroup.defaultGroup().modelNamed("RealEstate");
```

```
NSLog.out.appendln("connection dictionary keys: " +
model.connectionDictionary().allKeys());
```

The output of this code is:

```
connection dictionary keys: ("plugin", "jdbc2Info", "username", "driver", "password",
"URL")
```

Now that you know the names of the connection dictionary's keys, you need to set the values for some of them. You don't need to set the values for all of the keys, only for the keys you care about. When you force a model to use a new connection dictionary with the method discussed below, only the keys for which you explicitly set values are overridden. So if, for example, the original connection dictionary in the model specifies the database URL and you don't provide a value for that key in the dictionary of overrides, the value in the model's dictionary is still used even after you've provided an updated connection dictionary.

Where in an application's execution should you set the values? It depends on your application's requirements. If you provide each user with an independent connection to the database by giving each user a separate stack as described in "Providing Separate Stacks" (page 52), you should first get a user's database username and password and then set the values of the connection dictionary before instantiating the stack. After you set the values in the connection dictionary but before you instantiate a stack for each user, you need to invoke the method `EODatabaseContext.forceConnectionWithModel`. You pass to this method as arguments an EOModel object, a connection dictionary of keys to override with the values you supply programmatically, and an editing context.

The code in "Providing a Connection Dictionary in Code" (page 106) shows all these steps.

Connecting to Multiple Data Stores

Enterprise Objects makes it easy to work with multiple data stores in a single application. An application often includes enterprise object instances that represent data from different repositories.

The EOObjectStoreCoordinator is the object that manages multiple repositories in a single application. When a fetch is performed, it determines which repository can service the fetch. Similarly, when changes are committed, it determines which repository to save the changes to. When a fault is fired, it determines which repository can service the fault. And, Enterprise Objects is smart enough to generate database-specific or database-optimized SQL expressions within a single application that connects to multiple repositories.

There are, however, a few limitations when using multiple data sources in an application. They include:

- Within an EOModelGroup, all entity names must be unique.
- You can't model inheritance hierarchies across different data sources.
- You can't flatten attributes or relationships across data sources.
- Enterprise Objects doesn't support two-phase commit, so you have to be careful when a saving enterprise objects that are constituted from different data sources in a single invocation of `EOEditingContext.saveChanges()`.

If you understand these limitations, connecting to multiple data stores in an Enterprise Objects application should just work.

Database Channels

The default configuration of the Enterprise Objects core stack provides only a single database channel, regardless of the number of transactions a particular Enterprise Objects stack makes. The default behavior is usually sufficient but certain applications and configurations may benefit from using multiple database channels.

Conflicts within Enterprise Objects due to busy database channels may occur when a database context needs to perform an operation and its database channel is already busy with another operation such as fetching. Most of these types of conflicts are the result of inefficient fetching, such as when faults are inadvertently fired while other database operations are in progress, and can be avoided.

When an EODatabaseContext needs a new channel because all its current channels are busy, it posts an EODatabaseChannelNeededNotification. You can create database channels on demand by registering for this notification and providing an additional channel at that time. In the method you write to create database channels, you should set an upper limit on the number of channels that can be registered with a particular database context. It's very unusual for an EODatabaseContext to require more than two or three EODatabaseChannels. See "Providing Multiple Database Channels" (page 106) to learn how to add additional database channels to an application.

Providing a Connection Dictionary in Code

The code in "Listing 10-1" provides a connection dictionary programmatically using the procedure discussed in "Storing in Code" (page 104).

Listing 9-1 Setting connection dictionary programmatically

```
EOModel model = EOModelGroup.defaultGroup().modelNamed("RealEstate");
NSLog.out.appendln("connection dictionary keys:" +
model.connectionDictionary().allKeys());

NSMutableDictionary overrides = new NSMutableDictionary();
overrides.takeValueForKey("brent", "username");
overrides.takeValueForKey("secret", "password");
EODatabaseContext.forceConnectionWithModel(model, overrides, new
EOEditingContext());
```

Providing Multiple Database Channels

To create and register a new database channel with a particular database context, you can use the code in "Listing 10-2" (page 107). It's probably most appropriate to put this code in your Session class, as this listing does. The code listing also shows how to register for the notification that EODatabaseContext posts when it needs another database channel.

Listing 9-2 Creating and registering a new database channel

```
import com.webobjects.foundation.*;
import com.webobjects.appserver.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eoaccess.*;

public class Session extends WOSession {

    private final int DatabaseChannelCountMax = 3;

    public Session() {
        super();
        NSNotificationCenter.defaultCenter().addObserver(this, new
            NSSelector("registerNewDatabaseChannel",
            new Class[] { NSNotification.class } ),
            EODatabaseContext.DatabaseChannelNeededNotification, null);
    }

    public void registerNewDatabaseChannel(NSNotification notification) {
        EODatabaseContext databaseContext =
(EODatabaseContext)notification.object();
        if (databaseContext.registeredChannels().count() <
DatabaseChannelCountMax){
            EODatabaseChannel channel = new EODatabaseChannel(databaseContext);
            databaseContext.registerChannel(channel);
        }
    }

}
```

If you create database channels programmatically on a per-session basis, you may end up with a large number of database channels per application instance. "When Database Connections Are Closed" (page 103) describes how to check for and close extraneous database connections.

Closing Database Channels

The code in "Listing 10-3" demonstrates how to close an application's database connections at a recurring interval. The `closeDatabaseChannels` method assumes that the application has only one object store coordinator (per-session object store coordinators are not used) and that all of the cooperating object stores managed by the coordinator are database contexts, which is the usual case.

Listing 9-3 Close database channels at a specified interval

```
import com.webobjects.foundation.*;
import com.webobjects.appserver.*;
import com.webobjects.eocontrol.*;
import com.webobjects.eoaccess.*;
import java.util.*;

public class Application extends WOApplication {

    public static void main(String argv[]) {
        WOApplication.main(argv, Application.class);
```

```
    }

    public Application() {
        super();
        System.out.println("Welcome to " + this.name() + "!");
        setupDatabaseChannelCloserTimer();
    }

    public void setupDatabaseChannelCloserTimer() {
        Timer timer = new Timer(true);
        //Close open database connections every four hours.
       timer.scheduleAtFixedRate(new DBChannelCloserTask(), new Date(), 14400);
    }

    class DBChannelCloserTask extends TimerTask {

     public DBChannelCloserTask() {
      super();
     }

      public void run() {
       closeDatabaseChannels();
       NSLog.out.appendln("running timer");
      }

      public void closeDatabaseChannels() {
       int i, contextCount, j, channelCount;
       NSArray databaseContexts;
       EOObjectStoreCoordinator coordinator;

       coordinator =
      (EOObjectStoreCoordinator)EOObjectStoreCoordinator.defaultCoordinator();
       databaseContexts = coordinator.cooperatingObjectStores();
       contextCount = databaseContexts.count();

       //Iterate through all an app's cooperating object stores (database
contexts).
       for (i = 0; i < contextCount; i++) {
        NSArray channels =

((EODatabaseContext)databaseContexts.objectAtIndex(i)).registeredChannels();
        channelCount = channels.count();

          for (j = 0; j < channelCount; j++) {
          //Make sure the channel you're trying to close isn't performing a
transaction.
         if (!((EODatabaseChannel)channels.objectAtIndex(j)).adaptorChannel().
            adaptorContext().hasOpenTransaction()) {
             ((EODatabaseChannel)channels.objectAtIndex(j)).adaptorChannel().
              closeChannel();
           }
          }
        }
      }
    }
}
```

Concurrency

The default Enterprise Objects core configuration uses a single thread to perform its operations (not including threads in the Java virtual machine), which means that only a single operation within Enterprise Objects can occur at any given moment. More importantly, however, a single application instance by default has only one access layer stack. This means that each user in a single application instance shares the resources for connecting to, fetching from, and committing data to the application's data sources with the application's other users.

If user A initiates a fetch while user B's fetch is being performed, user A's fetch must wait until user B's fetch is done. If user B commits many records to a data source while user A is fetching data from that data source, user B's commit doesn't occur until user A's fetch is done. In an application that services multiple concurrent users, you can see that you'll likely consider instrumenting multithreading in your Enterprise Objects applications.

This chapter discusses threading in Enterprise Objects, including how to instrument multithreading in an Enterprise Objects Application. It is divided into the following sections:

- "Determining Requirements" (page 109) discusses if and when you should think about implementing concurrency.
- "Maintaining Thread Integrity" (page 110) discusses your responsibilities when you instrument an application for concurrency.

Determining Requirements

The first step in instrumenting multihreading in an Enterprise Objects application is determining if you really need to. Multithreaded applications of any type are inherently more difficult to program, debug, and maintain than those that use a single thread.

In the context of Enterprise Objects and WebObjects, it is usually better to allocate more application instances and more hardware on which to run those instances rather than to complicate your applications by instrumenting concurrency programmatically. Hardware today is rather cheap and sometimes using more servers is the most reasonable solution to service multiple concurrent users of the same database.

The second step in instrumenting multithreading in an Enterprise Objects application is determining what kind of concurrency you need. This requires knowing (or predicting) the bottlenecks within your application. Do bottlenecks occur at the database level when multiple users attempt concurrent

access to the data source so that adding more database channels alleviates the bottleneck? Do bottlenecks occur elsewhere in the access layer so that providing a separate access layer for each user alleviates the bottleneck?

The answers to these questions help determine the mechanism you need to use to instrument concurrency within an Enterprise Objects application. Two common design patterns for concurrency within Enterprise Objects applications are to:

- Provide each user with an independent access layer stack.
- Provide multiple database channels on demand.

Both of these options provide concurrency within the access layer, which is usually sufficient to achieve multithreading within Enterprise Objects.

The first design pattern is discussed in "Providing Separate Stacks" (page 52). The second is discussed in "Database Channels" (page 106).

Maintaining Thread Integrity

When instrumenting concurrency in any application, you must take responsibility for locking certain objects to ensure thread integrity. Even if you don't instrument concurrency in an application, that application still uses multiple threads—no Java application is truly single-threaded. On most Java virtual machines, the garbage collector and `finalize` methods each run in separate threads. To ensure that objects you manipulate aren't affected by one of these threads, *you must lock the objects you manipulate directly*. As part of instrumenting concurrency in an Enterprise Objects application, but also in Enterprise Objects applications in which you don't explicitly instrument concurrency, *it is your responsibility to lock the Enterprise Objects you use directly*.

In versions of Enterprise Objects prior to WebObjects 5.2, you were expected to explicitly lock and unlock EOEditingContext objects. Most other Enterprise Objects locked themselves in any methods that changed state (which includes most methods) or did not support locking.

These objects, however, which primarily include EOObjectStore objects and fault handlers in the access layer, had no way of knowing the context of their usage. The breadth of the Enterprise Objects API allowed them to be used in many different ways at many different times. For example, faults can be fired in many different scenarios. Consequently, these objects needed to lock and unlock frequently. This has undesirable performance characteristics.

Imagine an EOEditingContext fetches a thousand rows from a database. The EODatabaseContext method `initializeObject` is invoked once per row to create a corresponding EOEnterpriseObject. That method performs locking operations each time it is invoked. Since EODatabaseContext can service only one EOEditingContext at a time, nearly all of those locking operations are redundant.

A better design is to remove the locking operations from `initializeObject` and to mandate that the locking operations are instead performed on the EODatabaseContext object. Then, EOEditingContext manages the lock on the EODatabaseContext it uses to fetch data. Only a single set of locking operations on an EODatabaseContext object is required per fetch.

In this design, object stores automatically lock their parent object stores when they perform operations requiring access to those object stores. So, an editing context locks its parent object store once per fetch, making it unnecessary for locking operations to occur in each invocation of `initializeObject`.

This design reflects the updated concurrency architecture of Enterprise Objects in WebObjects 5.2. In the updated architecture, each lock is treated as a shared resource. *To ensure safe concurrent access to Enterprise Objects, it is your fundamental responsibility is to lock the Enterprise Objects you use directly.* The Enterprise Objects you manipulate directly and lock then lock any additional resources they use directly as needed.

Not all classes in the core Enterprise Object layers are suitable for concurrent access within this locking paradigm—this includes Enterprise Object classes that do not implement the NSLocking interface.

For example, EOEnterpriseObject does not implement the NSLocking interface. The framework assumes that enterprise object instances are used only by the thread that has locked their EOEditingContext. Since it probably never makes sense to provide concurrent access to EOEnterpriseObject instances separately from their EOEditingContext, it shouldn't be a problem that EOEnterpriseObject does not implement NSLocking.

You usually interact only with the EOEditingContext lock. It is vital to properly lock and unlock EOEditingContext objects to ensure the integrity of their EOEnterpriseObject instances. An editing context (an object store) automatically locks its parent object store (usually an instance of EOObjectStoreCoordinator). Obtaining a lock on an EOObjectStoreCoordinator causes it to lock all of its registered cooperating object stores.

Since EOObjectStoreCoordinator is the highest-level object in the access layer stack, and since it automatically locks the object stores registered with it, obtaining a lock on an EOObjectStoreCoordinator is sufficient to manipulate any access-layer objects underneath it. In other words, objects in the access layer can be used only by the thread that has obtained a lock on the object store coordinator that is the highest-level object in that particular access layer stack—you must secure a lock on a particular object store coordinator before using any of the objects it manages.

When you secure a lock for an object store coordinator, it automatically locks all of its registered cooperating object stores. When you release the lock on a particular object store coordinator, it automatically releases the locks it has on its cooperating object stores. If you directly manipulate any access layer objects that are not cooperating object stores that have been locked by an object store coordinator, you must lock and unlock those objects yourself. This includes objects like EODatabaseChannel and EOAdaptorChannel.

What are some of the consequences of irresponsible locking? Consider the case of an invocation of the `invalidateAllObjects` method on an editing context. This method propagates to every EOEditingContext in an application. If an unlocked EOEditingContext receives an `invalidateAllObjects` method, that editing context's EOEnterpriseObjects are forcefully turned into empty faults. The most favorable outcome of this scenario is that any outstanding changes in the application are lost. You can see that it's important to lock editing contexts (or other Enterprise Objects) to ensure that operations intended for a particular editing context (or other type of Enterprise Object) don't adversely affect other editing contexts.

Part of the responsibility in managing locks on Enterprise Objects is to responsibly discard locks. You must unlock any locks you explicitly obtain regardless of the circumstances. You can use `finally` blocks to achieve this requirement. Leaving locks in place after a nonfatal exception eventually deadlocks the application. Locked editing contexts can be garbage collected, so removing references to editing contexts also releases their locks.

Here are a few additional guidelines regarding locking in Enterprise Objects applications:

- In general, you should first secure the appropriate locks on EOObjectStoreCoordinators before posting notifications that other Enterprise Objects register to receive.

- Enterprise Object delegates do not need to worry about locking unless they attempt to access additional resources.
- Enterprise Objects uses more sophisticated locking objects than those built in to Java. These objects provide both you and Enterprise Objects with more control over the scope of critical regions within applications. This reduces contention and the possible scenarios that can generate deadlocks.
- Child (nested) editing contexts use their parent's lock.
- EOSharedEditingContext objects have a multireader, single-writer lock.
- Each EOObjectStore instance and each EOObjectStoreCoordinator instance may have its own lock.
- There is a global lock for loading EOModel files.

Problems with locking can be addressed by using NSLog. Set the debug level to at least `DebugLevelInformational` and the debug groups to include `DebugGroupMultithreading`. In the event of apparent deadlock, you can obtain a complete stack trace of all the threads within the Java virtual machine by sending the `java` process the `QUIT` signal. You can do this on the command line with `kill -3`*pid* or Control \, although these commands vary by Java platform.

Enterprise Objects in WebObjects

This appendix discusses how to access WebObjects in enterprise object classes and how to access enterprise objects in WebObjects Builder.

Accessing WebObjects in Enterprise Objects

In a well-designed WebObjects application, Enterprise Object classes don't have direct knowledge of WebObjects classes. In practical terms, this means that Enterprise Objects classes don't reference any classes in the WebObjects framework (`com.webobjects.appserver`). Enterprise Object classes designed this way are more portable and help you achieve the promise of reusable, highly maintainable business objects.

However, there are occasions in which you need to reference a WebObjects object such as WOSession or WOApplication. There is no direct API to do this. However, there are a few tricks that are commonly used to accomplish this.

The easiest way to access a WOSession object in an Enterprise Object class is to set the delegate of an editing context to be the session object. When you invoke the method `editingContext` on an enterprise object instance, it returns the Session's default editing context (this is the default behavior; your configuration may differ, depending on customizations you make).

Assuming your enterprise object classes use the default editing context configuration, to set the delegate for the editing context of any enterprise object class in your application, you invoke `defaultEditingContext().setDelegate(this)` in your Session class's constructor. Then, to get a reference to the Session object in an enterprise object class, you invoke the method shown in "Listing A-1" on an enterprise object instance.

Listing A-1 Accessing a Session object from an enterprise object

```
Session aSession = (Session)editingContext().delegate();
```

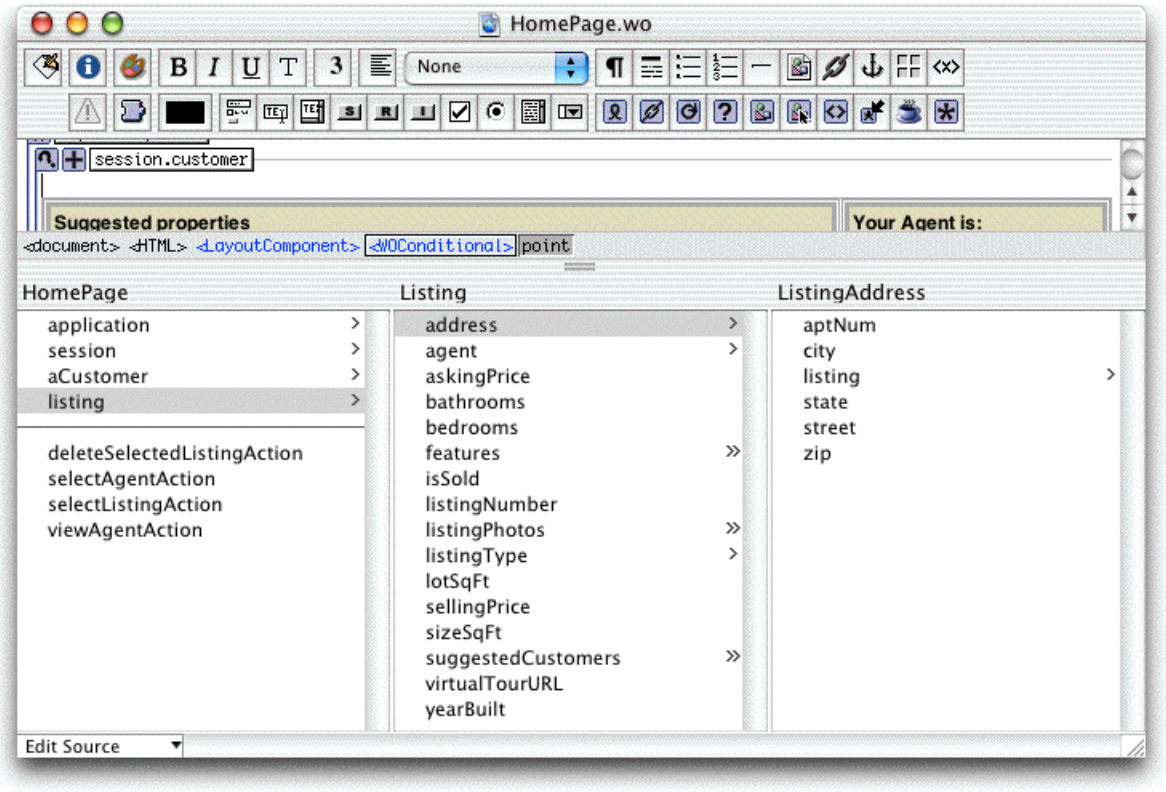
The code in "Listing A-1" assumes that an enterprise object's editing context is not `null` and that the enterprise object is in the editing context. In an enterprise object's constructor, this is not necessarily the case: The enterprise object may not yet be inserted into the editing context. So, the safe place to invoke the code is in `awakeFromInsertion` or `awakeFromFetch`.

While this is the easiet way to obtain references to WebObjects objects from enterprise objects, it compromises the purity and portability of the enterprise objects. A better approach would be to use notifications to access WebObjects objects from within enterprise objects. This approach maintains the decoupling between the two frameworks.

Enterprise Objects in WebObjects Builder

One of the most useful features of WebObjects Builder is its ability to access an application's EOModels and to extract from those models information about entities and an entity's class properties. It displays the entities and class properties in the lower pane of the split view while in layout view. This allows you to quickly and easily bind dynamic element bindings to properties of an enterprise object. "Figure A-1" shows the Listing entity and its class properties from the Real Estate model.

Figure A-1 Entity and attributes in WebObjects Builder



So how does WebObjects Builder know what entities to display? It does not display all of the entities in an application's models. Rather, it displays an entity and its attributes only if aWOComponent includes a key (a field or accessor methods) of a type that corresponds to an entity in the application's models.

For example, in "Figure A-1", the `listing` key corresponds to the Listing entity in the Real Estate model. The HomePage.wo component (from the iShacks example in `/Developer/Examples/JavaWebObjects/`) includes a field and accessors for the `listing` key.

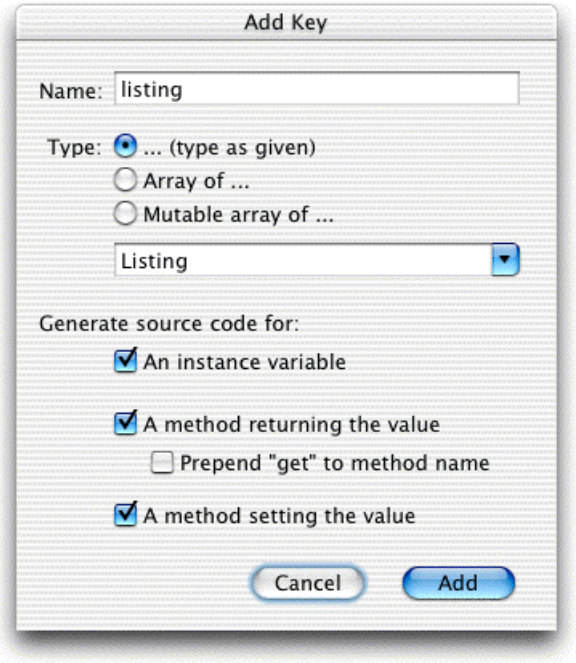
WebObjects Builder knows that the `listing` field and its accessors correspond to an enterprise object of type Listing because the `listing` field is declared to be of type Listing, which is an Enterprise Object class declared in the Real Estate framework that the iShacks example includes. The declarations of `listing` in the HomePage.wo component are shown in "Listing A-2".

Listing A-2 Declarations of the `listing` key

```
public Listing listing;
public Listing listing() {
    return listing;
}
public void setListing(Listing newListing) {
    listing = newListing;
}
```

But what if you add a key to a WOComponent that corresponds to an entity that doesn't use a customer enterprise object class (that is, the entity maps directly to EOGenericRecord)? To add a key to a WOComponent, you usually do it within WebObjects Builder. A window appears when you add a key, as shown in "Figure A-2".

Figure A-2 Add key window in WebObjects Builder



The combo box in the Type section lets you enter the name of an entity in the application's models. It includes an auto-complete feature, so if you want a new key to correspond to the Listing entity, typing "Li" displays the whole word "Listing." When you click Add, declarations for the new key are added to the current WOComponent. Each of the declarations includes a `@TypeInfo` comment above it, which WebObjects Builder uses to determine the entity of a key that is declared as an EOEnterpriseObject. Declarations for the `listing` key that maps to an EOGenericRecord rather than to an EOGenericRecord subclass called Listing are shown in "Listing A-3".

Listing A-3 `listing` key declarations with @TypeInfo

```
/** @TypeInfo Listing */
protected EOEnterpriseObject listing;

/** @TypeInfo Listing */
public EOEnterpriseObject listing() {
    return listing;
}

public void setListing(EOEnterpriseObject newListing) {
    listing = newListing;
}
```

Principal Methods

This chapter provides a list of the most commonly used methods in an Enterprise Objects application. The methods are categorized by the task in which they are used and listed alphabetically within each category.

Creating and Initializing Objects

This section lists the primary methods you use to instantiate and initialize enterprise objects.

EOEnterpriseObject: `awakeFromFetch(EOEditingContext`*editingContext*`)`
Override this method in custom enterprise object classes to provide custom initialization of an enterprise object when it is first fetched from a data source.

EOEnterpriseObject: `awakeFromInsertion(EOEditingContext`*editingContext*`)`
Override this method to provide custom initialization of an enterprise object when it is first created and inserted into an editing context.

Do not add any custom code to an enterprise object's constructor. Rather, override this method in your custom business logic classes and add your custom code there. This method is invoked automatically by the framework when a new enterprise object is created and has finished initializing. Before you can override this method for a particular enterprise object class, you need to create a custom class using EOModeler, which is discussed in "Business Objects" (page 23).

EOEditingContext: `insertObject(EOEnterpriseObject`*enterpriseObject*`)`
Use this method to insert a new enterprise object instance into an editing context. *When you instantiate an enterprise object, you must immediately insert it into an editing context.*

EOUtilities: `createAndInsertInstance(EOEditingContext`*editingContext*`, String`*entityName*`)`
This convenience method instantiates an enterprise object for the entity specified by *entityName* and inserts the new object into the editing context specified by *editingContext*.

Fetching and Accessing Data

The methods in this section fetch data from a data source and access the properties of enterprise object instances.

EOFetchSpecification constructor: `(String`*`entityName`*`, EOQualifier`*`qualifier`*`, NSArray`*`sortOrderings`*`)`

Use this method to construct a fetch specification that you use to retrieve data from a data source. Only the *`entityName`* parameter is mandatory. A fetch specification is transformed by the framework into a SQL expression in which *`entityName`* is used in the `FROM` clause, *`qualifier`* is used in the `WHERE` clause, and *`sortOrderings`* is used in the `ORDER BY` clause.

EOUtilities: `objectsForEntityNamed(EOEditingContext`***editingContext***`, String`*`entityName`*`)`

This convenience method fetches all the enterprise objects for the entity specified by *`entityName`*.

EOEditingContext: `objectsWithFetchSpecification(EOFetchSpecification`*`fs`*`)`

Use this method to retrieve data from a data source.

EOUtilities: `primaryKeyForObject(EOEditingContext`***editingContext***`, EOEnterpriseObject`*`enterpriseObject`*`)`

This convenience method retrieves the primary key for the enterprise object specified by *`enterpriseObject`*. You usually do not need to worry about primary keys when building an Enterprise Objects application.

EOQualifier: `qualifierWithQualifierFormat(String`*`format`*`, NSArray`*`arguments`*`)`

Use this method to construct a qualifier for use in a fetch specification or for use when sorting fetch results in memory.

EOUtilities: `rawRowsForSQL(EOEditingContext`***editingContext***`, String`*`modelName`*`, String`*`sqlString`*`)`

This convenience method evaluates the specified *`sqlString`* and uses information from the EOModel specified by *`modelName`* to retrieve the raw database rows specified by *`sqlString`*. In an Enterprise Objects application, you usually do not need to explicitly think about SQL or fetch raw rows.

EOCustomObject: `valueForKey(String`*`key`*`)`

This method returns an object for the property in an enterprise object specified by *`key`*. This method is part of the key-value coding infrastructure of Enterprise Objects, in which an object's properties are accessed by key rather than directly as fields or through accessor methods. This is discussed in more detail in "Accessing an Enterprise Object's Data" (page 34).

EOCustomObject: `valueForKeyPath(String`*`keyPath`*`)`

This method returns an object for the property in an enterprise object's relationship specified by *`keyPath`*, where *`keyPath`* is a String of the form "relationship.property". This method allows you to access relationships between enterprise objects using a chain of keys. It is part of the key-value coding infrastructure of Enterprise Objects, which is discussed in more detail in "Accessing an Enterprise Object's Data" (page 34).

Identifying and Tracking Objects

The methods in this section to identify and track enterprise objects in the object graph (in an editing context).

EOEditingContext: `faultForGlobalID(EOGlobalID`*`gid`*`, EOEditingContext`***editingContext***`)`

This method returns an enterprise object based on the global ID *`gid`* in the editing context *editingContext*. When an enterprise object is fetched from a persistent data source or otherwise instantiated, it is assigned a globally unique identifier called a global ID. You can use a global

ID to obtain a reference to the ID's enterprise object (which may be a fault). Uses for this method are discussed in "Working With Objects in Multiple Editing Contexts" (page 81).

EOEditingContext: `globalIDForObject(EOEnterpriseObject`*`enterpriseObject`*`)`
Use this method to obtain the global ID for an enterprise object in an editing context.

EOEnterpriseObject: `editingContext()`
Use this method to get a reference to the editing context in which a particular enterprise object is registered.

EOUtilities: `localInstanceOfObject(EOEditingContext`*`editingContext`*`, EOEnterpriseObject`*`enterpriseObject`*`)`
Use this method to create, in the editing context specified by *editingContext*, a copy of an enterprise object that exists in another editing context. This discussed in more detail in "Working With Objects in Multiple Editing Contexts" (page 81).

Working With Fetch Results

After you fetch data into an application, you often need to perform in-memory sorting or filtering of that data. The methods in this section provide support for both of those tasks.

EOQualifier: `filteredArrayWithQualifier(NSArray`*`array`*`, EOQualifier`*`qualifier`*`)`
Use this method to filter an array of enterprise objects based on the criteria specified by *`qualifier`*.

EOSortOrdering constructor: `(String`*`key`*`, NSSelector`*`selector`*`)`
Use this method to construct a sort ordering to use when sorting enterprise objects in memory. The *`key`* parameter specifies the property in the enterprise object on which to perform the sort. The *`selector`* parameter specifies the sorting order and is usually one of the static fields in the EOSortOrdering class, such as CompareAscending.

EOSortOrdering: `sortedArrayUsingKeyOrderArray(NSArray`*`array`*`, NSArray`*`sortOrderings`*`)`
Use this method to sort an array of enterprise objects in memory. The array of *`sortOrderings`* need contain only one EOSortOrdering object, which you construct with the EOSortOrdering constructor.

Manipulating and Changing Objects

The methods in this section manipulate enterprise objects and editing contexts.

EOCustomObject:
`addObjectToBothSidesOfRelationshipWithKey(EORelationshipManipulation`*`object`*`, String`*`key`*`)`
Use this method to add the enterprise object specified by *`object`* as the destination of the relationship specified by *`key`*. See "Manipulating Relationships" (page 45) for more information.

EOEditingContext: `deleteObject(EOEnterpriseObject`*`enterpriseObject`*`)`
Use this method to delete an enterprise object from an editing context. If your application connects to a relational database, this method usually results in the removal of a row or rows of data.

EOEditingContext: `lock()`

Obtains a lock on the receiver. To ensure the integrity of an editing context in multithreaded environments, you must lock an editing context before you invoke operations on it and unlock it afterwards. In a WebObjects application, if you use Session's `defaultEditingContext`, you do not need to lock and unlock it; it performs those operations internally.

EOEditingContext: `redo()`

Reapplies the last set of operations performed on enterprise objects in an editing context that were undone by the `undo` method. See "Undoing Changes" (page 78) for more information. You do not usually invoke this method explicitly. Rather, it is usually used only in a desktop application as the invocation target of the Edit menu's Redo item.

EOEditingContext: `revert()`

Use this method to restore an editing context to a stable state. See "Discarding Changes" (page 79).

EOCustomObject: `removeObjectFromBothSidesOfRelationshipWithKey(EORelationshipManipulation`*`object`*`, String`*`key`*`)`

Use this method to remove from the destination of the relationship specified by *`key`* the enterprise object specified by *`object`*. See "Manipulating Relationships" (page 45) for more information.

EOEditingContext: `saveChanges()`

Commits the changes made in an editing context to the data source. See "Flow of Data During a Save" (page 84) for detailed information on what occurs when this method is invoked.

EOCustomObject: `takeValueForKey(Object`*`value`*`, String`*`key`*`)`

This method sets the value of an enterprise object's property identified by *`key`* to *`value`*. This method is part of the key-value coding infrastructure of Enterprise Objects, in which an object's properties are accessed by key rather than directly as fields or through accessor methods. This is discussed in more detail in "Accessing an Enterprise Object's Data" (page 34).

EOCustomObject: `takeValueForKeyPath(Object`*`value`*`, String`*`keyPath`*`)`

This method sets the value of a property in an enterprise object that is the destination of an enterprise object's relationship. *`keyPath`* is a String of the form "relationship.property", in which "relationship" is the name of a relationship in the enterprise object on which this method is invoked and in which *`value`* mutates the property identified by the "property" portion of *`keyPath`*. This is discussed in more detail in "Accessing an Enterprise Object's Data" (page 34).

EOEditingContext: `undo()`

Reverses the last set of operations performed on enterprise objects in an editing context. See "Undoing Changes" (page 78) for more information.

EOEditingContext: `unlock()`

Releases a lock on the receiver. To ensure the integrity of an editing context in multithreaded environments, you must lock an editing context before you invoke operations on itm, and you must unlock it afterwards. In a WebObjects application, if you use Session's `defaultEditingContext`, you do not need to lock and unlock it; it performs those operations internally.

EOCustomObject: `validate`*`Key`*`(Object`*`value`*`, String`*`key`*`)`

This method is defined in EOCustomObject as `validateValueForKey`. You never directly invoke that method. Rather, you implement methods of the form `validate`*`Key`* in custom

enterprise object classes to validate certain properties. Those methods are then automatically invoked by `validateValueForKey`.

An implementation that validates the value of a `bathrooms` property of a Listing enterprise object would have the signature `publicObjectvalidateBedrooms(Object`*`value`*`,String`*`key`*`)`. See "Adding Validation" (page 41) for more information.

APPENDIX B

Principal Methods

Document Revision History

This table describes the changes to *WebObjects Enterprise Objects Programming Guide*.

Date	Notes
2005-08-11	Changed the title from "Enterprise Objects." Made minor bug fixes.
2005-01-11	Fixed line of sample code in "Connecting to a Database" chapter.
2004-12-02	Minor word change in preface to Figure 10-2.
2003-02-01	First version of *Enterprise Objects*.

Document Revision History

Glossary

access layer The classes in the package `com.webobjects.eoaccess`, which include the model-level classes EOEntity, EOAttribute, and EORelationship. You usually do not work with classes in this layer directly, but rather indirectly through an EOModel.

adaptor, database A mechanism that connects your application to a particular database server. For each type of server you use, you need a separate adaptor. WebObjects provides an adaptor for databases conforming to JDBC.

adaptor, WebObjects A process (or a part of one) that connects WebObjects applications to an HTTP server.

adaptor layer A sublayer of the access layer that provides classes that communicate directly with data sources.

application object An object (of the WOApplication class) that represents a single instance of a WebObjects application. The application object's main role is to coordinate the handling of HTTP requests, but it can also maintain application-wide state information.

attribute In Entity-Relationship modeling, an identifiable characteristic of an entity. For example, `lastName` can be an attribute of an Employee entity. An attribute typically corresponds to a column in a database table. See also "entity" (page 126); "relationship" (page 128).

batch faulting An feature that allows you to reduce round-trips to the database by firing multiple faults in a single fetch. See also "faulting" (page 126).

business logic The rules associated with the data in a database that typically encode business policies. An example is automatically adding late fees for overdue items.

CGI (Common Gateway Interface) A standard for interfacing external applications with information servers, such as HTTP or Web servers.

class In object-oriented languages such as Java, a prototype for a particular kind of object. A class definition declares fields and defines methods for all members of the class. Objects that have the same types of fields and have access to the same methods belong to the same class.

class property A field in an enterprise object that meets two criteria: It's based on an attribute in your model, and it can be fetched from the database. "Class property" can refer either to an attribute or to a relationship.

Cocoa An object-oriented application-development environment in Mac OS X.

column In a relational database, the dimension of a table that holds values for a particular attribute. For example, a table that contains employee records might have a column titled "LAST_NAME" that contains the values for each employee's last name. See also "attribute" (page 125).

component An object (of the WOComponent class) that represents a Web page or a reusable portion of one.

control layer The classes in the package `com.webobjects.eocontrol`, which include EOEditingContext and EOEnterpriseObject. You use classes in this layer to fetch, create, manage, and save persistent data to a data source.

data modeling The process of building a data model to describe the mapping between a relational database schema and an object model.

database server A data storage and retrieval system. Database servers typically run on a dedicated computer and are accessed by client applications over a network.

deep fetch An option available to fetch specifications that causes database fetches to occur against the root table and any leaf tables. Applicable to inheritance hierarchies.

deferred faults A special kind of fault that represents a number of other faults in a single object. See also "fault" (page 126); "faulting" (page 126).

delegation A way of extending a class in an object-oriented framework. In object-oriented design methodology, delegation is a form of class composition.

derived attribute An attribute in a data model that does not directly correspond to a column in a database. Derived attributes are usually calculated from a SQL expression.

dynamic element A dynamic version of an HTML element. WebObjects includes a list of dynamic elements with which you can build components.

enterprise object A Java object that conforms to the key-value coding protocol and whose properties (instance data) can map to stored data. An enterprise object brings together stored data with methods for operating on that data. It allows this data to persist in memory. See also "key-value coding" (page 127); "property" (page 128).

entity In Entity-Relationship modeling, a distinguishable object about which data is kept. An entity typically corresponds to a table in a relational database; an entity's attributes, in turn, correspond to a table's columns. An entity is used to map a relational database table to a Java class. See also "attribute" (page 125); "table" (page 128).

Entity-Relationship modeling A discipline for examining and representing the components and interrelationships in a database system. Also known as ER modeling, this discipline factors a database system into entities, attributes, and relationships. See also "object-relational mapping" (page 127).

EOModeler A tool used to create and edit models.

faulting A mechanism used by Enterprise Objects to increase performance whereby destination objects of relationships are not fetched until they are explicitly accessed.

fault A type of object in Enterprise Objects that represents a partially formed enterprise object instance. Faults are proxy or stand-in objects that provide performance benefits by delaying the retrieval of data in an enterprise object until it's absolutely needed.

fetch specification In Enterprise Objects applications, used to retrieve data from the database server into the client application, usually into enterprise objects.

fetch timestamp An attribute of an EOEditingContext that records the time of the most recent fetch of objects into that editing context.

flattened attribute An attribute that is added from one entity to another by traversing a relationship.

foreign key An attribute in an entity that gives it access to rows in another entity. This attribute must be the primary key of the related entity. For example, an Employee entity can contain the foreign key `deptID`, which matches the primary key in the entity Department. You can then use `deptID` as the source attribute in Employee and as the destination attribute in Department to form a relationship between the entities. See also "primary key" (page 127); "relationship" (page 128).

inheritance In object-oriented programming, the ability of a superclass to pass its characteristics (methods and fields) on to its subclasses, allowing subclasses to reuse these characteristics.

instance In object-oriented languages such as Java, an object that belongs to (is a member of) a particular class. Instances are created at runtime according to the specification in the class definition.

inverse relationship A relationship that goes in the reverse direction of another relationship. Also known as a back relationship.

Java Client A WebObjects development approach that allows you to create graphical user interface applications that run on the user's computer and communicate with a WebObjects server.

Java Foundation Classes A set of graphical user interface components and services written in Java. The component set is known as Swing.

JDBC An interface between Java platforms and databases.

join An operation that provides access to data from two tables at the same time, based on values contained in related columns.

key An arbitrary value (usually a string) used to locate a datum in a data structure such as a dictionary. See also "key-value coding" (page 127).

keypath A chain of keys supported by key-value coding that allows for the traversal of relationships between enterprise objects. See also "key-value coding" (page 127).

key-value coding The mechanism that allows the properties in enterprise objects to be accessed by name (that is, as key-value pairs) by other parts of the application. See also "keypath" (page 127).

locking A mechanism to ensure that data isn't modified by more than one user at a time and that data isn't read as it is being modified.

many-to-many relationship A relationship in which each record in the source entity may correspond to more than one record in the destination entity, and each record in the destination may correspond to more than one record in the source. For example, an employee can work on many projects, and a project can be staffed by many employees. In Enterprise Objects, a many-to-many relationship is composed of multiple relationships. See also "relationship" (page 128).

method In object-oriented programming, a procedure that can be executed by an object.

model An object (of the EOModel class) that defines, in Entity-Relationship terms, the mapping between enterprise object classes and the database schema. This definition is typically stored in a file created with the EOModeler application. A model also includes the information needed to connect to a particular database server.

notification A mechanism within Enterprise Objects that provides an asychronous communication infrastructure between objects.

object A programming unit that groups together a data structure (fields) and the operations (methods) that can use or affect that data. Objects are the principal building blocks of object-oriented programs.

object graph Refers to the graph of objects (especially enterprise object instances) that are contained by EOEditingContext objects.

object-relational mapping A system for transforming Entity-Relationship models to an object-oriented programming framework. Enterprise Objects performs object-relational mapping (mapping database structures to Java objects) with the help of Entity-Relationship models called EOModels. See also "Entity-Relationship modeling" (page 126).

prefetching A feature in Enterprise Object that allows you to suppress fault creation for an entity's relationships. Instead of creating faults, the relationship data is fetched when the entity is first fetched. See also "faulting" (page 126).

primary key An attribute in an entity that uniquely identifies rows of that entity. For example, the Employee entity can contain an `empID` attribute that uniquely identifies each employee.

property In Entity-Relationship modeling, an attribute or relationship. See also "attribute" (page 125); "relationship" (page 128).

prototype attribute A special type of attribute available in EOModeler to provide a template for creating attributes.

raw row fetching A possible option in a fetch specification that retrieves database rows without forming enterprise objects from those rows.

record The set of values that describes a single instance of an entity; in a relational database, a record is equivalent to a row.

referential integrity The rules governing the consistency of relationships.

reflexive relationship A relationship within the same entity; the relationship's source join attribute and destination join attribute are in the same entity.

relational database A database designed according to the relational model, which uses the discipline of Entity-Relationship modeling and the data design standards called normal forms.

relationship A link between two entities that's based on attributes of the entities. For example, the Department and Employee entities can have a relationship based on the `deptID` attribute as a foreign key in Employee, and as the primary key in Department (note that although the join attribute `deptID` is the same for the source and destination entities in this example, it doesn't have to be). This relationship would make it possible to find the employees for a given department. See also "foreign key" (page 126); "primary key" (page 127); "many-to-many relationship" (page 127); "to-many relationship" (page 128); "to-one relationship" (page 128).

relationship key A key (an attribute) on which a relationship joins.

request A message conforming to the Hypertext Transfer Protocol (HTTP) sent from a user's Web browser to a Web server to ask for a resource like a Web page. See also "response" (page 128).

request-response loop The main loop of a WebObjects application, which receives a request, responds to it, and awaits the next request.

response A message conforming to the Hypertext Transfer Protocol (HTTP) sent from the Web server to the user's Web browser that contains the resource specified by the corresponding request. The response is typically a Web page. See also "request" (page 128).

row In a relational database, the dimension of a table that groups attributes into records.

session A period during which access to a WebObjects application and its resources is granted to a particular client (typically a browser). Also an object (of the WOSession class) representing a session.

snapshotting Part of the Enterprise Objects optimistic locking mechanism in which snapshots of database rows in memory are compared with the data in the database.

table A two-dimensional set of values corresponding to an entity. The columns of a table represent characteristics of the entity and the rows represent instances of the entity.

target A blueprint for building a product from specified files in your project. It consists of a list of the necessary files and specifications on how to build them. Some common types of targets build frameworks, libraries, applications, and command-line tools.

template In a WebObjects component, a file containing HTML that specifies the overall appearance of a Web page generated from the component.

to-many relationship A relationship in which each source record has zero to many corresponding destination records. For example, a department has many employees.

to-one relationship A relationship in which each source record has one corresponding destination record. For example, each employee has one job title.

transaction A set of actions that is treated as a single operation that either succeeds completely (COMMIT) or fails completely (ROLLBACK).

uniquing A mechanism to ensure that, within a given context, only one object is associated with each row in the database.

update conflicts The realm of database applicataion development that deals with the problem of multiple users or processess accessing and updating the same set of data simultaneously.

update strategy A strategy for managing update conflicts. See also "update conflicts" (page 129).

validation A mechanism to ensure that user-entered data lies within specified limits.

WebObjects Builder A tool used to graphically edit WebObjects components.