What's New in Enterprise Objects Framework 3.0

This document describes changes made to the Enterprise Objects Framework product between release 2.2 and release 3.0. It tells you how to convert applications to 3.0, describes changes made to existing features, and then describes some new features you may want to start using in your application. It is organized into the following sections:

- Compatibility With Earlier Releases
- File Location Changes
- Changes to Java API
- Deprecated API
- Support for the OpenBase Lite Database
- New Convenience API
- Changes to EOModeler
- Binding to Complex Qualifiers
- Merging Object Changes
- Raw Row Fetching
- Support for Multi-Threaded Applications
- Changes to Key-Value Coding
- Changes to Enterprise Object Validation
- Changes to the Interface Layer

For a description of the changes to the WebObjects product, see "What's New in WebObjects 4.0."

Note: This document describes changes in both the Java and Objective-C implementations of Enterprise Objects Framework. Where Java and Objective-C method names are dissimilar, both method names are provided. For methods that take zero arguments, both languages use the same name. For single argument methods, the Java name is the Objective-C name minus the trailing colon (:).

Compatibility With Earlier Releases

Enterprise Objects Framework 3.0 is backward compatible with Enterprise Objects Framework 2.2; however, you must keep in mind the following:

- Release 3.0 is the first release of Enterprise Objects Framework that runs on Rhapsody and on Yellow Box for Windows NT. It does not run on OpenStep 4.2. Because of this change, the locations of Enterprise Objects Framework files have changed (see the section "File Location Changes").

- The file location changes require some changes to your Project Builder projects.
- Yellow Box uses a different version of the Java-wrapped APIs. The package names, class names, and some method names have changed. There is a script to help you convert your Java code.

If you have an existing application that uses both WebObjects and Enterprise Objects Framework (or that uses any of the Java APIs) and if you want to convert that application to the latest release, see the section "Converting an Existing WebObjects Application" in the document "What's New in WebObjects 4.0."

If you want to convert an application that doesn't use WebObjects, all you need to do is convert the project in Project Builder so that it points to the new locations for build tools. On Rhapsody systems, see the online document **/System/Developer/Makefiles/Conversion/DirectoryLayout/ConvertMakefilesReadMe.rtf**. On NT, see **NEXT_ROOT/Developer/Makefiles/Conversion/DirectoryLayout/ConvertMakefilesReadMe.rtf.**

File Location Changes

Enterprise Objects Framework 3.0 is the first release of Enterprise Objects Framework that runs on Rhapsody and on Yellow Box for Windows NT instead of OpenStep 4.2. Because of this change, the locations of Enterprise Objects Framework files have changed.

On Rhapsody, Enterprise Objects Framework files are installed in the **System** folder. On Windows NT, you still choose a folder in which to install the software, and the **NEXT_ROOT** environment variable points to that folder. The default has changed to **C:\Apple**.

The following table lists new directory names relative to the **System** folder or **NEXT_ROOT** and what each directory contains.

Location	Contains
Developer/Applications	The EOModeler application.
Developer/Examples/EnterpriseObjects	Enterprise Objects Framework examples.
Developer/Examples/WebObjects	WebObjects examples, including examples of using Enterprise Objects Framework with WebObjects.
Documentation/Developer	Developer documentation for Rhapsody, Yellow Box, Enterprise Objects Framework, and WebObjects.

Location	Contains
Library/Frameworks	Public frameworks such as EOAccess.framework.
Library/Executables	Framework DLLs (Windows NT systems only).
Library/Java	Java packages for Yellow Box, WebObjects, and Enterprise Objects Framework.
Local/Library	Prebuilt example frameworks and executables, sample EOModeler extension bundle, and 3.5 to 4.0 conversion scripts.

Changes to Java API

In Enterprise Objects Framework 3.0, the Java APIs have changed considerably. The changes to the Java APIs are summarized here:

- A two-letter prefix (EO) has been added to each Java class name. In almost all cases, the Java class name is now identical to its Objective-C counterpart.
- The Java package names have changed to the following:

  com.apple.yellow.eoaccess
  com.apple.yellow.eocontrol
  com.apple.yellow.eointerface
  com.apple.yellow.informixeoadaptor
  com.apple.yellow.odbceoadaptor
  com.apple.yellow.oracleeoadaptor
  com.apple.yellow.sybaseeoadaptor

  Note that the next.eo package has been split into two packages: eoaccess and eocontrol. Also note that Java APIs are now available for the EOInterface framework and the database-specific adaptor frameworks.
- The basic classes (for arrays, dictionaries, and data) have become more like their Foundation counterparts than their Java counterparts. For example, ImmutableVector is now named NSArray and responds to **count** instead of **size**. MutableHashtable is now named NSMutableDictionary and responds to **setObjectForKey** instead of **put**.

  Note that for numbers and strings, you still use the classes java.lang.Number and java.lang.String.

Also note that changing to Foundation-style methods for the dictionary class introduces a subtle change. The Java Hashtable classes take the arguments in the key-value order. For example, the **put** method takes the key and then the value. NSDictionary takes the value and then the key. The conversion scripts change the order of the arguments for you.

- DecimalNumber is no longer available. Use java.math.BigDecimal instead.
- CalendarDate is now named NSGregorianDate.
- The root object is now com.apple.yellow.foundation.NSObject.
- Delegate interfaces are now declared as inner classes (within the appropriate class). For example, EditingContextDelegates is now EOEditingContext.Delegate.

If you have existing Java code that you want to convert, see the document "What's New in WebObjects 4.0."

Deprecated API

Enterprise Objects Framework 3.0 introduces API improvements that require deprecating existing API. You'll still be able to use deprecated API, but you'll receive a warning at runtime. The following tables summarize the deprecated API.

Fetch Specification Hint Keys

Old API	New API
EOPrefetchingRelationshipHintKey (Objective-C) FetchSpecification.PrefetchingRelationshipHintKey and DatabaseContext.PrefetchingRelationshipHintKey (Java)	EOFetchSpecification's accessor methods, **prefetchingRelationshipKeyPaths** and **setPrefetchingRelationshipKeyPaths:**
EOFetchLimitHintKey (Objective-C) DatabaseContext.FetchLimitHintKey (Java)	EOFetchSpecification's accessor methods, **fetchLimit** and **setFetchLimit:**
EOPromptAfterFetchLimitHintKey (Objective-C) DatabaseContext.PromptAfterFetchLimitHintKey (Java)	EOFetchSpecification's accessor methods, **promptsAfterFetchLimit** and **setPromptsAfterFetchLimit:**

EOClassDescription

Old API	New API
delegate and **setDelegate:** class (Objective-C) or static (Java) methods	**classDelegate** and **setClassDelegate:** class (Objective-C) or static (Java) methods.

EOModelGroup

Old API	New API
delegate and **setDelegate:** class methods (Objective-C only)	**classDelegate** and **setClassDelegate:** class methods. Note that the corresponding Java static methods have always been named **classDelegate** and **setClassDelegate**.

NSObject Additions (Objective-C)
EOCustomObject (Java)

Old API	New API
useStoredAccessor class (Objective-C) or static (Java) method	This method still exists, but the default is now YES or **true**.
flushClassKeyBindings	None. Use **flushAllKeyBindings** instead.

EOUndoManager

Old API	New API
EOUndoManager class	NSUndoManager in Foundation. The EOUndoManager header file is no longer included in EOControl. If you want to continue using EOUndoManager, you'll have to include **EOControl/EODeprecated.h**, where it's now defined. EOUndoManager is no longer available in Java at all.

EOLoginPanel

Old API	New API
runPanelForAdaptor:validate: (Objective-C only) There wasn't an equivalent Java class in WebObjects 3.5.	**runPanelForAdaptor:** **validate:** **allowsCreation:** (Objective-C) **runPanelForAdaptor** (Java)

Support for the OpenBase Lite Database

Enterprise Objects Framework 3.0 adds support for a new database, OpenBase Lite, which ships with Enterprise Objects Framework 3.0 as an unsupported demo.

If you install OpenBase Lite, it installs the following two frameworks in **Local/Library/Frameworks:**

- OpenBaseLiteAPI.framework, the proprietary database implementation
- OpenBaseLiteEOAdaptor.framework, the OpenBase Lite adaptor

On NT, OpenBase Lite also installs the following DLLs in **Local/Library/Executables:**

- OpenBaseLiteAPI.dll
- OpenBaseLiteEOAdaptor.dll

Additionally, a preloaded OpenBase Lite database for the Movies and Rentals models is provided as a part of the installation process. All of the examples run against this database out-of-the-box (without any configuration).

OpenBase Lite is intended to be used as a single-user, single-machine database convenient for development, not as a deployment database. Only one process can access an OpenBase Lite database at a time (other processes are locked out until the first process releases the OpenBase Lite database). For more information on OpenBase Lite or if you are interested in the full-featured, client-server OpenBase database (as opposed to the bundled "Lite" version), contact OpenBase International:

OPENBASE INTERNATIONAL LTD.
58 Greenfield Road
Francestown, NH 03043 USA
TEL: (603) 547-8404
FAX: (603) 547-2423
e-mail: info@openbase.com
http://www.openbase.com

New Convenience API

Enterprise Objects Framework 3.0 introduces new API to facilitate common programming tasks. Tasks that used to take several lines of code now only take one. The following sections describe the new convenience API.

EOUtilities

Most of the new convenience API is implemented in EOUtilities. In Objective-C, EOUtilities is a category on EOEditingContext provided in EOAccess. In Java, it's a new class called EOUtilities in EOAccess. The Objective-C and Java methods work the same way, but you invoke them differently. For example, compare the invocations for the following method:

Objective-C

```
[editingContext objectsForEntityNamed:entityName];
```

Java

```
EOUtilities.objectsForEntityNamed(editingContext, entityName);
```

Both versions of the method require an editing context into which the objects should be fetched. In Objective-C, the editing context is the receiver of the message. In Java, the editing context must be passed as an argument.

Note: The Objective-C source code for EOUtilities is available as an example. On Rhapsody systems, see **/System/Developer/Examples/EnterpriseObjects/Sources/EOUtilities.** On NT, see **NEXT_ROOT/Developer/Examples/EnterpriseObjects/Sources/EOUtilities.**

The complete documentation for the Objective-C EOUtilities methods are documented in the "EOEditingContext Additions" class specification (EOAccess). The corresponding Java EOUtilities API documentation is available in the EOUtilities class specification (EOAccess). The following tables summarize the EOUtilities methods.

Note: All the Java EOUtilities methods are static methods.

Fetching Enterprise Objects

objectsForEntityNamed:	Fetches and returns the enterprise objects associated with the specified entity.
objectsOfClass:	Fetches and returns the enterprise objects associated with the specified class. Raises or throws an exception if more than one entity for the class exists.
objectsWithFetchSpecificationNamed:entityNamed:bindings: (Objective-C) objectsWithFetchSpecificationAndBindings (Java)	Fetches and returns the enterprise objects retrieved with the specified fetch specification and bindings. (For more information on bindings, see "Binding to Complex Qualifiers.")

Fetching Enterprise Objects

objectWithFetchSpecificationNamed:entityNamed:bindings: (Objective-C) objectWithFetchSpecificationAndBindings (Java)	Same as the corresponding **objects...** method (immediately above), except this method raises or throws an exception unless exactly one object is retrieved.
objectsForEntityNamed:qualifierFormat: (Objective-C) objectsWithQualifierFormat (Java)	Creates a qualifier with the provided format string and returns matching enterprise objects.
objectForEntityNamed:qualifierFormat: (Objective-C) objectWithQualifierFormat (Java)	Same as the corresponding **objects...** method (immediately above), except this method raises or throws an exception unless exactly one object is retrieved.
objectsMatchingValue:forKey:entityNamed: (Objective-C) objectsMatchingKeyAndValue (Java)	Creates an EOKeyValueQualifier with the specified key and value and returns matching enterprise objects.
objectMatchingValue:forKey:entityNamed: (Objective-C) objectMatchingKeyAndValue (Java)	Same as the corresponding **objects...** method (immediately above), except this method raises or throws an exception unless exactly one object is retrieved.
objectsMatchingValues:entityNamed: (Objective-C) objectsMatchingValues (Java)	Creates EOKeyValueQualifiers for each key-value pair in the specified dictionary, ANDs these qualifiers together into an EOAndQualifier, and returns matching enterprise objects.
objectMatchingValues:entityNamed: (Objective-C) objectMatchingValues (Java)	Same as the corresponding **objects...** method (immediately above), except this method raises or throws an exception unless exactly one object is retrieved.
objectWithPrimaryKeyValue:entityNamed: (Objective-C) objectWithPrimaryKeyValue (Java)	Fetches and returns the enterprise object identified by the specified primary key value. For use only with enterprise objects that have non-compound primary keys. Raises or throws an exception unless exactly one object is retrieved.
objectWithPrimaryKey:entityNamed: (Objective-C) objectWithPrimaryKey (Java)	Fetches and returns the enterprise object identified by the specified primary key dictionary. Raises or throws an exception unless exactly one object is retrieved.

Fetching Raw Rows

(For information on raw rows, see "Raw Row Fetching.")

rawRowsForEntityNamed:qualifierFormat: (Objective-C) rawRowsWithQualifierFormat (Java)	Creates a qualifier for the specified entity and with the specified qualifier format and returns matching raw row dictionaries.
rawRowsMatchingValue:forKey:entityNamed: (Objective-C) rawRowsMatchingKeyAndValue (Java)	Creates an EOKeyValueQualifier with the specified key and value and returns matching raw rows.
rawRowsMatchingValues:entityNamed: (Objective-C) rawRowsMatchingValues (Java)	Creates EOKeyValueQualifiers for each key-value pair in the specified dictionary, ANDs these qualifiers together into an EOAndQualifier, and returns matching raw rows.
rawRowsWithSQL:modelNamed: (Objective-C) rawRowsForSQL (Java)	Evaluates the specified SQL and returns the resulting raw rows.
rawRowsWithStoredProcedureNamed:arguments: (Objective-C) rawRowsForStoredProcedureNamed (Java)	Executes the specified stored procedure with the provided arguments and returns the resulting raw rows.
executeStoredProcedureNamed:arguments: (Objective-C) executeStoredProcedureNamed (Java)	Executes the specified stored procedure with the provided arguments. Returns the stored procedure's return values (if any). Use only with stored procedures that don't return results rows.
objectFromRawRow:entityNamed: (Objective-C) objectFromRawRow (Java)	Fetches and returns the object corresponding to the specified raw row (using EOEditingContext's **faultForRawRow:entityNamed:** in Objective-C or **faultForRawRow** in Java). This method can only be used on raw rows that include the row's primary key.

Accessing the Enterprise Objects Framework Stack

databaseContextForModelNamed:	Returns the database context used to service the specified model.
connectWithModelNamed:connectionDictionaryOverrides: (Objective-C) connectWithModelNamed (Java)	Connects to the database using the connection information in the specified model and the provided overrides dictionary. This method facilitates per-session database logins in WebObjects applications. Typically, you'd put a login name and password in the overrides dictionary and otherwise use the values in the model's connection dictionary.

Accessing Enterprise Object Data

primaryKeyForObject:	Returns the primary key dictionary for the specified enterprise object.
destinationKeyForSourceObject:relationshipNamed: (Objective-C) destinationKeyForSourceObject (Java)	Returns the foreign key for the rows at the destination entity of the specified relationship.
localInstanceOfObject:	Translates the specified enterprise object from one editing context to another.
localInstancesOfObjects:	Translates the specified enterprise objects from one editing context to another.

Accessing Model Information

modelGroup	Returns the model group associated with the editing context's root object store, an EOObjectStoreCoordinator.
entityNamed:	Returns the entity with the specified name. Raises or throws an exception if the specified entity can't be found.
entityForClass:	Returns the entity associated with the specified class. Raises or throws an exception if the specified entity can't be found or if more than one entity is associated with the class.
entityForObject:	Returns the entity associated with the provided enterprise object. Raises or throws an exception if the specified entity can't be found.

Supporting Convenience Methods

The following tables describe the remainder of the convenience API.

EOQualifier

qualifierToMatchAllValues:	Takes a dictionary of search criteria, from which the method creates EOKeyValueQualifiers (one for each dictionary entry). The method ANDs these qualifiers together, and returns the resulting EOAndQualifier.

EOQualifier

qualifierToMatchAnyValues:	Takes a dictionary of search criteria, from which the method creates EOKeyValueQualifiers (one for each dictionary entry). The method ORs these qualifiers together, and returns the resulting EOOrQualifier.

EODatabaseContext

forceConnectionWithModel: connectionDictionaryOverrides: editingContext: (Objective-C) forceConnectionWithModel: (Java)	Added to facilitate per-session database logins in WebObjects applications, this method connects to the database using the provided model's connection information and the provided overrides dictionary.

EOModelGroup

storedProcedureNamed:	Returns the stored procedure identified by the provided name.

Changes to EOModeler

EOModeler in release 3.0 has the following changes:

- Improvements in the way you create a model file and database from scratch
- Support for prototype attributes, which you can use to quickly set up attributes
- The ability to create and store complex queries (or EOFetchSpecifications)
- A different bundle loading procedure

Each is discussed in more detail in the following sections.

Improved Database Creation Support

The most common way to create a model file is to use information stored in an already-created database. Sometimes, however, it's useful to create the

model file first and use that model to create the empty database and generate the database tables. In release 3.0, EOModeler contains improvements that make it easier to create and delete the database:

- The adaptor login panels now allow you to create a new database or user.
- The SQL Generation panel now has options for creating a database and deleting a database.
- EOModeler now supports the definition of prototype attributes that you can use to quickly set up attributes in the new model (see the section "Prototype Attributes").

To create a model file and its database from scratch, do the following:

1. Choose New from the Model menu.
2. Select an adaptor.
3. Enter connection information for the new database (user name, password, and so on).

   Once the database is created, users will log into the database with this information.
4. Click Create.
5. Provide the administrator connection information.

   The information you provide in this panel will be used to log into the database server to create the new database. The login information you provide must be for an account that has database creation permissions.
6. Click Finish.
7. Define the entities and attributes that you want the model to represent.
8. Choose Generate SQL from the Property menu.
9. Select the Create Tables option.
10. Click the Execute SQL button.

API for Database Creation and Deletion

This section describes new API that supports the database creation feature in EOModeler.

EOLoginPanel

administrativeConnectionDictionaryForAdaptor:	Returns the administrative connection dictionary, which contains the values (user name and password) needed to connect to the database server as the administrator.
runPanelForAdaptor:validate:allowsCreation: (Objective-C) runPanelForAdaptor (Java)	Replaces **runPanelForAdaptor:validate:**.

EOAdaptor

createDatabaseWithAdministrativeConnectionDictionary: and dropDatabaseWithAdministrativeConnectionDictionary:	Creates or deletes the database specified in the receiver's connection dictionary, connecting to the database server using the information in the provided administrative connection dictionary.

EOSQLExpression

dropDatabaseStatementsForConnectionDictionary: administrativeConnectionDictionary: (Objective-C class method) dropDatabaseStatementsForConnectionDictionary (Java static method)	Generates the SQL statements to delete the database (or user for Oracle). Note that the statements generated only work if you are connected with administrative privileges.
createDatabaseStatementsForConnectionDictionary: administrativeConnectionDictionary: (Objective-C class method) createDatabaseStatementsForConnectionDictionary (Java static method)	Generates the SQL statements that will create a database (or user for Oracle) that can be accessed by the provided connection dictionary and administrative connection dictionary. Note that the statements generated only work if you are connected with administrative privileges.
schemaCreationStatementsForEntities:options: (Objective-C class method) schemeCreationStatementsForEntities (Java static method)	This method already exists, but takes these new keys to its option dictionary: EOCreateDatabaseKey EODropDatabaseKey

Option Keys for EOSQLExpression's schema creation methods

Objective-C For use with **schemaCreationScriptForEntities:options:** and **schemaCreationStatementsForEntities:options:**	Java For use with **schemaCreationScriptForEntities** and **schemaCreationStatementsForEntities**
EOCreateDatabaseKey	EOSQLExpression.CreateDatabaseKey
EOCreatePrimaryKeySupportKey	EOSQLExpression.CreatePrimaryKeySupportKey
EOCreateTablesKey	EOSQLExpression.CreateTablesKey
EODropDatabaseKey	EOSQLExpression.DropDatabaseKey
EODropPrimaryKeySupportKey	EOSQLExpression.DropPrimaryKeySupportKey
EODropTablesKey	EOSQLExpression.DropTablesKey
EOForeignKeyConstraintsKey	EOSQLExpression.ForeignKeyConstraintsKey
EOPrimaryKeyConstraintsKey	EOSQLExpression.PrimaryKeyConstraintsKey

Prototype Attributes

To allow easier model creation from scratch, EOModeler now supports the concept of prototype attributes. Prototype attributes are just what they sound like — special EOAttributes from which other EOAttributes derive their settings. A prototype can specify any of the characteristics you normally define for an attribute. When you create an attribute, you can associate it with one of these prototypes, and the attribute's characteristics are then set from the prototype definition.

For example, suppose your adaptor contains a date prototype that defines the value class to be NSCalendarDate and the external type to be DATE. When you create an attribute and associate it with this date prototype, the attribute's value class is dynamically resolved to NSCalendarDate and its external type is dynamically resolved to DATE. If any of the prototype information is not correct for your attribute, you can override it. Simply set the property of the attribute to the correct value. The remaining attribute properties will still dynamically resolve to the values set in the prototype.

To associate an attribute with a prototype, use the table mode of the Model Editor. In the row for your attribute, choose a prototype from the combo box in the Prototype column. (If EOModeler doesn't display the Prototype column, activate it from the Columns pull-down menu.) The prototypes in the combo box come from three places:

1. An EOEntity named **EO<adaptor-name>Prototypes**, where **<adaptor-name>** is the name of the adaptor for your model (EOOraclePrototypes, for example)

2. An EOEntity named **EOPrototypes**

3. The adaptor for your model

So to create your own prototype, create a prototype entity—an entity named either **EO<adaptor-name>Prototypes** or **EOPrototypes**—and add an attribute to it. Note that the **EO<adaptor-name>Prototypes** and **EOPrototypes** entities can be defined in the current model or in another model in the model group (all the models in your project are typically a part of the same model group).

When resolving a prototype name, Enterprise Objects Framework looks for prototypes in **EO<adaptor-name>Prototypes**, then in **EOPrototypes**, and finally in the adaptor for your model. This search path allows you to override the prototypes provided by each adaptor. Furthermore, if you don't want to use the adaptor-defined prototypes at all, you can hide them. Create an entity named **EOPrototypesToHide**. For each prototype you want to hide, create an attribute with that name; you don't need to specify other attribute properties.

API for Prototype Attributes

The following tables describe the API that has been added to support prototype attributes.

EOAdaptor

prototypeAttributes	Returns an array of prototype attributes specific to the adaptor.

EOModel

prototypeAttributeNamed:	Returns the prototype attribute identified by the specified name or **nil** or **null** if there isn't one by that name. Looks first for the prototype in the prototypes entity named EO<adaptorName>Prototypes, then in prototypes entity named EOPrototypes, and then in the list of prototypes provided by the adaptor itself (using EOAdaptor's **prototypeAttributes** method).
availablePrototypeAttributeNames	Returns an array of the names of all the prototypes available to the model.

EOAttribute

overridesPrototypeDefinitionForKey:	Returns NO or **false** if the requested key gets its value from the prototype attribute, or YES or **true** if the attribute overrides the prototype information for that key ("columnName", "valueClass", or so on). If the attribute doesn't have a prototype, this method returns NO or **false**.
setPrototype:	Sets the prototype attribute from which the receiver derives its settings. Invoking this method overrides any existing settings in the receiver.
prototypeName	Returns the name of the attribute's prototype, or **nil** or **null** if the attribute has none.
prototype	Returns the attribute's prototype, or **nil** or **null** if the attribute has none.

Query Builder

You can now use EOModeler to create a query, name it, and store it in the model file. To perform a query in Enterprise Objects Framework, you create an EOFetchSpecification object, which has associated with it an entity, a qualifier, a sort ordering for the fetched objects, and several other options. In previous releases, creating a fetch specification was usually done programmatically and could be quite complex and error prone. EOModeler now has a user interface that allows you to create the fetch specification, associate it with an entity, build the qualifier graphically, and specify the sort ordering and any other options, and test the complete fetch specification by dragging it into the Data Browser.

To create a query, select an entity and then click the New Fetch Specification button (second from the right in the tool bar).

API for Query Builder

This section describes API that has been added to support storing an EOFetchSpecification in the EOModel.

- A new class, EOQualifierVariable, defines objects that serve as placeholders in the qualifier. When you create a qualifier programmatically, you typically do something like this:

  In Objective-C

  ```
  aQual = [EOQualifier
    qualifierWithQualifierFormat:"dateReleased = %@", aDate];
  ```

  In Java

```
aQual = EOQualifier.qualifierWithQualifierFormat(
  "dateReleased = %@", aDate);
```

where **aDate** is a variable that contains the actual date you want to query upon. When you store the qualifier in an EOModel, there is no way to know the actual value to query upon or the variable that will contain that value. The EOQualifierVariable object acts as a placeholder for the actual variable that will represent the right side of the expression. You specify an EOQualifierVariable by using a $, as in the following:

```
dateReleased = $aDate
```

For more information, see the section "Binding to Complex Qualifiers."

- Methods have been added to EOEntity to retrieve the fetch specification by name. When you create an EOFetchSpecification programmatically, you pass it the entity with which it should be associated. When you create an EOFetchSpecification in EOModeler, you select the entity that it should be associated with and you assign a name to the fetch specification. The EOEntity now keeps a list of all fetch specifications associated with it and can retrieve them by name. Note that EOFetchSpecifications don't know their names; rather the owning entity keeps a fetch specification-to-name mapping.

The following tables describe the new API in more detail.

EOQualifierVariable (New Class)

variableWithKey: class method (Objective-C) EOQualifierVariable(String) constructor (Java)	Creates and returns a new EOQualifierVariable object with the specified name. For example, if your qualifier is "dateReleased = $aDate", then this method would be invoked with the key "aDate".
initWithKey: (Objective-C only)	Initializes a newly created EOQualifierVariable object for the specified name.
key	Returns the name with which the receiver is associated.

EOModelGroup

fetchSpecificationNamed:entityNamed: (Objective-C) fetchSpecificationNamed (Java)	Returns the fetch specification identified by the provided name from the specified entity.

EOEntity

fetchSpecificationNames	Returns an alphabetically sorted array of names of the entity's fetch specifications.
fetchSpecificationNamed:	Returns the fetch specification named with the provided name.
addFetchSpecification:withName: (Objective-C) addFetchSpecification (Java)	Adds the fetch specification and associates the provided name with it.
removeFetchSpecificationNamed:	Removes the fetch specification referred to by the provided name.

EODatabaseDataSource

initWithEditingContext: entityName: fetchSpecificationName: (Objective-C) EODatabaseDataSource(EOEditingContext, String, String) (Java)	Initializes a new data source with an editing context, an entity name, and a named fetch specification. If the provided fetch specification name is **nil** or **null**, this method creates a new fetch specification that fetches all of the entities objects and assigns this fetch specification to the new data source.
setFetchSpecificationByName:	Sets the database data source's fetch specification to the one identified by the provided name. This method is an alternative to **setFetchSpecification:**
fetchSpecificationName	Returns the name of the data source's fetch specification or **nil** or **null** if the data source's fetch specification doesn't have a name.

EOModeler Bundle Loading

EOModeler's Preferences panel now allows you to specify file system locations in which to look for EOModeler bundles. EOModeler loads all the **.EOMbundle** files it finds in those locations. In the past, you specified the bundles you wanted to load with a user default. This same default still exists (and is used by the Preferences panel), but its value has a slightly different meaning. Whereas you used to specify the full path to each bundle to load, you now specify paths in which to search for bundles.

Binding to Complex Qualifiers

In Enterprise Objects Framework 3.0, you can bind user interface elements directly to variables in a complex qualifier that you created using the new Query Builder in EOModeler.

For example, suppose you want to create a WebObjects application that allows users to perform a complex query on the Movies entity in the Movies database. Suppose you want to allow users to query on the title, the date released, and the studio. You could use the **queryMatch**, **queryMin**, and **queryMax** support in display group to easily construct such query. For example:

```
(title = $title) AND (dateReleased = $date) AND (studio = $studio)
```

However, **queryMatch** support is limited to ANDed criteria; it isn't sufficient for more complex queries such as:

```
(title = $title) OR (dateReleased = $date) OR (studio = $studio)
```

For this qualifier you could define the qualifier in EOModeler and then bind to it in your user interface. In general, you would set this up by following these steps:

1. In EOModeler, open the Movies model file and select the Movies entity.

2. Create a fetch specification associated with the Movies entity by clicking the New Fetch Specification button.

3. Use Query Builder's user interface to set up a query on the **title**, **dateReleased**, and **studio** attributes of the entity (see the section "Query Builder"). On the right side of each expression, use a $ syntax to denote the qualifier variable. For example, your fetch specification might look like this:

```
(title = $title) OR (dateReleased = $date) OR (studio = $studio)
```

Depending on the type of graphical user interface you build, you access the fetch specification's query bindings differently. In WebObjects Builder, you access the query bindings in the following way:

4. Use WebObjects builder to create a component that allows the user to enter the query criteria. You might create text fields for the title and date released and a pop-up list for the studio, for example.

5. Drag the fetch specification from EOModeler into your component. This has the effect of creating a new display group for your specification's entity.
6. Choose "Add and Configure."
7. Configure the new display group, setting its fetch specification to the one you defined in your model.
8. In WebObjects Builder, bind the user interface elements to the **queryBindings.title**, **queryBindings.date**, and **queryBindings.studio** keys of your display group (**movieDisplayGroup**, for example).

In Interface Builder, the steps are similar except that you bind the user interface elements to **@bindings.title**, **@bindings.date**, and **@bindings.studio** keys of your display group. The @bindings syntax represents the value associated with the named qualifier variable.

Qualifier bindings are also useful when you want to bind a value to more than one qualifier component as in the following:

```
(title like $searchString) OR (description like $searchString)
```

In this example, **searchString** could contain a user-provided keyword surrounded with wildcard characters.

The following tables describe the API added to support binding to complex qualifiers.

EOFetchSpecification

fetchSpecificationWithQualifierBindings:	Returns a new fetch specification created by resolving the bindings for the fetch specification's qualifier using the bindings dictionary passed as an argument to this method.
requiresAllQualifierBindingVariables and setRequiresAllQualifierBindingVariables:	Returns or sets whether a missing binding is ignored or whether it raises or throws an exception. If **requiresAllQualifierBindingVariables** is YES or **true**, a missing binding raises or throws an EOQualifierVariableSubstitutionException during variable substitution. If NO or **false**, any nodes for which there are no bindings should be pruned from the qualifier. The default is NO or **false**.

EOQualifier

qualifierWithBindings:requiresAllVariables: (Objective-C) qualifierWithBindings(NSDictionary, Boolean) (Java)	Returns a new qualifier created by substituting all EOQualifierVariables with the values contained in the provided argument. The object passed to this method is an NSDictionary containing the values to which the EOQualifierVariables are bound. (Typically the values are those entered by the user in the user interface fields.) If the second argument (a boolean value) is YES or **true**, then the new qualifier requires all its variables (meaning that if a value is not found for a variable in the provided object, an EOQualifierVariableSubstitutionException is raised or thrown). If the second argument is NO or **false**, then the new qualifier doesn't require all its variables; and if any variable is not found in the bindings dictionary (that is, the user has left that field blank), the node containing that variable is simply pruned from the qualifier tree.
bindingKeys	Returns an array of strings representing the binding keys for the EOQualifierVariables contained in this qualifier. For example, if you have a qualifier such as "dateReleased = $date", this method returns an array containing the single string "date". Multiple occurrences of the same variable only appear once in this list.
keyPathForBindingKey:	Returns the lefthand side of a qualifier binding. For example, if you have a qualifier such as "movie.dateReleased = $date", this method returns "movie.dateReleased" for the key "date".

EOClassDescription

defaultFormatterForKeyPath:	Similar to **defaultFormatterForKey:**, except this method takes a key path (roles.roleName, for example).

EODataSource

qualifierBindingKeys	Returns an array of strings representing all of the binding keys from the fetch specification's qualifier and the data source's auxiliary qualifier.
qualifierBindings and setQualifierBindings:	Returns or sets the NSDictionary containing the bindings that will be used for variable replacement on the fetch specification's qualifier and the auxiliary qualifier before the fetch is executed. The keys to the dictionary are the keys returned by **qualifierBindingKeys**. The values are typically the values that the application's user entered through the user interface.

New Exception	
EOQualifierVariableSubstitutionException	Raised or thrown when the EOFetchSpecification's **requiresAllQualifierBindingVariables** is YES or **true** and a value for one of the variables in the qualifier is missing from the bindings object.

Merging Object Changes

A change has been made to how peer editing contexts (that is, editing contexts that share an object store) handle database modifications. WebObjects applications often use peer editing contexts since each session has an editing context and an application can run multiple sessions.

When an application has more than one editing context, it's possible for each editing context to have its own in-memory copy of the same enterprise object. Suppose the users of the editing contexts each make changes to their copies of that object. Which change wins? In prior releases of Enterprise Objects Framework, the editing context that saved its changes first overwrote the changes made by the second editing context.

For example, suppose two editing contexts each have a copy of a Customer object. Suppose the user of the first editing context changes the customer's address and at the same time, the user of the second editing context changes the customer's phone number. Then suppose the first user saved the change to the address. In previous releases of Enterprise Objects Framework, the change to the phone number was lost. When the first editing context's changes were saved, the customer object for the second editing context was turned back into a fault, losing any uncommitted changes in that object. The next time the customer was accessed, it was refetched from the object store, and the phone number retrieved from the database was the one committed by the first editing context.

In Enterprise Objects Framework release 3.0, the default behavior is not to lose the changed phone number. EOEditingContexts now keep track of uncommitted changes. When an object (such as the customer object) is refaulted in the second editing context (due to the EOObjectsChangeInStoreNotification posted after the first editing context saved changes), the second editing context checks to see if it has uncommitted changes for that object. If it does, it fires the fault (refetching the object from the database) and merges the uncommitted changes with the information retrieved from the database. Thus, in the example above, the second editing context would refetch the customer from the in-memory snapshot, thereby picking up

the change to the address, and then overwrite the phone number retrieved from the database with the changed phone number.

Suppose, however, that the two users each change the same field. For example, suppose the first user changes the last name to “Primero” but doesn’t save the change immediately, and the second user changes the last name to “Segundo” and immediately saves the change (before the first user saves). The first editing context refaults and refetches the customer object containing the last name “Segundo” and then overwrites that field with its uncommitted change of “Primero.” You may not want this to happen. You may want the first user to decide whether to keep his uncommitted change or to replace it with the change made by the second user. If so, your editing context delegate should implement the method **editingContext:shouldMergeChangesForObject:** (**editingContextShouldMergeChangesForObject** in Java).

Remember that in a WebObjects application, the first editing context does not pick up the change until the first user submits a request. That is, the second user changes the last name to “Segundo” and clicks the submit button to save the change to the database. At this point, the first editing context refaults the customer object but has no way to update the user’s browser with information that the object has changed until that user submits another request. This request is likely to be the action of submitting the last name change to “Primero.” Thus, your application will probably have to display an alert component informing its user that the customer value in the database has changed and then prompt to see if its user wants to commit or discard his own changes.

The following tables describe the API added to support the change to the way peer editing contexts behave.

EOEditingContext Delegatation

editingContext:shouldMergeChangesForObject: (Objective-C) editingContextShouldMergeChangesForObject (Java)	When an EOObjectsChangedInStoreNotification is received, the editing context sends this message to its delegate once for each of the objects that has both uncommitted changes and an update from the EOObjectStore. This method is invoked before any updates actually occur. If this method returns YES or **true**, all of the uncommitted changes should be merged into the object after the update is applied, in effect preserving the uncommitted changes (the default behavior). The delegate method **editingContext:shouldInvalidateObject:globalID:** (**editingContextShouldInvalidateObject** in Java) will not be sent for the object in question. If this method returns NO or **false**, no uncommitted changes are applied. Thus, the object is updated to reflect the values from the parent object store exactly. This method should not make any changes to the object since it is about to be invalidated.
editingContextDidMergeChanges:	Invoked once after a batch of objects has been updated from the database. A delegate might implement this method to define custom merging behavior, most likely in conjunction with **editingContext:shouldMergeChangesForObject:** (**editingContextShouldMergeChangesForObject** in Java). It is safe for this method to make changes to the objects in the editing context.

EOEnterpriseObject Informal Protocol (Objective-C) or Interface (Java)

changesFromSnapsot:	Returns a dictionary similar to a snapshot except that the dictionary contains only those keys that refer to uncommitted changes in the enterprise object relative to the provided snapshot argument. For to-many keys, the uncommitted value is an array of two arrays: uncommitted additions and uncommitted deletions.
reapplyChangesFromDictionary:	Similar to **takeValuesFromDictionary:** but the dictionary argument can include values for to-many relationships as described above in the description of **changesFromSnapshot:**. Attribute and to-one keys refer to values that should replace the enterprise object's current value. An instance of EONull is used in the dictionary argument as a placeholder for **nil** or **null**.

Raw Row Fetching

When you perform a fetch in an Enterprise Objects Framework application, the information from the database is fetched and stored in a graph of enterprise objects. This object graph provides many advantages, but it can be large and complex. If you're creating a simple application, you may not need all of the benefits of the object graph. For example, a WebObjects application that merely displays information from a database without ever performing database updates and without ever traversing relationships might be just as well served by fetching the information into a set of dictionaries rather than a set of enterprise objects.

More specifically, suppose you want to display the first name, last name, and the department for a set of employees. Using objects, you would bind Employee's **firstName**, **lastName**, and **department.name** keys to your user interface. This configuration requires fetching of all of the attributes in an Employee entity—the ones you want to display (**firstName** and **lastName**) as well as the ones you don't (**salary**, **birthDate**, **address**, and so on, for example). In addition, this configuration requires faulting in (or perhaps prefetching) all of the related Department objects. Again you fetch all the Department attributes, those you want to display (**departmentName**) as well as those you don't (**budget**, **location**, and so on). In addition to fetching a large amount of data that your application doesn't use, this object-based fetch incurs the additional overhead of creating real enterprise objects from the returned data and of uniquing those objects in the EOEditingContext.

In this kind of display-only scenario, it might be preferable to fetch only the attributes that you need, and to fetch them as lightweight, non-uniqued, rows. In this example, you could fetch only the **firstName**, **lastName**, and **department.name** for each employee. In addition to fetching less data, you'd also fetch with one trip to the database instead of two (one for Employee objects and one for the related Departments).

Enterprise Objects Framework 3.0 supports this concept of a simplified fetch, called *raw row* fetching. In raw row fetching, each row from the database is fetched into an NSDictionary object.

To set up an application to perform raw row fetching, create an EOFetchSpecification, and send it a **setFetchesRawRows:YES** (or **setFetchesRawRows(true)** in Java) message. By default, the keys in the raw row dictionaries are the attribute names as given by the EOEntity's **attributesToFetch** method.

If you want more control over the attributes that are fetched for the raw row, use the **setRawRowKeyPaths:** method to specify the attribute paths you want. The key paths you provide can be simple attribute keys, such as **title**, as well as key paths, such as **studio.name**. After the fetch, each row is returned as a separate dictionary whose keys are the key paths you specified. If you use **setRawRowKeyPaths:**, you don't have to invoke **setFetchesRawRows:**; it's automatic.

When you use raw row fetching, you lose some important features:

- The NSDictionary objects are not uniqued.
- The NSDictionary objects aren't tracked by an editing context.
- You can't access to-many relationship information. (To access to-one relationship information, you use key paths such as "**movie.dateReleased**".)

Should you fetch a row into an NSDictionary and later want to fetch the corresponding enterprise object, send **faultForRawRow:entityNamed:editingContext:** (or **faultForRawRow** in Java) to the EOEditingContext. This creates a fault for the row (an EOFault object in Objective-C or an empty object of the correct enterprise object class in Java). The raw row dictionary must contain the primary key attributes for this method to work properly. When your code tries to access the object for that row, the fault forces another database fetch, and a true enterprise object is created.

The following tables describe the API added to support raw row fetching.

EOFetchSpecification

rawRowKeyPaths	Returns an array of attribute keys that should be fetched as raw data. The default value is **nil** or **null**, indicating that full enterprise objects are to be returned from the fetch. If the array contains no objects, the entity specifies which attributes to fetch (EOEntity's **attributesToFetch** method).
setRawRowKeyPaths:	Sets the array of attribute keys that should be fetched as raw data. You can disable the fetching of raw rows by sending **nil** or **null** to this method. If you want to perform raw row fetching, but you want the entity to specify which attributes to fetch, you can pass an empty array to this method or you can use the **setFetchesRawRows:** method to enable raw row fetching.
fetchesRawRows	Returns whether raw row fetching is performed. YES or **true** if **rawRowKeyPaths** is non-**nil** or non-**null**.

EOFetchSpecification

setFetchesRawRows:	Sets whether raw row fetching is performed. If the value passed to this method is YES or **true**, then the **rawRowKeyPaths** array is set to an empty array. If NO or **false**, then the **rawRowKeyPaths** array is set to **nil** or **null**.

EOObjectStore

faultForRawRow:entityNamed:editingContext: (Objective-C) faultForRawRow (Java)	Returns a fault for the given raw row dictionary. The raw row dictionary must include the primary key attributes for this method to work properly. If the dictionary does not include the primary key, this method raises or throws an exception. Note that as EOObjectStore subclasses, EOEditingContext and EODatabaseContext also provide this method.

In addition to these methods, EOUtilities also provides raw row fetching methods. For more information, see section "New Convenience API."

Support for Multi-Threaded Applications

Enterprise Objects Framework 3.0 provides thread-safe APIs. This means that you can now write a multi-threaded enterprise object application. (For information on multi-threading WebObjects applications, see "What's New in WebObjects 4.0.")

The following operations are now thread-safe:

- The initial setup of the object graph in the editing context
- Making changes to the object graph in the editing context
- Database fetches
- Faulting
- Database saves

Enterprise Objects Framework 3.0 allows one thread to be active in each EOEditingContext and one thread to be active in each EODatabaseContext. In other words, multiple threads can access and modify objects concurrently in different editing contexts, but only one thread can access the database at a time (to save, fetch, or fault). Consequently, the support for multi-threaded applications can't be used to increase the level of database concurrency, but this wasn't the goal. Rather, the goal for 3.0 was to provide *thread safety* for applications that perform

other types of operations using multiple threads. To increase the level of database concurrency in your WebObjects applications, simply run multiple instances. This technique has been proven to scale very well.

The following tables describe the new API provided to support multi-threaded applications:

EOEditingContext (conformance to NSLocking)

lock	Locks access to the EOEditingContext to prevent other threads from accessing it. You should lock the editing context when you are making changes to the object graph, when you are fetching objects, and when you are saving objects. Only one thread is allowed per editing context tree.
unlock	Unlocks access to the EOEditingContext so that other threads may access it.

EODatabaseContext (conformance to NSLocking)

lock	Locks access to the EODatabaseContext to prevent other threads from accessing it. You typically don't have to invoke this method yourself.
unlock	Unlocks access to the EODatabaseContext. You typically don't have to invoke this method yourself.

Changes to Key-Value Coding

In release 3.0, Enterprise Objects Framework makes slight changes to key-value coding:

- The implementations of the primitive key-value coding methods resolve their keys differently: the search order for resolving the key has changed and the **valueForKey** methods now support Java-style getter methods (such as **get*Key***). More detailed information on the search order is provided in the table below.
- A new method, **storedValueForKey:**, has been added to the set of primitive key-value coding methods. It is the corresponding getter method to **takeStoredValue:forKey:** (Objective-C) and **takeStoredValue** (Java).
- The methods **takeStoredValue:forKeyPath:** and **takeStoredValuesFromDictionary:** have been removed.

The new key-value coding API is summarized in the following table:

Key-Value Coding Primitives

valueForKey:	The search order for resolving the provided key has changed to the following: method get*Key*, *key* method _get*Key*, *_key* instance variable *_key*, *key*
takeValue:forKey:(Objective-C) takeValueForKey (Java)	The search order for resolving the provided key has changed to the following: method set*Key*, _set*Key* instance variable *key*, *_key*
storedValueForKey:	The search order for resolving the provided key has changed to the following: method _get*Key*, *_key* instance variable *_key*, *key* method get*Key*, *key*
takeStoredValue:forKey:(Objective-C) takeStoredValueForKey (Java)	The search order for resolving the provided key has changed to the following: method _set*Key*, *_key* instance variable *key*, set*Key*

Stored Value Methods

The stored value methods, **storedValueForKey:** and **takeStoredValue:forKey:** (**storedValueForKey** and **takeStoredValueForKey** in Java), are used by the framework when accessing properties of an enterprise object to get or set properties for state storage and restoration (either from the database or to an in-memory snapshot). This access is considered private to the enterprise object and is invoked by the Framework to effect persistence on the object's behalf.

On the other hand, the basic key-value coding methods, **valueForKey:** and **takeValue:forKey:** (**valueForKey** and **takeValueForKey** in Java), are the public interface to an enterprise object. They are invoked by clients external to the object (such as for interactions with user interface or other business object logic).

Enterprise object classes can take advantage of this distinction to perform additional processing in accessor methods except when the object is being initialized with values from an external store. For instance, suppose an object wanted to update a total whenever the bonus was set:

```
void setBonus(double newBonus) {
  willChange();
  _total += (newBonus - _bonus);
}
```

This code should be activated when the object is updated with values provided by a user through the application's user interface, but not when the **bonus** property is restored from the database. Since the Framework restores the property using **takeStoredValue:forKey:** (**takeStoredValueForKey** in Java) and since this method accesses the **_bonus** instance variable in preference to calling the accessor, the unnecessary (and possibly harmful) recomputation of **_total** is avoided. If the object actually wants to intervene when a property is set from the database, it has two options:

- Implement **_setBonus:**
- Turn off stored accessors by overriding the class (Objective-C) or static (Java) method **useStoredAccessor** to return NO or **false**.

Changes to Enterprise Object Validation

Enterprise Objects Framework 3.0 makes slight changes to validation. First, it adds new validation API, summarized in the following tables.

EOEnterpriseObject Informal Protocol (Objective-C) or Interface (Java)

validateTakeValue:forKeyPath: (Objective-C) validateTakeValue (Java)	Validates (and coerces) the provided value and assigns it to destination of the provided key path if the value is different from the current value.

New Exception

EOUnknownKeyException	The default implementation of valueForKey: and takeValue:forKey: (takeValueForKey in Java) raise or throw this exception when they are invoked with a key that doesn't correlate with a method or instance variable in the receiver. The userInfo dictionary is augmented with the target object (EOTargetObjectUserInfoKey) and the unknown key (EOUnknownUserInfoKey).

Second, the Java validation methods have changed. Where their Objective-C counterparts (and their release 2.2 equivalents) return an exception, the 3.0 methods throw the (previously returned) exception. For more information, see the interface specification for EOValidation.

Changes to the Interface Layer

In the EOInterface framework, three association classes have been added, and some changes have been made to EODisplayGroup.

New Associations

The following sections provide a high level overview of the new associations. For complete documentation, see the corresponding class specifications.

EOMatrixAssociation

Binds titles or images in an NSMatrix to string or image attributes in an enterprise object. Its aspects are:

title	String property of the enterprise object that should be displayed in the matrix.
image	NSImage property of the enterprise object that should be displayed in the matrix.
enabled	Boolean property of enterprise object indicating if the matrix is enabled or disabled.

EORecursiveBrowserAssociation

Binds display groups to an NSBrowser so that enterprise objects of a recursive nature (that is, enterprise objects where there is a parent-child relationship) are displayed in the browser. Its aspects are:

rootChildren	A display group that represents the root object, the parent of all children. Bind this aspect first in Interface Builder. If you do, Interface Builder creates a second display group (bound to the **children** aspect) that holds the browser's current selection.
isLeaf	Boolean property of the enterprise object indicating if it has children. If the value is NO or **false**, the enterprise object must be able to return a value for the **children** aspect.
children	A display group containing the children of this enterprise object. If you bind the **rootChildren** aspect first, a display group is created by Interface Builder and bound to this aspect. If this display group is created by Interface Builder, it always holds the browser selection.
title	String to display in the browser title for this enterprise object.

EOMasterCopyAssociation

Binds a display group to a relationship property of the selected object in another display group. This association contains one aspect, **parent**, which is

the key for the property in the master object. Any change performed in one of these display groups is reflected in the other display group. Both display groups always have the same set of enterprise objects for **allObjects** but, depending on the applied qualifier, not necessarily the same set of **displayedObjects**.

This association exists mainly to support EORecursiveBrowserAssociation.

EOColumnAssociation

Three new aspects have been added to EOColumnAssociation:

textColor	NSColor property of the enterprise object indicating the text color for the row.
italic	Boolean property of the enterprise object indicating whether row text should be italicized.
bold	Boolean property of the enterprise object indicating whether row text should be bold.

EODisplayGroup Changes

EODisplayGroup has changed to make its API similar to the WODisplayGroup class in the WebObjects Framework.

Changed Methods

valueForKey:object:	Is now **valueForObject:key:** (**valueForObject** in Java).

New Methods

insertedObjectDefaultValues	Returns a dictionary containing the default values to be used for newly inserted objects. The keys into the dictionary are the properties of the entity that the display group manages. If the dictionary returned by this method is empty, the **insert** method adds an object that is initially empty. Because the object is empty, the display group has no value to display in the user interface for that object, meaning that there is nothing for the user to select and modify. Use the **setInsertedObjectDefaultValues:** method to set up a default value so that there is something to display.
setInsertedObjectDefaultValues:	Sets the default values to be used for newly inserted objects. You use this method to provide at least one field that can be displayed for the newly inserted object. The possible keys into the dictionary are the properties of the entity managed by this display group. Note that EODisplayGroup already contains other hooks for this purpose, such as **displayGroup:shouldInsertObject:atIndex:** (**displayGroupShouldInsertObject** in Java).

New Methods

queryOperatorValues	Returns a dictionary of operators to use on items in the **equalToQueryValues** dictionary. If a key in the **equalToQueryValues** dictionary also exists in **queryOperatorValues**, that operator for that key is used. The possible values are defined in the EOQualifier header.
setQueryOperatorValues:	Sets a dictionary of operators to use on items in the **equalToQueryValues** dictionary.
defaultStringMatchOperator	Returns the operator used to perform pattern matching for NSString values in the **equalToQueryValues** dictionary. This operator is used for properties listed in the **equalToQueryValues** dictionary that have NSString values and that do not have an associated entry in the **queryOperator** dictionary. In these cases, the operator returned by this method is used to perform pattern matching. The default value for the query match operator is **caseInsensitiveLike**, which means that the query does not consider case when matching letters. The other possible value for this operator is **like**, which matches the case of the letters exactly.
setDefaultStringMatchOperator:	Sets the operator used to perform pattern matching for NSString values in the **equalToQueryValues** dictionary.
defaultStringMatchFormat	This format is used for properties listed in the **equalToQueryValues** dictionary that have NSString values and that do not have an associated entry in the **queryOperator** dictionary. In these cases, the value is matched using pattern matching and the format returned by this method specifies how it will be matched. The default format string for pattern matching is "%@*" which means that the string value in the **equalToQueryValues** dictionary is used as a prefix. For example, if the **equalToQueryValues** dictionary contains a value “Jo” for the key “name”, the query returns all records whose name values begin with “Jo”.
setDefaultStringMatchFormat:	Sets how pattern matching will be performed on NSString values in the **equalToQueryValues** dictionary.
queryBindingValues	Added to support the binding of complex qualifiers to the user interface (see the section “Binding to Complex Qualifiers”). This method returns an NSDictionary containing the actual values that the user wants to query upon.
setQueryBindingValues:	Added to support the binding of complex qualifiers to the user interface (see the section “Binding to Complex Qualifiers”). This method sets the NSDictionary containing the actual values that the user wants to query upon.