Model Object Implementation Guide

Cocoa > Design Guidelines

2008-02-08

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

Simultaneously published in the United States and Canada.

Contents

Listings

Managed Object Accessor Methods 21

Introduction to Model Object Implementation Guide

With every successive release of Mac OS X, the basic functionality Cocoa provides has increased—particularly with technologies such as bindings and Core Data. This document describes in detail aspects of design and implementation that you need to take advantage of the features Cocoa offers.

This document addresses questions such as, what are model objects? what do they do? what do you have to do to implement a model class? and why is this important?

Who Should Read this Document

You should read this document to learn how to implement Cocoa model classes.

You are expected to be familiar with Cocoa standards, conventions and so on as described in Naming Conventions and the section entitled "Defining a Class" in *The Objective-C 2.0 Programming Language* (for example, class names should start with a capital letter; instance variable names should start with a lower case letter; instance variables should not be public, and so on). In implementing a model object, you should adhere to the Model View Controller (MVC) design pattern as described in The Model-View-Controller Design Pattern) and the Object Modeling section in Cocoa Design Patterns.

Organization of This Document

The following articles describe the features a model object might have and explain why and how you might implement them:

- "Checklist and Design Considerations" (page 9) provides a checklist of features that a model object might have, and reasons why you should consider implementing them.
- "Basic Accessor Methods" (page 15) describes how to implement accessor methods for standard model objects.
- Managed Object Accessor Methods (page 21) describes how to implement accessor methods for Core Data's managed objects.
- "Key-Value Technology Compliance" (page 31) describes how to ensure that your model objects support key-value coding and key-value observing, and describes how to register dependent keys.
- "Model Object Validation" (page 35) describes how to implement validation methods.
- "Initialization" (page 37) describes how to customize the initialization of model objects.

- "Archiving" (page 39) describes how to support archiving.
- "Copying" (page 41) describes how to support copying.

Checklist and Design Considerations

The fundamental role of model objects is to encapsulate data, and to provide access to that data. Model classes can add value in the form of custom behavior. The following checklist enumerates the additional features a model object might have to fully integrate into the Cocoa environment. It provides links to sections that discuss the design considerations you should bear in mind when you decide whether or not to use the feature, and links to sections that describe the implementation details.

- Representation of instance variables

  For design considerations, see "Instance Variable Types" (page 10) below.

  For implementation details, see "Basic Accessor Methods" (page 15).

- Accessor methods

  For design considerations, see "Accessor Methods" (page 11) below.

  For implementation details, see "Basic Accessor Methods" (page 15), and for Core Data see Managed Object Accessor Methods (page 21).

- Key-value coding and key-value observing compliance

  For design considerations, see: "Accessor Methods" (page 11) and "Key-Value Coding and Key-Value Observing" (page 12) below.

  For implementation details, see "Basic Accessor Methods" (page 15), Managed Object Accessor Methods (page 21), and "Key-Value Technology Compliance" (page 31).

- Copying

  For design considerations, see "Copying" (page 12) below.

  For implementation details, see "Copying" (page 41).

- Archiving

  For design considerations, see "Archiving" (page 13) below.

  For implementation details, see "Archiving" (page 39).

- Key-value observing notifications for dependent values

  For design considerations, see "Business Logic" (page 14) below.

  For implementation details, see "Dependent Values" (page 32).

- Initialization

  For design considerations, see "Business Logic" (page 14) below.

  For implementation details, see "Initialization" (page 37).

- Validation

  For design considerations, see "Business Logic" (page 14) below.

  For implementation details, see "Model Object Validation" (page 35).

Design Considerations

As general design and implementation principles, you should ensure that you follow usual Cocoa standards such as naming conventions and so on as described in Naming Conventions and "Defining a Class" in *The Objective-C 2.0 Programming Language*. For example, class names should start with a capital letter; instance variable names should start with a lowercase letter; instance variables should not be public, and so on. In implementing a model object, you should adhere to the Model View Controller (MVC) design pattern as described in The Model-View-Controller Design Pattern) in *Application Architecture Overview*.

If you are creating a traditional Cocoa application, you typically subclass `NSObject`. If you are creating an application that uses Core Data, then for classes that represent persistent entities, you subclass `NSManagedObject`. Most of the principles described in this document apply to subclasses of both `NSObject` and `NSManagedObject`. Where there are differences, they are either called out inline, or a separate section is included that addresses Core Data–specific issues.

Instance Variable Types

You can represent attribute values with an object or with a scalar or a C structure (a `struct` such as `NSRect`). There are different considerations to bear in mind when using either type. If you use an object, you must ensure correct data encapsulation (see "Encapsulation" (page 11)). If you represent an attribute as a scalar value (such as `int` , `float` , or `double`) or as a `struct`, it may be easier for you to perform arithmetic calculations (you do not have to convert from an object representation to a scalar value) and there are no memory management issues.

The basic scalar types and a limited set of common structures (`NSRect`, `NSPoint`, `NSSize`, and `NSRange`) integrate transparently with key-value coding and key-value observing (see "Key-Value Technology Compliance" (page 31))—that is, the key-value technologies automatically convert between the scalar or `struct` representation and a corresponding object representation, such as an instance of `NSNumber` or `NSValue`.

The main disadvantage of using non-object types is that non-object types cannot unambiguously represent a nil value. In addition, integration with other technologies (notably key-value coding and key-value observing) may require conversion of an attribute's value to an object representation anyway, which incurs some overhead. Depending on the pattern of usage of your application, it may be that any savings made by using non-object representations are outweighed by the overhead of conversion to and from object form (at least in terms of runtime efficiency—programmer effort is also a consideration). Finally, note that the granularity of representation for some types may be insufficiently fine. In a financial application it may be inappropriate to represent numeric data using, for example, a `float` due to the inherent inaccuracy of the `float` type. It may be more appropriate to use an `NSNumber` representation, as `NSNumber` provides both greater accuracy and a rich set of rounding behaviors.

If you are using Core Data, then there are additional constraints on the types you can use to represent a *persistent* attribute. Core Data natively supports only strings, numbers, dates, and binary data. You can, however, use transformable or transient values to work around this restriction, as described in Non-Standard Attributes.

Accessor Methods

The primary goal of accessor methods is to provide access to property values. There are two basic forms of accessor—get accessors and set accessors, used (predictably) to get and and set a property value respectively. You should implement accessors to preserve encapsulation.

You can also use accessor methods to simplify and streamline memory management, and to facilitate integration with other Cocoa technologies (in particular key-value coding and key-value observing, and through them, Cocoa bindings—see "Key-Value Coding and Key-Value Observing" (page 12)). The key-value coding protocol defines patterns for **collection accessors** for sets and arrays which further extend the concept of encapsulation and provide additional functionality.

The same principles that apply to "standard" model classes also apply to `NSManagedObject` subclasses used with Core Data, in order to provide additional functionality Core Data requires a different set of implementations for accessor methods. These are described in Managed Object Accessor Methods (page 21).

If you use the Objective-C declared properties feature (see "Properties" in *The Objective-C 2.0 Programming Language*), most of these considerations are taken care of for you.

Encapsulation

If an attribute is represented by an object, that object may be accessible by other parts of your application. To ensure proper encapsulation of data, you should ensure that model objects maintain their own copies of attribute values, and that get accessors advertise the attribute value as being immutable. For example, an employee's `firstName` attribute may be declared as an `NSString`, but it is possible that at some point the value passed to the set accessor may be an instance of `NSMutableString`. If you simply set the model object's instance variable to that string, this would leave open the possibility that the contents of the string could be altered externally without the employee instance being aware of the change (thus violating the principle of encapsulation). In the set accessor, you should therefore copy the new attribute value (or if the property should be mutable, you make a mutable copy).

If there are mutable and immutable versions of a class you use to represent a property—such as `NSArray` and `NSMutableArray`—you should typically declare the return value of the get accessor as an immutable object even if internally the model uses a mutable object. Declaring the return value as an immutable object signals that the value should not be modified externally.

Memory Management

You should implement your accessor methods such that they take care of memory management, as described in"Basic Accessor Methods" (page 15). If your accessors provide a clean API for modifying property values, and if you use the accessors pervasively when modifying values, then you will avoid most memory management issues that arise in Cocoa.

Collection Accessors

The key-value coding protocol defines patterns for collection accessors for sets and arrays. By implementing accessor methods that follow these patterns you derive a number of benefits.

- These accessor methods are key-value observing compliant (see "Key-Value Technology Compliance" (page 31) for more details). That is, if you invoke any of the mutator methods (such as `insertObject...`, `removeObject...`, or `replaceObject...`), then suitable key-value observing notifications are sent. This typically makes it easier to modify a collection directly than by using the proxy returned by `mutableArrayValueForKey:` or `mutableSetValueForKey:`.
- If you do use `mutableArrayValueForKey:` or `mutableSetValueForKey:`, the collection accessors are invoked automatically when you make modifications to the collection proxy. If you do not implement these accessors, the collection proxy must replace the whole collection (using the simple set accessor) for each modification, which can incur unnecessary overhead.
- You can use the collection accessors to hide underlying implementation details. There is no need for a collection to be implemented using the corresponding collection object—that is, for example, an array relationship need not actually be represented using an array. All that is required is that the accessor methods get and set values appropriately.

Key-Value Coding and Key-Value Observing

Key-value coding (KVC) and key-value observing (KVO) are fundamental technologies that are required for integration with Cocoa bindings and with Core Data's change management mechanism.

- You can use key-value coding to access an object's property using the property name as a key. Key-value coding includes a consistent API for property value validation which is described in "Model Object Validation" (page 35).
- You can use key-value observing to detect changes to property values. You can also use it as a means of registering dependencies between keys to denote that a change in the value of one key will result in a change in the value of another dependent key. For example, a `fullName` key may depend on the values of `firstName` and `lastName`. This latter feature is described in "Key-Value Technology Compliance" (page 31).

See *Key-Value Coding Programming Guide* and *Key-Value Observing Programming Guide* for detailed information on these two technologies.

To complete the picture, you must also of course use the KVO-compliant methods. You can also use key-value coding methods such as `setValue:forKey:`, `mutableArrayValueForKey:`, and `mutableSetValueForKey:`.

Copying

In some simple situations it is clear what "copy" means. In many cases, however, you must decide what it actually means to "copy" an object, and in particular what are the limits you want to impose on the copied object graph. You must also decide whether it is appropriate to copy all an object's attributes—for example, should an employee ID be copied? You must pay particular attention to relationships. If you "copy" an employee object, does this imply that a related department object is also copied? If you copy a department object, does that imply that related employees are also copied? If the answer to both of these questions is

"yes," then copying a single employee implies copying (as a minimum) also the department to which they belong, and the other employees in that department… These issues are described in more detail in Deciding How to Implement Object Copy.

Archiving

You can use archiving as a mechanism for data serialization to save your model objects to a persistent store or to send to other processes. The role of archiving as a means to save your objects to a persistent store is largely superseded by Core Data, which manages object persistence for you. You may also, however, use archiving to support copy and paste operations and the transfer of data between applications. For more about archiving and serialization in general, see *Archives and Serializations Programming Guide for Cocoa*.

When archiving model objects you must decide what properties to add to the archive. Typically you should archive all an object's non-derived attributes (that is, the attributes that cannot be calculated or derived from other attributes). If you are using archiving to serialize an object graph to save to a file, then you should also add related objects to the archive so that when the archive is unarchived, those relationships are restored.

Archiving and Copying

A common use of archiving is to support copy and paste operations (or to transfer data to other applications). If—especially in the context of Core Data—you use archiving for these purposes, you must decide what it actually means to copy, and what are the limits you want to impose on the copied object graph. A requirement for archiving in these situations is likely to be semantically different from archiving in the context of object graph serialization. When you create an archive for data serialization to a persistent store, you typically want to record all aspects of an object, including its relationships to other objects. When you use archiving to support copy and paste, you typically do not want to traverse relationships.

Rather than using archiving per se, therefore, it is often more appropriate to define a custom method that returns a representation of the object either in property list form or encapsulated in an `NSData` object, and then to initialize a copy object using that representation.

Note that in some situations, you may not actually want to copy an object. If you want to support drag and drop of objects at the destination of a relationship (for example, if you want to use drag and drop to transfer employees from one department to another) or if you support cross-store relationships, you should probably use managed object IDs or URI representations of managed object IDs.

Classic and Keyed Archiving

There are two forms of archiving, classic archiving and keyed archiving. Using classic archiving you must add instance variables directly to an archive and extract them in the same order. Using keyed archiving you encode and decode values as key-value pairs. This approach gives you more flexibility than with the classic technique. First, the order in which variables are encoded and decoded does not matter. Second, the archive is more robust against schema changes—the decoder does not fail if a value is not present in an archive, and it ignores any extra values in the archive. Finally, the output file may be human-readable, which may aid in any debugging process.

You can combine archiving techniques to ensure that your model objects can be archived using either classic or keyed archiving. Typically (since it offers a richer and more robust approach) you should choose keyed archiving over classic archiving.

Versioning

Version handling is typically easier with keyed archives, especially if you make only minor modifications to the schema (adding, removing, or renaming attributes). Old versions can still read keyed archives—keys present in the archive that are not present in the old schema are simply ignored. Problems may still arise, however, if the old version depends on the presence of a key that is absent in the new schema. For more details, see Forward and Backward Compatibility for Keyed Archives in *Archives and Serializations Programming Guide for Cocoa*.

Business Logic

Business logic is a broad term that encompasses actions performed on or using model objects. You are free to implement whatever methods you wish to support your application. You can provide methods to calculate values ranging from simple examples—such as the full name of an employee represented by a concatenation of first and last names—to complex—such as salary overhead for a department.

Cocoa provides a special API to formalize logic for validation, initialization, and dependent values. In an initialization method you can specify default values for an object's attributes. In validation methods, you can ensure that property values meet various constraints. To integrate with key-value observing, you may also need to notify observers if a change is made to a property on which a derived value depends.

There are many situations in which the value of one property depends on that of one or more other properties. If the value of one attribute changes, then the value of the derived property should also be flagged for change—for example, if the `lastName` property of an employee changes, the `fullName` also changes.

You use validation methods to ensure that data values meet various criteria that you specify. There are two types of validation—property-level and inter-property. You use property-level validation methods to ensure the correctness of individual values; you use inter-property validation methods to ensure the correctness of combinations of values where individual values may be valid but a combination may not be. For example, a person object may have attributes `age` and `hasDriversLicense`, with corresponding values `14` and `YES`. The individual values may be valid, the combination of values is invalid.

Basic Accessor Methods

There are two basic forms of accessor, a "get accessor", and a "set accessor". You use the get accessor to retrieve a property value from an object; you use the set accessor to set a property value in an object. You can use the Objective-C 2.0 Properties feature to declare and implement accessor methods.

The implementation of an accessor method depends on the type of the property to which it provides access—that is, whether the property is an attribute or a relationship, and if it is a relationship whether it is a to-one or a to-many relationship—see Cocoa Design Patterns for more details. It also depends on whether you are using garbage collection.

Important: The accessor methods shown here are not thread-safe in a managed memory environment, where thread-safety requires the use of a lock which incurs considerable overhead. Typically you cannot express thread-safety at the level of an individual accessor method (see *Threading Programming Guide*).

Objective-C Properties

You can use the Objective-C 2.0 properties feature to avoid the need to write accessor methods yourself. In your class interface, you declare a specification for the properties using `@property`:

```
@interface MyClass : NSObject
{
    NSString *myString;
    BOOL valid;
}
@property (copy, nonatomic) NSString *myString;
@property (nonatomic, getter=isValid) valid;
@end
```

In the implementation, you use `@synthesize` to direct the compiler to generate accessor methods corresponding to the property specification:

```
@implementation MyClass
@synthesize myString;
@synthesize valid;
@end
```

For full details, see Properties.

In most cases, this should be all you need. Sometimes, however, you may need to implement your own accessor methods—for example, for relationships you may want to make a mutable copy of a new value in a setter method. Even if you do implement custom accessors, you are encouraged to declare properties since they make your intent explicit.

Attributes

Attributes are defining characteristics of a model object.

Object Attributes

For performance reasons, the get accessor typically simply returns the value.

```
- (NSString *)firstName
{
    return firstName;
}
```

In the set accessor, you should typically make a copy of the new value that is then private to the model object, as shown in this example:

```
- (void)setFirstName:(NSString *)aFirstName
{
    if (firstName != aFirstName)
    {
        [firstName release]; // omit if you only support a garbage-collected
environment
        firstName = [aFirstName copy];
    }
}
```

Note that this requires the attribute to implement the `NSCopying` protocol. Most of the basic Cocoa classes you might use as an attribute implement `NSCopying`. If you implement a custom class to represent an attribute, it is typically easy to also implement the copy method. In cases where a class is immutable, this might simply retain `self`.

Non-Object Attribute Types

The following examples illustrate accessor methods for non-object attribute types.

```
- (NSRect)bounds
{
    return bounds;
}

- (void)setBounds:(NSRect)newBounds
{
    bounds = newBounds;
}
```

You typically also choose a suitable means of representing a `nil` value using the given attribute type. To properly integrate with key-value coding (see "Key-Value Technology Compliance" (page 31)) you should implement `setNilValueForKey:` as illustrated in the following example.

```
- (void)setNilValueForKey:(NSString *)key
{
    if ([key isEqualToString:@"bounds"]
```

```
    {
        bounds = NSMakeRect(0,0,0,0);
    }
    else
    {
        [super setNilValueForKey:key];
    }
}
```

Relationships

The semantics of accessor methods for a relationship depend on whether the relationship is a to-one or a to-many relationship. In a to-one relationship, you maintain a reference to a related object; in a to-many relationship you need a private collection that maintains references to related objects. In addition, there are special accessors for to-many relationships that may make access more efficient and that allow you to represent the relationship using something other than a collection object.

To-One Relationship

In contrast to an attribute, a relationship is not a private characteristic of an object. As with the attribute, the get accessor typically simply returns the value.

```
- (Department *)department
{
    return department;
}
```

The set accessor does not copy a new value. If you use a managed memory environment, or if you need to support both managed memory and garbage collection, you release the old value and retain the new (`retain` and `release` are no-ops in a garbage-collected environment):

```
- (void)setDepartment:(Department *)newDepartment
{
    if (newDepartment != department)
    {
        [department release];
        department = [newDepartment retain];
    }
}
```

If you use garbage collection, you can simply assign the new value:

```
- (void)setDepartment:(Department *)newDepartment
{
    department = newDepartment;
}
```

To-Many Relationship

There are two forms of accessor for to-many relationships—the simple get and set form that follows the same pattern as attributes and to-one relationships, and the collection form. The latter is primarily used for integration with key-value coding and key-value observing.

```
- (NSArray *)employees
{
    return employees;
}
- (void)setEmployees:(NSMutableArray *)newEmployees
{
    if (employees != newEmployees)
    {
        [employees autorelease];
        employees = [newEmployees mutableCopy];
    }
}
```

Collection Accessors

Collection accessors follow patterns, different for sets and arrays. The patterns are described in *Key-Value Coding Programming Guide*, but here is a summary. Given a relationship named `<key>` :

- For an array, you implement `countOf<Key>` and `objectIn<Key>AtIndex:`. You may also implement `get<Key>:range:`. If you want to support mutations, you also implement `insertObject:in<Key>AtIndex:` and `removeObjectFrom<Key>AtIndex:`. Again to improve performance, you may also implement `replaceObjectIn<Key>AtIndex:withObject:`.
- For a set, you implement an `add<Key>Object:` and `remove<Key>Object:` pair, an `add<Key>:` and `remove<Key>:` pair, or both pairs. For greater efficiency, you can also implement `intersect<Key>:` .

The following example illustrates collection accessors for an array; the analogous methods for sets are illustrated in Managed Object Accessor Methods (page 21).

```
- (NSUInteger)countOfEmployees
{
    return [employees count];
}

- (id)objectInEmployeesAtIndex:(NSUInteger)idx
{
    return [employees objectAtIndex:idx];
}

- (void)insertObject:(id)anObject inEmployeesAtIndex:(NSUInteger)idx
{
    [employees insertObject:anObject atIndex:idx];
}

- (void)removeObjectFromEmployeesAtIndex:(NSUInteger)idx
{
    [employees removeObjectAtIndex:index];
}
```

```
- (void)replaceObjectInEmployeesAtIndex:(NSUInteger)idx withObject:(id)anObject
{
    [employees replaceObjectAtIndex:idx withObject:anObject];
}
```

Managed Object Accessor Methods

This article explains why you might want to implement custom accessor methods for managed objects, and how to implement them for attributes and for relationships. It also illustrates how to implement primitive accessor methods.

Overview

On Mac OS X v10.5, Core Data dynamically generates efficient public and primitive get and set attribute accessor methods and relationship accessor methods for managed object classes. Typically, therefore, there's no need to for you to write accessor methods for properties that are defined in the entity of a managed object's corresponding managed object model—although you may use the Objective-C 2 properties feature to declare properties to suppress compiler warnings. To get the best performance—and to benefit from type-checking—you use the accessor methods directly, although they are also key-value coding (KVC) compliant so if necessary you can use standard key-value coding methods such as `valueForKey:`. You do need to write custom accessor methods if you use transient properties to support non-standard data types (see "Non-Standard Persistent Attributes" in *Core Data Programming Guide*) or if you use scalar instance variables to represent an attribute.

Mac OS X v10.4: This article describes accessor methods for Mac OS X v10.5; if you are using Mac OS X v10.4, see "Mac OS X v10.4: Managed Object Accessor Methods" in *Core Data Programming Guide*.

Custom implementation

The implementation of accessor methods you write for subclasses of `NSManagedObject` is typically different from those you write for other classes.

- If you do not provide custom instance variables, you retrieve property values from and save values into the internal store using primitive accessor methods.
- You must ensure that you invoke the relevant access and change notification methods (`willAccessValueForKey:`, `didAccessValueForKey:`, `willChangeValueForKey:`, `didChangeValueForKey:`, `willChangeValueForKey:withSetMutation:usingObjects:`, and `didChangeValueForKey:`).

  `NSManagedObject` disables automatic key-value observing (KVO, see *Key-Value Observing Programming Guide*) change notifications, and the primitive accessor methods do not invoke the access and change notification methods.
- In accessor methods for properties that are *not* defined in the entity model, you can either enable automatic change notifications or invoke the appropriate change notification methods.

You can use the Xcode data modeling tool to generate the code for accessor methods for any modeled property.

Key-value coding access pattern

The access pattern key-value coding uses for managed objects is largely the same as that used for subclasses of `NSObject`—see `valueForKey:`. The difference is that, if after checking the normal resolutions `valueForKey:` would throw an unbound key exception, the key-value coding mechanism for `NSManagedObject` checks whether the key is a modeled property. If the key matches an entity's property, the mechanism looks first for an accessor method of the form `primitive`*Key*, and if that is not found then looks for a value for *key* in the managed object's internal storage. If these fail, `NSManagedObject` throws an unbound key exception (just like `valueForKey:`).

Dynamically-Generated Accessor Methods

By default, Core Data dynamically creates efficient public and primitive get and set accessor methods for modeled properties (attributes *and* relationships) of managed object classes. This includes the key-value coding mutable proxy methods such as `add<Key>Object:` and `remove<Key>s:`, as detailed in the documentation for `mutableSetValueForKey:`—managed objects are effectively mutable proxies for all their to-many relationships.

Note: If you choose to implement your own accessors, the dynamically-generated methods never replace your own code.

For example, given an entity with an attribute `firstName`, Core Data automatically generates `firstName`, `setFirstName:`, `primitiveFirstName`, and `setPrimitiveFirstName:`. *Core Data does this even for entities represented by* `NSManagedObject`. To suppress compiler warnings when you invoke these methods, you should use the Objective-C 2.0 declared properties feature (see Declared Properties), as described in "Declaration" (page 22).

The property accessor methods Core Data generates are by default `(nonatomic, retain)`—*this is the recommended configuration*. The methods are `nonatomic` because non-atomic accessors are more efficient than atomic accessors, and in general it is not possible to assure thread safety in a Core Data application at the level of accessor methods. (To understand how to use Core Data in a multi-threaded environment, see Multi-Threading with Core Data.)

In addition to always being `nonatomic`, dynamic properties only honor `retain` or `copy` attributes—`assign` is treated as `retain`. You should use `copy` sparingly as it increases overhead. You cannot use `copy` for relationships because `NSManagedObject` does not adopt the `NSCopying` protocol, and it's irrelevant to the behavior of to-many relationships.

Important: If you specify `copy` for a to-one relationship, you will generate a *run-time* error.

Declaration

You can use Objective-C 2 properties to declare properties of managed object classes—you typically do this so that you can use the default accessors Core Data provides without generating compiler warnings. *The easiest way to generate the declarations is to select the relationship in the Xcode modeling tool and choose Design > Data Model > Copy Obj-C 2.0 Method Declarations to Clipboard.* and then modify the code if necessary.

You declare attributes and relationships as you would properties for any other object, as illustrated in the following example. When you declare a to-many relationship, the property type should be `NSSet *`. (The value returned from the get accessor is *not* a KVO-compliant mutable proxy—for more details, see "To-many relationships" in *Core Data Programming Guide*.)

```
@interface Employee : NSManagedObject
{ }
@property(nonatomic, retain) NSString* firstName, lastName;
@property(nonatomic, retain) Department* department;
@property(nonatomic, retain) Employee* manager;
@property(nonatomic, retain) NSSet* directReports;
@end
```

If you are not using a custom class, you can declare properties in a category of `NSManagedObject`:

```
@interface NSManagedObject (EmployeeAccessors)
{ }
@property(nonatomic, retain) NSString* firstName, lastName;
@property(nonatomic, retain) Department* department;
@property(nonatomic, retain) Employee* manager;
@property(nonatomic, retain) NSSet* directReports;
@end
```

You can use the same techniques to suppress compiler warnings for the automatically-generated to-many relationship mutator methods, for example:

```
@interface Employee (DirectReportsAccessors)

- (void)addDirectReportsObject:(Employee *)value;
- (void)removeDirectReportsObject:(Employee *)value;
- (void)addDirectReports:(NSSet *)value;
- (void)removeDirectReports:(NSSet *)value;

@end
```

You typically retain attributes, although to preserve encapsulation where the attribute class has a mutable subclass and it implements the `NSCopying` protocol you can also use `copy`, for example:

```
@property(nonatomic, copy) NSString* firstName, lastName;
```

Implementation

You can specify an implementation using the `@dynamic` keyword, as shown in the following example—although since `@dynamic` is the default, there is no need to do so:

```
@dynamic firstName, lastName;
@dynamic department, manager;
@dynamic directReports;
```

There should typically be no need for you to provide your own implementation of these methods, unless you want to support scalar values. The methods that Core Data generates at runtime are more efficient than those you can implement yourself.

Inheritance

If you have two subclasses of `NSManagedObject` where the parent class implements a dynamic property and its subclass (the grandchild of `NSManagedObject`) overrides the methods for the property, those overrides cannot call `super`.

```
@interface Parent : NSManagedObject
@property(nonatomic, retain) NSString* parentString;
@end

@implementation Parent
@dynamic parentString;
@end

@interface Child : Parent
@end

@implementation Child
- (NSString *)parentString
{
    // this throws a "selector not found" exception
    return parentString.foo;
}
@end
```

Custom Attribute and To-One Relationship Accessor Methods

Important: You are strongly encouraged to use dynamic properties (that is, properties whose implementation you specify as `@dynamic`) instead of creating custom implementations for standard or primitive accessor methods.

If you want to implement your own attribute or to-one relationship accessor methods, you use the primitive accessor methods to get and set values from and to the managed object's private internal store. You must invoke the relevant access and change notification methods, as illustrated in Listing 1 (page 24). `NSManagedObject`'s implementation of the primitive set accessor method handles memory management for you.

Listing 1 Implementation of a custom managed object class illustrating attribute accessor methods

```
@interface Department : NSManagedObject
{
}
@property(nonatomic, retain) NSString *name;
@end

@interface Department (PrimitiveAccessors)
- (NSString *)primitiveName;
- (void)setPrimitiveName:(NSString *)newName;
@end
```

```
@implementation Department

@dynamic name;

- (NSString *)name
{
    [self willAccessValueForKey:@"name"];
    NSString *myName = [self primitiveName];
    [self didAccessValueForKey:@"name"];
    return myName;
}

- (void)setName:(NSString *)newName
{
    [self willChangeValueForKey:@"name"];
    [self setPrimitiveName:newName];
    [self didChangeValueForKey:@"name"];
}
@end
```

The default implementation does not copy attribute values. If the attribute value may be mutable and implements the `NSCopying` protocol (as is the case with `NSString`, for example), you can copy the value in a custom accessor to help preserve encapsulation (for example, in the case where an instance of `NSMutableString` is passed as a value). This is illustrated in Listing 2 (page 25). Notice also that (for the purposes of illustration) in this example the get accessor is not implemented—since it's not implemented, Core Data will generate it automatically.

Listing 2 Implementation of a custom managed object class illustrating copying setter

```
@interface Department : NSManagedObject
{
}
@property(nonatomic, copy) NSString *name;
@end

@implementation Department

@dynamic name;

- (void)setName:(NSString *)newName
{
    [self willChangeValueForKey:@"name"];
    // NSString implements NSCopying, so copy the attribute value
    NSString *newNameCopy = [newName copy];
    [self setPrimitiveName:newNameCopy];
    [newNameCopy release];
    [self didChangeValueForKey:@"name"];
}
@end
```

If you choose to represent an attribute using a scalar type (such as `NSInteger` or `CGFloat`), or as one of the structures supported by `NSKeyValueCoding` (`NSRect`, `NSPoint`, `NSSize`, `NSRange`), then you should implement accessor methods as illustrated in Listing 3 (page 26). If you want to use any other attribute type, then you should use a different pattern, described in Non-Standard Persistent Attributes.

Listing 3 Implementation of a custom managed object class illustrating a scalar attribute value

```
@interface Circle : NSManagedObject
{
    CGFloat radius;
}
@property CGFloat radius;
@end

@implementation Circle

- (CGFloat)radius
{
    [self willAccessValueForKey:@"radius"];
    float f = radius;
    [self didAccessValueForKey:@"radius"];
    return f;
}

- (void)setRadius:(CGFloat)newRadius
{
    [self willChangeValueForKey:@"radius"];
    radius = newRadius;
    [self didChangeValueForKey:@"radius"];
}
@end
```

Custom To-Many Relationship Accessor Methods

Important: You are strongly encouraged to use dynamic properties (that is, properties whose implementation you specify as `@dynamic`) instead of creating custom implementations for standard or primitive accessor methods.

You usually access to-many relationships using `mutableSetValueForKey:`, which returns a proxy object that both mutates the relationship and sends appropriate key-value observing notifications for you. There should typically be little reason to implement your own collection accessor methods for to-many relationships. If they are present, however, the framework calls the mutator methods (such as `add<Key>Object:` and `remove<Key>Object:`) when modifying a collection that represents a persistent relationship. (Note that fetched properties do not support the mutable collection accessor methods.) In order for this to work correctly, you must implement an `add<Key>Object:`/`remove<Key>Object:` pair, an `add<Key>:`/`remove<Key>:` pair, or both pairs. You may also implement other get accessors (such as `countOf<Key>:`, `enumeratorOf<Key>:`, and `memberOf<Key>:`) and use these in your own code, however these are not guaranteed to be called by the framework.

Important: For performance reasons, the proxy object returned by managed objects for `mutableSetValueForKey:` does not support `set<Key>:` style setters for relationships. For example, if you have a to-many relationship `employees` of a Department class and implement accessor methods `employees` and `setEmployees:`, then manipulate the relationship using the proxy object returned by `mutableSetValueForKey:@"employees"`, `setEmployees:` is *not* invoked. You should implement the other mutable proxy accessor overrides instead.

If you do implement collection accessors for model properties, they must invoke the relevant KVO notification methods. Listing 4 (page 27) illustrates the implementation of accessor methods for a to-many relationship—`employees`—of a Department class. *The easiest way to generate the implementation is to select the relationship in the Xcode modeling tool and choose Design > Data Model > Copy Obj-C 2.0 Method {Declarations/Implementations} to Clipboard.*

Listing 4 A managed object class illustrating implementation of custom accessors for a to-many relationship

```
@interface Department : NSManagedObject
{
}
@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSSet *employees;
@end


@interface Department (DirectReportsAccessors)

- (void)addEmployeesObject:(Employee *)value;
- (void)removeEmployeesObject:(Employee *)value;
- (void)addEmployees:(NSSet *)value;
- (void)removeEmployees:(NSSet *)value;

- (NSMutableSet*)primitiveEmployees;
- (void)setPrimitiveEmployees:(NSMutableSet*)value;

@end


@implementation Department

@dynamic name;
@dynamic employees;

- (void)addEmployeesObject:(Employee *)value
{
    NSSet *changedObjects = [[NSSet alloc] initWithObjects:&value count:1];

    [self willChangeValueForKey:@"employees"
          withSetMutation:NSKeyValueUnionSetMutation
          usingObjects:changedObjects];
    [[self primitiveEmployees] addObject:value];
    [self didChangeValueForKey:@"employees"
          withSetMutation:NSKeyValueUnionSetMutation
          usingObjects:changedObjects];

    [changedObjects release];
```

```
}

- (void)removeEmployeesObject:(Employee *)value
{
    NSSet *changedObjects = [[NSSet alloc] initWithObjects:&value count:1];

    [self willChangeValueForKey:@"employees"
          withSetMutation:NSKeyValueMinusSetMutation
          usingObjects:changedObjects];
    [[self primitiveEmployees] removeObject:value];
    [self didChangeValueForKey:@"employees"
          withSetMutation:NSKeyValueMinusSetMutation
          usingObjects:changedObjects];

    [changedObjects release];
}

- (void)addEmployees:(NSSet *)value
{
    [self willChangeValueForKey:@"employees"
          withSetMutation:NSKeyValueUnionSetMutation
          usingObjects:value];
    [[self primitiveEmployees] unionSet:value];
    [self didChangeValueForKey:@"employees"
          withSetMutation:NSKeyValueUnionSetMutation
          usingObjects:value];
}

- (void)removeEmployees:(NSSet *)value
{
    [self willChangeValueForKey:@"employees"
          withSetMutation:NSKeyValueMinusSetMutation
          usingObjects:value];
    [[self primitiveEmployees] minusSet:value];
    [self didChangeValueForKey:@"employees"
          withSetMutation:NSKeyValueMinusSetMutation
          usingObjects:value];
}
```

Custom Primitive Accessor Methods

Primitive accessor methods are similar to "normal" or public key-value coding compliant accessor methods, except that Core Data uses them as the most basic data methods to access data, consequently they do *not* issue key-value access or observing notifications. Put another way, they are to `primitiveValueForKey:` and `setPrimitiveValue:forKey:` what public accessor methods are to `valueForKey:` and `setValue:forKey:`.

Typically there should be little reason to implement primitive accessor methods. They are, however, useful if you want custom methods to provide direct access to instance variables for persistent Core Data properties. The example below contrasts public and primitive accessor methods for an attribute, `int16`, of type `Integer 16`, stored in a custom instance variable, `nonCompliantKVCivar`.

```
// primitive get accessor
- (short)primitiveInt16 {
```

```
    return nonCompliantKVCivar;
}

// primitive set accessor
- (void)setPrimitiveInt16:(short)newInt16 {
    nonCompliantKVCivar = newInt16;
}

// public get accessor
- (short)int16 {
    short tmpValue;
    [self willAccessValueForKey: @"int16"];
    tmpValue = nonCompliantKVCivar;
    [self didAccessValueForKey: @"int16"];
    return tmpValue;
}

// public set accessor
- (void)setInt16:(short)int16 {
    [self willChangeValueForKey: @"int16"];
    nonCompliantKVCivar = int16;
    [self didChangeValueForKey:@"int16"];
}
```

Key-Value Technology Compliance

There are a number of ways you can ensure that your model objects are key-value coding (KVC) and key-value observing (KVO) compliant, typically and most easily by implementing suitable accessor methods as described in "Basic Accessor Methods" (page 15). The main exception to this rule is Core Data, which imposes special constraints on the implementation of accessor methods. If you are using Core Data, you should read Managed Object Accessor Methods (page 21) to learn how to ensure your managed object classes are KVC and KVO compliant.

KVC Compliance

The key-value coding mechanism tries hard to find a value for a given key, so that it is actually difficult not to be KVC compliant for a given property. Although there are a number of ways to ensure compliance, it is recommended that you use accessor methods and follow standard naming conventions. The general requirements for KVC compliance are described in Key-Value Coding Accessor Methods in *Key-Value Coding Programming Guide*.

KVO Compliance

There are two ways you can implement KVO compliance—using automatic notification, or, using manual notification. As the name implies, if you use automatic notification you don't have to do anything other than implement (and use) standard accessor methods as described in "Basic Accessor Methods" (page 15). For most classes, automatic KVO notification is enabled by default, and there is typically no benefit in disabling automatic notification. The primary exception is in a subclass of `NSManagedObject`.

Important: `NSManagedObject` *disables* automatic notification by default. Moreover, you cannot enable automatic notification for modeled properties. If you implement accessors for model properties, you *must* invoke the relevant change notification methods. You can, however, enable automatic notification for unmodeled properties using `automaticallyNotifiesObserversForKey:`.

If you want to disable automatic notification, you implement `automaticallyNotifiesObserversForKey:` and return `NO` for the keys for which you want to provide manual notifications. In your set accessors, for simple attributes you then invoke `willChangeValueForKey:` and `didChangeValueForKey:` respectively before and after the property key is changed. For a to-many relationship, you also need to invoke the relevant notification methods indicating the type of change and the indexes of the objects changed.

You can implement "bulk" modifiers that allow you to, for example, make a large number of additions to an array in a single method call rather than requiring an individual method call for each insertion. The following example shows a custom bulk modifier method for an ordered to-many relationship.

```
- (void)addObjectsToEmployeesFromArray:(NSArray *)otherArray
{
```

```
    if ([otherArray count] > 0)
    {
        NSRange range = NSMakeRange([employees count], [otherArray count]);
        NSIndexSet *indexes = [NSIndexSet indexSetWithIndexesInRange:range];

        [self willChange:NSKeyValueChangeInsertion valuesAtIndexes:indexes
                forKey:@"employees"];
        [employees addObjectsFromArray:otherArray];
        [self didChange:NSKeyValueChangeInsertion valuesAtIndexes:indexes
                forKey:@"employees"];
    }
}
```

Dependent Values

There are many situations in which the value of one property depends on that of one or more other properties. If the value of one attribute changes, then the value of the derived property should also be flagged for change. How you ensure that key-value observing notifications are posted for these dependent properties depends on which version of Mac OS X you're using.

Mac OS X v10.5 and later

If you are targeting Mac OS X v10.5 and later, to trigger notifications automatically you should either override `keyPathsForValuesAffectingValueForKey:` or implement a suitable method that follows the pattern it defines for registering dependent keys.

For example, you could override `keyPathsForValuesAffectingValueForKey:` as shown in the following example:

```
+ (NSSet *)keyPathsForValuesAffectingValueForKey:(NSString *)key
{
    NSSet *keyPaths = [super keyPathsForValuesAffectingValueForKey:key];

    if ([key isEqualToString:@"fullNameAndID"])
    {
        NSSet *affectingKeys = [NSSet setWithObjects:@"lastName", @"firstName",
@"employeeID", nil];
        keyPaths = [keyPaths setByAddingObjectsFromSet:affectingKeys];
    }
    return keyPaths;
}
```

Or, to achieve the same result, you could just implement `keyPathsForValuesAffectingFullNameAndID` as illustrated in the following example:

```
+ (NSSet *)keyPathsForValuesAffectingFullNameAndID
{
    return [NSSet setWithObjects:@"lastName", @"firstName", @"employeeID", nil];
}
```

Important: Note that you cannot set up dependencies on to-many relationships. For example, suppose you have an Order object with a to-many relationship (`orderItems`) to a collection of OrderItem objects, and OrderItem objects have a `price` attribute. You might want the Order object have a `totalPrice` attribute that is dependent upon the prices of all the OrderItem objects in the relationship. You can *not* do this by implementing `keyPathsForValuesAffectingValueForKey:` and returning `orderItems.price` as the keypath for `totalPrice`. You must observe the `price` attribute of each of the OrderItem objects in the `orderItems` collection and respond to changes in their values by updating `totalPrice` yourself.

Mac OS X v10.3 and later

If you are targeting Mac OS X v10.3 and later, you should use `setKeys:triggerChangeNotificationsForDependentKey:` to trigger notifications automatically. You set up the dependencies as illustrated in the following example:

```
+ (void)initialize
{
    NSArray *keys = [NSArray arrayWithObjects:
        @"firstName", @"lastName", nil];
    [self setKeys:keys triggerChangeNotificationsForDependentKey:
        @"fullName"];
}
```

Important: Note that you cannot set up dependencies on key *paths*. For example, suppose you have an Order object with a to-many relationship (`orderItems`) to a collection of OrderItem objects, and OrderItem objects have a `price` attribute. You might want the Order object have a `totalPrice` attribute that is dependent upon the prices of all the OrderItem objects in the relationship. You can *not* do this with `setKeys:triggerChangeNotificationsForDependentKey:` passing `orderItems.price` as the key. You must observe the `price` attribute of each of the OrderItem objects in the `orderItems` collection and respond to changes in their values by updating `totalPrice` yourself.

Model Object Validation

There are two types of validation—property-level and inter-property. You use property-level validation methods to ensure the correctness of individual values; you use inter-property validation methods to ensure the correctness of combinations of values.

Property-Level Validation

The `NSKeyValueCoding` protocol specifies a method—`validateValue:forKey:error:`—that provides general support for validation methods in a similar way to that in which `valueForKey:` provides support for accessor methods. You typically do not override `validateValue:forKey:error:`; instead you implement custom validation methods that follow the pattern `validate<Key>:error:`. In the method implementation, you check the proposed new value and if it does not fit your constraints you return `NO`. If the error parameter is not null, you also create an `NSError` object that describes the problem, as illustrated in the following example.

```
-(BOOL)validateAge:(id *)ioValue error:(NSError **)outError {
    if (*ioValue == nil) {
        // trap this in setNilValueForKey? new NSNumber with value 0?
        return YES;
    }
    if ([*ioValue floatValue] <= 0.0) {
        if (outError != NULL) {
            NSString *errorStr = NSLocalizedStringFromTable(
                @"Age must greater than zero", @"Employee",
                @"validation: zero age error");
            NSDictionary *userInfoDict = [NSDictionary
dictionaryWithObject:errorStr
                forKey:NSLocalizedDescriptionKey];
            NSError *error = [[[NSError alloc]
initWithDomain:EMPLOYEE_ERROR_DOMAIN
                code:PERSON_INVALID_AGE_CODE
                userInfo:userInfoDict] autorelease];
            *outError = error;
        }
        return NO;
    }
    else {
        return YES;
    }
    // . . .
```

It is important to note that the input value is a pointer to object reference (an `id *`). This means that in principle you can change the input value. Doing so is, however, strongly discouraged, as there are potentially serious issues with memory management (see Key-Value Validation in *Key-Value Coding Programming Guide*). You should not invoke `validateValue:forKey:error:` within a custom property validation method. If you do, you will create an infinite loop when `validateValue:forKey:error:` is invoked at runtime.

Inter-Property Validation

It is possible for the values of all the individual attributes of an object to be valid and yet for the combination of values to be invalid. Consider, for example, an application that stores information about people including their age and whether or not they have a driving license. For a Person object, `12` might be a valid value for an `age` attribute, and `YES` is a valid value for a `hasDrivingLicense` attribute, but (in most countries at least) this combination of values would be invalid.

The `NSKeyValueCoding` protocol does not define a method for inter-property validation. Core Data, however, defines `validateForUpdate:` which you can co-opt for classes that do not inherit from `NSManagedObject`. Using `validateForUpdate:` also makes it easier for you to migrate your classes to Core Data in the future should you wish. An example implementation is shown in Managed Object Validation in *Core Data Programming Guide*.

Initialization

There are several situations in which an object's properties may be initialized. Most obviously, when an object is first created—you typically establish values in this case in `init` or another custom initializer. You may also, however, want to initialize an object in a different way when it is extracted from an archive. In this situation, you can customize the initialization in `initWithCoder:`. Core Data provides special methods for initialization at different stages in an object's life-cycle.

Core Data Initialization

Core Data defines a number of methods to support initialization, validation, and reporting on saving. Core Data also allows you to put default initial values for attributes into the managed object model. This may obviate the need for many custom initialization methods. Should you nevertheless need to perform additional initialization, Core Data provides special methods—`awakeFromInsert` and `awakeFromFetch`—that you can override to perform initialization in two different circumstances. Note that, should you wish to perform initialization in all circumstances, it also specifies a designated initializer for `NSManagedObject`: `initWithEntity:insertIntoManagedObjectContext:`.

The initializer methods `awakeFromInsert` and `awakeFromFetch` allow you to discriminate between two different situations. `awakeFromInsert` is invoked automatically when a newly created managed object is first inserted into a managed object context. This happens only once in the entire lifetime of the object. You can use this method to, for example, set a creation date stamp. `awakeFromFetch` is invoked on subsequent occasions when a managed object is retrieved from a persistent store, whether as a result of your executing a fetch request, or as a result of a fault firing. You can use `awakeFromFetch` to, for example, calculate derived property values.

Because you must initialize an `NSManagedObject` instance with `initWithEntity:insertIntoManagedObjectContext:`, you cannot readily use managed objects with an archiver (put another way, you cannot an easily implement an `initWithCoder:`: method for a managed object class). You might instead implement a custom class that handles the decoding and then returns a real managed object instead of `self` from `initWithCoder:`.

Archiving

Archiving is supported by archiver objects and the `NSCoding` protocol. The protocol consists of two methods: `initWithCoder:` and `encodeWithCoder:`. There are two forms of archiving, classic and keyed. Both techniques support versioning.

Keyed Archiving

Using keyed archiving you encode and decode values as key-value pairs, as illustrated in the following example:

```
- (void)encodeWithCoder:(NSCoder *)encoder
{
    [encoder encodeObject:department forKey:@"department"];
    [encoder encodeObject:lastName forKey:@"Last"];
    [encoder encodeObject:firstName forKey:@"First"];
    [encoder encodeInt: employeeID forKey:@"EmpID"];
}

- initWithCoder:(NSCoder *)decoder
{
    [self setDepartment:[decoder decodeObjectForKey:@"department"]];
    employeeID = [decoder decodeIntForKey:@"EmpID"];
    [self setLastName:[decoder decodeObjectForKey:@"Last"]];
    [self setFirstName:[decoder decodeObjectForKey:@"First"]];
    return self;
}
```

The order in which variables are encoded and decoded does not have to match.

Classic Archiving

If you need to support versions of Mac OS X prior to 10.2, you cannot use keyed archiving. Using classic archiving, you must encode and decode instance variables in the same order, as illustrated in the following example:

```
- (void)encodeWithCoder:(NSCoder *)encoder
{
    [encoder encodeObject:firstName];
    [encoder encodeObject:lastName];
    [encoder encodeObject:department];
    [encoder encodeValueOfObjCType:@encode(int) at:&employeeID];
}

- initWithCoder:(NSCoder *)decoder
```

```
{
    [self setFirstName:[decoder decodeObject]];
    [self setLastName:[decoder decodeObject]];
    [self setDepartment:[decoder decodeObject]];
    [decoder decodeValueOfObjCType:@encode(int) at:&employeeID];
    return self;
}
```

Combining Archiving Techniques

Classic archiving is deprecated, so you should migrate your archives to the keyed format. If you need to support classic and keyed archiving, you can combine archiving techniques, as illustrated in the following example:

```
- (void)encodeWithCoder:(NSCoder *)encoder
{
    if ([encoder allowsKeyedCoding])
    {
        [encoder encodeObject:firstName forKey:@"First"];
        [encoder encodeObject:lastName forKey:@"Last"];
        [encoder encodeInt: employeeID forKey:@"EmpID"];
        [encoder encodeObject:department forKey:@"department"];
    }
    else
    {
         [encoder encodeObject:firstName];
         [encoder encodeObject:lastName];
         [encoder encodeObject:department];
         [encoder encodeValueOfObjCType:@encode(int) at:&employeeID];
    }
}
```

The corresponding `initWithCoder:` method follows a similar pattern.

Versioning

The archiving mechanism allows version information to be stored in archives. You set a class's version number with the `setVersion:` method—typically in the class's `initialize` method.

You use the `NSCoder` method `versionForClassName:` to retrieve the class's version from an archive. If you need to obtain the version from within an `NSCoding` protocol or other method, you should use the class name explicitly (for example, `version = [MyClass version]`). If you simply send `version` to the return value of `class`, a subclass's version number may be returned instead.

If you use Core Data for object persistence, then you do not use `NSCoder`-based archiving or versioning. On Mac OS X v10.5, Core Data provides an infrastructure for data migration based on versioned managed object models—for more details, see *Core Data Model Versioning and Data Migration Programming Guide*. (On Mac OS X v10.4, you can add version information to the metadata for a store using the persistent store coordinator method `setMetadata:forPersistentStore:`. For more details, see Versioning.)

Copying

To support copying, you adopt the `NSCopying` protocol. The protocol has a single method—`copyWithZone:`—that you must implement to allow your objects to be copied. If your application distinguishes between mutable and immutable versions of an entity, you should adopt `NSCopying` for the immutable version, and `NSMutableCopying` for the mutable version (you implement `mutableCopyWithZone:`). The implementation of the `copy` method is described in detail in Implementing Object Copy.

A "cheap" way to provide customized copy behavior in some situations is to use key-value coding. You can create a dictionary representation of an object using `dictionaryWithValuesForKeys:`, create a new uninitialized instance of the same class, then use the dictionary to initialize the new instance with `setValuesForKeysWithDictionary:`.

Document Revision History

This table describes the changes to *Model Object Implementation Guide*.

Date	Notes
2008-02-08	Corrected the implementation example for setNilValueForKey:; updated the examples for dependent keys.
2007-10-31	Updated for Mac OS X v10.5.
2007-07-10	Corrected minor typographical errors.
2007-02-08	Corrected a link to "Managed Object Validation."
2006-09-05	Reorganized Validation article.
2006-04-04	Revised reference to Data Modeling article.
2006-03-08	Added example of interproperty validation; made minor correction to validation method (to check for null error parameter).
2006-01-10	Corrected a typographical error.
2005-06-04	New document that describes issues relating to the design and implementation of model objects