# Cocoa Bindings Programming Topics

General

2009-03-08

# Contents

# Figures and Listings

# Introduction to Cocoa Bindings Programming Topics

Cocoa bindings is a collection of technologies you can use in your applications to fully implement a Model-View-Controller paradigm where models encapsulate application data, views display and edit that data, and controllers mediate between the two. Cocoa bindings reduces the code dependencies between models, views and controllers, supports multiple ways of viewing your data, and automatically synchronizes views when models change. Cocoa bindings provides extensible controllers, protocols for models and views to adopt, and additions to classes in Foundation and the Application Kit. You can eliminate most of your glue code by using bindings available in Interface Builder to connect controllers with models and views.

## Who Should Read This Document

Cocoa bindings is ideal for developers writing new applications who have some familiarity with Cocoa, and for developers of existing applications who want to simply clean up or eliminate their existing glue code. In most cases, Cocoa bindings can be used to replace traditional Cocoa mechanisms such as target-action, delegation, and some data source protocols. However, great care has been taken to ensure that both approaches can be used side by side within the same application.

This document assumes that you have read *Key-Value Coding Programming Guide*, *Key-Value Observing Programming Guide* and *Value Transformer Programming Guide*.

**Important:** Cocoa bindings is available to Cocoa Objective-C applications running Mac OS X version 10.3 and later.

## Organization of This Document

The following articles cover key concepts in understanding how Cocoa bindings works:

- "What Are Cocoa Bindings?" (page 11) describes the advantages that Cocoa bindings offers developers; provides a brief summary of how they work; and what design patterns you should use to adopt the technology.
- "How Do Bindings Work?" (page 23) describes in detail the technologies supporting Cocoa bindings and how they interact.
- "User Defaults and Bindings" (page 55) describes the role of the NSUserDefaultsController and how it works with NSUserDefaults.
- "Providing Controller Content" (page 35) describes how to set and modify the content of NSObjectController and its subclasses.
- "Working With a Controller's Selection" (page 41) describes how to get a controller's selection and change the current selection.

- "Bindings Message Flow" (page 47) illustrates the flow of messages between model, view and controller objects in a bindings application.

These articles contain tasks that teach you how to use Cocoa bindings:

- "Creating a Master-Detail Interface" (page 59) explains how to implement a basic master-detail interface where a table view is used in the master interface to display a collection of objects, and other views used in the detail interface to display the selected object in the collection.
- "Displaying Images Using Bindings" (page 67) describes the various options when displaying images in columns and contains an example of a custom value transformer.
- "Implementing To-One Relationships Using Pop-Up Menus" (page 73) explains how to implement editable to-one relationship as pop-up menus.
- "Filtering Using a Custom Array Controller" (page 77) explains how to add a search field to the master interface to filter the objects it displays.
- "Controller Key-Value Observing Compliance" (page 79) details the properties for which the controller classes provide key-value observing change notifications.
- "Troubleshooting Cocoa Bindings" (page 83) describes a number of common problems encountered with Cocoa bindings applications and provides methods for correcting the issues.

## See Also

There are other technologies, not fully covered in this topic, that are fundamental to how bindings work. You may want to read these topics if you want a better understanding of the underpinnings of Cocoa bindings, or if you want to use these technologies independent from bindings. For example, this topic does not explain how to use the methods defined in the key-value observing protocol. Refer to these documents for more details:

- *Cocoa Application Tutorial Using Bindings* takes you through the steps of building the familiar Currency Converter application using Cocoa bindings.
- *Cocoa Bindings Reference* enumerates the classes that support Cocoa bindings and provides descriptions of the bindings for each class, along with the supported options and placeholders.
- *Key-Value Coding Programming Guide* covers all the features of the key-value coding protocol that allows objects to indirectly access the properties of other objects.
- *Key-Value Observing Programming Guide* covers all the features of the key-value observing protocol that allows objects to observe changes in other objects.
- *Value Transformer Programming Guide* describes how to use value transformers to convert values from one type to another.
- *Sort Descriptor Programming Topics* describes how to use sort descriptors that specify how collections are sorted.

# What Are Cocoa Bindings?

In the simplest functional sense, the Cocoa bindings technology provides a means of keeping model and view values synchronized without you having to write a lot of "glue code." It allows you to establish a mediated connection between a view and a piece of data, "binding" them such that a change in one is reflected in the other.

This article describes what the technology offers and how it makes writing applications easier. It also introduces the idea that rather than completely reimplementing an existing application to make use of bindings, you can incorporate bindings in stages.

This article also describes on a conceptual level how Cocoa bindings work, and the design patterns you should adopt. It gives a brief overview of the Model-View-Controller design pattern, and why it is beneficial. It then gives a conceptual overview of how the main technologies that underpin Cocoa bindings—key-value coding, key-value observing, and key-value binding—work, and how they inter-relate. The article finally explains the role of the controller classes that Cocoa bindings provide and why you should use them.

"How Do Bindings Work?" (page 23) describes the supporting technologies in greater detail.

## The Advantages of Using Bindings

The Cocoa bindings technology offers a way to increase the functionality and consistency of your application while at the same time decreasing the amount of code you have to write and maintain. It takes care of most aspects of user interface management for you by allowing you to off load the work of custom glue code onto reusable pre-built controllers. It helps you build polished, easy-to-use applications that leverage object relationships, provide sortable tables, and include intelligent selection management.

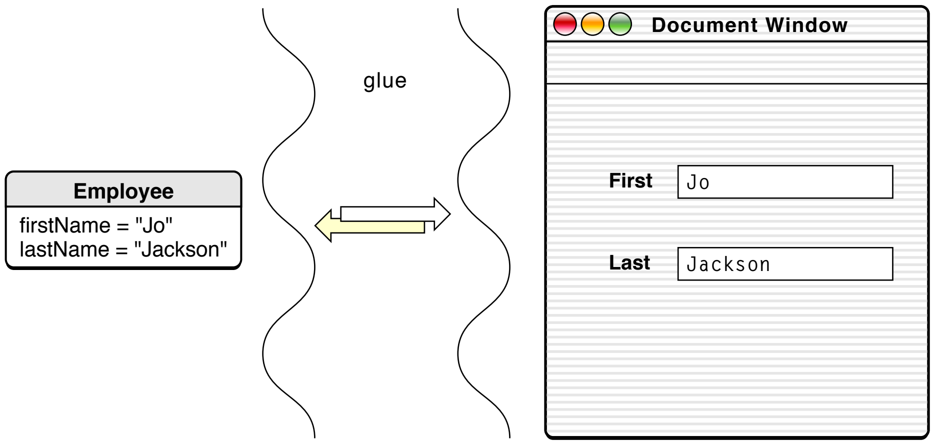
Typically you do not need to completely rewrite your application in order to adopt Cocoa bindings. For example, it is likely that you can factor out User Preferences to be managed by Cocoa bindings without affecting the rest of an application. You will find it easier to make use of Cocoa bindings if your application adopts the recommended design patterns.

## The Model-View-Controller Design Pattern

Cocoa applications generally adopt the Model-View-Controller (MVC) design pattern. When you develop a Cocoa application, you typically use model, view, and controller objects, each of which performs a different function. Model objects represent data and are typically saved to a file or some other permanent data store. View objects display model attributes. Controller objects act as go-betweens, to make sure that what a view displays is consistent with the corresponding model value and that any updates a user makes to a value in a view are propagated to the model. An understanding of the MVC design pattern is essential to fully understand and leverage Cocoa bindings. If you need to know more, read "The Model-View-Controller Design Pattern."

If you adopt the MVC design pattern, much of your application code is easier to reuse and extend—you can reuse model and view classes in different applications. Much of the implementation of a controller object consists of what is commonly referred to as "glue code." Glue code is the code that keeps the model values and views synchronized, and is unique to each application. It is typically tedious and cumbersome to write, contributes little to the fundamental function of the application, but you must do it well to provide a good user experience.

**Figure 1** Controllers provide glue code



Cocoa bindings replace much of the glue code with reusable controllers and provide an infrastructure that allows you to connect the user interface with an application's data.

Cocoa uses a number of terms that are commonly used in computer science. To avoid misunderstanding, they are defined in the "Terminology" section of *Key-Value Coding Programming Guide* with their particular meaning in the context of Cocoa bindings.

## What Is a Binding?

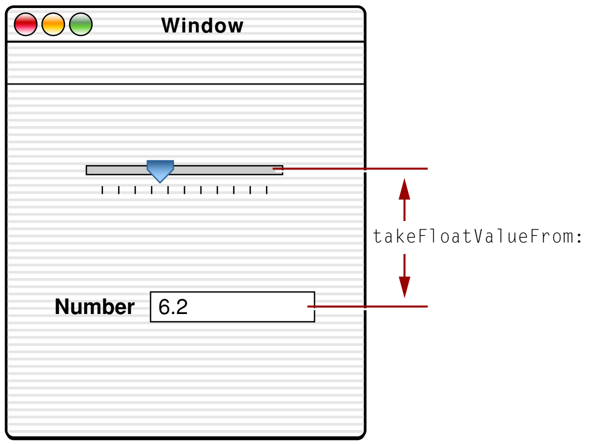
A binding is an attribute of one object that may be bound to a property in another such that a change in either one is reflected in the other. For example, the "value" binding of a text field might be bound to the temperature attribute of a particular model object. More typically, one binding might specify that a controller object "presents" a model object and another binding might specify that the value of a text field be tied to the temperature property of the object presented by the controller.

Although the following examples concentrate on simple cases, bindings are not restricted to the display of textual or numeric values. Among other things, a binding might specify the color in which text should be displayed, whether a view is hidden or not, or what message and arguments should be sent when a button is pressed.

## A Simple Example

Take as an example a very simple application in which the values in a text field and a slider are kept synchronized. Consider first an implementation that does *not* use bindings. The text field and slider are connected directly to each other using target-action, where each is the other's target and the action is `takeFloatValueFrom:` as shown in Figure 2. (If you do not understand this, you should read Getting Started With Cocoa.)

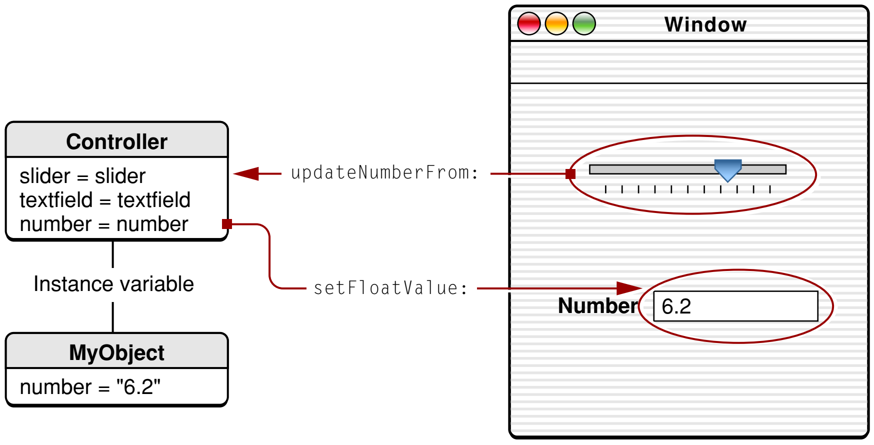
**Figure 2** A simple Cocoa application



This example illustrates the dynamism of the Cocoa environment—the values of two user interface objects are kept synchronized without writing any code, even without compiling. It also serves to illustrate the target-action design pattern (for more details, read "The Target-Action Paradigm").

The major flaw from which this example suffers is that, as it is, it has almost no real-world application. In order to find out the value to which either the slider or the text field has been set, and update a model attribute accordingly, you need connections to the text field and slider, and have to write some code. You typically use a controller that is connected to both (using outlets) and to which both are connected (using target-action), as illustrated in Figure 3 (page 13).

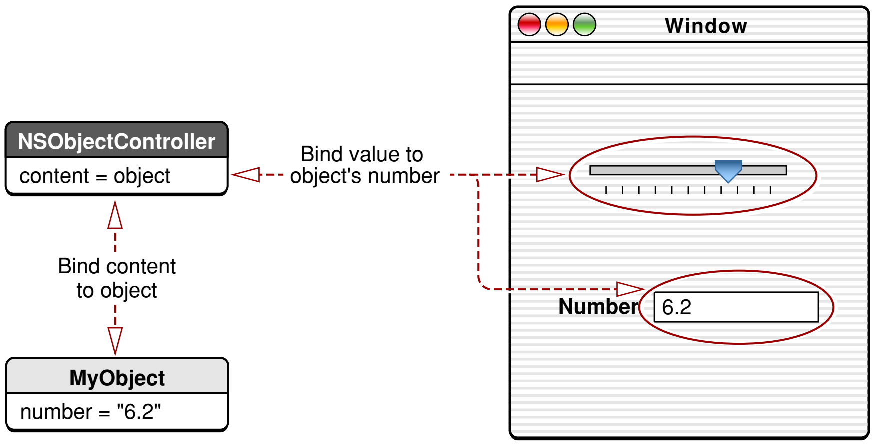
**Figure 3** Slider example using target-action



When a user moves the slider, it sends an action message to its target (the controller). The controller in turn updates the value in the model, and synchronizes the user interface (the text field and the slider). Although this example is not particularly difficult, the situation becomes more complex if you use more complicated models and displays, especially if you use, for example, table views that allow multiple selections, or if a value may be displayed in a different window. And you have to write all the code to support this functionality.

Cocoa bindings uses prebuilt controller objects (subclasses of NSController) and supporting technologies to keep values synchronized automatically. The application design for an implementation of the slider example that uses bindings is shown in Figure 4.

**Figure 4** Slider demonstration using bindings



Note that this implementation does not use the target-action pattern. The slider does not send an action message to the controller. Instead, as the slider moves, it informs the controller directly that the value of its content's number has changed and what the value is. The controller updates the model and in turn informs the text field and slider that the value they are displaying has changed. (In examples as simple as this controllers are not really necessary, however in most cases they are.) The mechanisms used to relay information are explained later in this article and in greater detail in "How Do Bindings Work?" (page 23), but it is important to appreciate that in most cases you will not have to write any glue code.

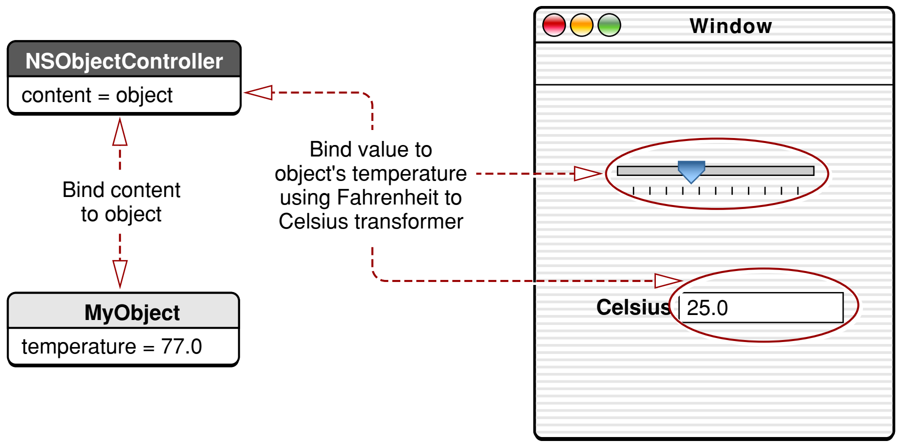
## Binding Options

Many bindings allow you to specify options to customize their behavior. There are three types of option: value transformers, placeholders, and other parameters.

A value transformer, as its name implies, applies a transformation to a value. A value transformer may also allow reverse transformations. The Foundation framework provides the abstract NSValueTransformer class and several convenient transformers, including one that negates a value—that is, it turns a Boolean `YES` into `NO` (and vice versa). You can also implement your own transformers.

To see how transformers might be useful, suppose that in the previous example the number in the model represents temperature in degrees Fahrenheit, but that you want to display the value in Celsius. You could implement a reversible value transformer that converts values from one scale to the other. If you then specify it as the transformer option for the text field and slider, as shown in Figure 5 (page 15), the user interface displays the temperature in Celsius, and any new values entered using the slider or text field converted to Fahrenheit.

**Figure 5** Displaying temperature using transformers



To learn more about transformers read *Value Transformer Programming Guide* (the article also shows an implementation of the Fahrenheit to Celsius transformer).

Placeholder options allow you to specify what a view should display: if the value of the property to which it is bound is null (nil); if there is no selection; if there is a multiple selection; or if for some other reason the value is not applicable.

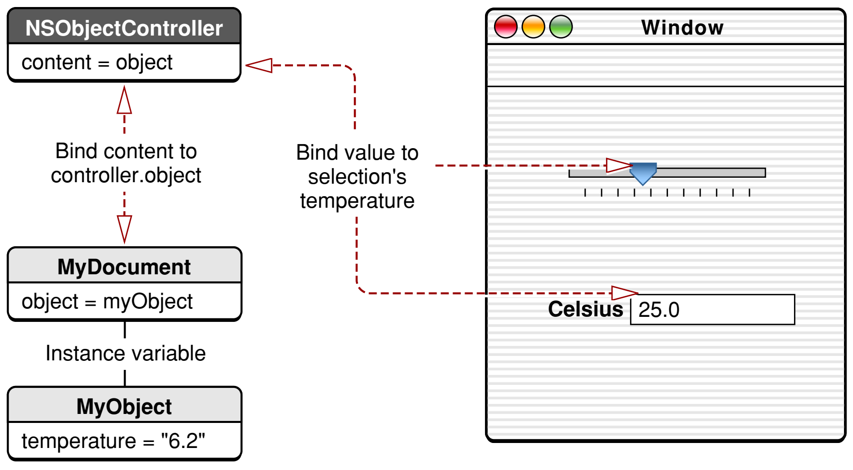
In addition to value transformers and placeholders, some bindings offer a variety of other options, such as whether the value of the binding is updated as edits are made to the user interface item, or whether the editable state of a user interface item is automatically configured based on the controller's selection. For a complete list of all the binding options available, see *Cocoa Bindings Reference*.

## Extending the MVC Design Pattern

The Cocoa bindings architecture extends the traditional Cocoa MVC configuration, where there is a single custom-built controller that manages the user interface. It provides a set of reusable controller classes that inherit from an abstract superclass, NSController. In a bindings-based application there may be several controllers—your own (such as an NSWindowController subclass, managing a document's user interface) and others that are subclasses of NSController and manage different parts of the user interface. You might also create your own subclasses of the standard Application Kit controller classes—in particular you might subclass NSArrayController to customize sorting and filtering behavior.

Other figures in this document present a convenient shorthand. Although the NSController instance is conceptually bound directly to its model object, in most situations the binding will be "indirect," to a variable in your document object, as shown in Figure 6.

**Figure 6** Typical bindings configuration using existing controller



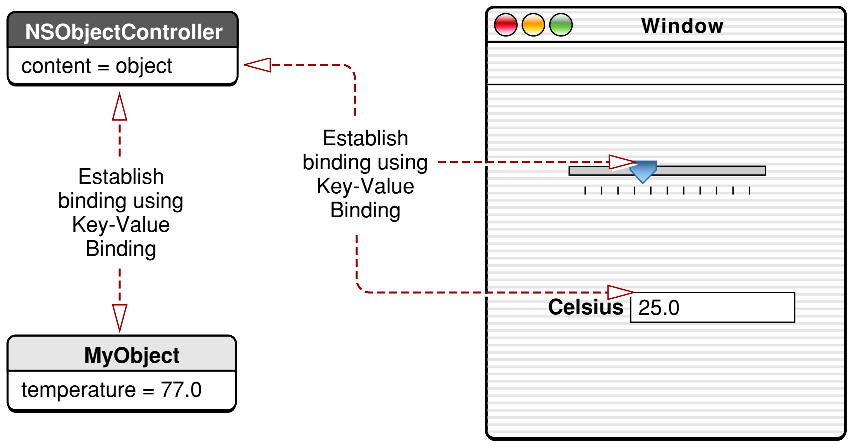
## Supporting Technologies

Cocoa bindings rely primarily on two other technologies, key-value coding (KVC) and key-value observing (KVO). Bindings themselves are established using key-value binding (KVB) as illustrated in Figure 7. In practice you typically need to understand these technologies only if you want to create your own custom view with bindings. If you want to use bindings, the only requirement that is imposed on you is that your model classes must be compliant with key-value coding conventions for any properties to which you want to bind.

### Key-Value Binding

A binding is established with a `bind:toObject:withKeyPath:options:` message which tells the receiver to keep its specified attribute synchronized—modulo the options—with the value of the property identified by the key path of the specified object. The receiver must watch for relevant changes in the object to which it is bound and react to those changes. The receiver must also inform the object of changes to the bound attribute. After a binding is established there are therefore two aspects to keeping the model and views synchronized: responding to user interaction with views, and responding to changes in model values.

**Figure 7** Bindings established using key-value binding


NSObjectController
content = object
Establish binding using Key-Value Binding
Establish binding using Key-Value Binding
MyObject
temperature = 77.0
Window
Celsius
25.0

In a view-initiated update a value changed in the user interface is passed to the controller, which in turn pushes the new value onto the model. To preserve the abstraction required to allow this to work with any controller or model object, the system uses a common access protocol—key-value coding.

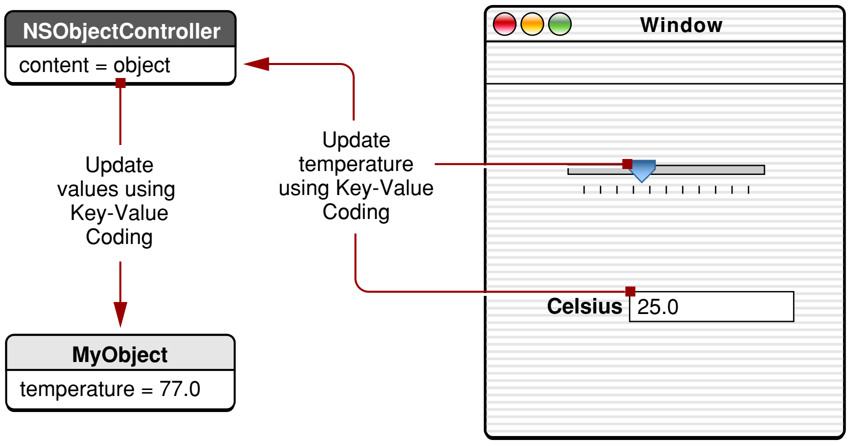
In a model-initiated update models notify controllers, and controllers notify views, of changes to values in which interest has been registered using a common protocol—key-value observing. Note that a model-initiated update can be triggered by direct manipulation of the model—for example by a Scripted Apple event—or as the result of a view-initiated update—a change to the temperature made by editing the Celsius field must be propagated back to the slider.

## Key-Value Coding

Key-value coding is a mechanism whereby you can access a property in an object using the property's name as a string—the "key." You can also use key paths to follow relationships between objects. For example, given an Employee class with an attribute `firstName`, you could retrieve an employee's first name using key-value coding with the key `firstName`. If Employee has a relationship called "manager" to another Employee, you could retrieve an employee's manager's first name using key-value coding with the key path `manager.firstName`. For more details, see *Key-Value Coding Programming Guide*.

Recall that a binding specifies the key path to a property to which a given attribute is bound. If the value in the slider or the text field is changed, it uses key-value coding—using the key path specified by the binding as the key—to communicate that change directly to the controller, as illustrated in Figure 8. Note that the arrows in this figure represent the direction in which messages are sent and in which information flows. The new value is passed from the user interface widget to the controller, and from the controller to the model.

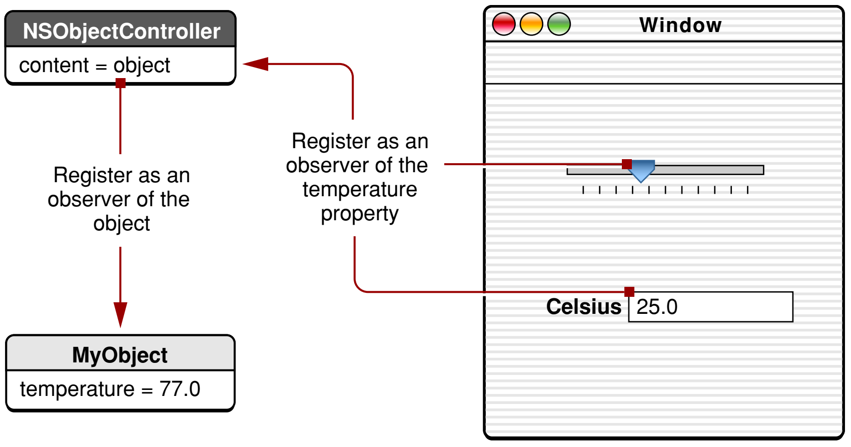
**Figure 8** Using key-value-coding to update values


NSObjectController
content = object
Update values using Key-Value Coding
MyObject
temperature = 77.0
Update temperature using Key-Value Coding
Window
Celsius
25.0

## Key-Value Observing

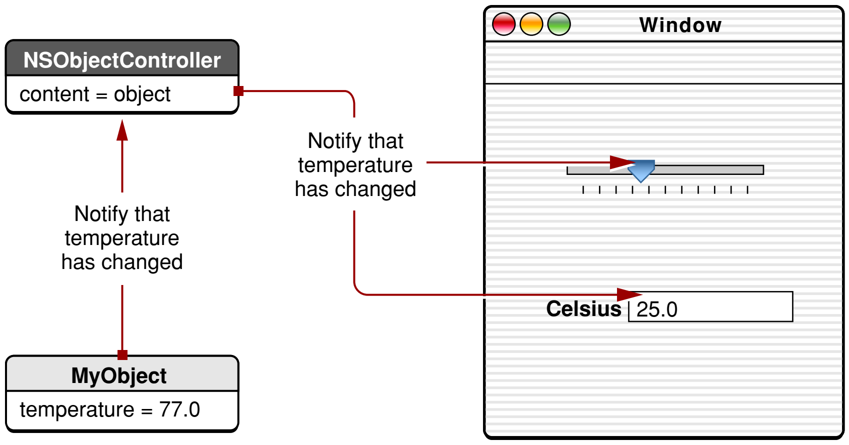
Key-value observing is a mechanism whereby an object can register with another to be notified of changes to the value of a property. When one object is bound to another object, it registers itself as an observer of the relevant property of that object. In the current example, the text field and slider register as observers of the temperature property of the controller's content, as illustrated in Figure 9.

**Figure 9** Key-value observing—registering observers


NSObjectController
content = object
Register as an observer of the object
MyObject
temperature = 77.0
Register as an observer of the temperature property
Window
Celsius
25.0

Note that the arrows shown in Figure 9 indicate direction of observation, not of data flow. Observation is a "passive" process (akin to registering to receive notifications from an NSNotificationCenter). When a value changes, the observed object sends a message to interested observers to notify them, as illustrated in Figure 10. The arrows in Figure 10 show the direction in which messages are sent.

**Figure 10** Key-value observing—notification of observers



## Why Are NSControllers Useful?

Bindings can, in principle, be made between almost any two objects, provided that they are KVC-compliant and KVO-compliant. A view could bind directly to a model object. Bindings-based applications, however, use controller objects to manage individual model objects and collections of model objects and to interface to the user preferences system.

It is possible to make bindings directly to your model objects or to controllers that do not inherit from NSController—however you lose (or must reimplement) functionality provided by the Application Kit's controller objects.

- NSController instances manage their current selection and placeholder values. This allows a view to display an appropriate value if the controller's selection is null, or if there is a multiple selection.
- NSController (and Application Kit user interface elements that support binding) implements the NSEditor and NSEditorRegistration protocols. The NSEditorRegistration protocol provides a means for an editor (a view) to inform a controller when it has uncommitted changes. The NSEditor protocol provides a means for requesting that the receiver commit or discard any pending edits.

  For example, if a user is typing in a text field and then clicks a button, the controller ensures that the model object is updated with the complete contents of the text field before the button action takes place.

  Although the methods are typically invoked on user interface elements by a controller they can also be sent to a controller, for example in response to a user's attempt to save a document or quit an application.

### NSController Classes

NSController is an abstract class. Its concrete subclasses are NSObjectController, NSUserDefaultsController, NSArrayController, and NSTreeController. NSObjectController manages a single object and provides the functionality discussed so far. NSUserDefaultsController provides a convenient interface to the preferences system.

NSArrayController and NSTreeController manage collections of model objects and track the current selection. The collection controllers also allow you to add objects to, and remove objects from, the content collection. The objects that the collection controllers manage don't even have to be in an array—your container can implement suitable methods ("indexed accessor" methods, defined in the NSKeyValueCoding protocol) to present the values to the controller as if they were in an array.

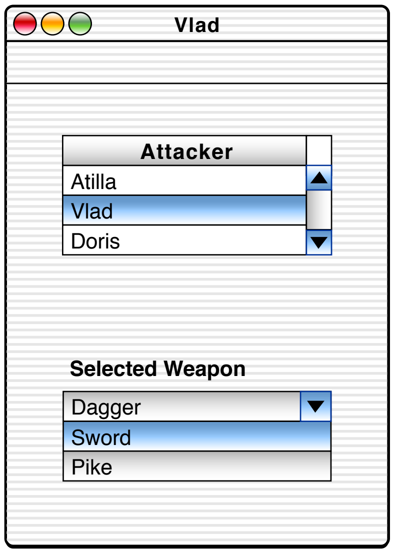
## What Can You Bind?

You can make bindings for most of the Application Kit view classes, such as NSButton and NSTableView. Using an array controller, for example, you can bind the contents of a pop-up menu to objects in an array. The remainder of this article presents an example that is moderately complex. Although the details are intentionally left vague it nevertheless serves to illustrate a number of points, and provides examples of more complex bindings.

## Real-World Example

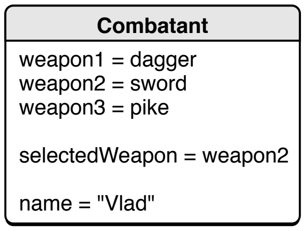
Consider a game application in which the user manages a number of combatants, from which they can select one as an attacker. A combatant can carry three weapons, one of which is selected at any time. In the application, the list of combatants is shown in a table view, the window's title shows the attacker's name, and a pop-up menu shows the currently selected weapon, as shown in Figure 11 (page 20).

**Figure 11** User interface for Combatants application



Combatants are represented by instances of the Combatant class. In the Combatant class, each weapon is referenced as a separate instance variable, as shown in Figure 12. By implementing suitable "indexed" accessor methods (defined by the key-value coding protocol), however, the Combatant class can allow an array controller to access the weapons as if they were in an array.

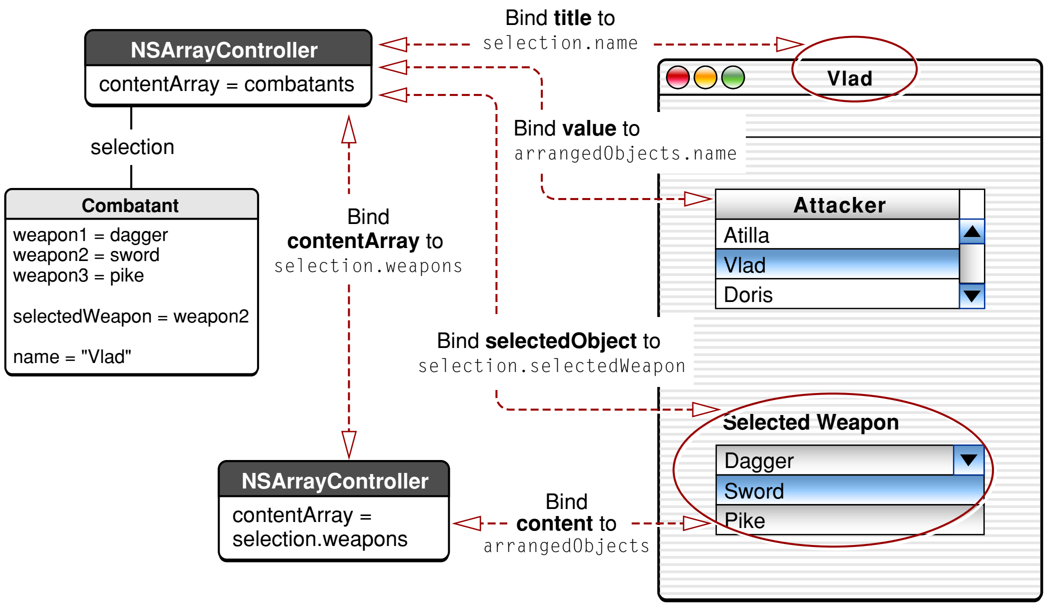
**Figure 12** Combatant class



When the user chooses an attacker from the table view, the window title is updated to reflect the attacker's name, and the title of the pop-up menu is updated to reflect the attacker's selected weapon. When the user activates the pop-up its contents are created dynamically from the set of weapons carried by the combatant. When the user selects a menu item, the combatant's selected weapon is set to that corresponding to that menu item. If a different attacker is selected, the pop-up, selection and window title update accordingly.

Figure 13 (page 21) illustrates how the user interface of the Combatants application can be implemented using bindings. The table view is bound to an array controller that manages an array of combatants. The window title is bound to the name of the selected combatant. The pop-up menu retrieves its list of items from an array controller bound to the attacker's weapons "array," and its selection is bound to the attacker's selected weapon.

**Figure 13** Combatants application managed by bindings


NSArrayController
contentArray = combatants
selection
Combatant
weapon1 = dagger
weapon2 = sword
weapon3 = pike
selectedWeapon = weapon2
name = "Vlad"
Bind title to selection.name
Bind value to arrangedObjects.name
Bind contentArray to selection.weapons
Bind selectedObject to selection.selectedWeapon
Bind content to arrangedObjects
NSArrayController
contentArray = selection.weapons
Vlad
Attacker
Atilla
Vlad
Doris
Selected Weapon
Dagger
Sword
Pike

This example illustrates a number of points:

- In an application you can use more than one controller object.
- Different aspects of a user interface element may be bound to different controllers.
- You can use your own custom model classes.

Finally, it should be emphasized that the example requires no actual code to set up the user interface—the controllers and bindings can all be created in Interface Builder. This represents a considerable reduction in programming effort compared with the traditional target-action based approach.

# How Do Bindings Work?

This article provides a conceptual explanation of how Cocoa bindings work. It describes:

- How connections between model and controller, and controller and view, are established using key-value binding
- Unbinding
- The NSEditor and NSEditorRegistration protocols
- The technologies that Cocoa bindings use to support communication between the model, view, and controller, namely key-value coding, and key-value observing
- How the various technologies interact

You should already be familiar with the concepts presented in "What Are Cocoa Bindings?" (page 11).

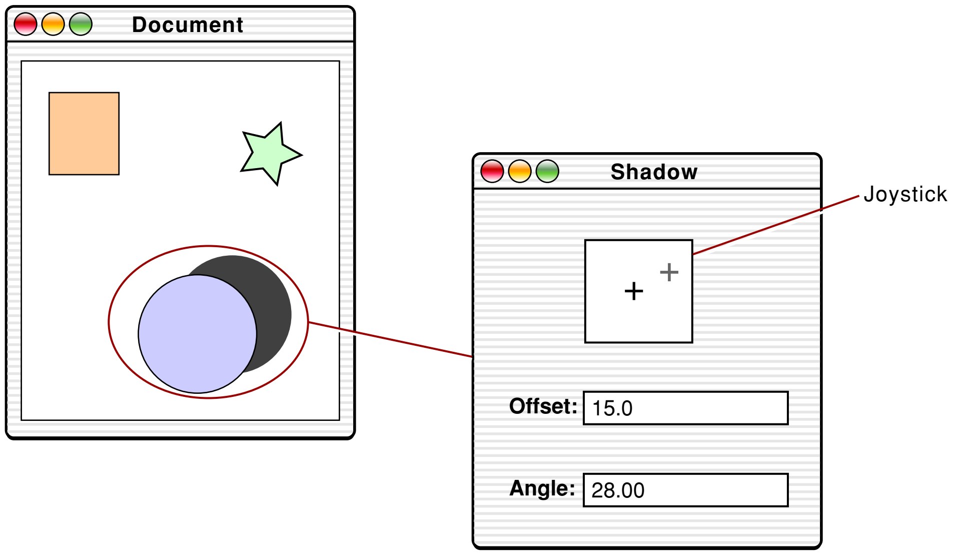
## Overview of the Supporting Technologies

This section presents an overview of the technologies that make Cocoa bindings work and how they interact. They are discussed in greater detail in the following sections.

Cocoa bindings rely on other technologies—key-value coding (KVC) and key-value observing (KVO)—to communicate changes between objects, and on key-value binding (KVB) to bind a value in one object to a property in another. Cocoa bindings also use two protocols—NSEditor and NSEditorRegistration—that help to ensure that any pending edits are either discarded or committed before user interface elements are disposed of.

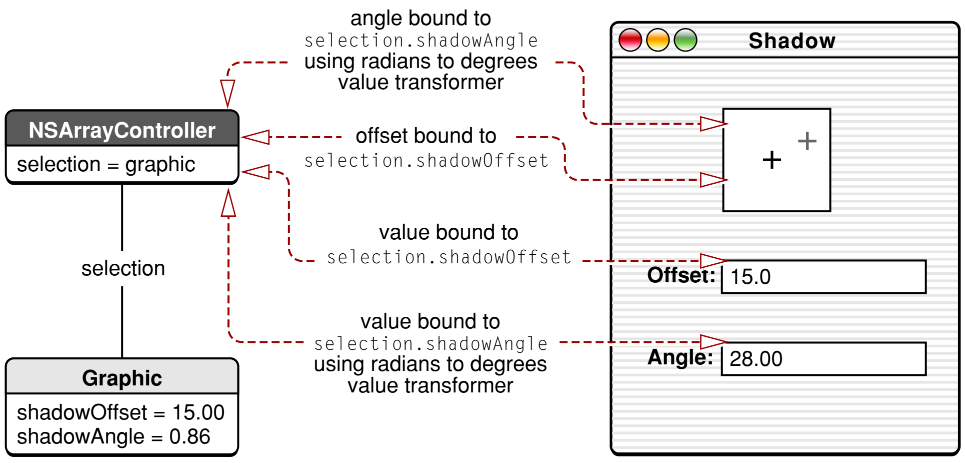
To understand how these technologies work together, consider a drawing application that allows the user to draw graphic objects such as circles and rectangles on screen. Among other properties, a graphic object has a shadow that may be offset a variable distance from the center of the graphic at an arbitrary angle. An inspector displays the offset and angle of the selected graphic object's shadow in a pair of text fields and a custom view—a joystick—as shown in Figure 1.

**Figure 1** Example drawing application



The implementation of the inspector is illustrated in Figure 2. Both the text fields and the joystick are bound to the `selection` of an NSArrayController. The controller's `contentArray` is bound to an array of Graphic objects. A Graphic has instance variables to represent its shadow's angle in radians and its offset from its center.

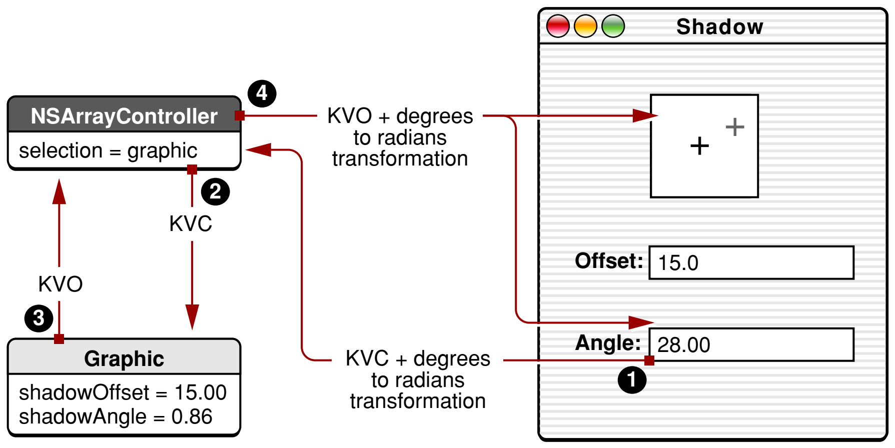
**Figure 2** Bindings for example graphics application



The text fields' values are bound to the angle and offset of the graphic object's shadow; the joystick provides a graphical representation of angle and offset. The angle in the text field is displayed in degrees, and the angle used internally by the joystick is specified in radians. The bindings for both of these specify a reversible value transformer that converts between radians and degrees.

The complete sequence of events that occurs when a user edits a value in the angle text field is illustrated in Figure 3 (page 25).

**Figure 3** The complete edit cycle



1. The user enters a new value in the Angle text field. The text field uses the NSEditorRegistration protocol to indicate that an edit has begun, and when it is complete. The text field's binding specifies a reversible radians-to-degrees value transformer, so the new value is converted to radians.

2. Using KVC, through the controller the view updates the model object's `shadowAngle` variable.

3. Through KVO, the model informs the controller that an update has been made to its `shadowAngle` variable.

4. Through KVO, the controller informs the joystick and the angle text field that an update has been made to its content's `shadowAngle` variable.

Notice that the Offset text field was not involved in the cycle in any way. Cocoa bindings impose no more overhead than is necessary.

The next sections explain in greater detail how bindings are established and how the underlying technologies operate.

## The Supporting Technologies in Detail

This section first describes the technologies that support bindings and shows how they play their parts. It also explains what steps you must take in order to take advantage of these technologies.

### Establishing Bindings with Key-Value Binding

Key-value binding is used to establish bindings. The NSKeyValueBindingCreation informal protocol declares methods to establish and remove bindings between objects. In addition, it provides a means for a class to advertise the bindings that it exposes.

In most cases you need to use `bind:toObject:withKeyPath:options:`, and then only when you establish bindings programatically. Use of the `unbind:` is discussed in "Unbinding." The other methods—the class method `exposeBinding:` and the instance methods `exposedBindings` and `valueClassForBinding:`—are useful only in an Interface Builder palette.

## NSEditor/NSEditorRegistration

Together the NSEditorRegistration and NSEditor protocols allow views to notify a controller that an edit is underway and to ensure that any pending edits are committed as and when necessary.

The NSEditorRegistration informal protocol is implemented by controllers to provide an interface for a view—the editor—to inform the controller when it has uncommitted changes. When an edit is initiated, the view sends the controller an `objectDidBeginEditing:` message. When the edit is complete (for example when the user presses Return) the view sends an `objectDidEndEditing:` message.

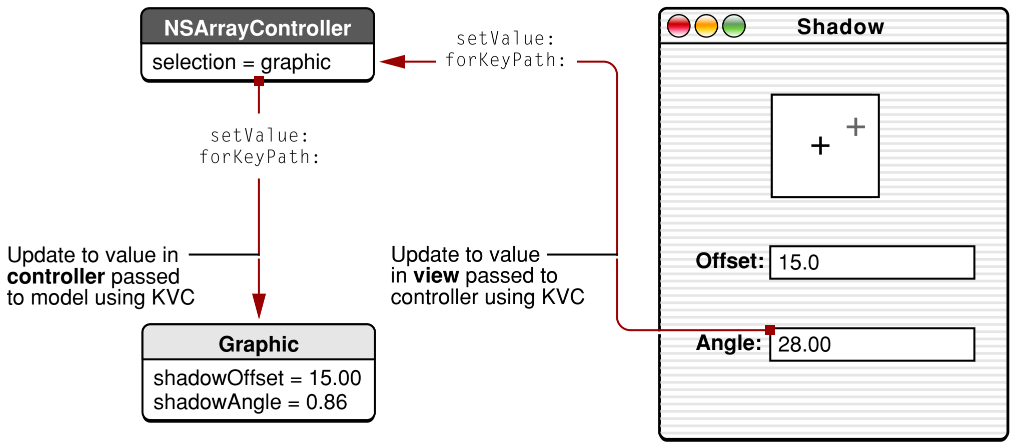
The controller is responsible for tracking which editors have uncommitted changes and requesting that they commit or discard any pending edits when appropriate—for example, if the user closes the window or quits the application. The request takes the form of a `commitEditing` or `discardEditing` message, defined by the NSEditor informal protocol. NSController provides an implementation of this protocol, as do the Application Kit user interface elements that support binding.

## Key-Value Coding

Key-value coding (KVC) provides a unified way to access an object's properties by name (key) without requiring use of custom accessor methods. The NSKeyValueCoding protocol specifies among others two methods, `valueForKey:` and `setValue:forKey:`, that give access to an object's property with a specified name. In addition, the methods `setValue:forKeyPath:` and `valueForKeyPath:` give access to properties across relationships using key paths of the form *relationship.property*, for example, `content.lastName`.

A binding for a given property specifies an object and a key path to a property of that object. Given this information, the bound object can use key-value coding—specifically `setValue:forKeyPath:`—to update the object to which it is bound without having to hard-code an accessor method, as illustrated in Figure 4.

**Figure 4** Key-value coding in Cocoa bindings



Since Cocoa bindings rely on KVC, your model and controller objects must be KVC-compliant for other objects to be able to bind to them. To support KVC you simply have to follow a set of conventions for naming your accessor methods, briefly summarized here:

- For an attribute or to-one relationship named `<key>`, implement methods named `<key>` and, if the attribute is read-write, `set<Key>:`—the case is important.
- For a to-many relationship implement either a method named `<key>` or both `countOf<Key>` and `objectIn<Key>AtIndex:`. The latter pair is useful if the related objects are not stored in an array. It is up to you to retrieve an appropriate value for the specified index—it does not matter how the value is derived. For a mutable to-many relationship, you should also implement either `set<Key>` (if you implemented `<key>`) or `insertObject:in<Key>AtIndex:` and `removeObjectFrom<Key>AtIndex:`.

For full details of all the methods declared by the NSKeyValueCoding protocol, see *Key-Value Coding Programming Guide*.

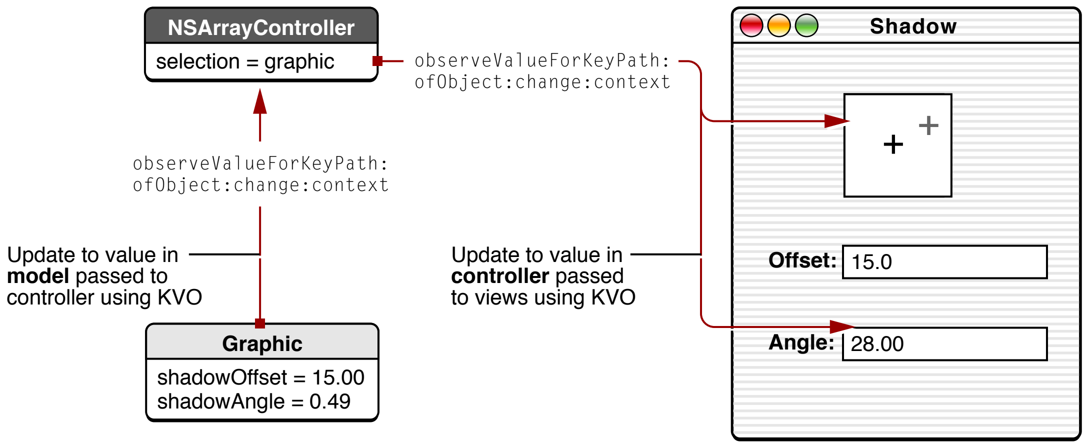
## Key-Value Observing

KVO is a mechanism that allows an object to register to receive notifications of changes to values in other objects. To register, an observer sends the object it wants to observe an `addObserver:forKeyPath:options:context:` message that specifies:

- The object that is observing (very often `self`)
- The key path to observe (for example, `selection.name`)
- What information will be sent on notification (for example, old value and new value)
- Optionally, contextual information to be sent back on notification (for example, a flag to indicate what binding is being affected)

An observed object communicates changes directly to observers by sending them an `observeValueForKeyPath:ofObject:change:context:` message (defined by the NSKeyValueObserving informal protocol)—there is no independent notifier object akin to NSNotificationCenter. Figure 5 illustrates how a change to a model value (the shadow angle) is communicated to views using KVO.

**Figure 5** Key-value observing in Cocoa bindings


NSArrayController
selection = graphic
observeValueForKeyPath:ofObject:change:context
observeValueForKeyPath:ofObject:change:context
Update to value in **model** passed to controller using KVO
Update to value in **controller** passed to views using KVO
Graphic
shadowOffset = 15.00
shadowAngle = 0.49
Shadow
Offset: 15.0
Angle: 28.00

The message is sent to observers for every change to an observed property, independently of how the change was triggered. (Note that the protocol itself makes no assumptions about what the observer actually does with the information.)

One other important aspect of KVO is that you can register the value of a key as being dependent on the value of one or more others. A change in the value associated with one of the "master" keys triggers a change notification for the dependent key's value. For example, the drawing bounds of a graphic object may be dependent on the offset and angle of the associated shadow. If either of those values changes, any objects observing the drawing bounds must be notified.

The main requirement to make use of KVO is that your model objects be KVO-compliant. In most cases this actually requires no effort—the runtime system adds support automatically. By default, all invocations of KVC methods result in observer notifications. You can also implement manual support—see *Key-Value Observing Programming Guide* for more details.

### Unbinding

Typically the only reason you would explicitly unbind an object is if you modify the user interface programatically and want to remove a binding. If you change an objects binding's values it should first clear any preexisting values.

If you implement a custom view or controller with custom bindings, you should ensure that it clears any bindings before it is deallocated—in particular it should release any objects that were retained for the specified binding in the `bind:toObject:withKeyPath:options:` method and should deregister as an observer of any objects for which it registered as an observer. It may be convenient to implement this logic in an `unbind:` method that you then call as the first step in `dealloc`.

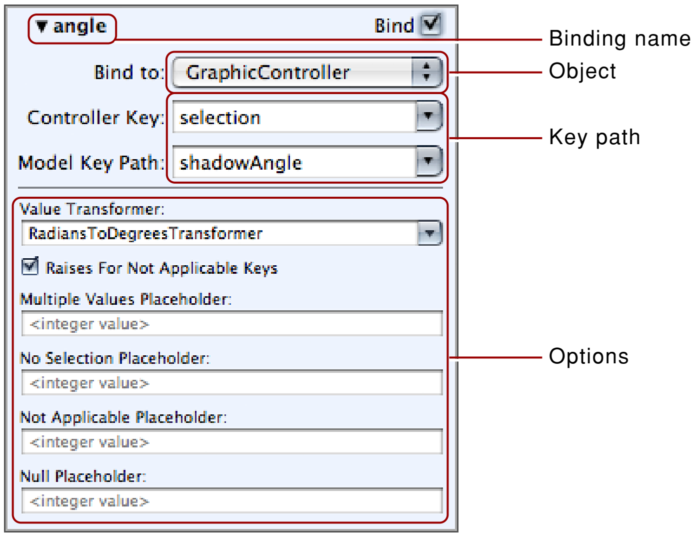
## Bindings in More Detail

This section examines bindings in more detail. It decomposes—from the perspective of a custom view, a joystick—what happens when a binding is established. This serves two purposes: it makes explicit what code is involved in establishing a binding and responding to changes, and it gives a conceptual overview of how to create your own bindings-enabled views.

A binding specifies what aspect of one object should be bound to what property in another, such that a change in either is reflected in the other. For a given binding, an object therefore records the target object for the binding, the associated key path, and any options that were specified.

You can establish bindings easily in Interface Builder using the Bindings pane of the Info window. In Interface Builder, the `angle` binding for a joystick that displays the offset and angle of graphics object's shadow might look like Figure 6.

**Figure 6** Bindings for a joystick's angle in Interface Builder



Bind to: GraphicController
Controller Key: selection
Model Key Path: shadowAngle
Value Transformer: RadiansToDegreesTransformer
✓ Raises For Not Applicable Keys
Multiple Values Placeholder: <integer value>
No Selection Placeholder: <integer value>
Not Applicable Placeholder: <integer value>
Null Placeholder: <integer value>

Establishing this binding in Interface Builder is equivalent to programatically sending this message to the joystick:

```
[joystick bind:@"angle" toObject:GraphicController
withKeyPath:@"selection.shadowAngle" options:options];
```

The arguments have the following meanings:

`@"angle"`
: Identifies an attribute whose value can be bound to the value of a property in another object. Note that the binding name need not necessarily correspond to the name of an actual instance variable.

`GraphicController`
: The object containing the bound-to property.

`@"selection.shadowAngle"`
: The key path that identifies the bound-to property.

`options`
: A dictionary that specifies any options such as placeholders, or in this case, a value transformer.

The information defined by the arguments can be stored in the bound object (in this case the joystick) as instance variables, as discussed next.

This example, and those that follow, assume that the joystick is represented by the class Joystick with instance variables as defined in the interface shown in Listing 1.

**Listing 1** Interface for the Joystick class

```
@interface Joystick : NSView
{
    float angle;
    id observedObjectForAngle;
    NSString *observedKeyPathForAngle;
    NSValueTransformer *angleValueTransformer;
    // ...
}
```

In its `bind:toObject:withKeyPath:options:` method an object must as a minimum do the following:

- Determine which binding is being set
- Record what object it is being bound to using what keypath and with what options
- Register as an observer of the keypath of the object to which it is bound so that it receives notification of changes

The code sample in Listing 2 shows a partial implementation of Joystick's `bind:toObject:withKeyPath:options:` method dealing with just the `angle` binding.

**Listing 2** Partial implementation of the bind:toObject:withKeyPath:options method for the Joystick class

```
static void *AngleBindingContext = (void *)@"JoystickAngle";

- (void)bind:(NSString *)binding
 toObject:(id)observableObject
 withKeyPath:(NSString *)keyPath
 options:(NSDictionary *)options
{
 // Observe the observableObject for changes -- note, pass binding identifier
 // as the context, so you get that back in observeValueForKeyPath:...
 // This way you can easily determine what needs to be updated.

if ([binding isEqualToString:@"angle"])
 {
    [observableObject addObserver:self
                   forKeyPath:keyPath
                  options:0
                  context:AngleBindingContext];

    // Register what object and what keypath are
    // associated with this binding
    observedObjectForAngle = [observableObject retain];
    observedKeyPathForAngle = [keyPath copy];

    // Record the value transformer, if there is one
    angleValueTransformer = nil;
    NSString *vtName = [options objectForKey:@"NSValueTransformerName"];
    if (vtName != nil)
    {
        angleValueTransformer = [NSValueTransformer
            valueTransformerForName:vtName];
    }
 }
 // Implementation continues...
```

This partial implementation does not record binding options other than a value transformer (although it may simply be that the binding does not allow for them). It nevertheless illustrates the basic principles of establishing a binding. Notice in particular the contextual information passed in the `addObserver:forKeyPath:options:context:` message; this is returned in the `observeValueForKeyPath:ofObject:change:context:` method and can be used to determine which binding is affected by the value update.

## Responding to Changes

As noted earlier, there are two aspects to change management—responding to view-initiated changes that must be propagated ultimately to the model, and responding to model-initiated changes that must be reflected in the view. This section illustrates both, and shows how KVC and KVO play their parts.

### View-Initiated Updates

Recall that the joystick was bound to the controller with the following method:

```
[joystick bind:@"angle" toObject:GraphicController
withKeyPath:@"selection.shadowAngle" options:options];
```

From the perspective of view-initiated updates the method can be interpreted as follows:

`bind: @"angle"`
: If whatever is associated with `angle` changes,

`toObject: GraphicController`
: tell the specified object (GraphicController) that

`withKeyPath: @"selection.shadowAngle"`
: the value of its (the GraphicController's) `selection.shadowAngle` has changed

`options: options`
: using any of these options that may be appropriate.

If the value associated with `angle` changes—typically when a user clicks or drags the mouse within the view—the joystick should pass the new value to the controller using KVC, as illustrated in Figure 4. The joystick should therefore respond to user input as follows:

- Determine new values for angle and offset
- Update its own display as appropriate
- Communicate new values to the controller to which it is bound

Listing 3 shows a partial implementation of an update method for the Joystick class. The excerpt deals just with the `angle` binding. It illustrates the use of key-value coding to communicate the new value (transformed by the value transformer if appropriate) to the observed object.

**Listing 3** Update method for the Joystick class

```
-(void)updateForMouseEvent:(NSEvent *)event
{
    float newAngleDegrees;
    // calculate newAngleDegrees...

    [self setAngle:newAngleDegrees];

    if (observedObjectForAngle != nil)
    {
        NSNumber *newControllerAngle = nil;
        if (angleValueTransformer != nil)
        {
            newControllerAngle =
                [angleValueTransformer reverseTransformedValue:
                    [NSNumber numberWithFloat:newAngleDegrees]];
        }
        else
        {
            newControllerAngle = [NSNumber numberWithFloat:newAngleDegrees];
        }
        [observedObjectForAngle setValue: newControllerAngle
                  forKeyPath: observedKeyPathForAngle];
    }
    // ...
}
```

Note that this example omits several important details, such as editor registration and checking that the value transformer allows reverse transformations.

## Model-Initiated Updates

Recall again that the joystick was bound to the controller with the following method:

```
[joystick bind:@"angle" toObject:GraphicController
withKeyPath:@"selection.shadowAngle" options:options];
```

From the perspective of model-initiated updates the method can be interpreted as follows:

`toObject: GraphicController`
: If the GraphicController's

`withKeyPath:@"selection.shadowAngle"`
: `selection.shadowAngle` changes

`bind:@"angle"`
: update whatever is associated with the exposed `angle` key

`options:options`
: using the options specified (for example, using a value transformer).

The receiver therefore registered as an observer of the specified object's key path (`selection.shadowAngle`) in its `bind:toObject:withKeyPath:options:` method, as was shown in Listing 2. Observed objects notify their observers by sending them an `observeValueForKeyPath:ofObject:change:context:` message. Listing 4 shows a partial implementation for the Joystick class for handling the observer notifications that result.

The fundamental requirement of the `observeValueForKeyPath:ofObject:change:context:` method is that the value associated with the relevant attribute is updated. This excerpt also shows how it can capture placeholder information that might be used in the display method to give visual feedback to the user, in this case using an instance variable that indicates that for some reason the angle is "bad."

**Listing 4** Observing method for the Joystick class

```
- (void)observeValueForKeyPath:(NSString *)keyPath
              ofObject:(id)object
```

```
            change:(NSDictionary *)change
               context:(void *)context
{
    // You passed the binding identifier as the context when registering
    // as an observer--use that to decide what to update...

    if (context == AngleObservationContext)
    {
        id newAngle = [observedObjectForAngle
            valueForKeyPath:observedKeyPathForAngle];
        if ((newAngle == NSNoSelectionMarker) ||
            (newAngle == NSNotApplicableMarker) ||
            (newAngle == NSMultipleValuesMarker))
        {
            badSelectionForAngle = YES;
        }
        else
        {
            badSelectionForAngle = NO;
            if (angleValueTransformer != nil)
            {
                newAngle = [angleValueTransformer
                    transformedValue:newAngle];
            }
            [self setValue:newAngle forKey:@"angle"];
        }
    }
    // ...

    [self setNeedsDisplay:YES];
}
```

In most controls the display method alters the visual representation depending on the current selection.

# Providing Controller Content

Controllers require content to manipulate and there are a number of options for setting this content. It can be done programmatically, through bindings, or automatically in response to actions configured in Interface Builder. This article describes the various methods of setting and modifying a controller's content.

## Setting the Content of a Controller

NSObjectController and its subclasses are initialized with the method `initWithContent:`, passing a content object or `nil` if you intend to use the content bindings. You can explicitly set the content of an existing controller using the `setContent:` method. It is far more common to provide content for controllers by establishing a binding to one of their exposed Controller Content bindings.

NSObjectController exposes a single binding for content called `contentObject`. You can establish a binding from `contentObject` to any object that is key-value-coding and key-value-observing compliant for the keys that you intend to have the controller operate on.

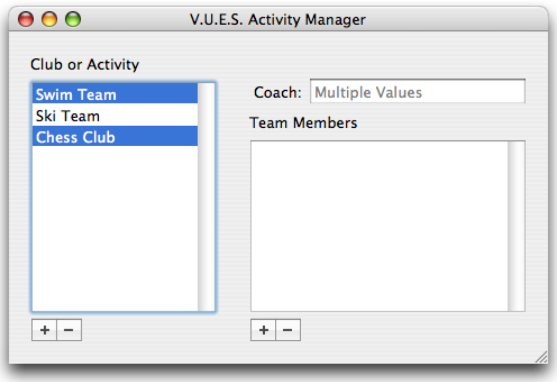
The collection controllers expose additional bindings: `contentArray`, `contentSet`, and `contentArrayForMultipleSelection`.

The `contentArray` binding is bound to an NSArray or an object that implements the appropriate array indexed accessor methods. Similarly the `contentSet` binding is bound to an NSSet object or an object that implements the appropriate set indexed accessor methods. The indexed accessor patterns are described in Indexed Accessor Patterns for To-Many Properties in the *Key-Value Coding Programming Guide*.

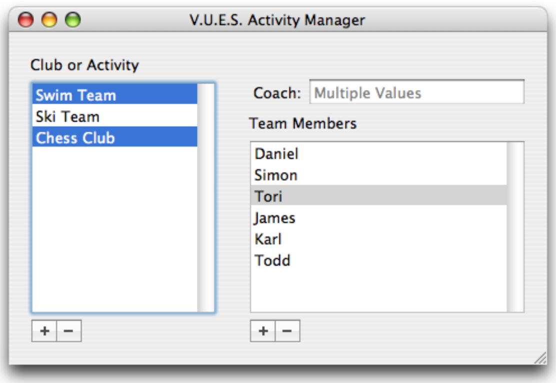
The `contentArrayForMultipleSelection` bindings is a special binding that is enabled only after establishing the `contentArray` or `contentSet` binding. The `contentArrayForMultipleSelection` binding is used as a fallback for the content of the controller when the `contentArray` or `contentSet` bindings return the multiple values marker. It allows you to use a different object and key path as the collection content in these cases and is often used when implementing a master-detail style interface.

For example, Figure 1 shows a typical master-detail interface. The array controller that provides the list of activities is designated as the master controller and the names of the activities are displayed in the table view on the left. A second array controller is the detail controller and provides the names of members displayed in the table view on the right.

**Figure 1** Master-detail interface with and without contentArrayForMultipleSelection



When only the contentArray is bound to `selection.members` the detail table view is empty when the master array controller has a multiple selection.



When the contentArrayForMultipleSelection binding is also bound to `selection.@distinctUnionOfArrays.members` the detail table view displays the members in all the selected activities.

The detail array controller's `contentArray` binding is bound to the master array controller object with the key path `selection.members`. The `value` binding of the column in the detail table view is bound to the detail array controller's `arrangedObjects.name` key path. When a single activity is selected in the master table view, the detail table view displays the names of the to-many members relationship.

However, what happens if the master table view is configured to allow multiple selection? If only the detail array controller's `contentArray` is bound, the detail table view is empty. While this is logical, it isn't necessarily the desired results. A better option might be to display a unique list of the members in the selected activities. This is where the `contentArrayForMultipleSelection` binding and the key-value coding collection operators come into play.

By establishing a binding from `contentArrayForMultipleSelection` to the master array controller using the key path `selection.@distinctUnionOfArrays.members`, the detail table view will be populated with the names of the users in all the selected activities. Because the `@distinctUnionOfArrays` operator was used, members that are common to both activities do not appear as duplicate names in the detail table view.

The `contentArrayForMultipleSelection` binding expects an array of data; it is not directly compatible with set collections that are bound to the `contentSet` binding. You must use the `@distinctUnionOfSets` or `@unionOfSets` set operator to convert the set to an array. The collection operators are described in "Set and Array Operators" in the *Key-Value Coding Programming Guide*.

Note that when the master array controller has a multiple selection the detail array controller's add and remove buttons are disabled. The buttons' `enabled` bindings are bound to the detail array controller's `canAdd` and `canRemove` methods. The detail array controller automatically knows that it is unable to add and remove items to the composite array and updates the `canAdd` and `canRemove` state, causing the buttons to be disabled.

## Traversing Tree Content with an NSTreeController

An NSTreeController requires that you describe how it should traverse the tree of objects, by specifying a child key path. This key path can be set programmatically using the NSTreeController method `setChildrenKeyPath:` or specified in the tree controller's inspector panel in Interface Builder.

All child objects for the tree must be key-value-coding compliant for the same child key path. If necessary you should implement accessor methods in your model classes, or categories on those classes, that map the child key to the appropriate class-specific method name.

An optional count key path can be specified that, if provided, returns the number of child objects available. The count key path is set programmatically using the `setCountKeyPath:` method, or in the controller's inspector panel in Interface Builder. Your model objects are expected to update the value of the count key path in a key-value-observing compliant method.

**Warning:** Providing the count key path to an NSTreeController instance disables the `add:`, `addChild:`, `remove:`, `removeChild:`, or `insert:` methods.

You can optionally provide a leaf key path that specifies a key in your model object that returns `YES` if the object is a leaf node, and `NO` if it is not. Providing this key path prevents the NSTreeController from having to determine if a child object is a leaf node by examining the child object and as a result improve performance. The leaf key path is set programmatically using the `setLeafKeyPath:` method, or in the controller's inspector panel in Interface Builder. This key path affects how an NSOutlineView or NSBrowser bound to the tree controller displays disclosure triangles:

- If the leaf key path for the model object returns `YES`, the outline view or browser does not show a disclosure triangle.
- If the leaf key path for the model object returns `NO`, then it always shows the disclosure triangle.
- If no leaf key path is configured for the controller, then the count or child key path of the model is queried to determine how many child objects are present. If there are 0 child objects, the disclosure triangle is not displayed, otherwise it is.

## Specifying the Class of a Controller's Content

In order for a controller to create new content objects automatically or in response to the target-action methods, it must know the appropriate class to use.

Controllers can be configured in one of two modes: object mode or entity mode. In object mode the content class is specified by the method `setObjectClass:` or in the controller inspector panel in Interface Builder. If the controller is configured in entity mode, the class is determined by the name of the entity or by the relationship that the entity defines for the key. The entity name is set using `setEntityName:` or in the controller inspector panel in Interface Builder.

If the controller is in object mode, the method `newObject` is used to create new objects. The default implementation simply allocates a new object of the class specified by `objectClass` or the `entityName` and sends the object a standard `init` message with no arguments. If your content objects require more complex initialization, you can subclass the appropriate controller class and override the `newObject` method.

An NSObjectController expects the content object to be of the class specified by the object class or entity name. When using the NSArrayController and NSTreeController the object class refers to the individual content objects, rather than the collection that holds the objects. In both cases the collections are expected to be key-value-coding compatible with arrays or sets, depending on the binding providing the content for the controller.

You are not restricted to having content of a single object class. You can create and insert objects of any class using one of the programmatic manipulation methods discussed in "Programmatically Modifying a Controller's Contents" (page 38).

## Automatically Prepares Content

NSObjectController and its classes provide support for automatically creating content for a controller when it is loaded from a nib file. This is typically configured in the controller inspector in Interface Builder by enabling the "Automatically Prepares Content" option. When this option is enabled, the controller creates and populates the content object or content collection when the controller is loaded from a nib file by calling the controller's `prepareContent` method.

For example, when an NSObjectController that has an object class of NSMutableDictionary is loaded from a nib and automatically prepares content is selected, the content of the object controller will be set to a newly instantiated, empty NSMutableDictionary instance.

Similarly, if an NSArrayController that has an object class of Activity is loaded, the content is set to a newly instantiated NSMutableArray containing a single instance of Activity. NSTreeController acts the same way.

If the controller that is loaded is in entity mode, then the data corresponding to the entity name is fetched and is filtered using the controller's configured filter predicate.

## Programmatically Modifying a Controller's Contents

When you modify a controller's content object the only restriction is that you must do it in a key-value-observing compliant manner so that the controller is informed of the changes. NSObjectController and its subclasses provide a number of methods that allow you to modify the contents of a controller programmatically.

NSObjectController offers the `addObject:` and `removeObject:` methods. When used with an NSObjectController, they are synonymous with the `setContent:` method, passing the parameter object or `nil` respectively.

The `addObject:` and `removeObject:` methods have somewhat different behavior for NSArrayController. In this case their behavior is the same as NSArray's `addObject:` and `removeObject:` methods. Unlike NSArray's implementations, these methods inform the array controller of the changes so that they can be reflected in the user interface.

NSArrayController extends the basic add and remove functionality with the following methods:

```
- (void)addObjects:(NSArray *)objects;
- (void)removeObjects:(NSArray *)objects;
- (void)removeObjectsAtArrangedObjectIndexes:(NSIndexSet *)indexes;
- (void)removeObjectAtArrangedObjectIndex:(unsigned int)index;

- (void)insertObjects:(NSArray *)objects atArrangedObjectIndexes:(NSIndexSet
*)indexes;
- (void)insertObject:(id)object atArrangedObjectIndex:(unsigned int)index;
```

The `addObjects:` and `removeObjects:` methods add or remove the objects passed as parameters from the collection. The method `removeObjectsAtArrangedObjectIndexes:` method iterates through the passed indexes, removing each object from the collection. The method `removeObjectAtArrangedObjectIndex:` removes the single object at the specified index.

The `insertObjects:atArrangedObjectIndexes:` iterates through the array of objects passed as the first parameter, inserting each object into the arranged collection at the corresponding index in the NSIndexSet. Similarly, the `insertObject:atArrangedObjectIndex:` method inserts the single object specified as the first parameter into the collection at the specified index.

NSTreeController provides four additional methods that operate in a similar fashion, but use NSIndexPaths to specify the location in the collection rather than simple indexes:

```
-(void)removeObjectsAtArrangedObjectIndexPaths:(NSArray *)indexPaths;
-(void)removeObjectAtArrangedObjectIndexPath:(NSIndexPath *)indexPath;

-(void)insertObject:(id)object atArrangedObjectIndexPath:(NSIndexPath *)indexPath;
-(void)insertObjects:(NSArray *)objects atArrangedObjectIndexPaths:(NSArray
*)indexPath;
```

**Note:** NSArrayController and NSTreeController are optimized for insertions and deletions made with these methods. Using these methods can provide a performance increase over modifying the model objects directly and relying on key-value observing.

## Modifying Controller Content by Target-Action

In addition to the programmatic add, insert, and remove methods, the controller classes implement several target-action methods for modifying the controller's content. These methods are typically configured as the actions for buttons in Interface Builder.

**Note:** In Mac OS X v10.4 the target-action methods are deferred so that the error mechanism can provide feedback as a sheet.

NSObjectController provides the following target-action methods:

```
- (void)add:(id)sender;
- (void)remove:(id)sender;
```

The `add:` method creates a new content object using the controller's `newObject` method and sets it as the content object of the controller if there is currently no content. The `remove:` method sets the content object to `nil`.

NSArrayController overrides NSObjectController's `add:` and `remove:` methods and adds the following method:

```
- (void)insert:(id)sender;
```

With an array controller `add:` creates a new object using the controller's `newObject` method and appends it to the controller's content collection. The `remove:` method removes the currently selected objects in the array controller from the collection. The `insert:` method creates a new object using the controller's `newObject` method, and inserts it after the current selection.

When the controller is in entity mode the remove semantics are dependent on the configuration of the binding and on the entities definition of the relationship. A remove may result in the current selection being removed from a relationship or being removed from the object graph entirely.

NSTreeController adds the following methods:

```
- (void)addChild:(id)sender;
- (void)insertChild:(id)sender;
```

The `addChild:` method creates and inserts a new object at the end of the tree controller's collection. The `insertChild:` method inserts a new child object relative to the current selection.

Again, the semantics are slightly different if the controller is in entity mode. The `add:` and `insert:` actions use the `newObject` method to create the object that is added to the collection. In object mode the `addChild:`, and `insertChild:` create objects of the class specified by `objectClass`, but do not use the `newObject` method to do so. In entity mode or if the parent object is a managed object subclass, the entity defines the class of object created for and `newObject` is never called.

In order to enable and disable the target-action buttons as appropriate, the object controller classes provide several methods that return Boolean values indicating which actions are currently possible, taking into account the controller's current selection and configuration.

```
//NSObjectController
- (BOOL)canAdd;
- (BOOL)canRemove;
- (BOOL)isEditable;

//NSArrayController
- (BOOL)canInsert;

//NSTreeController
- (BOOL)canAddChild;
- (BOOL)canInsertChild;
```

The enabled binding of controls are typically bound to the controller using one of these methods. As the controller's selection changes the values returned by these methods are updated and the bound user interface items are automatically enabled or disabled as appropriate.

## De-coupling a Controller from its Content Bindings

In order to programmatically de-couple a controller from a content object that is bound to `contentObject`, `contentArray` or `contentSet`, you must break the binding connection and set the controller's content to `nil`.

```
    [theController unbind:@"contentArray"];
    [theController setContent:nil];
```

# Working With a Controller's Selection

NSObjectController and its subclasses NSArrayController and NSTreeController support tracking of the currently selected object or objects. This article explains how to get a controller's current selection, change the current selection, and fine-tune the selection behaviors.

## Getting a Controller's Currently Selected Objects

There are two methods that are commonly used to access the objects that are currently selected: `selection` and `selectedObjects`.

NSObjectController and its subclasses implement the `selection` method. This method returns a proxy object that represents the receiver's current selection. The proxy is fully key-value-coding compliant.

When you request a key's value from the selection proxy it returns the value, or a **selection marker**. Placeholder markers provide additional information about the selection. There are three placeholder markers defined in the NSPlaceholders informal protocol:

- NSNoSelectionMarker

  The NSNoSelectionMarker indicates that there are no items selected in the controller.

- NSNotApplicableMarker

  The NSNotApplicableMarker indicates that the underlying object is not key-value-coding compliant for the requested key.

- NSMultipleValuesMarker

  The NSMultipleValuesMarker indicates that more than one object is selected in the controller and the values for the requested key aren't the same.

  By default controllers return the NSMultipleValuesMarker only when the values for the requested key differ. For example, if the value for `selection.name` returns an array containing three strings—"Tony", "Tony", "Tony"—the string "Tony" is returned instead of the NSMultipleValuesMarker.

  A collection controller can be configured—either programmatically using the method `setAlwaysUsesMultipleValuesMarker:` or by checking the "Always uses multiple values marker" checkbox in Interface Builder—such that it always returns NSMultipleValuesMarker when multiple items are selected, even if the values are equal.

Some bindings, such as NSTextField's `value` binding, allow you to replace selection markers with custom values that are treated as placeholder values by the controls. These replacement values are specified in the Bindings inspector in Interface Builder. Bindings established programmatically can provide values for the `NSNoSelectionPlaceholderBindingOption`, `NSNotApplicablePlaceholderBindingOption`, `NSNullPlaceholderBindingOption` or `NSRaisesForNotApplicableKeysBindingOption` keys, defined in the NSKeyValueBindingCreation informal protocol, in the options dictionary passed to `bind:toObject:withKeyPath:options:`. The NSPlaceholders protocol also provides two class methods `setDefaultPlaceholder:forMarker:withBinding:` and `defaultPlaceholderForMarker:withBinding:` that allow you to provide default placeholders for the specified selection markers for a binding.

Often you need to directly access the objects currently selected by the controller, rather than the proxy object returned by `selection`. NSObjectController and its subclasses provide the `selectedObjects` method to allow you to do just that. This method returns an array containing the objects that are currently selected by the receiver. NSObjectController's implementation returns an array containing a single object, the content object.

**Note:** You can establish bindings to a controller's `selection` method or the `selectedObjects` method. However, you should avoid binding *through* the `selectedObjects` array, for example `selectedObjects.name`. Instead, you should use `selection.name`. Similarly, you should avoid observing keys through the array returned by `selectedObjects`. The proxy returned by `selection` is more efficient at managing changes in key-value observing as the selection changes.

## Changing the Current Selection

The collection controllers provide methods that allow you to modify the current selection by adding and removing objects, or replacing the selection entirely.

All the methods that change a controller's selection return a boolean value that indicates if the selection was successfully changed. This is because an attempt to change the selection may cause a `commitEditing` message to be sent which may fail or be denied, perhaps due to a value failing validation. If the selection is changed successfully, these methods return `YES`, otherwise they return `NO`.

### Changing the Selection by Object

NSArrayController provides the following methods for replacing the controller's selection completely or adding and removing objects from the current selection:

```
- (BOOL)addSelectedObjects:(NSArray *)objects;
- (BOOL)removeSelectedObjects:(NSArray *)objects;
- (BOOL)setSelectedObjects:(NSArray *)objects;
```

All three methods require that you pass an array containing the objects as the parameter.

### Getting and Setting Selection by Index

The collection controller classes provide additional methods to get the current selection as a set of indexes to the objects in the collection.

NSArrayController provides the following methods for getting and setting the selection by index:

```
- (unsigned int)selectionIndex;
- (BOOL)setSelectionIndex:(unsigned int)index;

- (NSIndexSet *)selectionIndexes;
- (BOOL)setSelectionIndexes:(NSIndexSet *)indexes;
- (BOOL)addSelectionIndexes:(NSIndexSet *)indexes;
- (BOOL)removeSelectionIndexes:(NSIndexSet *)indexes;
```

The `selectionIndexes` method returns an NSIndexSet that specifies the indexes of all the selected objects. The convenience method `selectionIndex` returns the index of the first selected object as an unsigned integer.

You can explicitly set the selection of the controller using the `setSelectionIndexes:` method, passing an NSIndexSet that specifies the indexes of the items that should become the selection. The `setSelectionIndex:` method is a convenience method that lets you set the selection to a single index. If the selection is changed successfully, these methods return `YES`.

The `addSelectionIndexes:` method attempts to add objects specified in the NSIndexSet parameter to the current selection. Similarly, the `removeSelectionIndexes:` attempts to remove the objects specified by index in the NSIndexSet parameter from the selection. If the selection is changed successfully, the methods return `YES`.

NSTreeController treats data as an array of dictionaries of arrays, so a simple index of the selection isn't sufficient. Instead NSTreeController uses an NSIndexPath to specify the location of an object in the tree. An NSIndexPath specifies the "path" to the selection as a series of indexes into the arrays, for example "1.4.2.3".

NSTreeController provides the following methods for getting and setting the selection by index path:

```
-(NSIndexPath *)selectionIndexPath
-(NSArray *)selectionIndexPaths

-(BOOL)setSelectionIndexPath:(NSIndexPath *)indexPath
-(BOOL)setSelectionIndexPaths:(NSArray *)indexPaths

-(BOOL)addSelectionIndexPaths:(NSArray *)indexPaths
-(BOOL)removeSelectionIndexPaths:(NSArray *)indexPaths
```

The `selectionIndexPaths:` method returns an array of NSIndexPath objects that represent each of the currently selected objects. The convenience method `selectionIndexPath` returns the first selected object as an NSIndexPath.

You explicitly set the current selection using the `setSelectionIndexPaths:` method, passing an NSArray of NSIndexPath objects as the parameter. The convenience method `setSelectionIndexPath:` sets the selection to the single object specified by the NSIndexPath parameter.

The methods `addSelectionIndexPaths:` and `removeSelectionIndexPaths:` add or remove the objects at the index paths specified in the array from the selection. As with the other selection modifying methods they return a boolean value of `YES` if the selection was successfully changed.

## Setting the Selection Behaviors

The collection controllers provide several methods that allow you to fine-tune how selection is maintained by a controller, and how values are returned by the `selection` method.

## Avoids Empty Selection

Often it is desirable to attempt to always have at least one item in a collection selected after an action such as removing an item. This is the default behavior of the collection controllers.

You can modify this behavior using the method `setAvoidsEmptySelection:` passing `NO` as the parameter. This allows the controller to have an empty selection, even if there are objects in the content. You can query the current behavior using the method `avoidsEmptySelection`.

Interface Builder allows you to change this behavior for a collection controller by unchecking the "Avoids empty selection" checkbox in the controller's inspector panel.

## Selecting Objects Upon Insertion

By default a collection controller automatically selects objects as they are inserted. While this is often the correct behavior, if you are inserting many objects into the collection it is inefficient and can degrade performance.

You can turn off this behavior using the `setSelectsInsertedObjects:` method. Passing `YES` as the parameter causes all newly inserted objects to be selected. If `NO` is passed as the parameter, the current selection is unchanged as objects are inserted. The method `selectsInsertedObjects` returns a boolean indicating if newly inserted objects will be selected.

You can also change this setting in Interface Builder by unchecking the "Selects inserted objects" checkbox in the controller's inspector panel.

## Always Uses Multiple Values Marker

The NSMultipleValuesMarker indicates that more than one object is selected in the controller and the values for the requested key aren't the same. If the values are the same the value is returned rather than the selection marker. While this allows editing of that common value across all the selected objects, it may not be the desired result.

You can force all multiple selections to always return the NSMultipleValuesMarker using the method `setAlwaysUsesMultipleValuesMarker:`, passing `YES` as the parameter. You query the state of this setting using the method `alwaysUsesMultipleValuesMarker`.

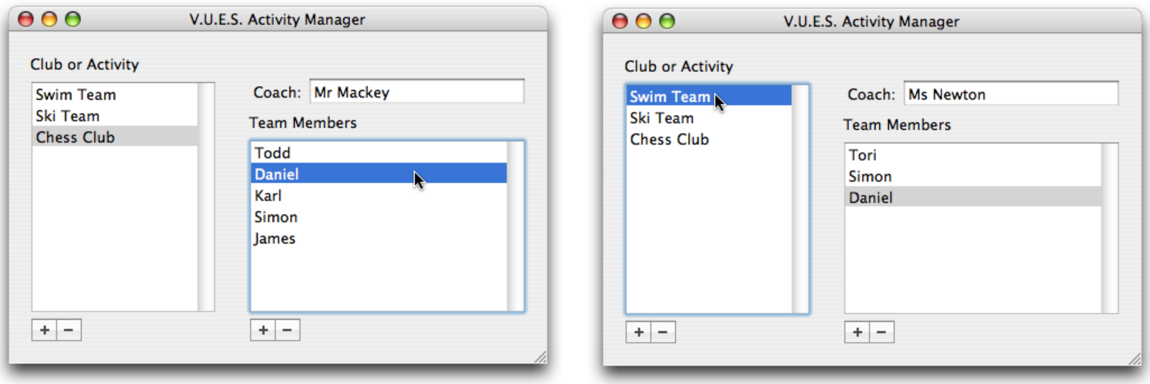
This setting can also be changed in Interface Builder by checking the "Always use multiple values marker" checkbox.

With large selections, enabling this option can improve performance.

## Preserves Selection

When the content of a collection controller changes, the default behavior is to attempt to find matching objects for the current selection in the new content. Figure 1 shows a master-detail interface in which the selected object in the detail NSTableView is automatically selected when the user selects a different activity in the master NSTableView.

**Figure 1** Preserves selection example



Daniel is selected in the "Chess Club" activity list.

Changing the selection to "Swim Team" in the activity list, the Daniel member is selected.

This behavior can be disabled by calling the `setPreservesSelection:` method, passing `NO` as the parameter. The current state is queried using the `preservesSelection` method which returns a boolean. You can also change this setting in Interface Builder by unchecking the "Preserves selection" option in the controller's inspector.

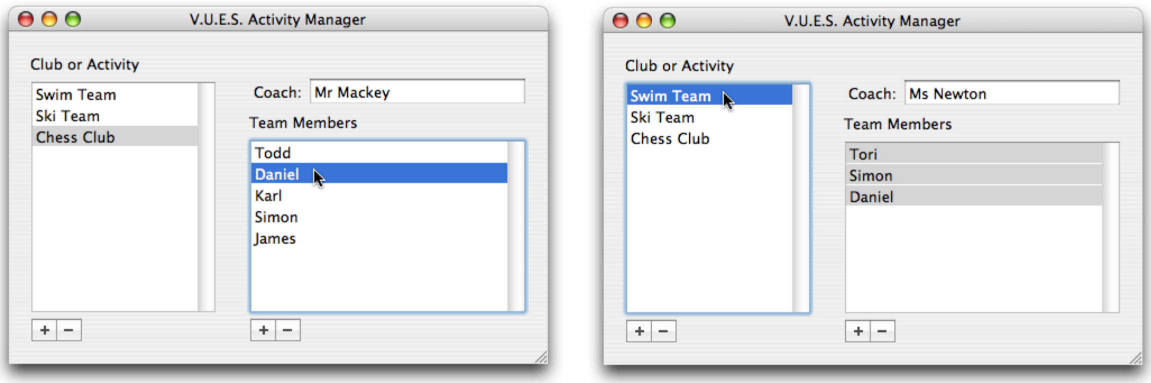
While the default behavior is often appropriate, there can be performance implications. When the content changes, the current selection must first be cached, and then the new content collection must be searched for matching objects. This can become costly when dealing with large collections and multiple selections.

## Selects All When Setting Content

There is a per-binding option that allows you further control over the selection when using an NSArrayController.

The `NSSelectsAllWhenSettingContentBindingOption` causes the array controller to automatically select all the items in the array when the application changes the `contentArray`, `contentSet`, `contentObject` or `contentArrayForMultipleSelection` value of the controller.

**Figure 2** Selects all when setting content example



Initially Daniel is selected in the "Chess Club" member list.

Changing the selection to "Swim Team" causes all team members to be selected.

Figure 2 shows an application with a master-detail interface. The detail NSTableView displays a single member selection "Daniel". When the user selects the "Swim Team" item in the master NSTableView all the members are selected automatically.

This option is set either in the Bindings inspector in Interface Builder, or by setting an NSNumber object with a boolean value of `YES` as the `NSSelectsAllWhenSettingContentBindingOption` value in the options dictionary passed to `bind:toObject:withKeyPath:options:`.

The "Selects All When Setting Content" option is also useful when creating inspectors for master-detail interfaces that allow multiple selections to occur in the master interface. The selection that is to be inspected is predetermined by the master array controller. The inspector array controller is bound to the master array controller's `selectedObjects` binding, specifying the "Selects All When Setting Content" option. This ensures that all items in the master controller's current selection are always selected in the inspector. In an NSDocument-based application, the detail array controller's `contentArray` binding is typically bound to the NSApplication instance using the `mainWindow.windowController.document.<yourmastercontroller>` key path, substituting the master array controller's key for the final key in the key path.

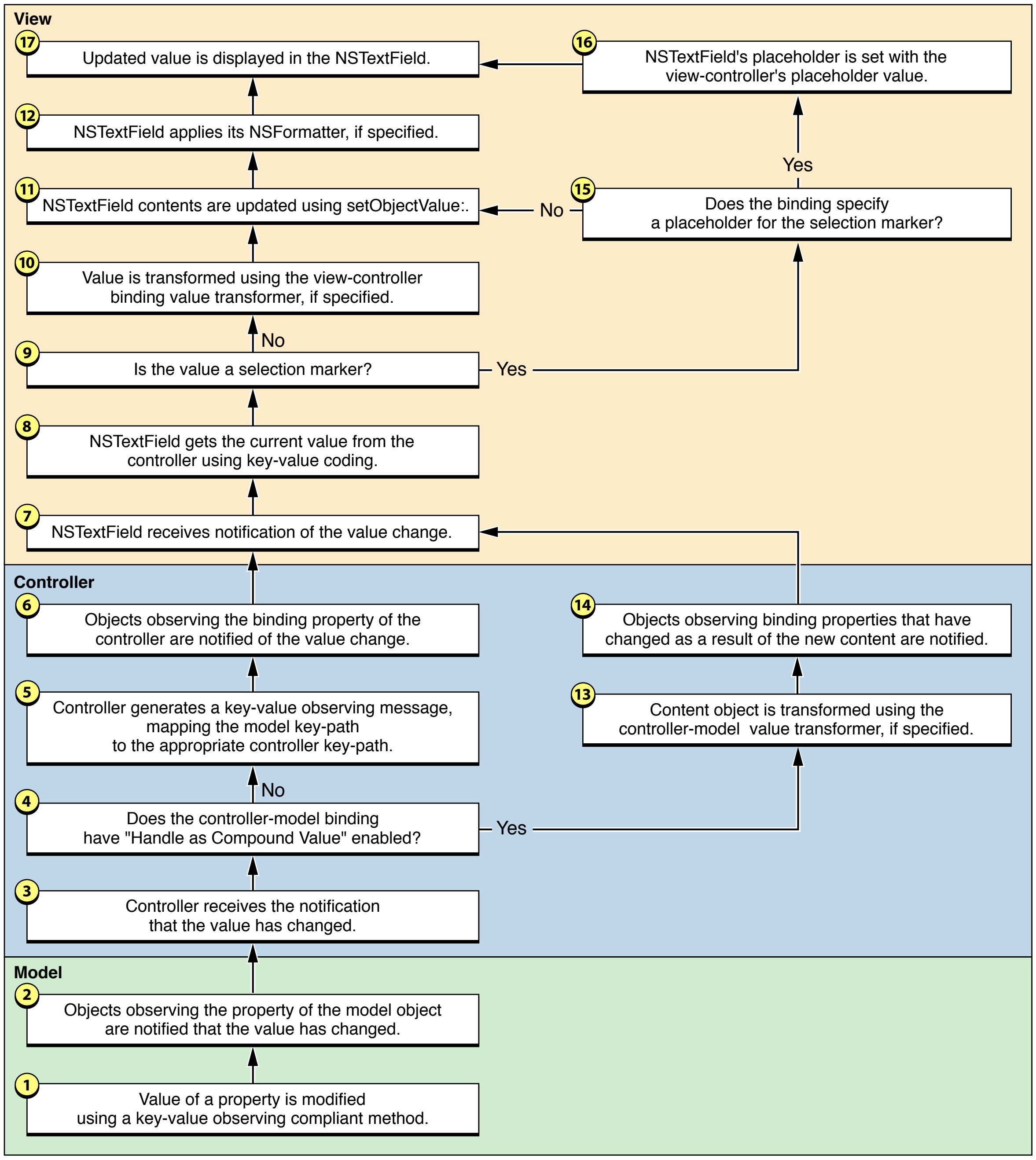
# Bindings Message Flow

A binding has a number of possible options that can be configured, and many of these will influence the flow of messages between the controller, view and model objects. Value transformers, NSFormatters, key-value validation, placeholders, selection value markers, and other binding options, all affect changes made in the user interface, and in the model values. This article describes the flow of messages between the objects for common interactions with the view, controller, and model objects.

## Changing the Value of a Model Property

The diagram in Figure 1 shows the flow of messages between the model, controller and view objects in response to changing the value of a property in the model object.

**Figure 1** Message flow when modifying the value of a model object property


View
17 Updated value is displayed in the NSTextField.
16 NSTextField's placeholder is set with the view-controller's placeholder value.
12 NSTextField applies its NSFormatter, if specified.
Yes
11 NSTextField contents are updated using setObjectValue:.
No
15 Does the binding specify a placeholder for the selection marker?
10 Value is transformed using the view-controller binding value transformer, if specified.
No
9 Is the value a selection marker?
Yes
8 NSTextField gets the current value from the controller using key-value coding.
7 NSTextField receives notification of the value change.
Controller
6 Objects observing the binding property of the controller are notified of the value change.
14 Objects observing binding properties that have changed as a result of the new content are notified.
5 Controller generates a key-value observing message, mapping the model key-path to the appropriate controller key-path.
13 Content object is transformed using the controller-model value transformer, if specified.
No
4 Does the controller-model binding have "Handle as Compound Value" enabled?
Yes
3 Controller receives the notification that the value has changed.
Model
2 Objects observing the property of the model object are notified that the value has changed.
1 Value of a property is modified using a key-value observing compliant method.

Specifically, this example is of an NSTextField with its `value` binding bound to an NSObjectController's selection using the `selection.firstName` key path. The NSObjectController's `content` binding is bound to a model object that encapsulates a person's name and address.

1. A property of the model object is changed using a key-value-observing compliant means.

2. The objects that are registered with the model object to receive key-value observing change notifications are sent an `observeValueForKeyPath:ofObject:change:context:` message by the model object's inherited key-value observing implementation.

3. The controller receives the `observeValueForKeyPath:ofObject:change:context:` message. The controller is registered with the model object as a result establishing the binding to the model object.

4. The controller tests to see if the controller-model binding has the "Handles as Compound Value" option set. If "Handles as Compound Value" is enabled, proceed to Step 13.

5. The controller generates a key-value observing message for the value that has changed, mapping the model key-path to the appropriate controller key-path. For example, if the key path of the property in the model object is "`firstName`" then the key-value observing change is mapped to "`selection.firstName`".

6. The objects that are registered with the controller object to receive key-value observing change notifications are sent an `observeValueForKeyPath:ofObject:change:context:` message by the controller object's inherited key-value observing implementation.

7. The view receives the `observeValueForKeyPath:ofObject:change:context:` message. The view is registered with the controller object as a result establishing the binding to the controller object.

8. The view objects gets the current value for the changed property from the controller using key-value coding.

9. The view tests the value retrieved in Step 8. If the value is one of the selection markers, or `nil`, proceed to Step 15.

10. If the view-controller binding specifies a value transformer, the value retrieved in Step 8 is passed to the value transformer's `transformedValue:` method. The value returned by the `transformedValue:` method is passed to the next step.

11. The view sets its contents to the updated, possibly transformed, value using the view's `setObjectValue:` method.

12. If the view has an NSFormatter attached to it, the value is formatted by the NSFormatter instance. Proceed to Step 17.

13. If the controller-model binding specifies a value transformer the entire model is transformed using the `transformedValue:` method.

14. The controller generates a key-value observing message, mapping the key path to the selection. This generates the same type of key-value observing change notification that would result in replacing the content object. Proceed to Step 7.

15. If the binding specifies a placeholder for the view-controller binding for the value marker it is used in Step 16. If no custom placeholder value is specified as part of the binding, and a custom default placeholder has been set for the view class and this particular binding, it is used in Step 16. Otherwise, flow returns to Step 11.

    An application can assign a custom default placeholder for a class and binding combination using the NSPlaceholders class method `setDefaultPlaceholder:forMarker:withBinding:`.

16. The placeholder value is set for the view using the `setPlaceholderString:` or `setPlaceholderAttributedString:` method.

17. The updated value is displayed in the user interface.

## User Updates a Value in the User Interface

The diagrams in Figure 2 and Figure 3 (page 52) show the flow of messages between the model, controller and view objects in response to changing the value of a property in the model object.

**Figure 2** Message flow in response to the user changing a value in an NSTextField

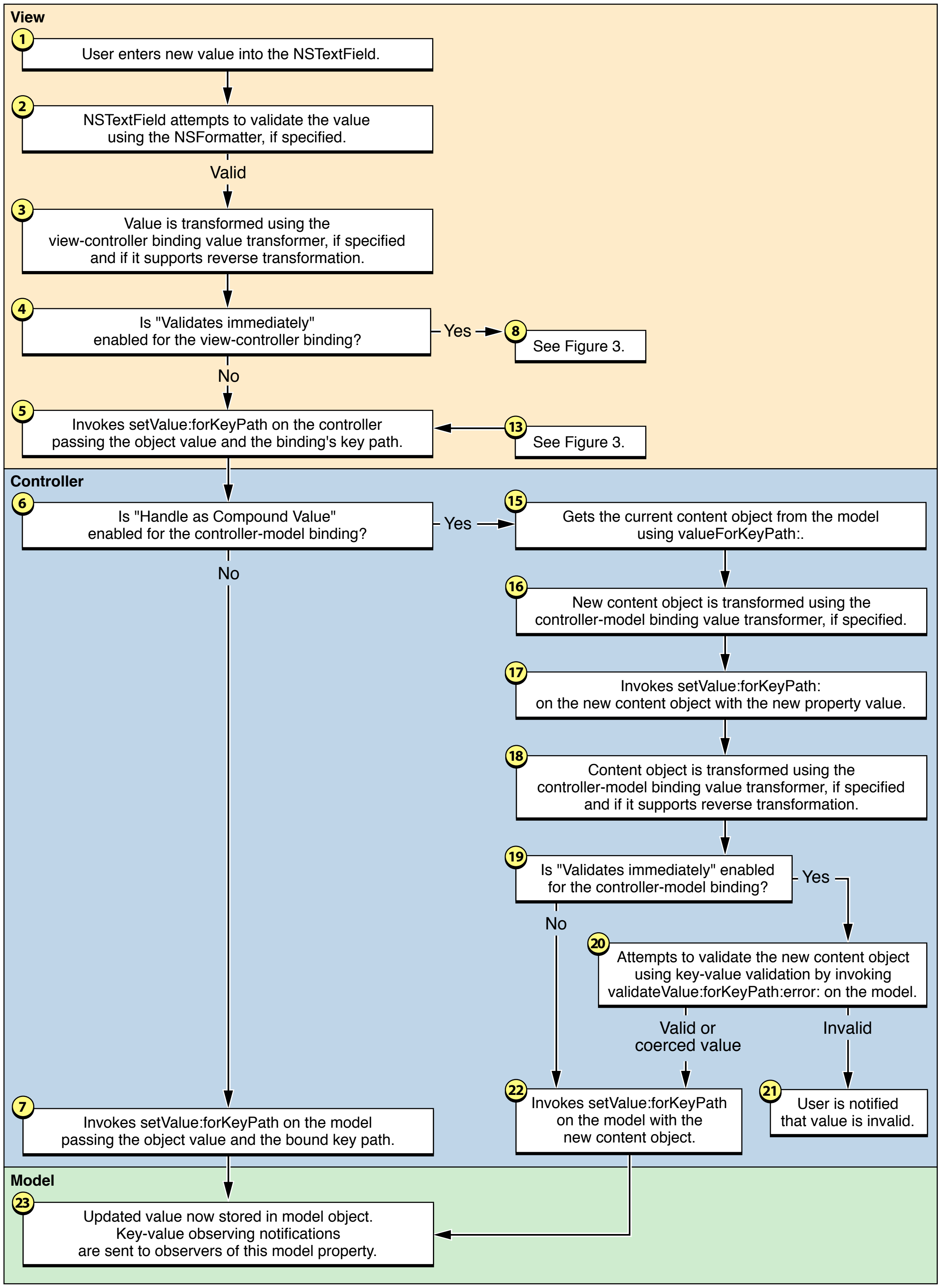
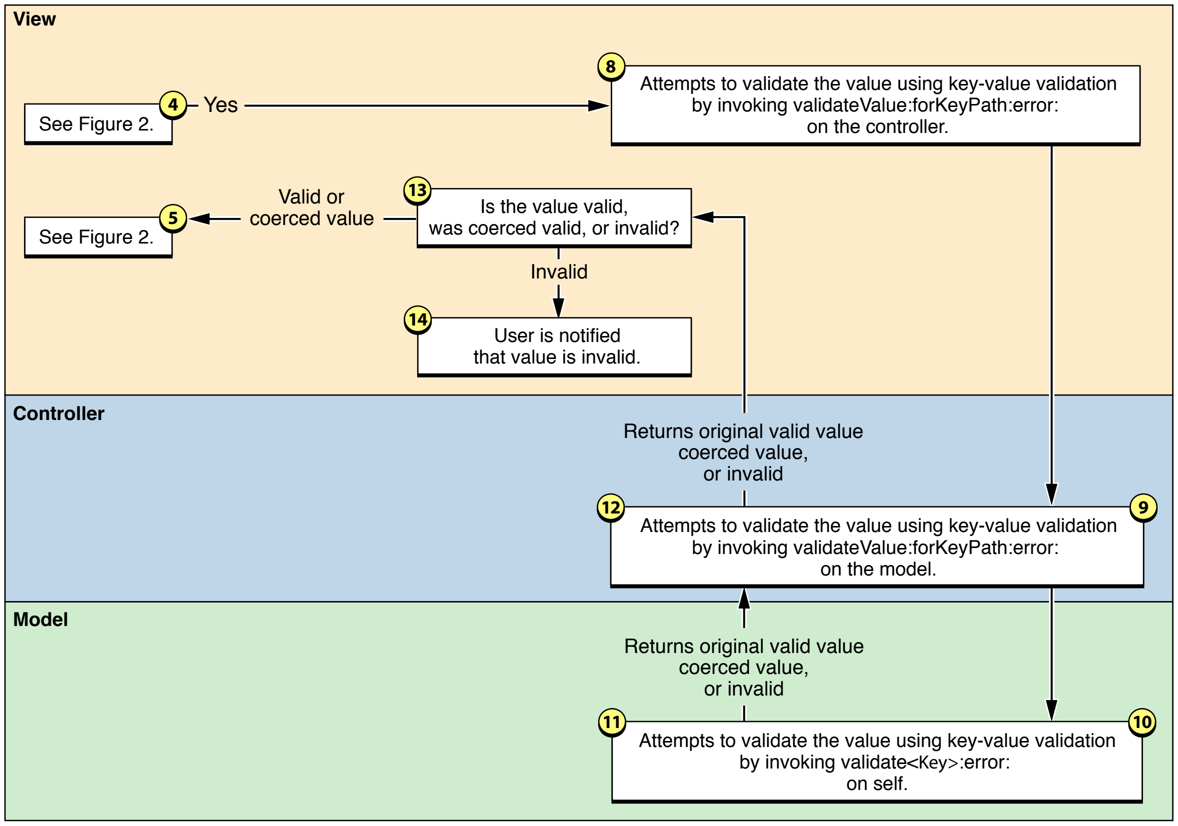

View
1 User enters new value into the NSTextField.
2 NSTextField attempts to validate the value using the NSFormatter, if specified.
Valid
3 Value is transformed using the view-controller binding value transformer, if specified and if it supports reverse transformation.
4 Is "Validates immediately" enabled for the view-controller binding?
Yes
8 See Figure 3.
No
5 Invokes setValue:forKeyPath on the controller passing the object value and the binding's key path.
13 See Figure 3.
Controller
6 Is "Handle as Compound Value" enabled for the controller-model binding?
Yes
15 Gets the current content object from the model using valueForKeyPath:.
No
16 New content object is transformed using the controller-model binding value transformer, if specified.
17 Invokes setValue:forKeyPath: on the new content object with the new property value.
18 Content object is transformed using the controller-model binding value transformer, if specified and if it supports reverse transformation.
19 Is "Validates immediately" enabled for the controller-model binding?
Yes
No
20 Attempts to validate the new content object using key-value validation by invoking validateValue:forKeyPath:error: on the model.
Valid or coerced value
Invalid
7 Invokes setValue:forKeyPath on the model passing the object value and the bound key path.
22 Invokes setValue:forKeyPath on the model with the new content object.
21 User is notified that value is invalid.
Model
23 Updated value now stored in model object. Key-value observing notifications are sent to observers of this model property.

**Figure 3** Message flow in response to the user changing a value in an NSTextField, view-controller validation


View
See Figure 2.
4
Yes
8
Attempts to validate the value using key-value validation by invoking validateValue:forKeyPath:error: on the controller.
See Figure 2.
5
Valid or coerced value
13
Is the value valid, was coerced valid, or invalid?
Invalid
14
User is notified that value is invalid.
Controller
Returns original valid value coerced value, or invalid
12
9
Attempts to validate the value using key-value validation by invoking validateValue:forKeyPath:error: on the model.
Model
Returns original valid value coerced value, or invalid
11
10
Attempts to validate the value using key-value validation by invoking validate<Key>:error: on self.

As in “"Changing the Value of a Model Property" (page 47)” this specific example is of an NSTextField with its `value` binding bound to an NSObjectController’s selection using the `selection.firstName` key path. The NSObjectController’s `content` binding is bound to a model object that encapsulates a person’s name and address.

1. The user enters a new value into the NSTextField.

2. If an NSFormatter is attached to the NSTextField the formatter attempts to validate the value. If the formatter fails to validate the value it provides failure feedback and returns control to the user.

   The view-controller binding option “Continuously Updates Value” determines when the view notifies the controller of changes in the text field. If it this option is specified, the controller will be updated for each keystroke. If it is disabled then the controller is only updated when the user hits return, tab, or the text field loses first responder status.

3. If the view-controller binding specifies a value transformer and the value transformer supports inverse transformations, the new value is transformed using the `inverseTransformedValue:` method. The value returned by the `inverseTransformedValue:` method is passed to the next step.

4. If the view-controller binding option “Validates Immediately” is enabled, then proceed to Step 8.

5. The view object invokes `setValue:forKeyPath:` on the controller object, passing the new, possibly transformed, object value and the binding’s key path as arguments.

6. If the controller-model binding option “Handle as Compound Value” is enabled, then proceed to Step 15.

7. The controller invokes `setValue:forKeyPath:` on the model object, passing the new content value from Step 5 and the key path for the property in the model object as arguments. Proceed to Step 23.

8. The view attempts to validate the value from Step 4 using key-value validation. It does this by invoking `validateValue:forKeyPath:error:` on the controller, passing the proposed value, the binding’s key path, and an NSError pointer reference as arguments.

9. The controller receives the `validateValue:forKeyPath:error:` message and forwards the validation request to the model object’s `validateValue:forKeyPath:error:` implementation, modifying the key path so that it references the appropriate model object property.

10. The model object receives the `validateValue:forKeyPath:error:` message and attempts to validate the proposed value by invoking `validate<Key>:error:` on self. The specific method signature will be dependent on the name of the property.

    The model object’s implementation of `validate<Key>:error:` returns a Boolean value indicating if the value was valid. If the result is `YES`, then the proposed value was valid, or was replaced with a validated value. If the result is `NO`, then the proposed value is invalid and the error reference should contain a description of why the validation failed.

11. The validation result, value, and a possible error are returned by the model’s implementation of `validate<Key>:error:` to the controller’s `validateValue:forKeyPath:error:` method.

12. The validation result, value, and a possible error are returned by the controller’s `validateValue:forKeyPath:error:` method to the view.

13. If the validation result is `YES`, then the valid or coerced value are returned to Step 5, otherwise proceed to Step 14.

14. The user is notified that the value is invalid.

    How the error is presented to the user is dependent on the view-controller binding’s “Always Presents Application Modal Alert Panels” option. If this option is enabled, then the error is displayed to the user as a modal alert panel. If it is not enabled, then the error is presented to the user as a sheet.

15. The controller gets the current content object using `valueForKeyPath:` on the model object.

    This is the message flow path for “Handle as Compound Value”. The original content object specified by the `contentObject`, `contentArray` or `contentSet` binding is retrieved from the model, transformed, the new value from the user inserted, the content object is inverse transformed, and set as the new content object.

16. If the controller-model binding specifies a value transformer the content object retrieved in Step 15 is transformed using the `transformedValue:` method. The value returned by the `transformedValue:` method is passed to the next step.

17. The controller invokes `setValue:forKeyPath:` on the retrieved, possibly transformed, content object. The value from Step 5 and the key path for the property in the model object as arguments.

18. If the controller-model binding specifies a value transformer and the value transformer supports inverse transformations, the content object from Step 17 is transformed using the `inverseTransformedValue:` method. The value returned by the `inverseTransformedValue:` method is passed to the next step.

19. If the controller-model binding option "Validates Immediately" is enabled, proceed to Step 20, otherwise proceed to Step 22.

20. The controller attempts to validate the new, possibly transformed, content object by invoking `validateValue:forKeyPath:error:` on the model object.

    If `validateValue:forKeyPath:error:` returns `YES`, then the valid or coerced valid value is passed to Step 22. If the result is invalid, proceed to Step 21.

21. The user is notified that the value is invalid.

    How the error is presented to the user is dependent on the controller-controller binding's "Always Presents Application Modal Alert Panels" option. If this option is enabled, then the error is displayed to the user as a modal alert panel. If it is not enabled, then the error is presented to the user as a sheet.

22. The controller invokes `setValue:forKeyPath:` on the model object using the new, possibly validated content object and the content key path as arguments.

23. The updated value is now stored in the model object.

    Changing the model object results in key-value observing change notifications being sent to the observers of the model property.

# User Defaults and Bindings

Many applications provide a preferences window that allows the user to customize an application's settings. NSUserDefaultsController provides a layer on top of NSUserDefaults and allows you to bind attributes of user interface items to the corresponding key in an application's user defaults.

## What Is NSUserDefaultsController?

NSUserDefaultsController is a concrete subclass of NSController that implements a bindings-compatible interface to NSUserDefaults. Properties of an instance of NSUserDefaultsController are bound to user interface items to access and modify values stored using NSUserDefaults.

NSUserDefaultsController is typically used when implementing your application's preference window interface, or when you can bind a user interface item directly to a default value. NSUserDefaults remains the primary programmatic interface to your application's default values for the rest of your application.

By default NSUserDefaultsController immediately applies any changes made to its properties. It can be configured so that changes are not applied until it receives an `applyChanges:` message, allowing the preferences dialog to support an Apply button. NSUserDefaultsController also supports reverting to the last applied set of values, using the `revert:` method.

NSUserDefaultsController also allows you to provide a dictionary of factory defaults that can be used to reset the user configurable values for your application, usually done in response to a user clicking a Revert to Factory Defaults button.

## The Shared User Defaults Controller

NSUserDefaultsController provides a shared instance of itself via the class method `sharedUserDefaultsController`. This shared instance uses the NSUserDefaults instance returned by the method `standardUserDefaults` as its model, has no initial values, and immediately applies changes made through its bindings.

Care must be taken that changes to the settings of the shared user defaults controller are made before any nib files containing bindings to the shared controller are loaded. To ensure that these changes are made before any nib files are loaded, they are often implemented in the `initialize` class method of the application delegate, or in your preferences window controller.

## Binding to the Shared User Defaults Controller

The shared NSUserDefaultsController is always available as a bindable controller in the Bindings Info window in Interface Builder. When establishing a binding to a user default, set the Controller Key to `values`, and the Model Key Path to the key of the default.

Creating bindings programmatically requires that you retrieve the shared user defaults controller using the NSUserDefaultsController class method `sharedUserDefaultsController`. You then provide that object as the *`observableController`* to the `bind:toObject:withKeyPath:options:` method.

The example in Listing 1 establishes a binding between an NSTextField (`theTextField`) and the `userName` default using the shared user defaults controller.

**Listing 1** Binding the userName defaults key to an NSTextField programmatically

```
 [theTextField bind:@"value"
             toObject:[NSUserDefaultsController sharedUserDefaultsController]
          withKeyPath:@"values.userName"
              options:[NSDictionary dictionaryWithObject:[NSNumber
numberWithBool:YES]

forKey:@"NSContinuouslyUpdatesValue"]];
```

## initialValues Versus NSUserDefaults registerDefaults:

The initial values dictionary allows you to provide a means to reset the user configurable default values to the factory defaults. Typically these values represent a subset of the defaults that your application registers using the NSUserDefaults method `registerDefaults:`.

Calling the NSUserDefaultsController method `setInitialValues:` should not be considered a replacement for registering your application's preference defaults using NSUserDefault's `registerDefaults:` method.

The example in Listing 2 loads the default values from a file in the application wrapper, registers those values with NSUserDefaults, and then registers a subset of the values as the initial values of the shared user defaults controller. The `setupDefaults` method would be called from your application delegate's `initialize` class method.

**Listing 2** Changing the initial values of the sharedUserDefaultsController instance

```
+ (void)setupDefaults
{
    NSString *userDefaultsValuesPath;
    NSDictionary *userDefaultsValuesDict;
    NSDictionary *initialValuesDict;
    NSArray *resettableUserDefaultsKeys;

    // load the default values for the user defaults
   userDefaultsValuesPath=[[NSBundle mainBundle] pathForResource:@"UserDefaults"
                               ofType:@"plist"];
    userDefaultsValuesDict=[NSDictionary
dictionaryWithContentsOfFile:userDefaultsValuesPath];
```

```
    // set them in the standard user defaults
    [[NSUserDefaults standardUserDefaults]
registerDefaults:userDefaultsValuesDict];

    // if your application supports resetting a subset of the defaults to
    // factory values, you should set those values
    // in the shared user defaults controller
    resettableUserDefaultsKeys=[NSArray
arrayWithObjects:@"Value1",@"Value2",@"Value3",nil];
    initialValuesDict=[userDefaultsValuesDict
dictionaryWithValuesForKeys:resettableUserDefaultsKeys];

    // Set the initial values in the shared user defaults controller
    [[NSUserDefaultsController sharedUserDefaultsController]
setInitialValues:initialValuesDict];
}
```

## Search Order for Defaults Values

When a method that is key-value coding compliant attempts to get a value for a key from an NSUserDefaultsController the following search pattern is used:

1. The value of a corresponding key in `values`
2. The value of a corresponding key in the NSUserDefaults instance returned by the NSUserDefaultsController method `defaults`.
3. The value of a corresponding key in the initial values dictionary

If no corresponding value is found, `nil` is returned.

The search path is somewhat different when you retrieve the result directly from the NSUserDefaults instance associated with the NSUserDefaultsController. In that case, any unapplied values in the NSUserDefaultsController, as well as the values in the initial values dictionary are ignored.

## Programmatically Accessing NSUserDefaultsController Values

Although NSUserDefaults should remain your primary programmatic interface to the user defaults, some circumstances require that you get and set the default values contained in an NSUserDefaultsController instance directly. For example, when implementing portions of your preferences window that don't directly interact with an existing binding, such as setting a font or choosing a directory path.

The NSUserDefaultsController method `values` returns a KVC-compliant object that is used to access these default values. To get the value of a default, use the `valueForKey:` method.

```
[[theDefaultsController values] valueForKey:@"userName"];
```

Similarly, to set a value for a default, use `setValue:forKey:`.

```
[[theDefaultsController values] setValue:newUserName
                                  forKey:@"userName"];
```

The NSUserDefaultsController automatically provides notification of the value change to any established bindings for that key path.

# Creating a Master-Detail Interface

This article explains how to use bindings to implement a basic master-detail interface. In the master interface portion, a table view is used to display your collection of objects. In the detail interface portion, other views are used to display the selected object. This article contains examples of binding table columns to array controllers.

A master-detail interface can be extended in a variety of ways that are not fully covered in this article. In particular, there are many aspects of a table view that work seamlessly with Cocoa bindings. Refer to these other articles for specific tasks:

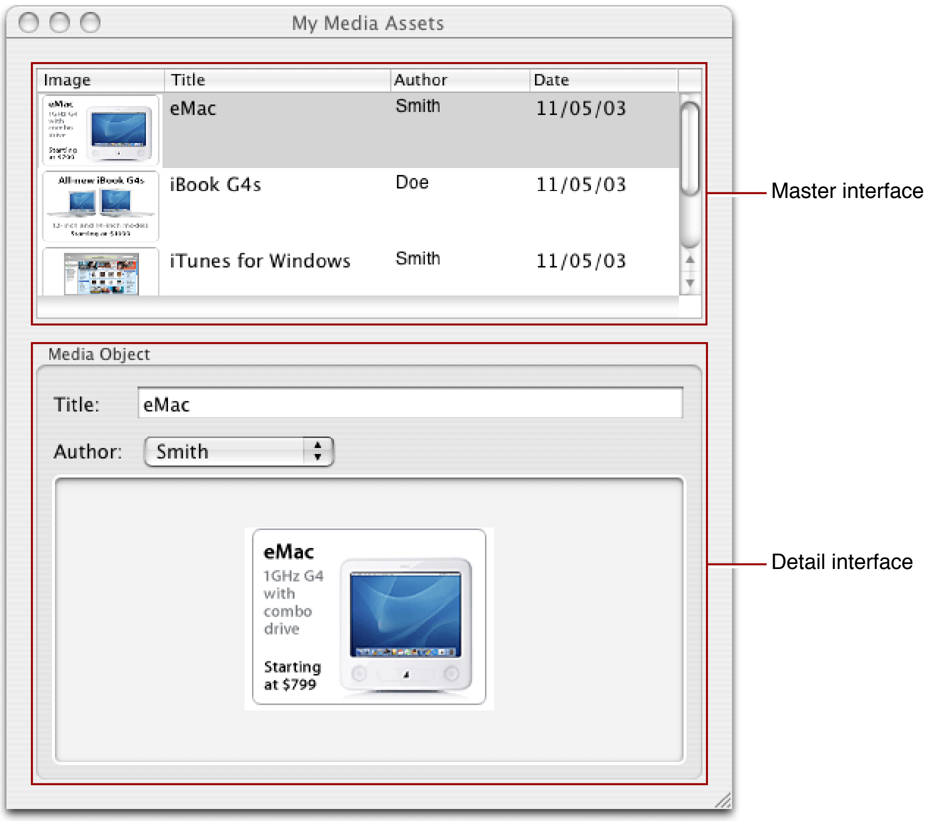
- "Displaying Images Using Bindings" (page 67) describes the various options used when displaying images in columns and contains an example of a custom value transformer.
- "Implementing To-One Relationships Using Pop-Up Menus" (page 73) explains how to implement editable to-one relationship as pop-up menus.
- "Filtering Using a Custom Array Controller" (page 77) explains how to add a search field to the master interface in order to filter the objects it displays.

## What's a Master-Detail Interface?

In a **master-detail interface**, the user can select objects from a list of objects and inspect the selected objects. The **master interface** displays the collection of objects, and the **detail interface** implements an inspector of the selected object. Whenever the user changes the selection in the master interface, the detail interface is updated to show the new selection. If no object is selected, the detail interface displays nothing or disables itself (if it is editable). If multiple objects are selected (assuming multiple selection is allowed), the detail interface is disabled or it applies to all selected objects. Typically, such an interface also allows users to add and remove objects from the collection.

What components you use to implement a master-detail interface depend on your application. Often, the master interface is implemented by a table view and the detail interface is a collection of views located above or below the master interface. Figure 1 shows a master-detail interface in a media asset application. In this example, the table view displays a collection of media objects and the views below display the properties of the selected media object.

**Figure 1** A master-detail interface



| Image | Title | Author | Date |
|---|---|---|---|
| | eMac | Smith | 11/05/03 |
| | iBook G4s | Doe | 11/05/03 |
| | iTunes for Windows | Smith | 11/05/03 |

Title: eMac

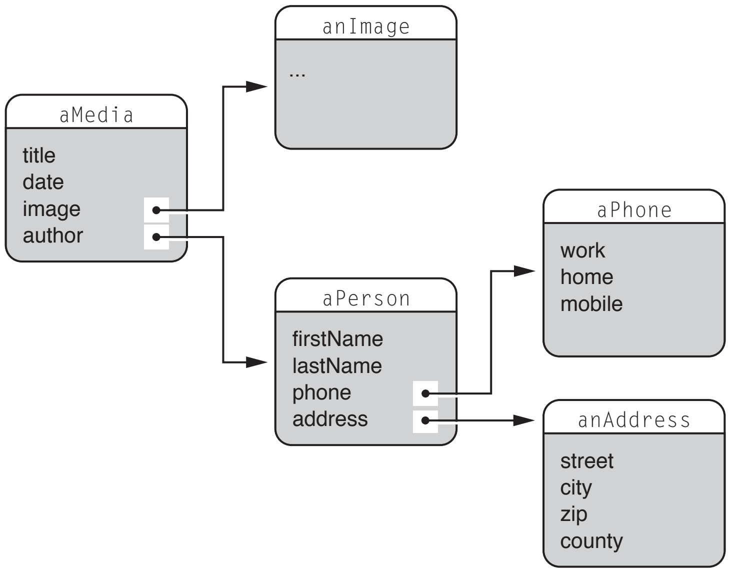
Author: Smith

## Creating Models

This article assumes that you have already designed your model classes, the objects that encapsulate your application data and provide methods to operate on that data. A master-detail interface is used to display a collection of model objects. Therefore, your root model object is expected to be a collection of these objects.

For example, Figure 2 illustrates the object diagram of the models in a media asset application. In this example, assume the application uses a document-based architecture in which MyDocument (the File's Owner of the `MyDocument.nib` file) has a `mediaAssets` instance variable that is initialized to an array of Media objects.

**Figure 2** An object model



## Creating Views

The master interface shown in Figure 1 (page 60) uses a table view in which each column in the table corresponds to a property of the displayed objects. Note that the table view displays attributes such as the title and date, as well as relationships such as the author. The author property is a to-one relationship (see Figure 2 (page 61)) in which the destination object is a Person object. You can display to-one relationships in a variety of ways. For example, the master interface displays only the author's last name. The detail interface lets you select the author from a pop-up menu.

Using Interface Builder, lay out your master-detail interface similar to Figure 1 (page 60). In this example, an image cell was dragged to the Image column to display a scaled version of the media image and a date formatter was dragged to the Date column. The Author column displays the last name of the authors but could be configured to display a menu as well by dragging a pop-up menu cell, NSPopUpButtonCell, to the Author column.

## Creating Controllers

Next you need to connect your views to your models via a controller. NSArrayController is specifically designed to manage selectable collections of objects. For this reason, it is ideal for implementing a master-detail interface. After you create an array controller, you specify its content to be your collection of model objects. If the array controller is editable, it can also create instances of models and add them to the collection. For this to work, you also need to tell the array controller the class name the entity it is managing.

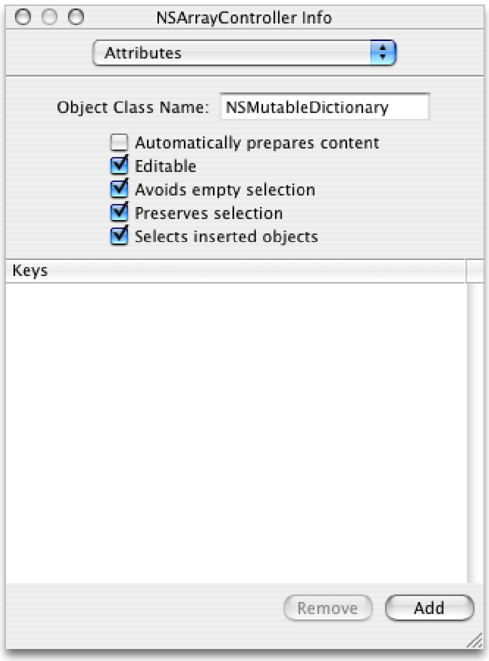
Follow these steps to create and configure an array controller for a master-detail interface:

1. Create an array controller by dragging an NSArrayController from the Interface Builder Controllers palette to your nib file, and optionally change the name of the controller. For example, change the name to `MediaAssetsController`.

2. Enter the class name of your model in the Object Class Name field on the Attributes pane of the Info window as shown in Figure 3. In the media asset application example, you would enter Media as the class name.

3. Deselect Editable on the Attributes pane if you don't want the user to edit the models, or add and remove them from the collection.

**Figure 3** An array controller Attributes pane



Object Class Name: NSMutableDictionary

- [ ] Automatically prepares content
- [x] Editable
- [x] Avoids empty selection
- [x] Preserves selection
- [x] Selects inserted objects

## Binding Controllers to Models

Each view and controller class exposes a set of bindings. You can inspect the bindings of an object using Interface Builder by selecting it and displaying the Bindings pane in the Info window. Select your array controller in the nib file and display its bindings as shown in Figure 4 (page 63).

You specify the content of an array controller by configuring its `contentArray` binding. For example, assuming the File's Owner maintains a collection of Media objects called `mediaAssets`, you would configure the `contentArray` binding as follows:

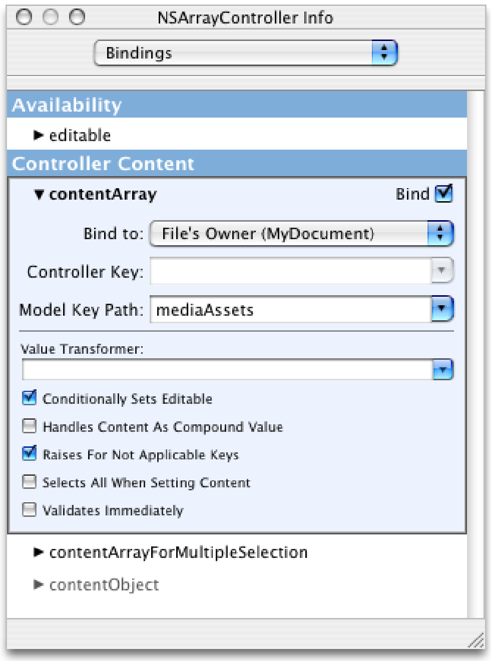
1. Set the `Bind to` aspect to the object that maintains the collection of model objects (for example, the File's Owner).

2. Leave the `Controller Key` blank.

3. Set the `Model Key Path` to the name of the array (for example, `mediaAssets`).

Figure 4 shows what the Bindings pane should look like when the `contentArray` binding is revealed.

**Figure 4** An array controller Bindings pane



contentArray — Bind [✓]

Bind to: File's Owner (MyDocument)

Controller Key:

Model Key Path: mediaAssets

Value Transformer:

- [✓] Conditionally Sets Editable
- [ ] Handles Content As Compound Value
- [✓] Raises For Not Applicable Keys
- [ ] Selects All When Setting Content
- [ ] Validates Immediately

Now this controller is configured to manage a collection of Media objects including adding and removing Media objects from the array. Go back to the Attributes pane if you want to change the default behavior of this controller—deselect Editable if you don't want the user to edit the collection.

## Binding Views to Controllers

Next, connect your views to your array controller by configuring the bindings of your view objects—bind each view in the master and detail interfaces to the array controller.

Setting some of these bindings can be complex if, for example, you want to represent an editable to-one relationship. This article covers the most common types of value bindings. See "Displaying Images Using Bindings" (page 67) and "Implementing To-One Relationships Using Pop-Up Menus" (page 73) for examples of more complex bindings.

In general, it helps to remember that most views that display some content have a binding called `value` or a binding that contains the word "value."

Also, because you bind a view to a controller, you need to be familiar with the properties of controllers. These are the primary properties of an array controller that you typically use as the value of the `Controller Key` aspect:

- `arrangedObjects`—specifies the collection of objects being displayed.
- `selection`—specifies the selected object in `arrangedObjects`.
- `selectedObjects`—specifies the selected objects in `arrangedObjects` for a multiple selection.

## Binding the Master Interface

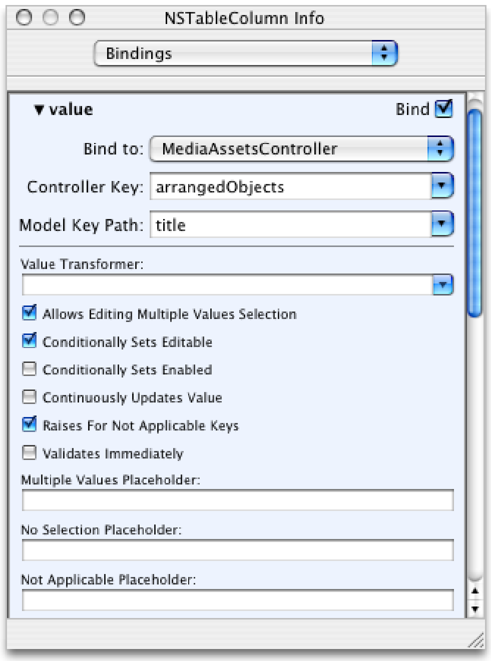
In the master interface, you actually configure the bindings of the table columns, not the bindings of a table view. Even if you use a special type of cell in the table column, for example, an image cell or NSPopUpButtonCell, you configure the table column, not the cell. The bindings for an NSTableColumn may change depending on its cell.

In this example, each row in the table view needs to correspond to a single Media object, and each column needs to display a property of that Media object. To display properties, you first select a table column in the table view, display the Bindings pane, and reveal the `value` binding. You will then bind the table column to the controller's `arrangedObjects` key because you are specifying the contents of the entire column, not a single cell. Configure each `value` binding to display a property of the Media object as follows:

1. Set the `Bind to` aspect to your array controller object. For example, `MediaAssetsController`.
2. Set the `Controller Key` to `arrangedObjects` (the collection of objects being displayed).
3. Set the `Model Key Path` to the property you want to appear in that column. For example, in the media asset application, set the `Model Key Path` to `date` in the Date column.

Figure 5 shows what the Bindings pane should look like when the `value` binding of the Title column is revealed.

**Figure 5** NSTableColumn Bindings pane



Bind to: MediaAssetsController
Controller Key: arrangedObjects
Model Key Path: title
Value Transformer:
✓ Allows Editing Multiple Values Selection
✓ Conditionally Sets Editable
☐ Conditionally Sets Enabled
☐ Continuously Updates Value
✓ Raises For Not Applicable Keys
☐ Validates Immediately
Multiple Values Placeholder:
No Selection Placeholder:
Not Applicable Placeholder:

In the case of the author property, you can set the `Model Key Path` to `author.lastname` if you want to simply display the last name. However, by default column cells are editable, so when you run your application, users will be able to change the author's last name by editing the text in the Author column. If you don't want this behavior, select the column and deselect Editable on the Attributes pane in the Info window.

Note that some Cocoa views already display lists of objects that work well with key-value coding. The `Controller Key` and `Model Key Path` aspects of a binding are concatenated to retrieve the value from the bound object. In the case of a table column, the `Controller Key` identifies an array; therefore the returned value is an array. For example, the Author table column uses the `arrangedObjects.author.lastname` key path to get an array of last names, instances of NSString, from the bound object.

## Binding the Detail Interface

You bind the views in the detail interface similarly to the way you bind the master interface except that you set the `Controller Key` for each binding to `selection`, the currently selected object.

For example, you configure the `value` binding for the Title text field in the detail interface depicted in Figure 1 (page 60) as follows:

1. Set the `Bind to` aspect to your array controller object. For example, `MediaAssetsController`.
2. Set the `Controller Key` to `selection` (the currently selected object).
3. Set the `Model Key Path` to the property you wish displayed in that view. For example, `title`.

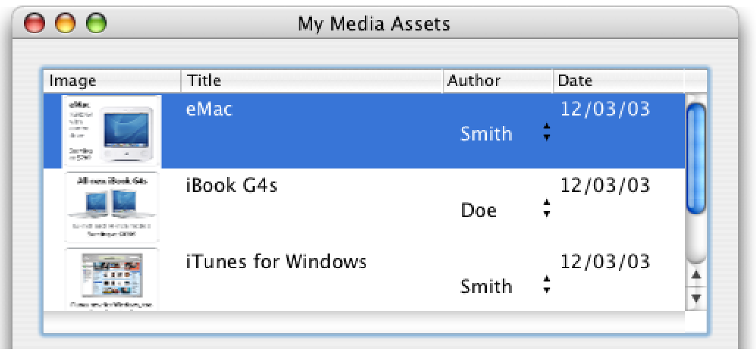
# Displaying Images Using Bindings

You typically display images using either an NSImageCell or an NSImageView. You might display images in table columns in a master interface, or an image view in a detail interface. Both of these components have similar bindings and therefore are both discussed in this article. When using these components, you also need to make similar decisions about how you want to store and access your images. The value bindings of these components support a variety of formats. You can specify the value binding using an NSImage, a file path or a URL.

The example presented in this article is an extension of those in "Creating a Master-Detail Interface" (page 59). Specifically, this article explains how to add an NSImageCell to a table column in the master interface (shown in Figure 1) and an NSImageView to the detail interface. This article also contains an example of a custom value transformer that converts image filenames to absolute paths.

See "Creating a Master-Detail Interface" (page 59) for the steps to create a basic master-detail interface.

**Figure 1** Using an NSImageCell in an NSTableColumn



| Image | Title | Author | Date |
|---|---|---|---|
|  | eMac | Smith | 12/03/03 |
|  | iBook G4s | Doe | 12/03/03 |
|  | iTunes for Windows | Smith | 12/03/03 |

## Creating Models

How you define image properties in your models depends on your application—it depends on how you want to store and access the images. If the images are stored on disk or in your project folder on the same computer, then you can access them via a file path. If the images are stored not on disk but on a remote server, then you can access them via a URL. If the images are stored in some data source, then you might load the images directly into memory. If you load them, you might define your image properties as simply NSImage objects. Fortunately, Cocoa bindings supports all these options.

In this example, the model uses an NSImage to represent the `image` property of a Media object as shown in Figure 2 (page 61) in "Creating a Master-Detail Interface" (page 59).

In the case of a file path or URL, you typically store only the filename in the model not the absolute path or URL (for example, you don't want to update your models every time you move the image folder). However, the bindings expect an absolute path or URL. One solution is to implement a custom value transformer that takes an image filename and returns a file path or URL based on some variables you define. See "Using a Value Transformer to Convert Paths" (page 69) to modify this example to use file paths instead of NSImage objects.

## Creating Views

Next, you create the image views by either dragging an NSImageView to a window or an NSImageCell to the column that will display the images.

## Creating Controllers

This example assumes you already have a working master-detail interface. Follow the steps in "Creating Controllers" (page 61) in “Creating a Master-Detail Interface” if you need to create an array controller.

## Binding Views to Controllers

Next, you bind the image views to the controllers. A subset of the value bindings of an NSImageView and an NSTableColumn (containing an NSImageCell) to choose from are:

- `value`—an NSImage object.
- `valuePath`—an absolute path to the image file.
- `valueURL`—a URL that returns the image file.

**Note:** When using an NSImageCell in a table column, you need to configure the bindings of the table column, not the cell. When you drag an NSImageCell to a column, the Value category on the NSTableColumn Bindings pane changes to reveal additional value bindings (displays the same value bindings as an NSImageView).

For example, you configure the `value` binding for the Image table column in Figure 1 (page 67) as follows:

1. Set `Bind to` to your array controller object. For example, `MediaAssetsController`.
2. Set `Controller Key` to `arrangedObjects` (the collection of objects being displayed).
3. Set `Model Key Path` to the NSImage property to display in that column. For example, in the media asset application, set the key path to `image`.

You configure the `value` binding for the NSImageView in the detail interface similarly, except that you set the `Controller Key` value to `selection` (that is, the currently selected object).

See "Using a Value Transformer to Convert Paths" (page 69) for a variation of this example that uses the `valuePath` binding.

## Using a Value Transformer to Convert Paths

If you want to access your images using a file path, then you bind your views to your controllers using the `valuePath` binding instead. However, `valuePath` is expected to be an absolute file path. Typically, you do not store absolute paths in your models, just filenames or relative paths. You can convert a filename or relative path to an absolute path using a custom value transformer as follows.

### Creating Custom Value Transformers

First create a custom value transformer that takes the filename or relative path and converts it to an absolute path. You do this by subclassing NSValueTransformer and overriding the `transformedValueClass` and `allowsReverseTransformation` class methods, as shown in Listing 1 (page 69).

You implement the `transformedValue:` method to perform the transformation. For example, the `transformedValue:` method implementation in Listing 1 assumes the images are located in the project `Resources` folder and uses NSBundle's `resourcePath:` method to convert the filename to an absolute path. Note that you need to modify this example if you store the images somewhere else on the file system.

**Listing 1** PathTransformer implementation file

```
#import "PathTransformer.h"

@implementation PathTransformer

+ (Class)transformedValueClass
{
    return [NSString self];
}

+ (BOOL)allowsReverseTransformation
{
    return NO;
}

- (id)transformedValue:(id)beforeObject
{
    if (beforeObject == nil) return nil;
    id resourcePath = [[NSBundle mainBundle] resourcePath];
    return [resourcePath stringByAppendingPathComponent:beforeObject];
}

@end
```

### Registering Value Transformers

In order to use your custom value transformer, you must first register it with NSValueTransformer. Note that you register instances of a value transformer, not a subclass. You typically register value transformers in an `initialize` method or the application delegate's `applicationDidFinishLaunching:` method. When you register a value transformer, you give it a logical name you can use later when configuring a bindings. For example, add the following code fragment to `applicationDidFinishLaunching:` to register an instance called `PathTransformer`:

```
id transformer = [[[PathTransformer alloc] init] autorelease];
[NSValueTransformer setValueTransformer:transformer forName:@"PathTransformer"];
```

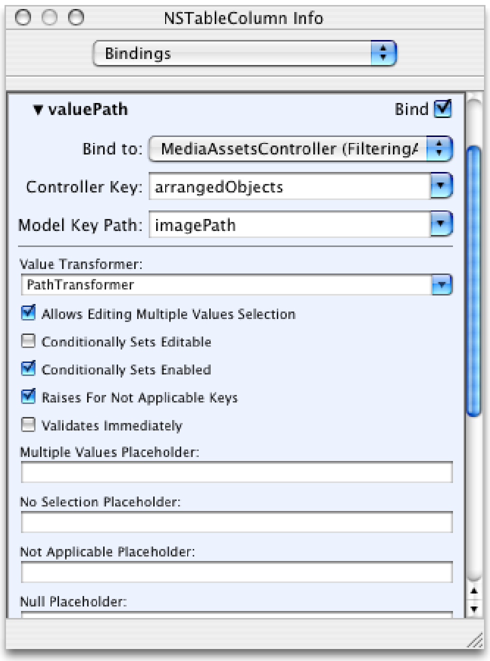
## Binding Views to Controllers Using Transformers

Finally, you specify the value transformer when binding your views to your controllers using the `valuePath` binding. For example, you configure the `valuePath` binding for the Image table column as follows:

1. Set the `Bind to` aspect to your array controller object—for example, `MediaAssetsController`.
2. Set the `Controller Key` aspect to `arrangedObjects` (the collection of objects being displayed).
3. Set the `Model Key Path` aspect to the property containing the image filename. For example, in the media asset application, set the key path to `imagePath`.
4. Enter the transformer's logical name, `PathTransformer` , in the Value Transformer text field.

Figure 2 shows the Bindings pane of an NSTableColumn in Interface Builder with the `valuePath` binding revealed and configured to use a custom value transformer.

**Figure 2** Bindings pane of an NSTableColumn



▼ valuePath — Bind ✓

Bind to: MediaAssetsController (Filtering/

Controller Key: arrangedObjects

Model Key Path: imagePath

Value Transformer: PathTransformer

✓ Allows Editing Multiple Values Selection
☐ Conditionally Sets Editable
✓ Conditionally Sets Enabled
✓ Raises For Not Applicable Keys
☐ Validates Immediately

Multiple Values Placeholder:

No Selection Placeholder:

Not Applicable Placeholder:

Null Placeholder:

To use the `valueURL` binding instead, implement a similar value transformer to convert filenames or relative paths to an appropriate NSURL object. Optionally, you can enhance the PathTransformer class to handle other types of filename transformations. In the latter case, register different instances using different names (for example, `PathTransformer` and `URLTransformer`) to handle each type of transformation.

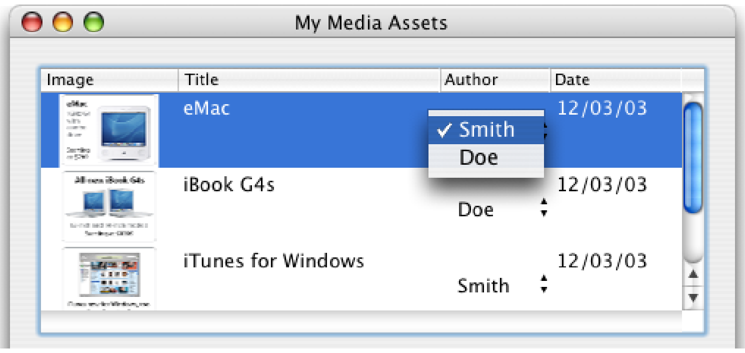
# Implementing To-One Relationships Using Pop-Up Menus

You can implement an editable to-one relationship using an NSPopUpButtonCell or NSPopUpButton. That is, allow the user to select a menu item from a pop-up menu in order to change the destination object of a to-one relationship in your model.

This article extends the example presented in "Creating a Master-Detail Interface" (page 59) by adding a pop-up menu to the table view in the master interface. Instead of displaying the author's last name in the Author column, you might allow users to select an author from a pop-up menu, as shown in Figure 1. Note that if you display just the last name, you have essentially flattened the to-one relationship. Consequently, if the user enters text in an editable Author column text cell, the user changes the value of the last name property belonging to the Person object, not the destination of the to-one relationship belonging to the Media object. Using pop-up menus is one way to implement an editable to-one relationship.

See "Creating a Master-Detail Interface" (page 59) for the steps to create a master-detail interface.

**Figure 1** Using pop-up menus to represent to-one relationships



## Creating Models and Controllers

To do this, you first need the supporting model and controller objects. Assuming you already have a master interface, you create an array of models that will be displayed in the pop-up menu, and you create an array controller to manage it.

How you initialize the collection of objects to display in the pop-up depends on your application. This example assumes that a collection already exists and is a property of the File's Owner (for example, a property called `authors` containing Person objects).

Create an NSArrayController by dragging it from the Cocoa-Controllers palette to your nib file, and rename it appropriately (for example, `AuthorsController`). It's common to have multiple controller objects per nib file so renaming them helps to identify them. Now, bind the model to the controller as follows. Select the array controller, display the Bindings pane in the Info window, reveal the `contentArray` binding, and configure it as follows:

1. Set the `Bind to` aspect to the object that maintains the collection of model objects (for example, the File's Owner).
2. Leave the `Controller Key` blank.
3. Set the `Model Key Path` to the name of the array (for example, `authors`).

Also enter the appropriate class name in the Object Class Name field on the Attributes pane (for example, enter the Person class name).

## Creating Views

Next, you create the pop-up views by dragging an NSPopUpButton to a window or an NSPopUpButtonCell to the column that will display the to-one relationship.

## Binding Views to Controllers

The primary bindings of an NSPopUpMenu and NSTableColumn (containing an NSPopUpButtonCell) that you will use to set up an editable to-one relationship are:

- `content`—the collection of objects to display in the pop-up menu.
- `contentValues`—the property of the objects in `content` that you want to display in the pop-up menu.
- `selectedObject`—the to-one relationship that users change when selecting an item from the pop-up menu.

**Note:** When using an NSPopUpButtonCell, you need to configure the bindings of the table column, not the cell. When you drag an NSPopUpButtonCell to a column, the Value category on the NSTableColumn Bindings pane changes to a Value Selection category containing the same bindings as an NSPopUpButton.

For example, if you want to display a pop-up menu in a column, configure the `content` binding to specify the content of the pop-up menu as follows:

1. Set the `Bind to` aspect to `AuthorsController`.
2. Set the `Controller Key` aspect to `arrangedObjects` .
3. Leave the `Model Key Path` aspect blank.

Then configure the `contentValues` binding to specify what should be displayed in the menu items as follows:

1. Set the `Bind to` aspect to `AuthorsController`.
2. Set the `Controller Key` aspect to `arrangedObjects`.

3. Set the `Model Key Path` aspect to `lastName`.

Finally, configure the `selectedObject` binding to specify the actual to-one relationship this pop-up menu changes as follows:

1. Set the `Bind to` aspect to `MediaAssetsController`.

2. Set the `Controller Key` aspect to `arrangedObjects`.

3. Set the `Model Key Path` aspect to `author` (a to-one relationship).

Now, when you run your application, a pop-up menu appears in each of the table column cells, displaying the current value of the to-one relationship. When the user selects another item from the menu, the controller changes the destination object of that to-one relationship.

Follow the same steps above to configure an NSPopUpButton as an editable to-one relationship. If you are implementing a master-detail interface and this pop-up button appears in the detail interface, then set the `Controller Key` for the `selectedObject` binding to `selection`, that is, to the currently selected object.

# Filtering Using a Custom Array Controller

This article explains how to use a search field and custom array controller to filter a collection of objects. Although you typically use a search field in conjunction with a table view to implement this style of interface, the tasks discussed in this article are not limited to these components.

## Overriding arrangeObjects:

The `arrangeObjects:` method sorts the content of the array controller using the current sort descriptors. To restrict the displayed content to a subset of the content you must override `arrangeObjects:`, create a new array containing the subset, and then call the superclass's implementation of `arrangeObjects:` to provide any appropriate sorting.

The `arrangeObjects:` implementation in Listing 1 performs an anchored search of the title property. In an anchored search, only those objects whose title property matches `searchString` are added to the returned array.

**Listing 1** Filtering implementation of arrangeObjects:

```
- (NSArray *)arrangeObjects:(NSArray *)objects {

    if (searchString == nil) {
        return [super arrangeObjects:objects];
    }

   NSMutableArray *filteredObjects = [NSMutableArray arrayWithCapacity:[objects
 count]];
    NSEnumerator *objectsEnumerator = [objects objectEnumerator];
    id item;

    while (item = [objectsEnumerator nextObject]) {
        if ([[item valueForKeyPath:@"title"] rangeOfString:searchString
options:NSAnchoredSearch].location != NSNotFound) {
            [filteredObjects addObject:item];
        }
    }
    return [super arrangeObjects:filteredObjects];
}
```

## Updating the Search String

You implement the `search:` action method to invoke `rearrangeObjects`, which triggers the invocation of the `arrangeObjects:` method.

Listing 2 shows an implementation of a search: action. The `searchString` is set to the string value of the `sender` object, and `rearrangeObjects` is called.

**Listing 2** Updating searchString

```
- (void)search:(id)sender {
    // set the search string by getting the stringValue
    // from the sender
    [self setSearchString:[sender stringValue]];
    [self rearrangeObjects];
}


- (void)setSearchString:(NSString *)aString
{
    [aString retain];
    [searchString release];
    searchString=aString;
}

@end
```

The `search:` action is invoked by an NSSearchField. The target of the search field is set to the array controller, and the action to the `search:` method. The search field should be configured so that it does not send the whole string as it changes. Turning this option off causes the `search:` method to be invoked each time that a keystroke occurs in the search field.

# Controller Key-Value Observing Compliance

The Cocoa bindings controller classes only provide key-value observing notifications for selected properties. This article enumerates the key-value-observing compliant properties for each class.

**Important:** The Cocoa bindings controller classes do not provide change values when sending key-value observing notifications to observers. It is the developer's responsibility to query the controller to determine the new values.

## NSUserDefaultsController

NSUserDefaultsController is key-value-observing compliant for the following properties:

- appliesImmediately
- defaults
- hasUnappliedChanges
- initialValues
- values

## NSObjectController

NSObjectController is key-value-observing compliant for the following properties:

- canAdd
- canRemove
- content
- isEditable
- objectClass
- selectedObjects
- selection

## NSArrayController

NSArrayController is key-value-observing compliant for the following properties:

- alwaysUsesMultipleValuesMarker
- arrangedObjects
- avoidsEmptySelection
- canAdd
- canInsert
- canRemove
- canSelectNext
- canSelectPrevious
- clearsFilterPredicateOnInsertion
- content
- filterPredicate
- isEditable
- preservesSelection
- selectedObjects
- selection
- selectionIndex
- selectionIndexes
- selectsInsertedObjects
- sortDescriptors

## NSTreeController

NSTreeController is key-value-observing compliant for the following properties:

- alwaysUsesMultipleValuesMarker
- arrangedObjects
- avoidsEmptySelection
- canAdd
- canAddChild
- canInsert
- canInsertChild
- canRemove

- canSelectNext
- canSelectPrevious
- content
- isEditable
- preservesSelection
- selectedObjects
- selection
- selectionIndexPath
- selectionIndexPaths
- selectsInsertedObjects
- sortDescriptors

**Note:** The value returned by NSTreeController's arrangedObjects method is opaque. You should only observe this property to determine that a change has occurred.

# Troubleshooting Cocoa Bindings

Applications that use Cocoa bindings can provide a challenge when attempting to troubleshoot problems. This article outlines some of the common issues encountered in applications that use Cocoa bindings and provides clues as to correcting the problem.

## My collection controller isn't displaying the current data

This is typically due to your application modifying the collection content in a manner that is not key-value-observing compliant. Modifying an array using `addObject:` or `removeObject:` is not sufficient.

You must either implement the indexed accessors for the collection key in your model class, or ask the model for a key-value-observing aware collection proxy using one of the following methods:

```
- (NSMutableArray *)mutableArrayValueForKey:(NSString *)key
- (NSMutableArray *)mutableArrayValueForKeyPath:(NSString *)keyPath

- (NSMutableSet *)mutableSetValueForKey:(NSString *)key
- (NSMutableSet *)mutableSetValueForKeyPath:(NSString *)keyPath
```

These return collection proxy objects that you can then use with the NSMutableArray or NSMutableSet mutation methods, and the appropriate key-value observing notifications are sent.

You can also modify the contents through the collection controller using the methods described in "Programmatically Modifying a Controller's Contents" (page 38).

## Changing the value in the user interface programmatically is not reflected in the model

If you change the value of an item in the user interface programmatically, for example sending an NSTextField a `setStringValue:` message, the model is not updated with the new value.

This is the expected behavior. Instead you should change the model object using a key-value-observing compliant manner.

## Changing the value of a model property programmatically is not reflected in the user interface

If changes made to a model value programmatically are not being reflected in the user interface, this typically indicates that the model object is not key-value-observing compliant for the property, or that you are modifying the value in a manner that is bypassing key-value observing. You should ensure that:

- The model class has automatic key-value observing enabled or implements manual key-value observing for the property.
- That you are changing the value using an accessor method, or using a key-value-coding compliant method. Changing the value of an instance variable directly does not provide key-value observing change notifications.
- If your model property is a collection, that you're modifying the content in a key-value-observing compliant manner. See "My collection controller isn't displaying the current data" (page 83) for more information.

## Binding to the incorrect key path

A common source of problems when using bindings is that a binding is established to the wrong key path. It is easy to miss the debug messages in the run log or console when the application doesn't behave as expected.

The following is an example of a debug message that is output to the run log or console:

```
2005-05-10 03:53:29.764 VuesActivity[2216] *** NSRunLoop ignoring exception
'[<Member 0x379180> valueForUndefinedKey:]:
this class is not key value coding-compliant for the key named.' that raised
during posting of delayed perform
with target 37f6d0 and selector 'invokeWithTarget:'
```

This message indicates that the instance of the Member class is not key-value-coding compliant for the key "named". In this case the error is that the correct Member key is in fact "name".

You can get more information about an error by setting the bindings debug level default. This can be done on the command line as follows:

```
defaults write com.yourdomain.yourapplication NSBindingDebugLogLevel 1
```

You can also set the debugging level in Xcode:

1. Select the application executable in the Executables group.
2. Choose the Get Info menu item in the File menu.
3. Select the Arguments tab in the executables info panel.
4. Add `-NSBindingDebugLogLevel 1` to the "Arguments to be passed at launch" list.

After setting the debug level default, the same problem that generated the above error message now generates the following error:

```
2005-05-10 04:02:32.146 VuesActivity[2241] Cocoa Bindings: Error accessing value
 for key path selection.named
of object <NSArrayController: 0x349ba0>[object class: Member, number of selected
 objects: 1]
(from bound object <NSTextField: 0x344380>): [<Member 0x37a660>
valueForUndefinedKey:]: this class
is not key value coding-compliant for the key named.
```

This provides more information that can aid you in tracking down which item in Interface Builder has been bound to the wrong key path. In this case it indicates that the binding is between an NSTextField and an NSArrayController with an object class of Member, using the `selection.named` key path.

With the binding debug level set to 1, any binding made to a non-existent key will cause a debug message to be printed to the console, even if the binding has disabled the “Raises For Not Applicable Keys” option.

Currently only a debug level of 1 is supported.

**Note:** The bindings error messages are logged after the exception is raised. If your application is set to break on [NSException raise] you will not see the error message in the run log or `gdb`. Be sure to check your `.gdbinit` file for a breakpoint if you still do not see the bindings error messages.

## The bindings of my custom view are disabled in Interface Builder

If you implement a subclass of NSView and create an Interface Builder palette for that subclass, your subclass’ implementation of `bind:toObject:withKeyPath:options:` must call the super implementation. If you don’t when you establish a binding in Interface Builder, the binding values disappear and the binding names become disabled and uneditable.

## An uneditable/visible/disabled NSTextField becomes editable/hidden/enabled

If you disable editing for an NSTextField in Interface Builder, and then establish a binding to it, you may find that the field then becomes editable when your application is running. This is typically due to the value binding having the “Conditionally Sets Editable” binding option enabled.

Similarly, if a control becomes hidden or enabled, you should check that the “Conditionally Sets Hidden” and “Conditionally Sets Enabled” options of the value binding have the expected settings.

## Binding a control to a value that is an unsigned int causes an exception

Key-value coding handles the conversion of NSNumbers and NSValues to some well-known scalar and structure types, but unsigned int is not currently supported. In the context of bindings, you should consider transforming the value using an NSValueTransformer, or in the specific case of NSTextField, NSFormatter.

## The view bound to an NSTreeController is not displaying data

This is often caused because the NSTreeController's children key path has not been set. You can verify that this is the cause by checking the console or run log for debug messages. See "Traversing Tree Content with an NSTreeController" (page 37) for more information.

```
2005-06-23 02:46:53.198 MyApplication[6777] Cocoa Bindings: Cannot perform
operation if childrenKeyPath is nil.
```

## A table column containing NSPopUpButtonCell items is not sortable

Tableview columns that contain NSPopUpButtonCell, or any cell type other than NSTextField can't be sorted if the cell's selection is bound using the `selectedObject` or `selectedObjects` bindings. If the selection is bound to selectedValue, selectedTag or selectedIndex, the column is sortable.

# Document Revision History

This table describes the changes to *Cocoa Bindings Programming Topics*.

| Date | Notes |
|---|---|
| 2009-03-08 | Added warning about lack of key-value observing change values to Controller Key-Value Observing Compliance. |
| 2007-05-22 | Clarified why you would use a controller. |
| 2006-07-24 | Corrected minor typos. |
| 2006-06-28 | Updated the prerequisite section in the introduction. |
| 2006-05-23 | Clarified Figure 3 in "Bindings Message Flow." |
| 2006-03-08 | Corrected the NSBindingDebugLogLevel flag in the "Troubleshooting Cocoa Bindings." |
| 2005-08-11 | Added "Controller Key-Value Observing Compliance " article. Added additional troubleshooting information. Corrected minor typos. |
| 2005-07-07 | Added four new articles on controller content, controller selection, message flow between model-view-controller, and troubleshooting Cocoa bindings. Made other minor changes. |
| 2005-04-29 | Corrected minor typo. Added retain to example code in "How Do Bindings Work?". |
| 2004-08-31 | Corrected minor typos. |
| 2004-04-20 | This version represents a major revision of *Cocoa Bindings*. "What Are Cocoa Bindings?" (page 11) and "How Do Bindings Work?" (page 23) replace the "Why Use Cocoa Bindings?" and "Support for the MVC Paradigm" articles. The "How to Use Bindings" article was removed. "User Defaults and Bindings" (page 55) was added to the topic. "Filtering Using a Custom Array Controller" (page 77) was edited to remove model dependencies. "Example: Currency Converter With Bindings" was moved to a separate book *Cocoa Application Tutorial Using Bindings*. "Example: Enhanced Currency Converter" and "Example:Preferences Pane" were removed from the topic. |
| 2004-02-22 | Added a link to *Cocoa Bindings Reference* from "Introduction to Cocoa Bindings Programming Topics" (page 9). Corrected a bug in "Filtering Using a Custom Array Controller" (page 77) that caused the NSSearchField to loose focus. |

| Date | Notes |
|---|---|
| 2003-12-16 | Renamed to *Cocoa Bindings* from *Controller Layer*. Made minor edits to "Supporting the MVC Paradigm" and "Object Modeling". Revised "Core Classes and Protocols" and renamed it "How to Use Bindings". Added four new articles on how to create a master-detail interface, display images using bindings, use pop-up menus to represent to-one relationships, and filter an array controller's arranged objects. |
| 2003-10-14 | Renamed to *Controller Layer* from *The Cocoa Controller Layer*. Revised and reorganized all the conceptual articles in this topic. Enhanced the tutorials by numbering the steps, adding a Converter model, and fixing other errors. |
| 2003-08-07 | First version of *The Cocoa Controller Layer*. |