Chapter 9

NIBWARE AND THE BASIC APPLICATION

CHAPTER 9 NIBWARE AND THE BASIC APPLICATION

Goal

To understand the basic objects and connections used to create nibware, the fundamental building block of applications.

Prerequisites

Experience with Objective-C, Project Builder, Interface Builder, and creating basic single-window applications.

Objectives

At the end of this section, you will be able to:

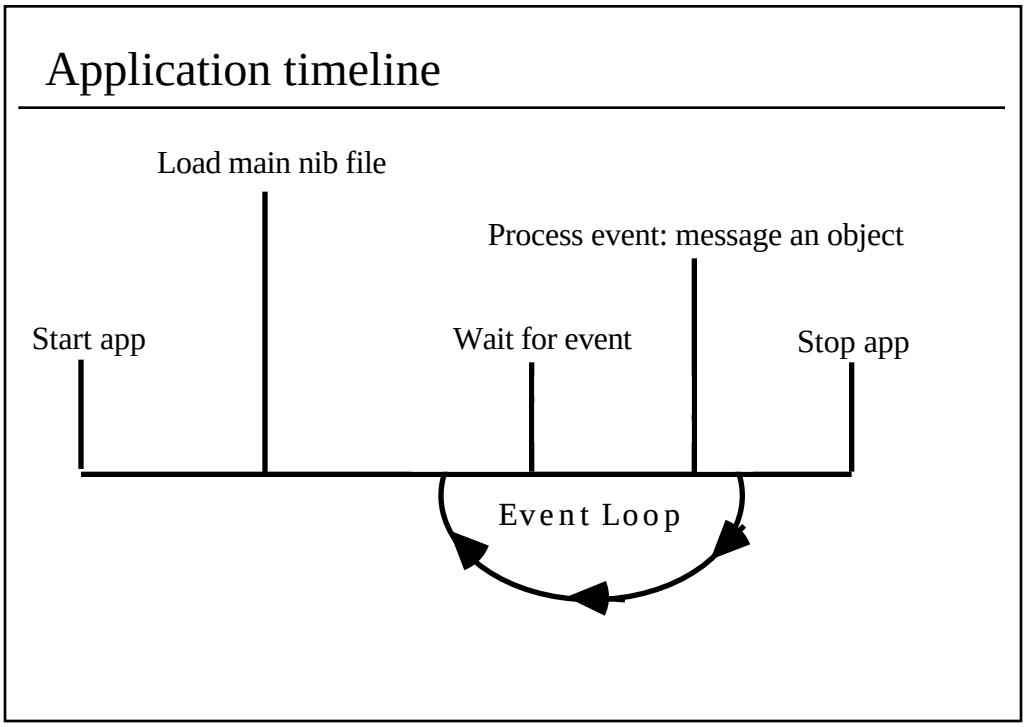
- Explain the contents of a nib file including basic controls, connections and design patterns
- Describe the useful attributes of buttons and text field controls
- Explain the purpose of **awakeFromNib** and identify the operations that typically would be performed there
- Build a basic single-window application using Interface Builder

Reading

For an introduction to the frameworks, Interface Builder, nib files, and the basics of application design, see:

Developer Tutorial

Development: Tools and Techniques



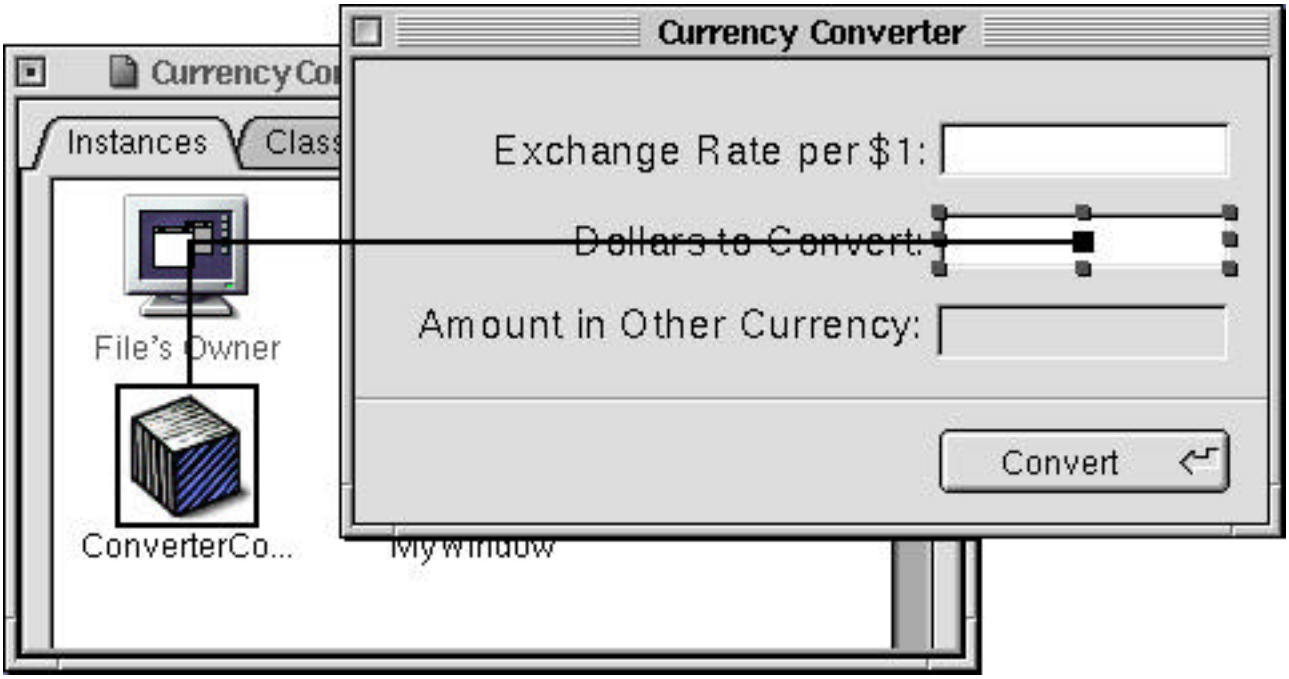
Application timeline

Our graphical applications follow this basic pattern:

- Load the initial user interface from the main nib file
- Repeatedly react to user-generated events by messaging application objects until a terminate message is encountered

The basic paradigm of a user driving the application through the graphical interface, makes this style of programming different from more traditional procedural programming.

The event loop, which processes events one at a time, once per cycle, is an important focal point that impacts many aspects of designing our applications. It is useful to keep this picture clearly in the back of your mind.



What's in a nib file?

A nib file is an archive of object instances generated by Interface Builder. Unlike the product of many user interface building systems, a nib file is not generated code. The objects in the nib file are graphically instantiated and edited in Interface Builder and include:

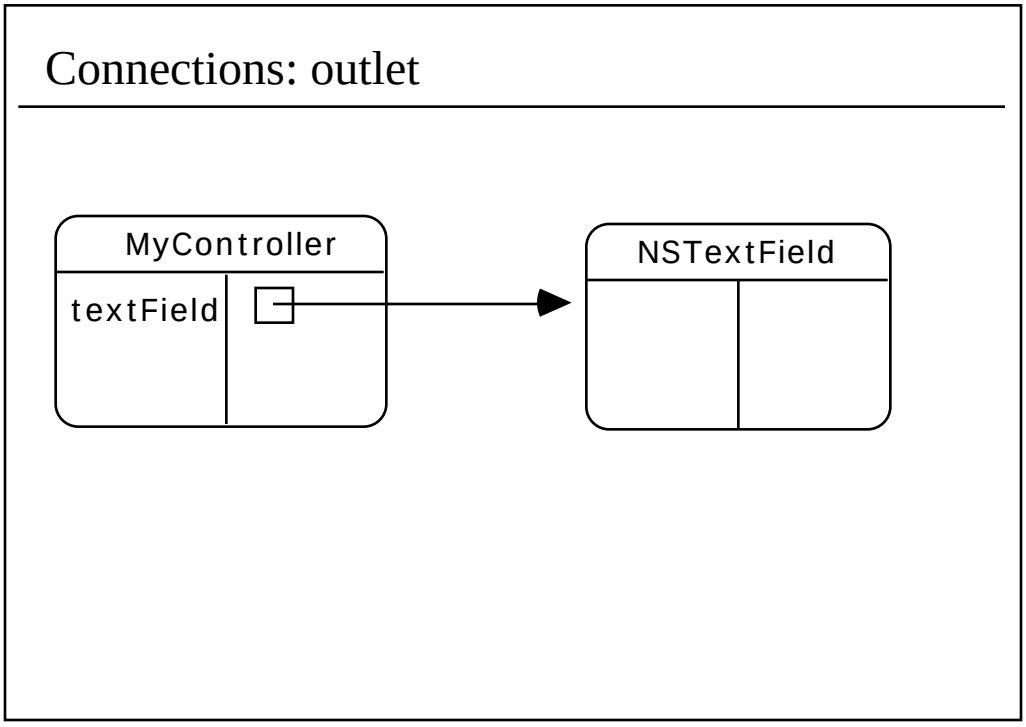
» Application Kit Objects

» Your custom objects

Application Kit objects include windows, buttons, titles, text fields, and the like. Your custom objects implement the interface portion of your application and are likely to be targets, delegates, and controllers that interact with the Application Kit objects to form a coherent system.

Most objects have attributes that are specifically configured to suit the purposes of the user interface, usually via the Interface Builder inspector. Among the most powerful attributes of an object are the various connections it has to other objects in the interface.

When the nib file is loaded at runtime, this graph of objects is instantiated, configured and connected just as it was in Interface Builder.



Connections: outlet

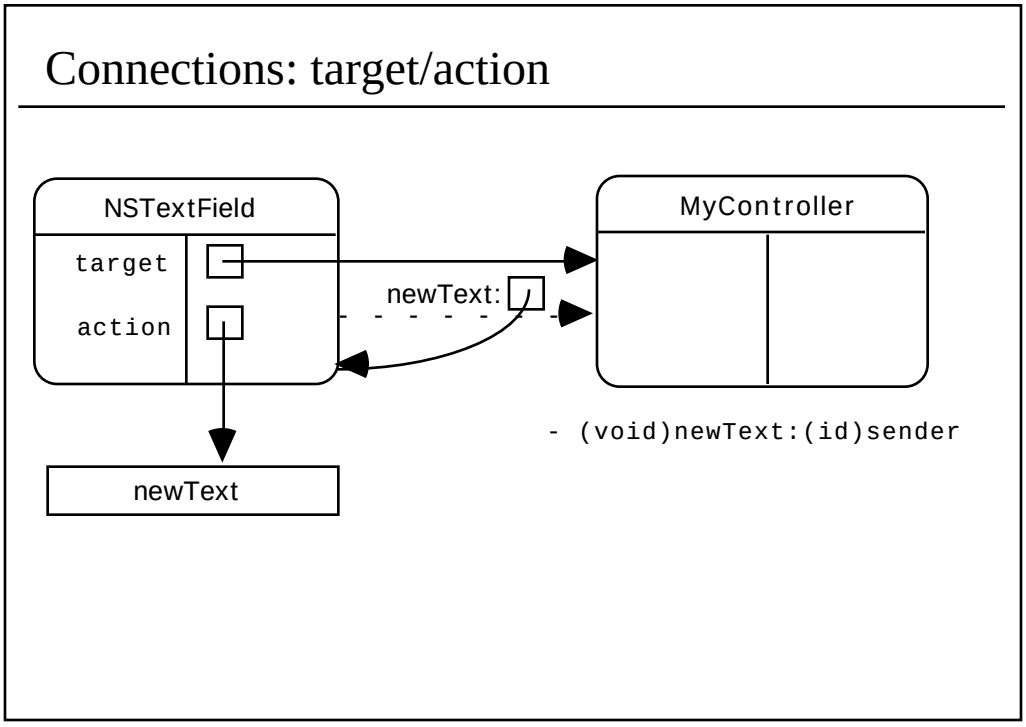
The most fundamental connection is the outlet: one object points to another via one of its instance variables. Once an object has the pointer to another, it can send it messages or let other objects know about it. This forms the basis for many kinds of inter-object communications like target/action and delegation. This also allows you to further configure and query objects dynamically at runtime.

Your custom objects will have outlets to Application Kit objects and vice versa. This is one way in which a generic reusable framework can be customized for a specific application.

Interface Builder only recognizes certain instance variables as outlets:

```
id anInstanceVariable;
```

They must be of type id—dynamically typed—and must appear in your interface file, one variable per line. If not, your instance variable will not show up as a possible outlet in the connections inspector.



Connections: target/action

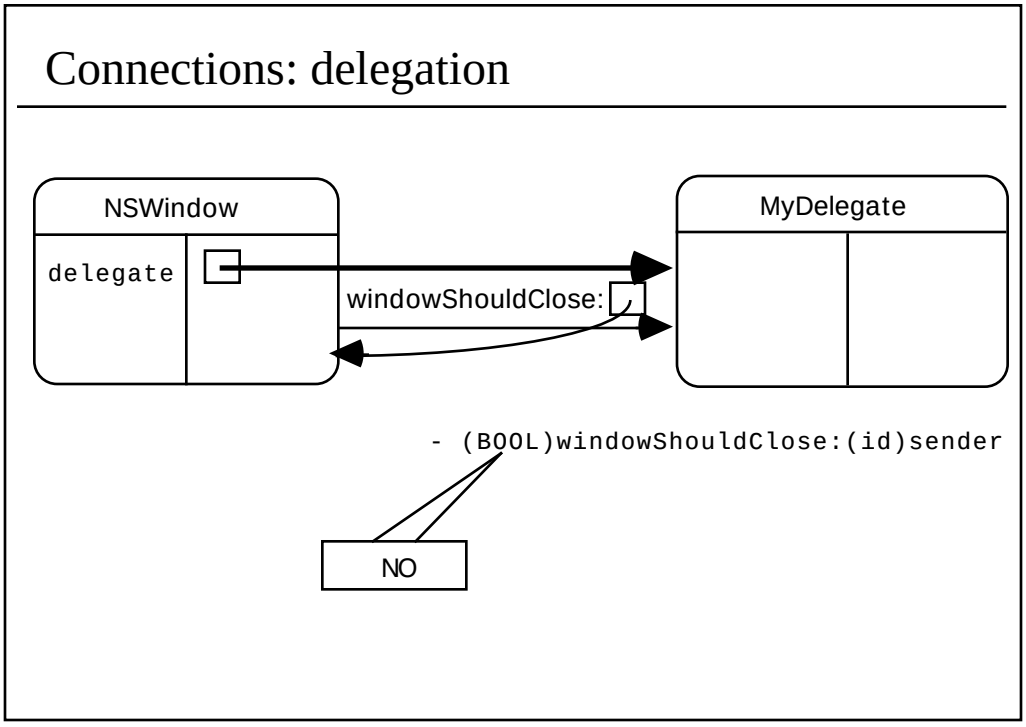
Many Application Kit objects are controls that give the user a way to make the application respond to a specific request. These controls respond by messaging a target object that will carry out the request in an application specific way. The target object is typically one of your own custom objects.

Target/action requires both an outlet to the target object, and an indication of which message to send. The latter is not an outlet, but simply a selector, a language specific way of storing a message for future use. The target/action connection requires two instance variables

Messages eligible for action messages must follow a strict prototype:

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

Action messages have no return value and take a single parameter—a dynamically typed object instance. With access to the message sender, the target object can send messages back to the control, querying for information or reconfiguring its state. If your method prototype does not match this format, it will not show up as a possible action in the Interface Builder connections inspector.



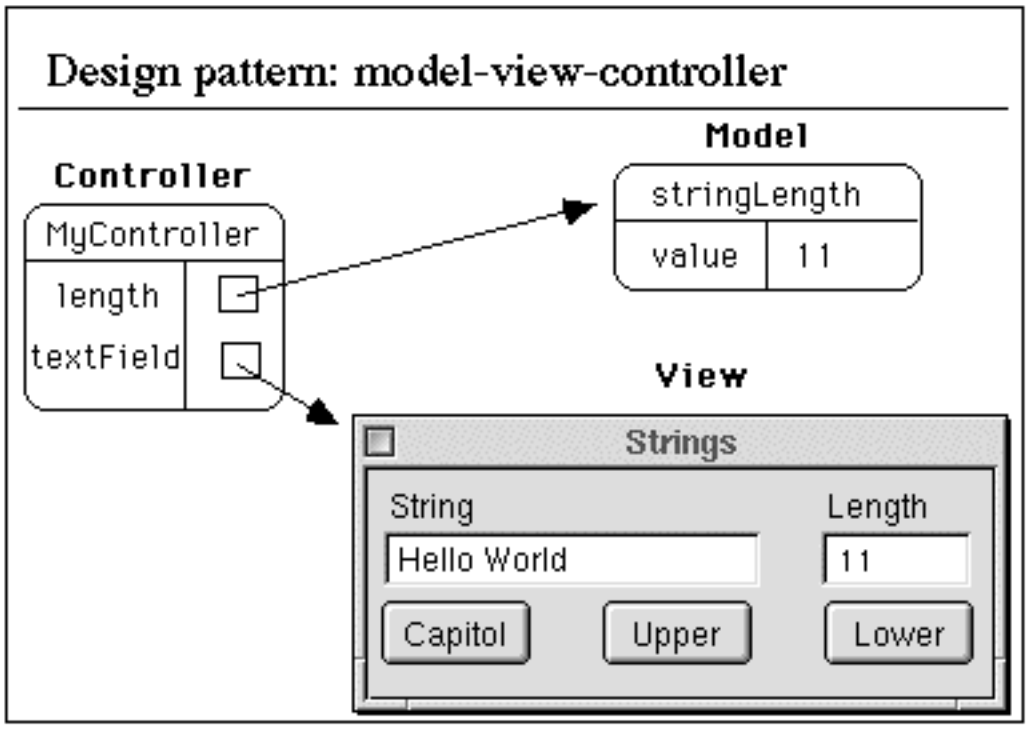
Connections: delegation

Delegation is also implemented with an outlet. Again, the delegate outlet is typically in an Application Kit object such as a window or text view, and it usually points at one of your own custom objects. You supply the delegate.

Which messages are sent and when depends on the particular Application Kit class and whether or not your delegate responds to one or more of the given messages. The list of possible messages is fixed by the Application Kit object. Your delegate can implement as many or few as you wish in order to get the job done.

Delegates can be asked whether or not something should happen, and be notified of status changes which will or have already taken place regardless. In both cases, a delegate can supplement the behavior of the Application Kit object, customizing the behavior to suit the goals of a specific application.

Application Kit objects which use delegates message them directly for some messages and notify them indirectly for others, using NSNotificationCenters. When your custom object is attached as a delegate, it will automatically be registered for notifications as well.

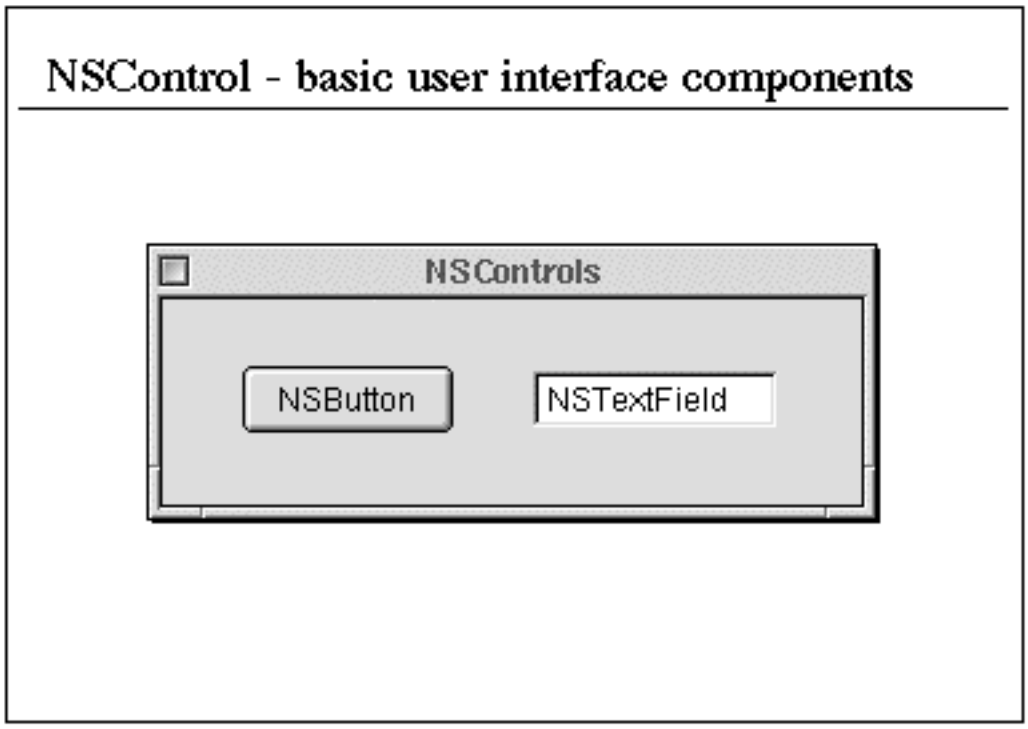


Design pattern: model-view-controller

Application Kit objects are designed specifically for implementing graphical interfaces: a user's view of the application and the data it manipulates. The data, on the other hand, usually exist regardless of the user's view; it is independent of the application itself. It may live in a file, a database, even on a remote server. Connecting the model to the view is the job of a third player, your custom object. It is often referred to as a controller because it controls the relationship between the model and the view, namely, how changes in one affect and are reflected in the other.

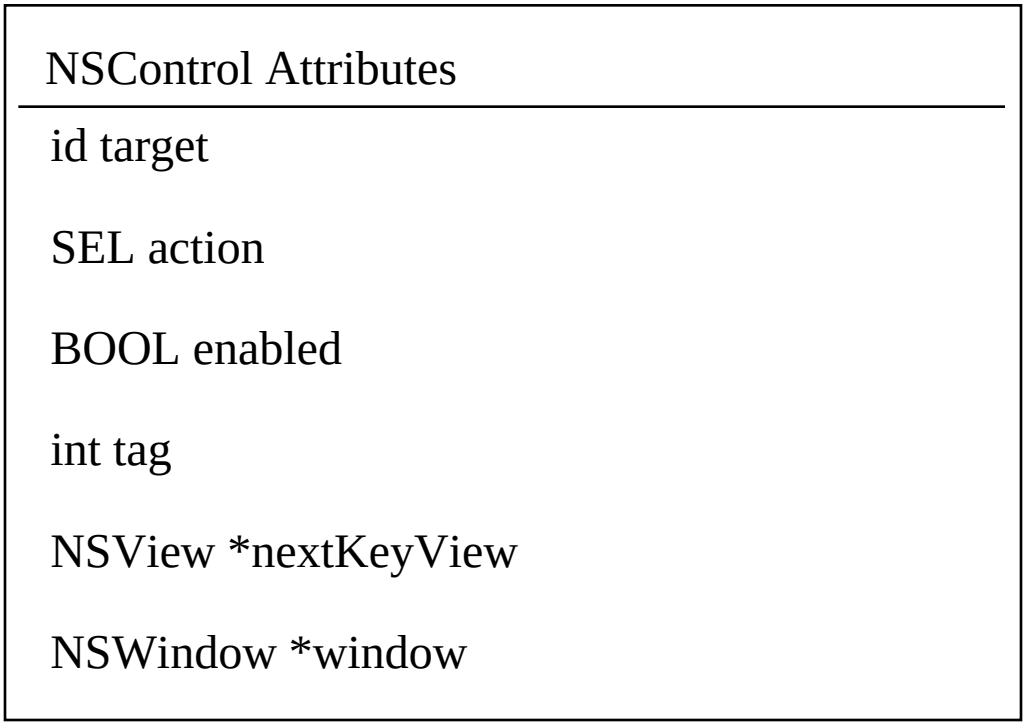
The model class often comes from another source and is likely to remain quite constant in terms of its interface and design. It is reused throughout a variety of applications in the enterprise. Likewise, the Application Kit objects are reusable, with fixed API, requiring no design on your part. The bulk of your work involves the custom controller objects that connect the model and the view objects in a way that is tailored to your specific application needs.

This trio of objects, a model, a view and a controller, forms a basic design pattern that is used throughout the Application Kit and in many areas of application design. This is another picture you would do well to keep in the back of your mind. Be aware of how often this design pattern is used even when these precise names are not mentioned.



NSControl—basic user interface components

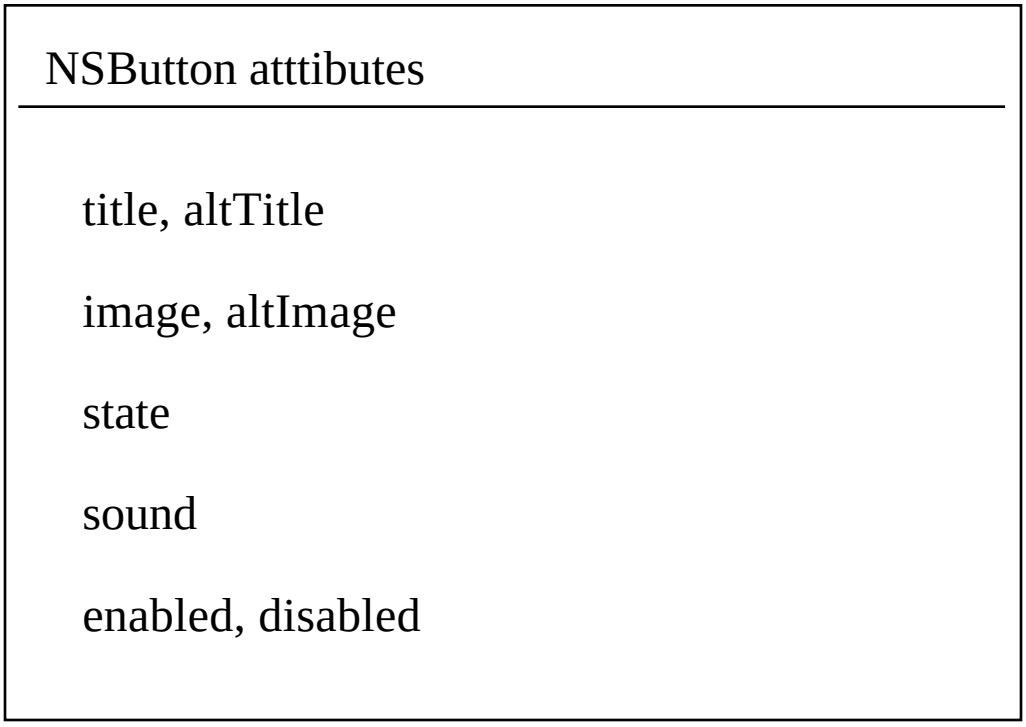
The basic component of just about any user interface is some kind of control. As its name suggests, it gives the user control over the behavior of the application. Because of attributes and behavior they share, all controls are subclasses of the abstract superclass, NSControl.



NSControl attributes

There are a number of specialized controls—buttons, text fields, sliders, and matrices, and so on. They all inherit attributes and behavior from their super classes, and from NSControl in particular. These attributes are useful and generic to all controls:

- **target, action:** all controls use target/action to message your custom objects.
- **enabled:** whether or not the control is actually available for the user. Controls are disabled when state changes in the application make them inappropriate. This status is likely to change dynamically as the application runs.
- **tag:** a unique integer for distinguishing between several controls.
- **nextKeyView:** the next view (usually a. control) reached via the TAB key.
- **window:** the window object within which the control lives.



NSButton attributes

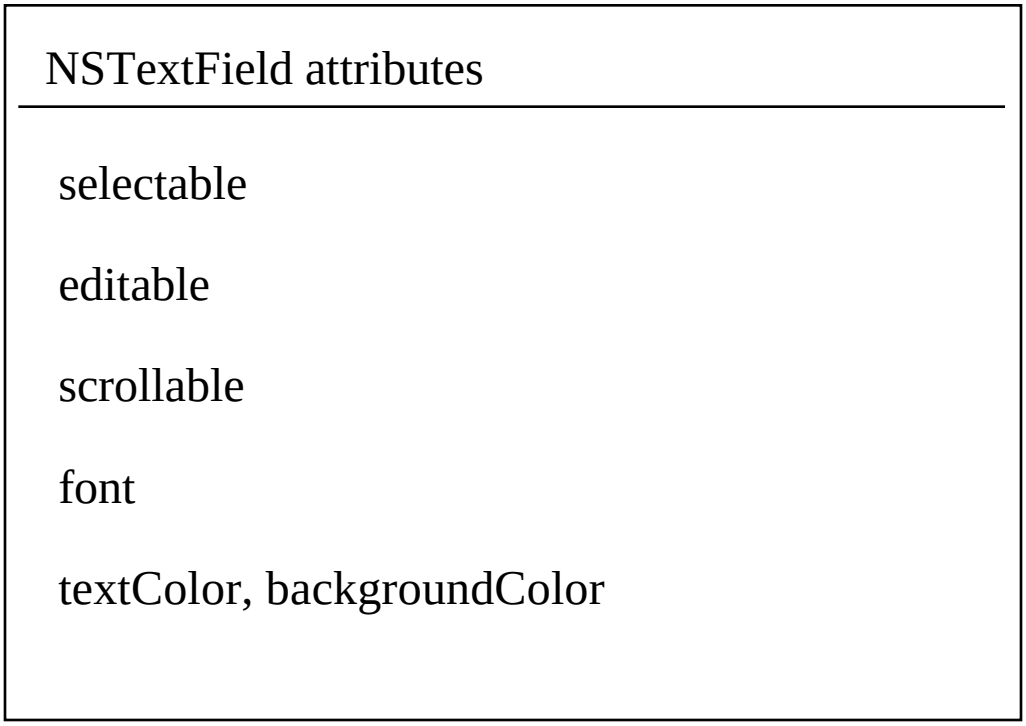
A button is the most basic control. A user presses it and triggers an action somewhere in the application, namely, the target object to which it is connected. With a small set of attributes, an NSButton can be configured into a surprisingly wide range of different kinds of buttons.

To indicate their function in the interface, buttons know how to label themselves with titles and/or images.

The most important distinction among button types is whether they are momentary-push or two-state buttons. The first is used simply to trigger an action—always the same action. A two-state button, like a toggle, check or radio button, also tracks the state of the button. This can be visually reinforced with alternate titles and/or images. While the action is still triggered with each button press, its behavior will be conditional on whether the button is on or off, black or white, or yes or no.

A button may be configured to continously send the target/action message as long as it is held in a pressed position. The time interval between messages is likewise configurable. By default, buttons are not continuous.

In addition to images, buttons can have a sound associated with them in order to produce audible feedback when they are pressed.

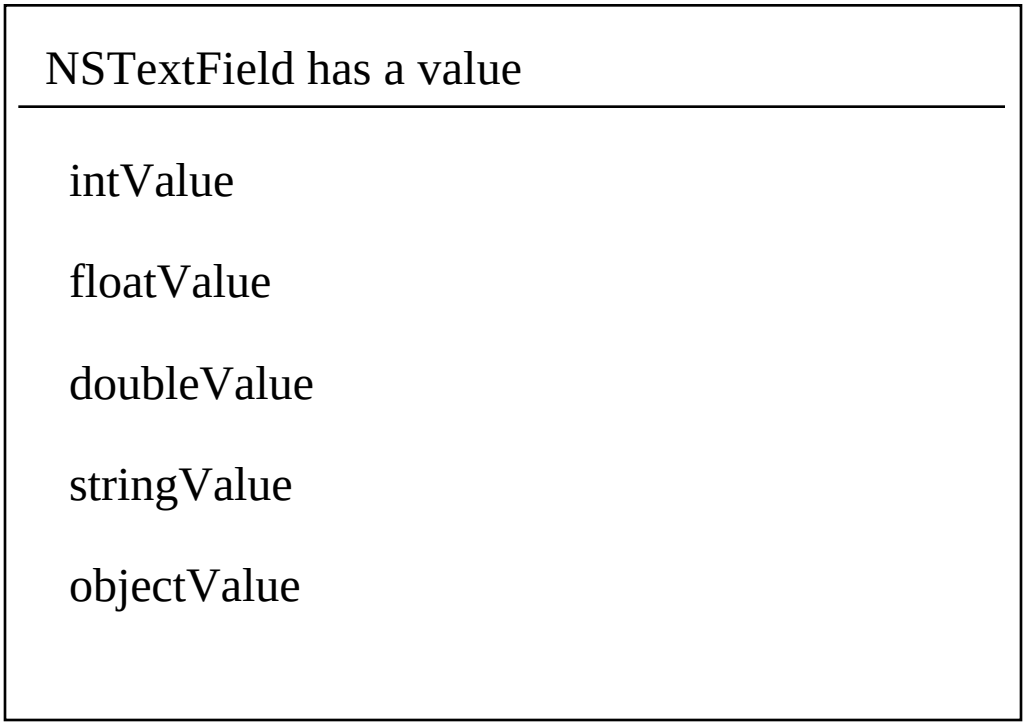


NSTextField attributes

NSTextField adds the ability to display arbitrary values in the UI, most likely an attribute of an Enterprise Object instance. NSTextField likewise provides the user with a data entry control, permitting arbitrary input values from the user interface. NSTextFields essentially provide a one line, full rich text editor just for the occasion! NSTextFields send a target/action message when the user presses Return.

Like NSButton, NSTextField has some useful attributes that enable you to configure a variety of different flavors for different user interface needs:

- **selectable**: whether the user can set the focus to the text field, a requirement for editing. Titles are non-selectable NSTextFields.
- **editable**: whether the user can edit the text fieldValue. Read-only TextFields are common and often use a non-white background color.
- **scrollable**: whether the NSTextField can scroll to view and edit a value that is larger than its visible frame. Non-scrollable fields are useful for fixed width attributes.
- **font**: for emphasis, or aesthetic appeal, a variety of fonts can be set both programmatically and by the user with the font panel.
- **textColor**, **backgroundColor**: useful for conveying extra meaning or for aesthetic appeal, NSTextField works with colors. Non-echoing password fields set the same text and background colors rendering typing invisible.



NSTextField has a value

NSTextFields maintain an arbitrary value, whether set programmatically or interactively by the end user via the UI. To suit a variety of application needs, a TextField can accept or return a number of different value types. You can work with primitive C types:

- int
- float
- double

Or with objects:

- string
- object

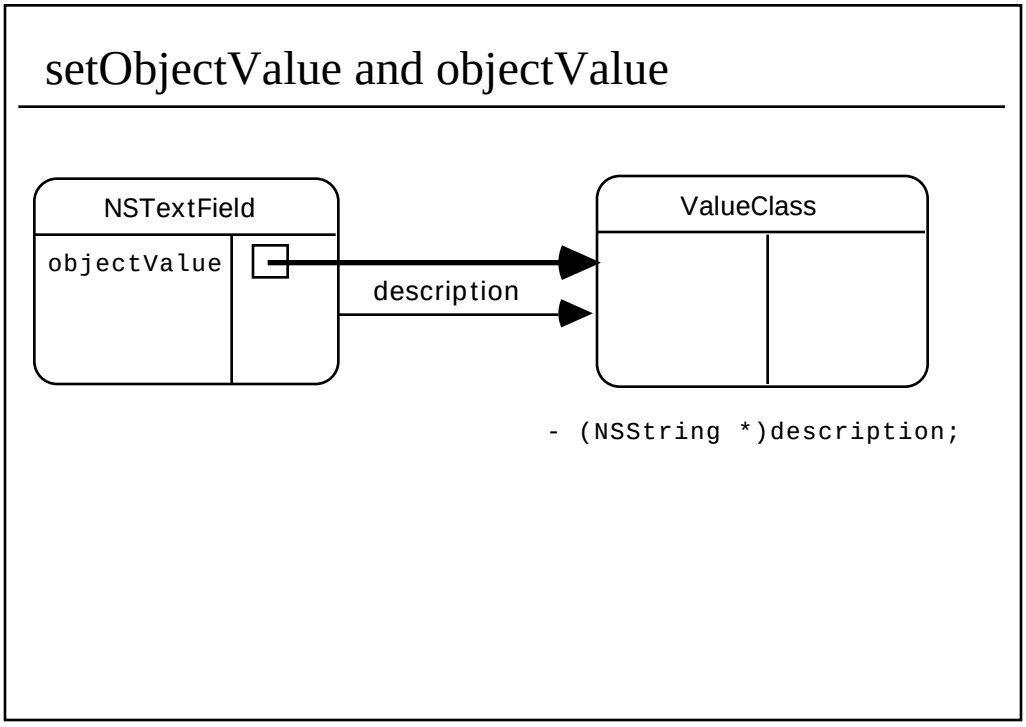
Each of these types provides a pair of accessor methods using the following format:

```
- (id)objectValue;

- (void)setObjectValue:(id)object;
```

where id is replaced with the specific value type, i.e. int, float, double, or NSString *.

Other NSControl subclasses dealing with values—such as NSSlider—will likewise respond to these messages.

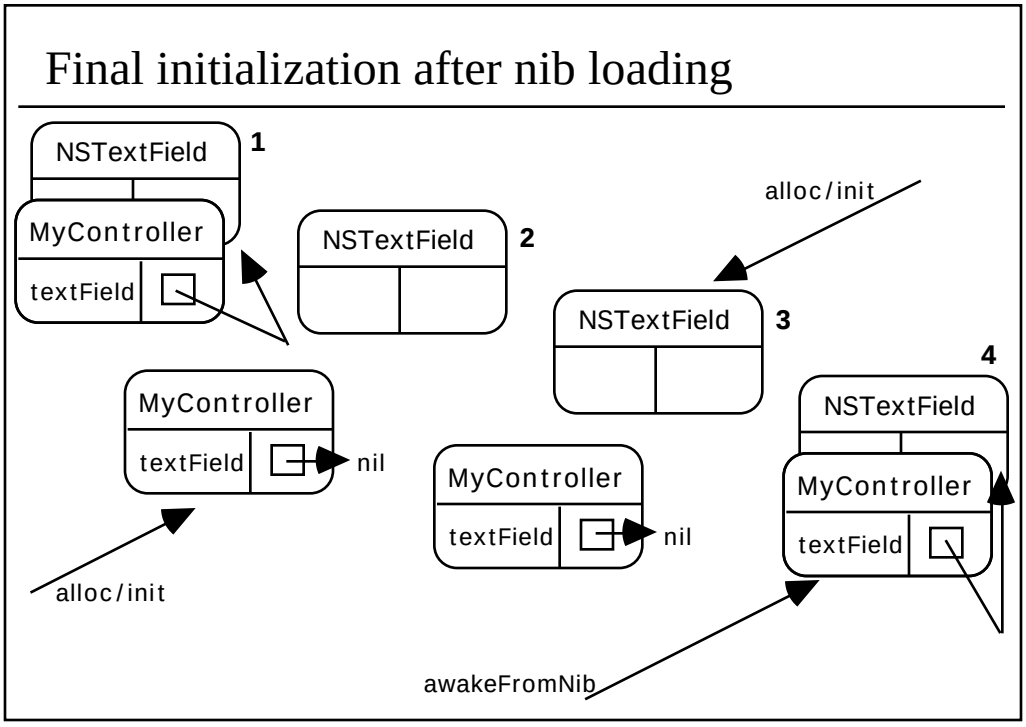


setObjectValue and objectValue

With **objectValue**, NSTextField can work with arbitrary value object classes such as NSNumber, NSCalendarDate or a custom value object of your design. How does the text field know what to display? The value object must respond to the **description** selector which returns a string representation of the object's value. Ultimately, a text field can only display NSStrings.

The relationship between a TextField and its **objectValue** becomes more interesting, and even more powerful, when a third object is placed between them to act as a sort of filter. This is the case with formatters.

Note, although you can display an arbitrary value object class with **setObjectValue**, without a formatter, the **objectValue** method returns an NSString instance. This is because user input arrives from the user interface as a string and cannot be converted into a non-string value object without the help of a formatter.



Final initialization after nib loading

While nearly everything in a nib file can be configured with Interface Builder, some final adjustments are typically necessary at run time, after the nib has been loaded. Consider a text field that displays the current date or an account balance fetched from a database.

Who should update the value of the NSTextField? Most likely, it should be your controller object that has an outlet pointing to the NSTextField. It lives in the nib file along with the NSTextField. When?

When a nib file is loaded, the archived objects are instantiated and configured to reflect their attribute values established in Interface Builder. Once all the objects are in memory, they are connected via outlets. **init** is too early for your controller to message the NSTextField since its outlet is not properly connected, nor it is clear if the NSTextField has even been instantiated.

After the nib loading is complete, when all the objects are unarchived and connected, each object from the NIB that responds to the message will be sent **awakeFromNib**. At this point, it is guaranteed that your controller's outlet is connected to a fully configured NSTextField instance. Now is the time for your controller to fetch the relevant data, and update the NSTextField's value. Although this example focuses on a text field in particular, the technique applies to final initialization of any Application Kit object— buttons, slides, text, table views, and so on.

awakeFromNib

```
- (void)awakeFromNib
{
   [textField setFloatValue: [rate value]];
   [[textField window] makeKeyAndOrderFront:nil];
}
```

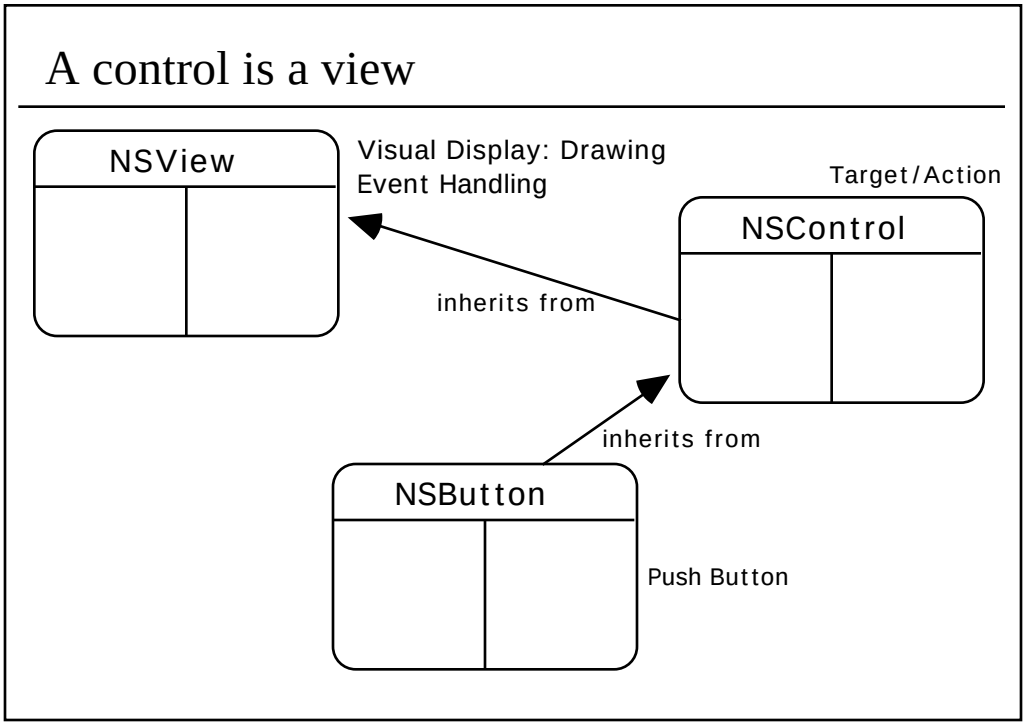
awakeFromNib

awakeFromNib is sent to each object in the nib file that responds to it. It is sent after all the objects are unarchived and connected but just before the interface is made visible to the user. Typically, **awakeFromNib** is implemented by your controller objects which are responsible for the state of the Application Kit objects they control.

This code sample shows how you might go about updating an NSTextField using a value obtained from another object. **awakeFromNib** will be called before the interface becomes visible to the user.

Another typical chore for **awakeFromNib** is to message the window object from the nib, ask it to come to the front of the screen and become the active window. This is not done by default because some interfaces do not need to become key when they are presented. Remember, evey control can reach the window it is on. Any window can be placed in front as the key window with **makeKeyAndOrderFront:**. More about NSWindows and window ordering later.

Additional Points to Ponder

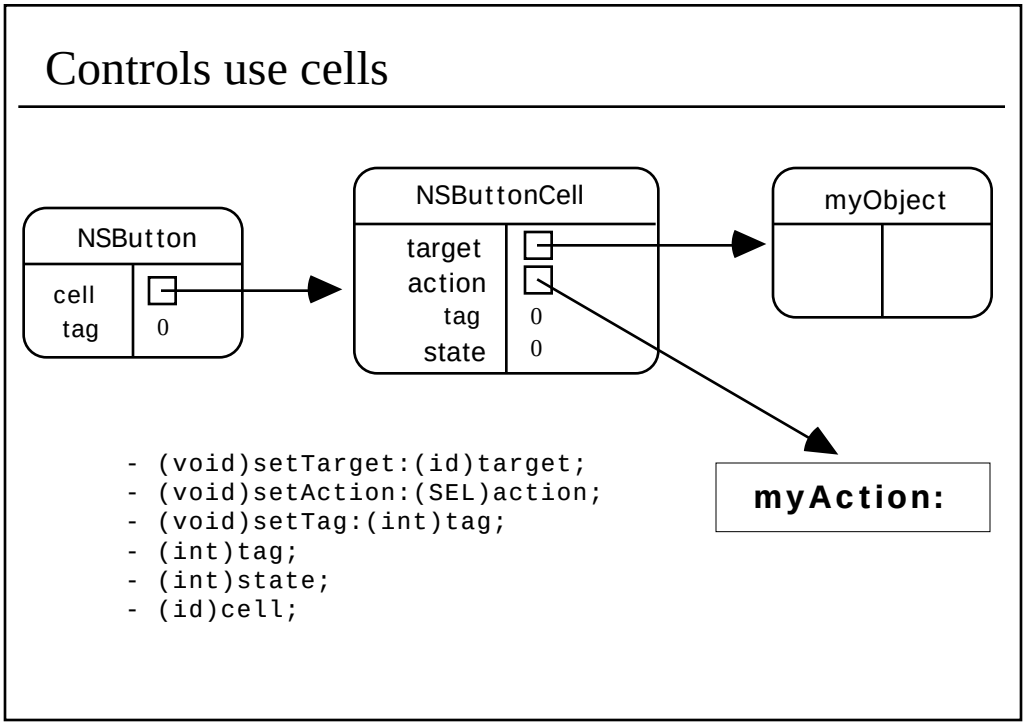


A control is a view

A control is a user interface component—it is visible. Accordingly, NSControl is a subclass of NSView. NSView will be explored in greater detail later, but it is useful to see now what NSView contributes to NSControl via inheritance:

- The ability to display or become visible
- The ability to handle user events such as mouse and keyboard

As an abstract super class, NSControl adds the ability to implement target/action behavior. Individual NSControl subclasses further specialize into specific kinds of visual display and target/action handling such as NSButton, NSTextField, or NSSlider.

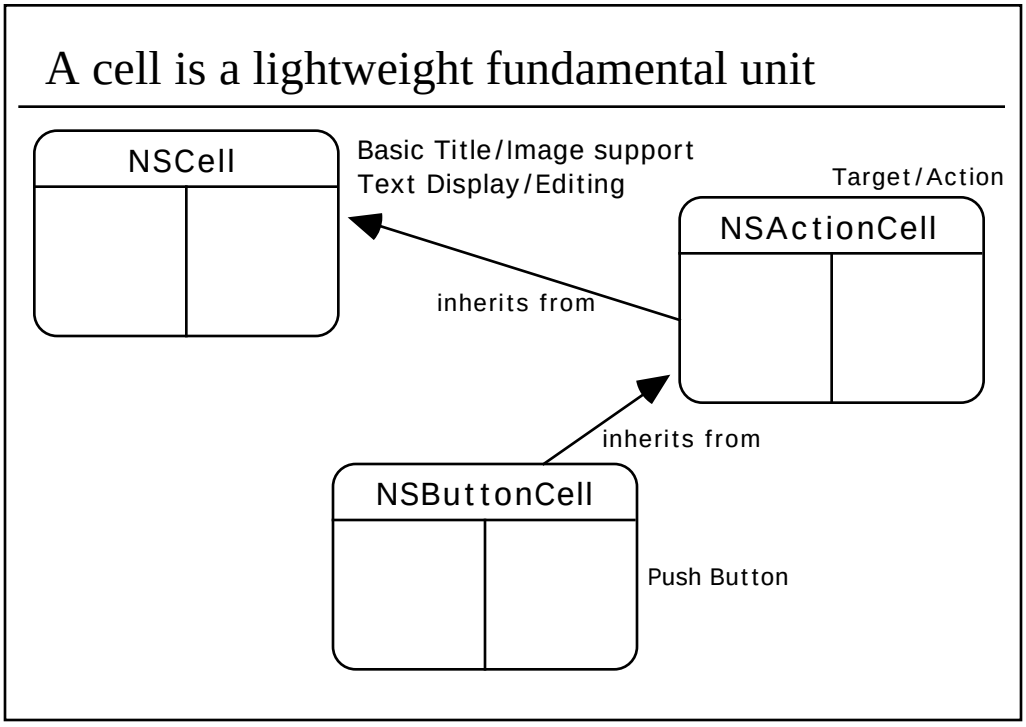


Controls use cells

Every NSControl subclass uses a companion object to help get its job done. Every control uses a **cell**, a subclass of NSCell. Notice that key NSControl attributes like **target**, **action**, or **state** may not even live in the control itself but in the associated NSCell instance. Many of NSButton's accessor methods really just "pass through" to the corresponding NSCell methods.

In addition to storing these instance variables, cells also know how to draw themselves. An NSButtonCell knows how to draw its 3-dimensional border, title and icon. An NSTextFieldCell knows how to display text.

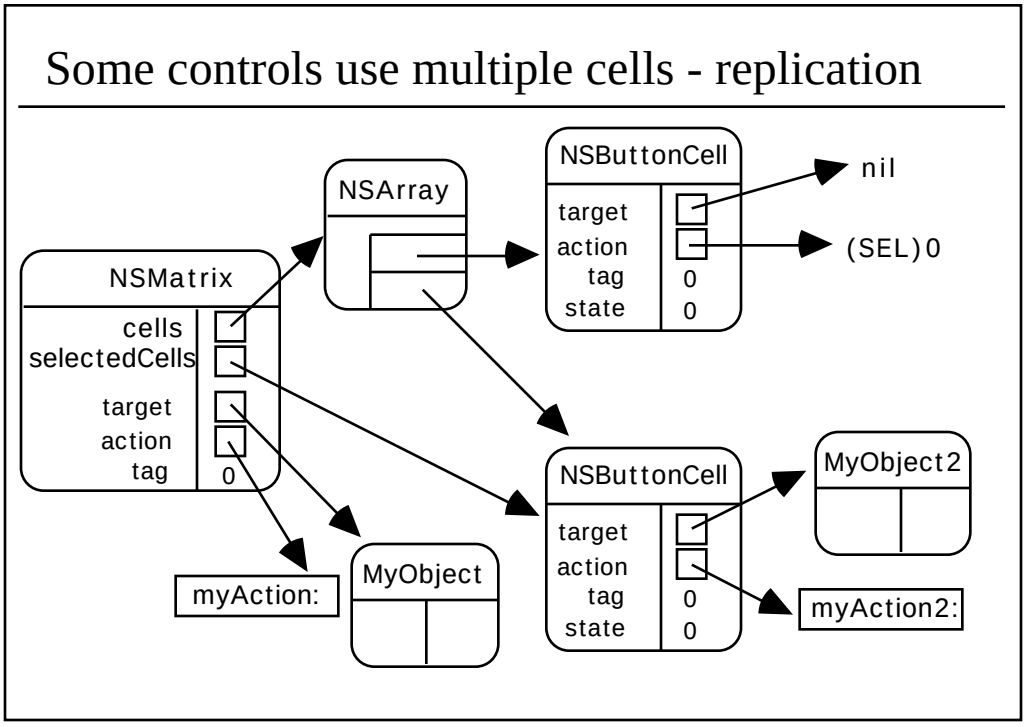
Why is this the case?



A cell is a lightweight fundamental unit

NSCell is a subclass of NSObject, not NSView. A Cell is not a View. Because of its ability to draw and handle events, NSView is a very heavy class. By contrast, NSCell is light and efficient but it requires a View to obtain the full context for displaying and receiving mouse and keyboard events.

Cells make little sense when you consider only the one-to-one mapping of NSButton/NSButtonCell or NSSlider/NSSliderCell. Cells were designed as a performance gain for the one-to-many cases where one NSControl maps to multiple NSCells.

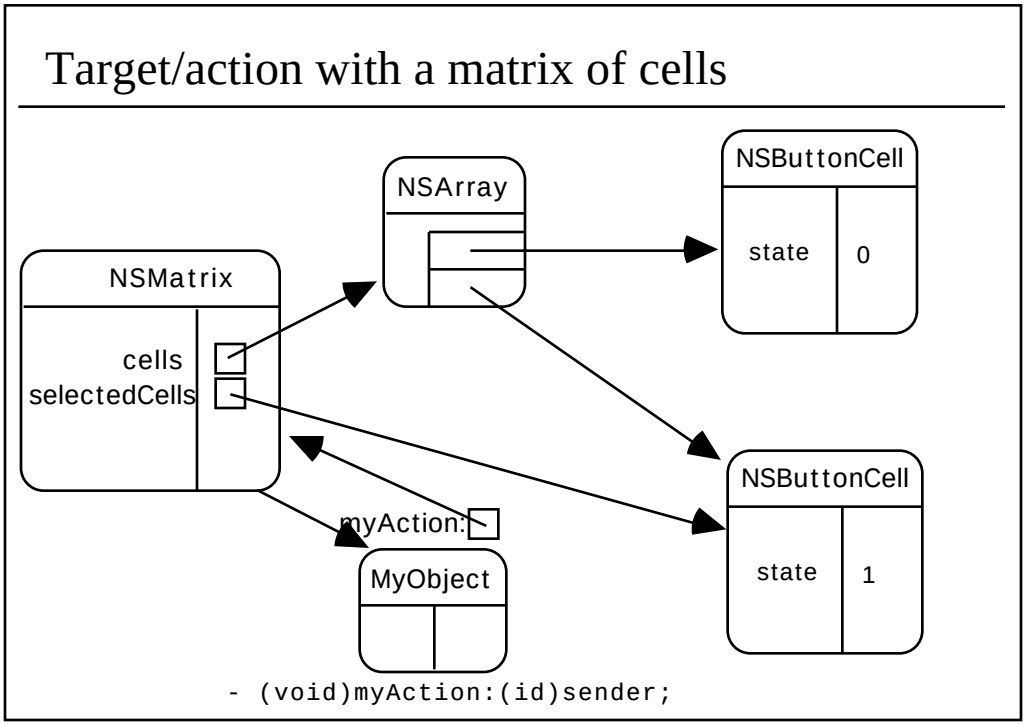


Some controls use multiple cells—replication

The motivation for Cells is most apparent if you consider a matrix of NSButtonCells. NSMatrix is an NSControl which in turn is a costly NSView. Within, the matrix can be subdivided into any number of regions, each of which must:

- Display itself as distinct from the others, e.g. image, title, state
- Claim a region in which it is responsible for any user events, e.g. mouse clicks
- Maintain separate instance variables, ideally with the possibility for each to have a separate target/action pair

The solution is to abstract these points into a separate class which can then be replicated for a collection, all within the scope of a single encompassing NSControl—in this case an NSMatrix. In the case of a control, like a matrix of radio buttons where only one can be selected at a time, it is crucial that all button cells be grouped and tracked by a single object containing them.



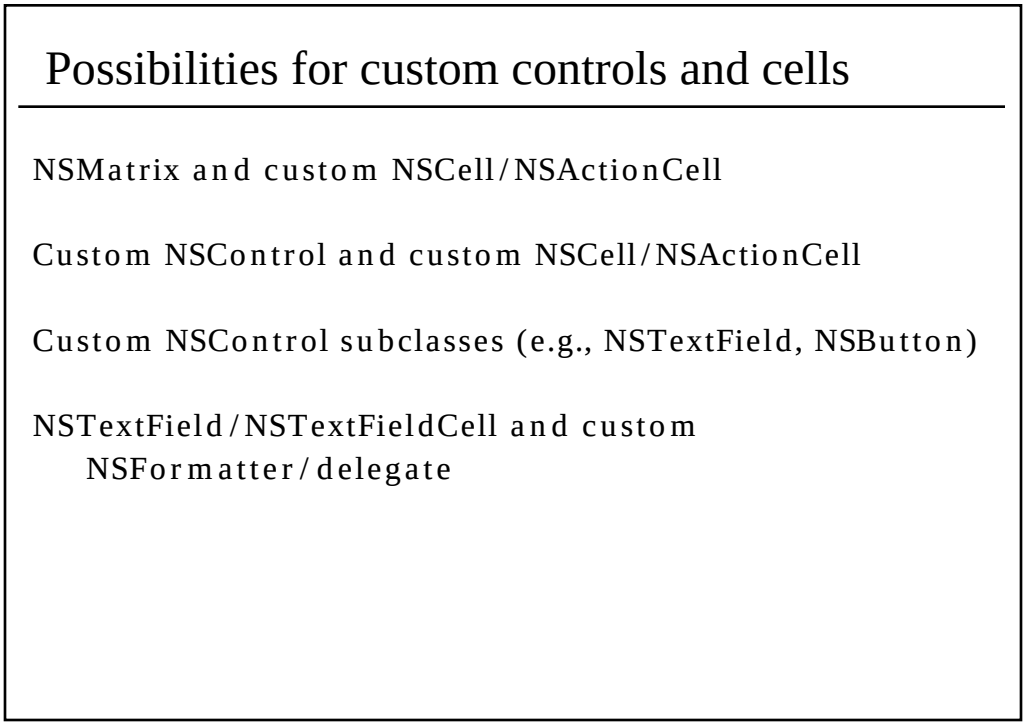
Target/action with a matrix of cells

The matrix is the Control. The matrix is therefore the **sender** of the target/action message. This will create some surprises if you think **sender** refers to a Button or ButtonCell. To determine the state of the selected NSButtonCell within the Matrix, you cannot directly ask sender:

```
[sender state];
```

This is a mistake and would probably terminate your application since NSMatrix does not respond to **state**. Rather, you must first determine the **selectedCell** within the Matrix and ask it:

```
[[sender selectedCell] state];
```



Possibilities for custom controls and cells

Given that most controls use cells, there are several different combinations for creating customized controls of your own. You might design a new cell that forms the basis for a replicated set within a matrix. You might design a new control which implements its behavior with the help of standard text or button cells. You can even use an existing control/cell arrangement, such as NSTextField, but subclass it to add specialized behavior.

Lastly, text-based controls—such as text fields, matrices, and table views—can be customized with auxiliary objects without modifying the control or cell classes directly. An NSFormatter subclass and your custom control delegate can be used to tailor such a control to meet the sophisticated needs of your application.

Important ideas from this section

» Applications are driven by the event loop

» User interfaces are implemented with nib files that contain:

    Archived object instances

    Connections among the object instances

» Application Kit objects work in conjunction with your custom classes and depend on basic interconnections:

    outlets

    target/action

    Delegation—with related notifications

» Controls are the most basic user interface components that include:

    NSButton—giving the user a trigger for an action

    NSTextField—displaying and allowing the user to input arbitrary object values

» awakeFromNib is where final initialization of the user interface should occur

» You can invoke target/action programmatically given an NSControl subclass instance

» Every NSControl subclass uses a corresponding NSCell subclass

Classes featured in this section

» NSControl

» NSButton

» NSTextField

» NSCell, NSActionCell, NSButtonCell, NSTextFieldCell

» NSMatrix

REVIEW NIBWARE AND THE BASIC APPLICATION

1. What happens when a nib file is loaded?

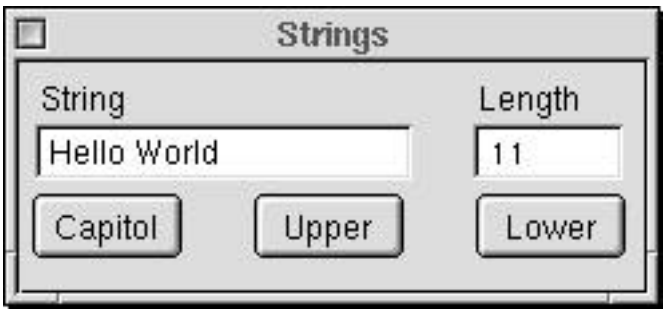
2. What's the function of a Control?

3. What happens when you send a message to nil? What is the value of an outlet that has not been connected in IB? What happens when you press a button whose target outlet has not been connected?

4. Why might you disable a button? How would you do it?

EXERCISE 9.1 A BASIC APPLICATION—STRINGS

In this exercise you build a simple application that demonstrates the target/action paradigm. The application allows the user to type in a string and displays its length in a separate field. It provides three buttons which capitalize, uppercase and lowercase the string. Use the NSString class to store and manipulate the user's string.



Objectives

After completing this exercise, you will be able to:

- Construct a user interface with Interface Builder
- Develop and build an application using Project Builder
- Connect objects in the application together using target/action
- Understand how the Model–View–Controller design pattern maps to a basic application

Exercise

1. Create a new application project called Strings. Open the interface file **Strings.nib** and construct the user interface as shown above.

2. From the classes window in Interface Builder, subclass NSObject to create a controller class for the application named **AppController**. Include outlets for the two text fields the controller object will change—the **stringField** and **lengthField**. You also need target/action methods to respond to the user interface controls. For example:

```
- (void)capitalize:(id)sender
- (void)lower:(id)sender
- (void)upper:(id)sender
- (void)stringChanged:(id)sender
```

3. Instantiate the AppController.

4. Make target/action connections in Interface Builder so the controller reacts to the buttons and changes in the text field.

5. From the classes window in Interface Builder, create the files for AppController and add them to the project. Save the interface, and switch to Project Builder.

6. Your controller needs an NSString instance to hold the current string, so add a third instance variable to **AppController.h** called **string**. NSString provides all the support you need with:

```
- (NSString *)capitalizedString;
- (NSString *)lowercaseString;
- (NSString *)uppercaseString;
```

   Check the NSString documentation for additional information.

   Hint: Because you will be modifying the string, an NSMutableString is convenient. On the other hand, you might simply use NSString, releasing the old and retaining the new each time it changes.

7. Now you can fill in the logic needed to process the user's actions. Implement the methods in **AppController.m** to handle all the actions you defined. You may want to create your NSString instance in an **init** method and provide an **awakeFromNib** method to set up the length field from this initial string value.

8. Build the project. Test it from the Project Builder launcher panel.

Enhancements

» Add code to **awakeFromNib** that makes the application window key after the nib has been loaded. You won't notice this change when launching the application from Project Builder. Launch it from the file viewer instead.

» Make sure the length text field is read-only, via Interface Builder's inspector. Use a grey background in the interface to visually reinforce this constraint.

» Change the project name to **StringApp** using the Project inspector—this is independent of the nib file name. This will change the name of the built application from **Strings.app** to **StringApp.app**. You can also change the title in the main menu and the window.

» You can send the **selectText:** message to the text field so that its string is selected and automatically edited by the user's keystrokes. Add this message to the end of each target/action method and see if you prefer this behavior.