Chapter 3

FORMATTING AND VALIDATING TEXT

CHAPTER 3 FORMATTING AND VALIDATING TEXT

Goal

To outline the role that formatters and control delegates play in managing text display and editing.

Prerequisites

Strong understanding of NSTextField in terms of target/action, object value, cell, delegate, and editing behavior.

Objectives

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

- Attach your own custom formatter to a text-based control
- Utilize NSString and NSCharacterSet to implement a formatter
- Distinguish formatting from validation and describe how NSControl and its delegate cooperate to accomplish the latter

Reading

NSFormatter class reference in the Foundation

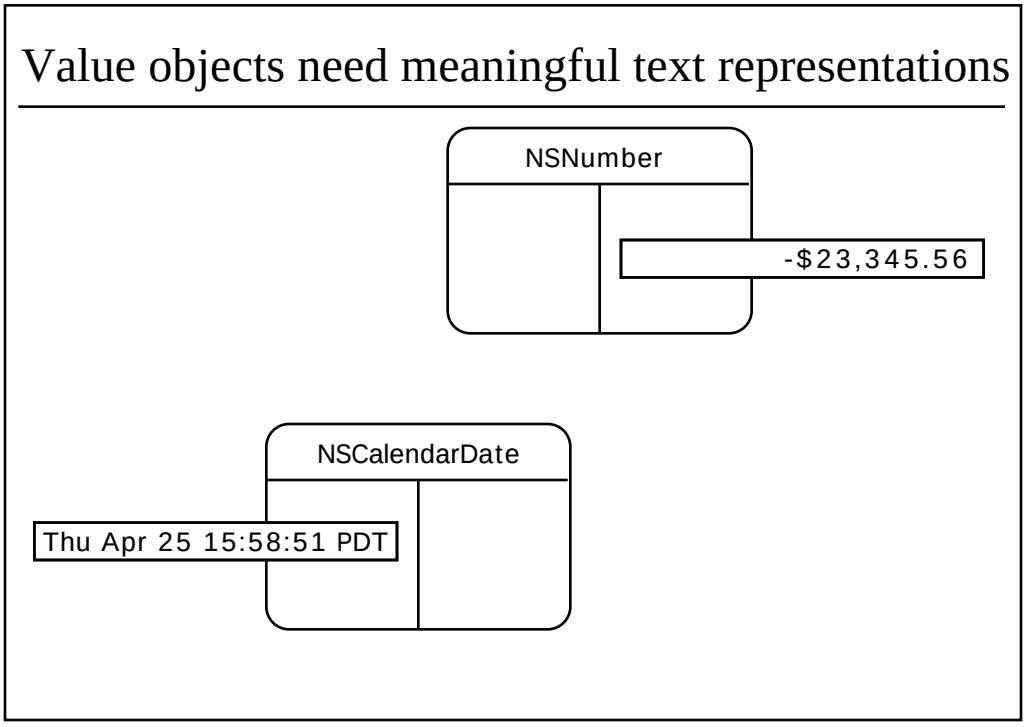
NSControl class reference in the Application Kit

NSString class reference in the Foundation

NSAttributedString class reference in the Foundation

NSCharacterSet class reference in the Foundation

NSScanner class reference in the Foundation

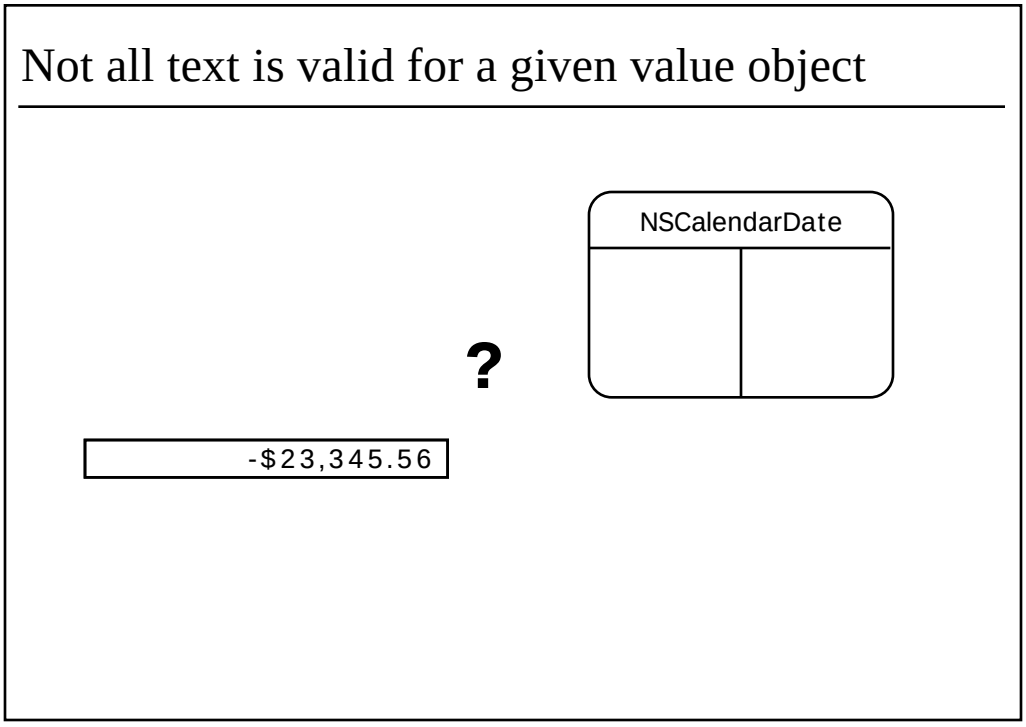


Value objects need meaningful text representations

NSTextField can take arbitrary value objects and display them as strings. A value object an afford the service of providing its value as a simple string object by implementing the **description** method. It is easy to connect objects and display vanilla string values.

Good user interfaces should make a larger step towards the sophistication of the user driving them. Personal preferences, cultural and language differences, custom business logic and procedure, and greater ease and clarity demand more meaningful text representations. Money should have a currency symbol, perhaps green, or red and black for debt or credit. Dates can have a huge variety of formats that select the few fields of interest, change the order, the month and day names, the manner of telling clock time. The possibilities are limitless.

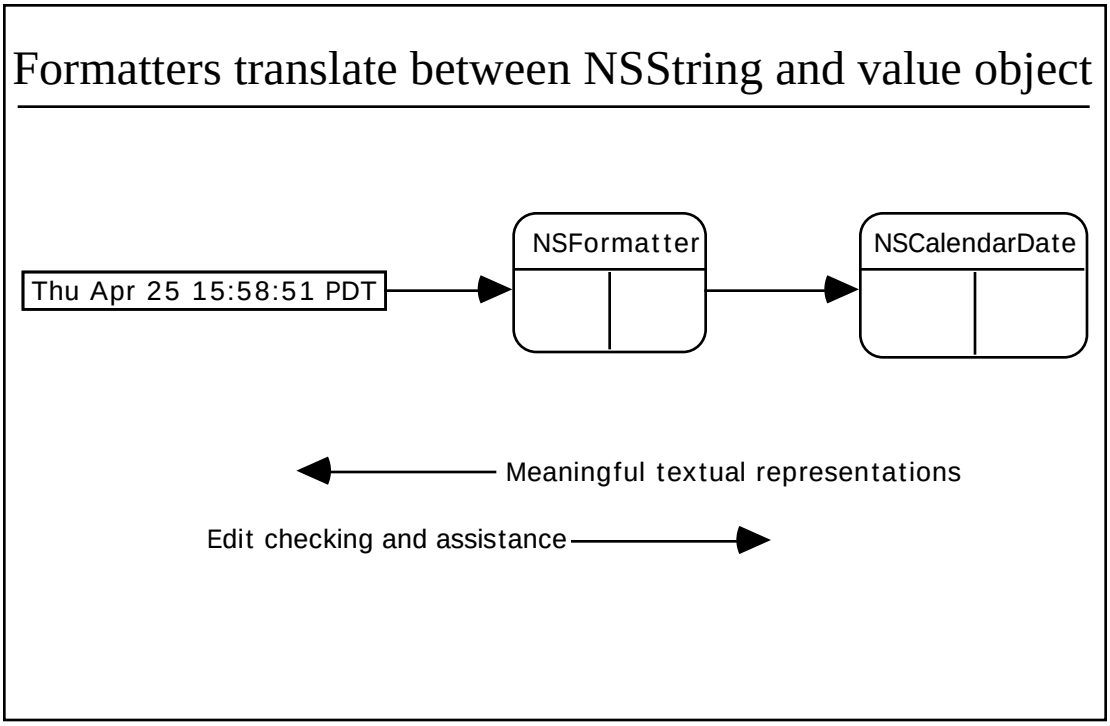
NSTextField, a powerful and generally reusable text machine should not be cluttered with a myriad of diverse value objects and the way they should be displayed. The value object may provide some efficient means of customizing its appearance but typically on the character values themselves, not arbitrary graphical elements like color values and fonts. A value object does not want to interface with a graphical Application Kit object any more that handing out its value when asked.



Not all text is valid for a give value object

During text input, a user has full rein to type in arbitrary characters and perform random edits. Mistakes are likely. The range and shape of a valid text value may not be readily apparent. Someone needs to check the user's entry and verify that it maps to a valid object value. It would be nice if someone could also automate some data entry chores like automatically adding dashes for phone or social security numbers.

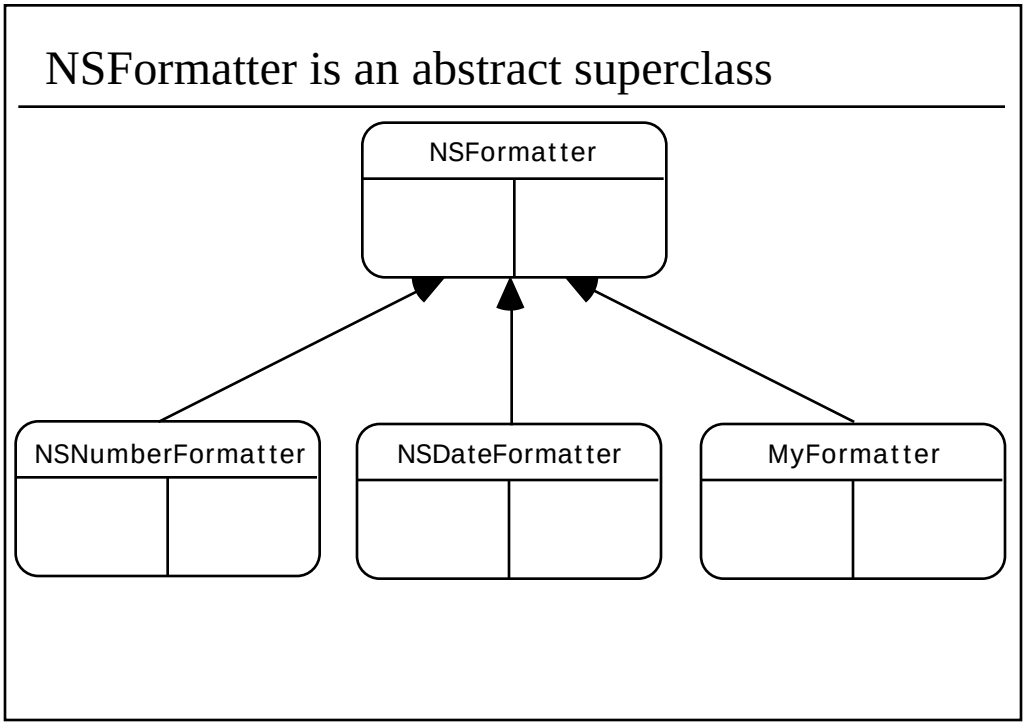
Once again, good design suggests that the generic NSTextField object should not be overly burdened with elaborate and specialized tasks like these. And it is clear that our value objects should be small, efficient, and completely unconcerned with the complexities of the user interface mechanism.



A formatter translates between NSString and value object

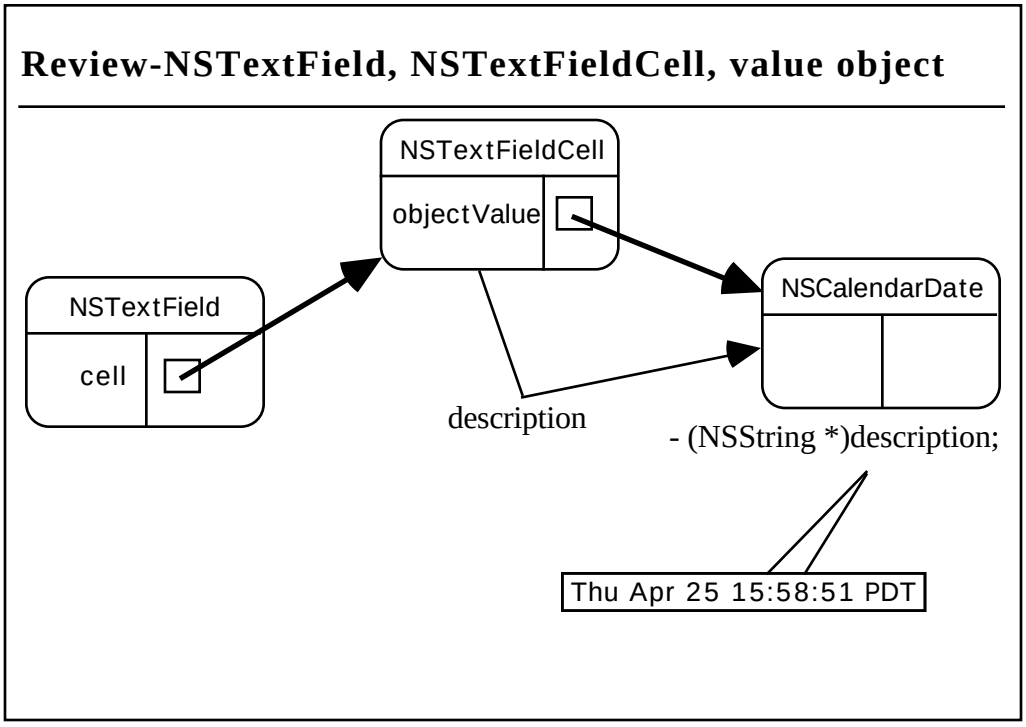
The solution is a third object. It is quite valid to view this design pattern as yet another instance of Model-View-Controller and, perhaps at the most fundamental level, a single value object as the model is exposed to the users view through the narrow view of a text field. The formatter is essentially a controller. It controls how the model should be displayed and how user text should be reflected back in the model.

- Model—value object, in this case, NSCalendarDate
- View—NSTextField
- Controller—NSFormatter subclass



NSFormatter is an abstract superclass

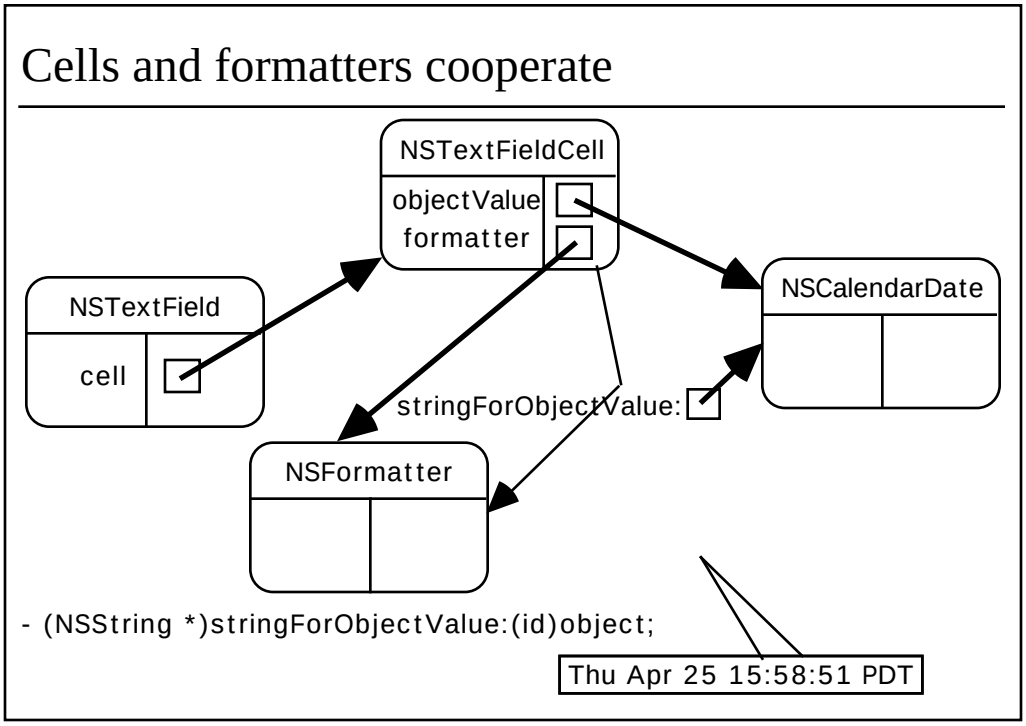
With a nicely designed abstract superclass, implementing a custom formatter subclass is straightforward.The Foundation provides two concrete subclasses of NSFormatter: NSNumberFormatter and NSDateFormatter. These work with NSNumber and NSDate value objects with a rich array of configuration options. You can easily design your own formatters whether they are non-configurable “filters” that do one job well or are configurable, generalized formatters, reusable in the finest sense of the word.



Review—NSText Field, NSTextFieldCell, value object

To clearly see how a formatter works, remember the key objects involved:

- NSCalendarDate—the model. It can return a string representation of this value with **description.**
- NSTextField—the view. This is an NSControl that utilizes a corresponding NSCell subclass.
- NSTextFieldCell—the meat of the text-based control. It contains an outlet for the value object. For display, it gets a string representation of the object by sending it the **description** message.



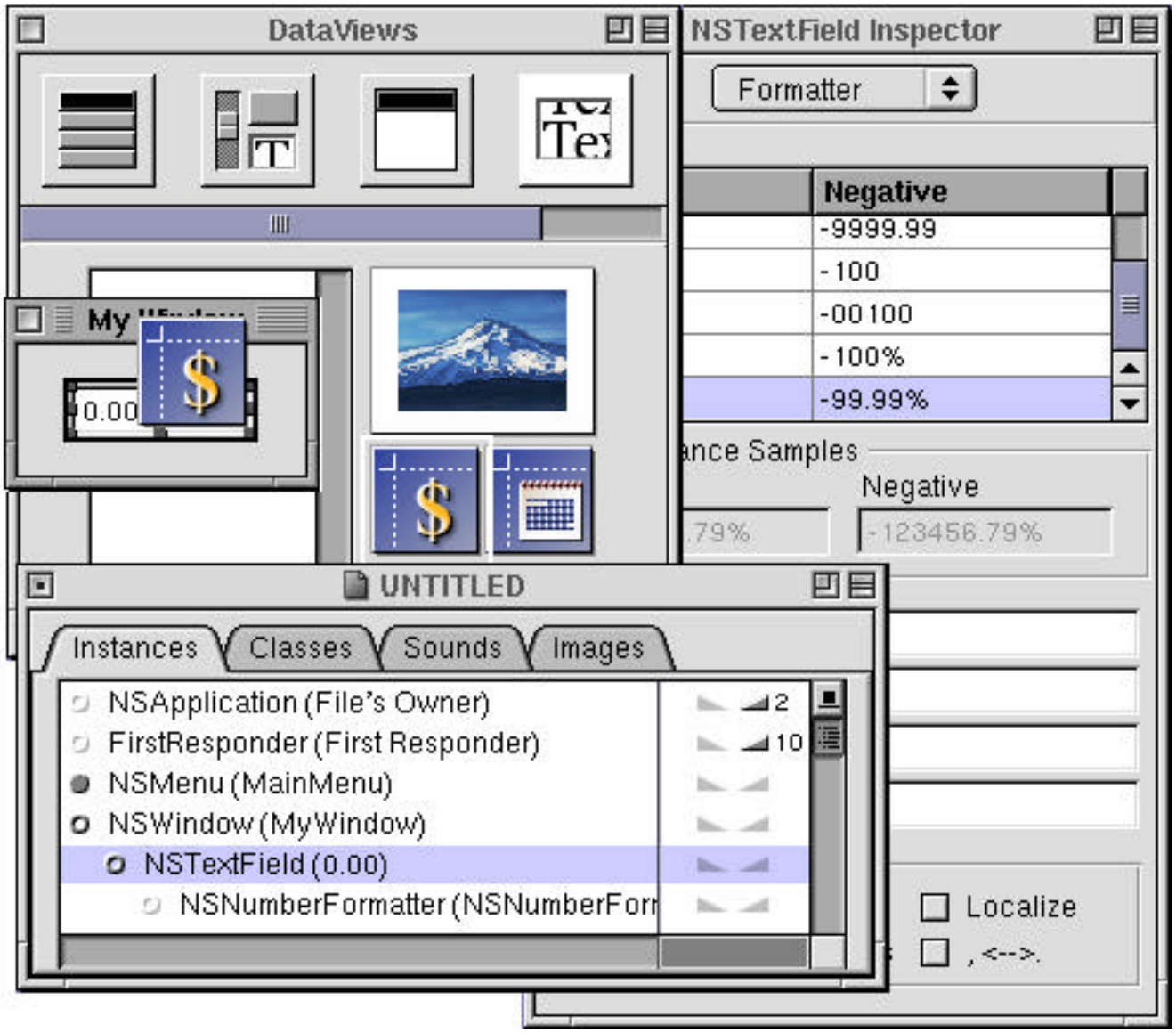
Cells and formatters cooperate

NSCell has a formatter outlet. If a formatter is available, it is used, instead of the value object's **description** method, to manage the display and editing of the attached value object.

When necessary, NSCell asks the formatter instance for a string that represents the value of the object passed as an argument. The formatter maintains a simple interface with the NSCell—return an NSString instance. It is decoupled from the cell's view or control container so that it is not relevant whether the cell is in a text field, a table view or any manner of custom view or control subclass. On the other hand, the formatter instance is intimately familiar with the value object. It is likely designed with NSCalendarDate specifically in mind.

By design, the NSCell and NSCalendarDate objects remain unchanged, pure and general in their purpose. Any one of a wide range of different custom formatters could be applied as a controller, mediating between the model and its projection on to a view.

For more elaborate display features, a formatter can implement **(NSAttributedString *) attributedStringForObjectValue:**. Attributed strings extend NSString objects by associating display attributes such as color and font, potentially down to the level of individual characters. If a formatter responds to this message, when possible, it will be used rather than **stringForObjectValue**.



Attaching a formatter

Given the implementation of a custom NSFormatter subclass, how can you attach it to a given NSCell? If the NSFormatter is properly palettized, provided to Interface Builder via a custom palette bundle, it can be instantiated and attached in the familiar manner. Drag the formatter over the interface component and drop. Note:

- NSMatrix—Once you have attached a formatter to a TextField, you can expand it into a matrix of TextFieldCells with the expected behavior: each TextFieldCell is cloned from the original and is attached to its own cloned formatter instance. For an existing NSMatrix, you can attach the formatter to one of the cells then configure it as the cell prototype.
- NSTableView—Each column in a table view has a column header that works in much the same way. You can attach a formatter instance by dragging and dropping over the header cell and it will be instantiated and attached to the prototype data cell for that column.

Using Interface Builder's outline mode, you can see which formatter is attached. The inspector popup button will include a "Formatter" item which switches the panel view to the custom formatter inspector if this particular formatter class provides one.

In lieu of a palette, you can attach a formatter programmatically. For an NSTextField instance:

```
[[textField cell] setFormatter: formatter];
```

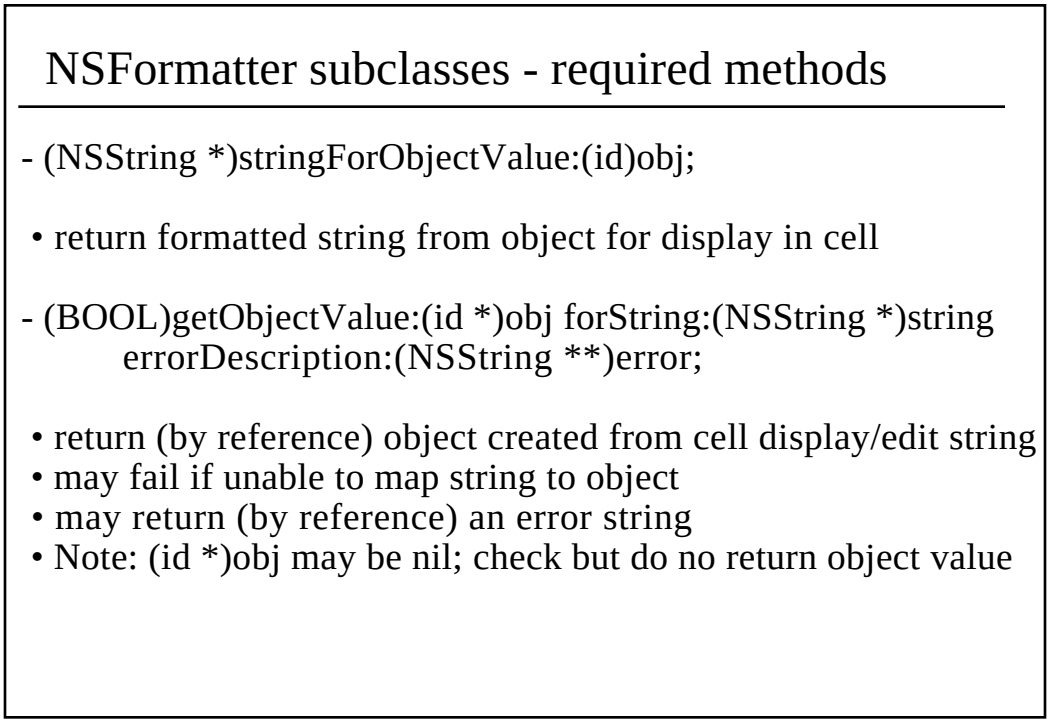
For a particular column in an NSTableView:

```
[[[tableView columnWithIdentifier: @”date”] dataCell]
   setFormatter: formatter]];
```

And for an NSMatrix of cells:

```
[[matrix prototype] setFormatter: formatter];
```

How is the formatter instantiated? You can programmatically instantiate with **alloc** and **init** or you can instantiate in Interface Builder and connect a controller outlet to it for future reference.



NSFormatter subclasses—required methods

NSFormatters get involved in both directions of data transfer—providing a displable string from the object value and creating a new value object instance from a given display string. Since **getObjectValue:forString:serrorDescription:** returns a boolean value, whether or not the formatter was able to create a valid object value. The other two parameters must be returned by reference:

» Object value—return a pointer to a pointer to the new value object instance. If this method fails, because it cannot provide a value object, return the pointer passed in, which will be nil.

» Error description—a pointer to a pointer to an NSString instance that describes the failure to deliver an object value. If this method fails, but provides no error description string, return the pointer passed in, which will be nil.

What happens when this method fails? The cell flags it objectValue as invalid with two consequences:

» The user cannot leave the cell. The control refuses to reliquish first responder status.

» **objectValue** for the containing control returns nil.

Both methods must be implemented by all NSFormatter subclasses to be functional.

Dynamic editing - character by character intervention

```
- (BOOL)isPartialStringValid:(NSString *)partialString
        newEditingString:(NSString **)newString
         errorDescription:(NSString **)error;
```

- called after every keystroke, before cell displays it
- if NO, keystroke does not appear
- if newString (by reference), it is displayed instead

Dynamic editing—character by character intervention

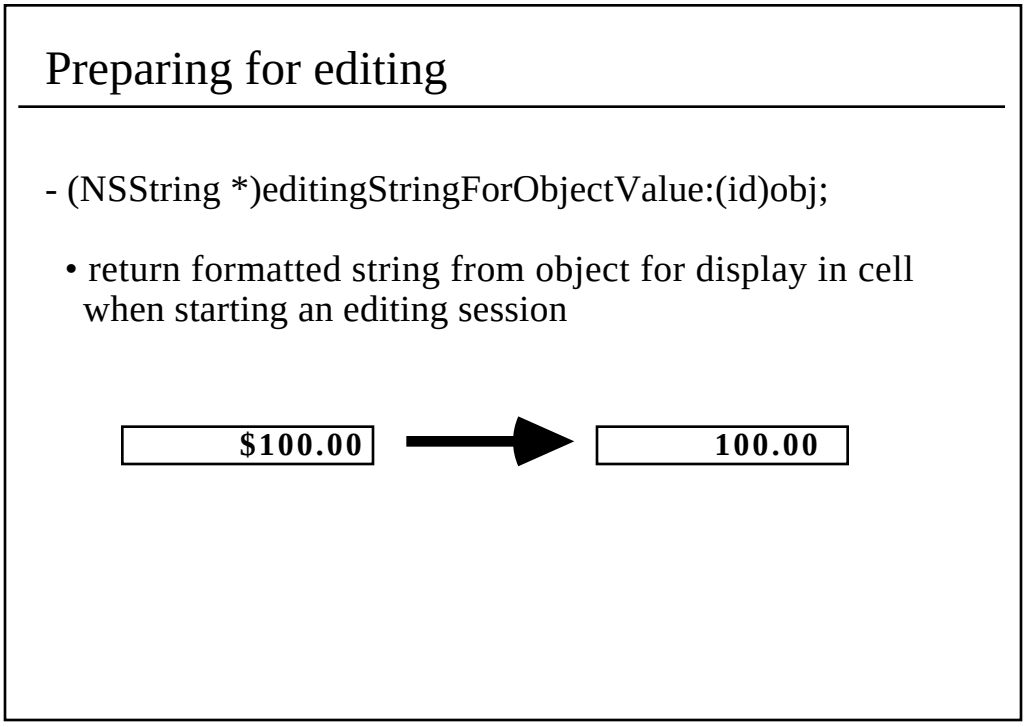
Your formatter can monitor each keystroke applied to the cell during user editing. After each event, but before the results are displayed in the cell, your formatter will be passed a partial string. Your formatter has the following options:

» Accept the string. Simply return YES.

» Reject the string. Return NO. The user edit will not be accepted and the cell will remain as though the user typed nothing. Optionally, return by reference an error string describing in a terse manner why the string was failed—used by the control's delegate.

» Accept the string but modify it. Return YES and pass a new editing string back by reference. This allows the formatter to automatically insert formatting characters, such as dashes in phone numbers, to save the user from entering them manually. Another example is simply formatting on the fly, for example, capitalizing a character as soon as it is typed.

Note, with pasting and flexible keyboard editing, user edits can happen anywhere in the string, not just at the end. Your formatter should check the entire partial string with each invocation of this method.



Preparing for editing

A formatter can implement an optional interface for providing an initial edit string, again, derived from a given value object. The allows a formatter to distinguish the display string, which might include extra characters such as a currency sign or punctuation, from an edit version of the value, typically the most vanilla representation of the value. A user will ideally want to input only the essentials during editing. The formatter can get fancy when it's time to display the value with **stringForObjectValue:**.

Additional optional formatter methods

- (id)defaultObjectValue;

- return a default instance of formatter-specific object value class
- by Interface Builder when formatter is attached to a cell

- (NSAttributedString *)attributedStringForObjectValue:(id)object
withDefaultAttributes:(NSDictionary *)attributes

- if the formatter responds to it, used instead of stringForObjectValue
- can associate attributes like color and font for special effect

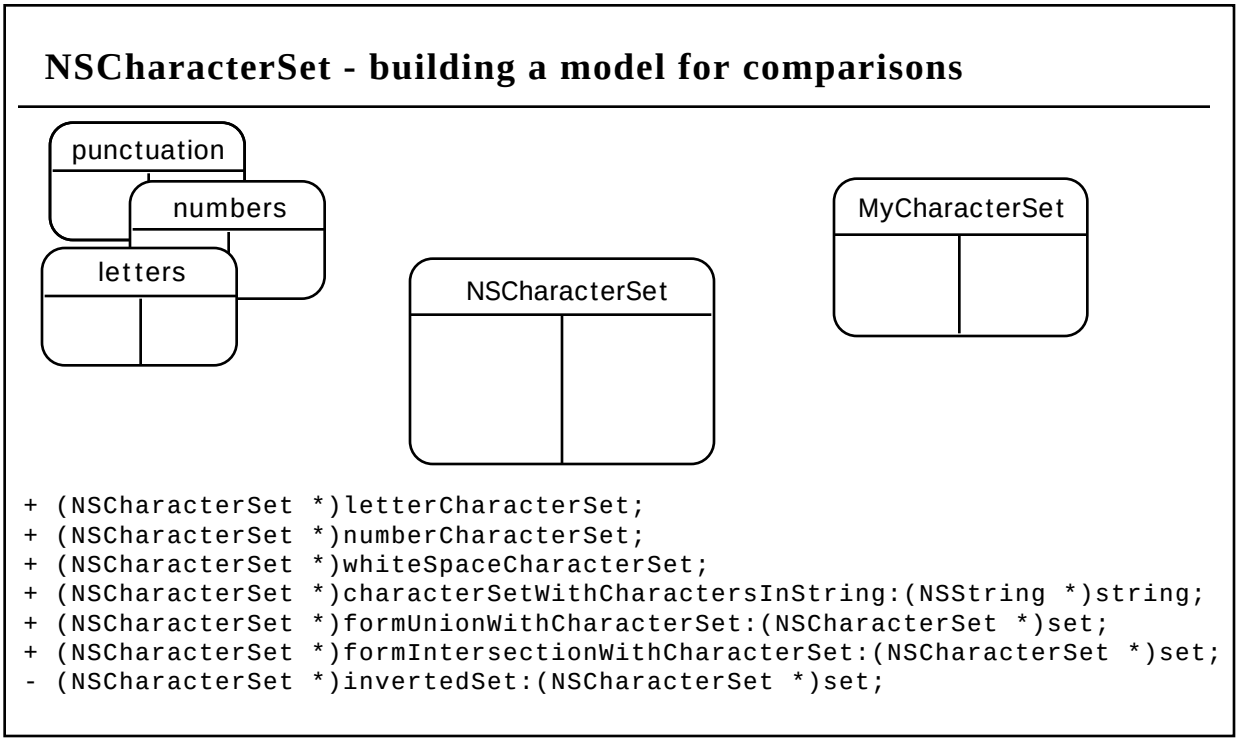
Additional optional formatter methods

```
- (id)defaultObjectValue
```

If your formatter is palettized, Interface Builder uses this method when necessary. When your formatter is first dragged and dropped on a cell in the user interface, it may be that the string value already there is incompatible with your formatter's value object type. The formatter will be asked for a default and should return an appropriate value object instance based on reasonable default values.

```
- (NSAttributedString*)attributedStringForObjectValue:
(id)object withDefaultAttributes:(NSDictionary *)attributes
```

If your formatter responds to this selector, this method will be used instead of **stringForObjectValue:**. NSAttributedString combines a plain NSString with a set of attributes that associate ranges of the string with special fonts, colors and other text related features. A number formatter might choose to represent a negative number in red.



NSCharacterSet—building a model for comparisons

Formatters need some skill with handling strings. A common task is testing a string or partial string against a reference set of characters. If a formatter allows only numbers, it will want to check the string against a set of valid number characters to see that nothing outside of this set is allowed.

NSCharacterSet is a Foundation class that objectifies an arbitrary set of characters. It provides access to several commonly used instances such as the set of all letters, numbers or punctuation symbols. It provides methods that allow you to customize a set by adding or deleting characters in a number of ways. You can also perform set operations such as union and intersection, creating a new set instance out of old ones. Once you have a set that matches what is valid, you can generate the inversion of the set. This is useful for finding invalid characters by asking, “Are there any characters in the string that belong to this character set?”

Like many value and collection classes in the Foundation, NSCharacterSet is immutable and provides a mutable subclass, NSMutableCharacterSet. There are methods to convert between the two forms and some methods apply only to mutable instances. Like most Foundation objects, NSCharacterSets can be archived, unarchived and copied. They conform to NSCoding, NSCopying, and NSMutableCopying.

Useful NSString methods

Basic
  length, cString, stringWithCString:, stringWithFormat:

Extraction
  characterAtIndex:, substringWithRange:

Formatting
  uppercaseString, lowercaseString, capitalizedString

Comparing
  compare:, rangeOfCharacterFromSet:

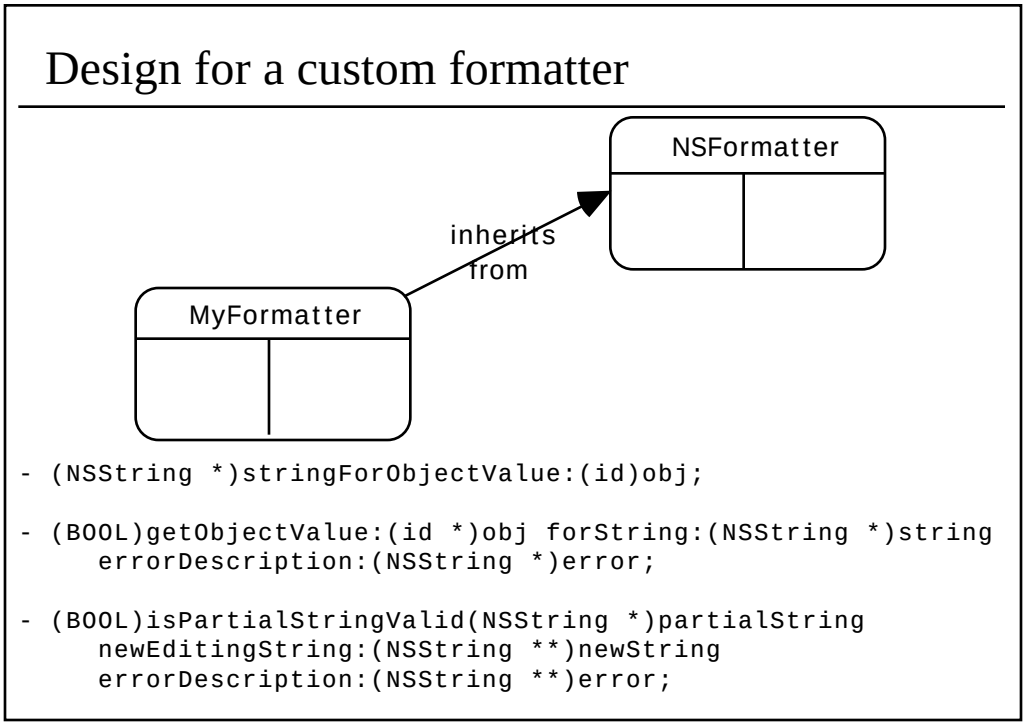
Building
  stringByAppendingFormat:(NSString *)format, ...;

Useful NSString methods

The heart of the formatter's skill is its ability to adroitly handle NSString instances. In the object-oriented world,- NSString is fortunately quite adept at manipulating itself. A formatter merely has to message the string to do so. NSString offers sizable API for a myriad of operations. These are some common ones organized into basic functional areas:

- Basic—length is useful for checking empty or fixed length values. You can obtain a primitive C char * if you need to operate using traditional pointers, macros or functions. Note, you may need to convert back to an NSString if you a building a new string.
- Extraction—you can extract a character or substring of characters.
- Formatting—A string can produce different formatted instances of itself.
- Comparing—comparing two strings or for comparing a string against a character string.
- Building—for editing assistance in particular, a formatter will simply want to append to the partial string and return the new instance.

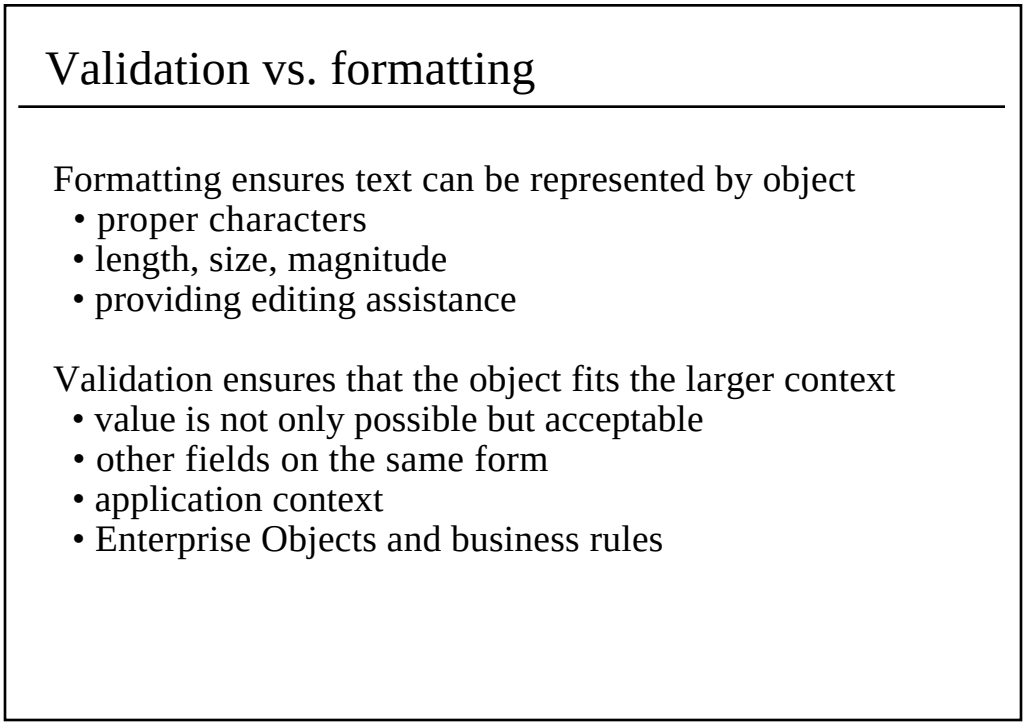
There are numerous other possible operations involving NSString, NSCharacterSet, NSRange and possibly NSScanner. This is enough to get you thinking in the right direction.



Design for a custom formatter

This is the standard design for a full-featured custom formatter. It includes both required methods and includes partial string validation as a hook for following each user edit action. This might be the complete interface for even a powerful formatter but it does not include API for parameterized features. If you design a more general configurable formatter, you would need to add accessor methods for getting and setting options.

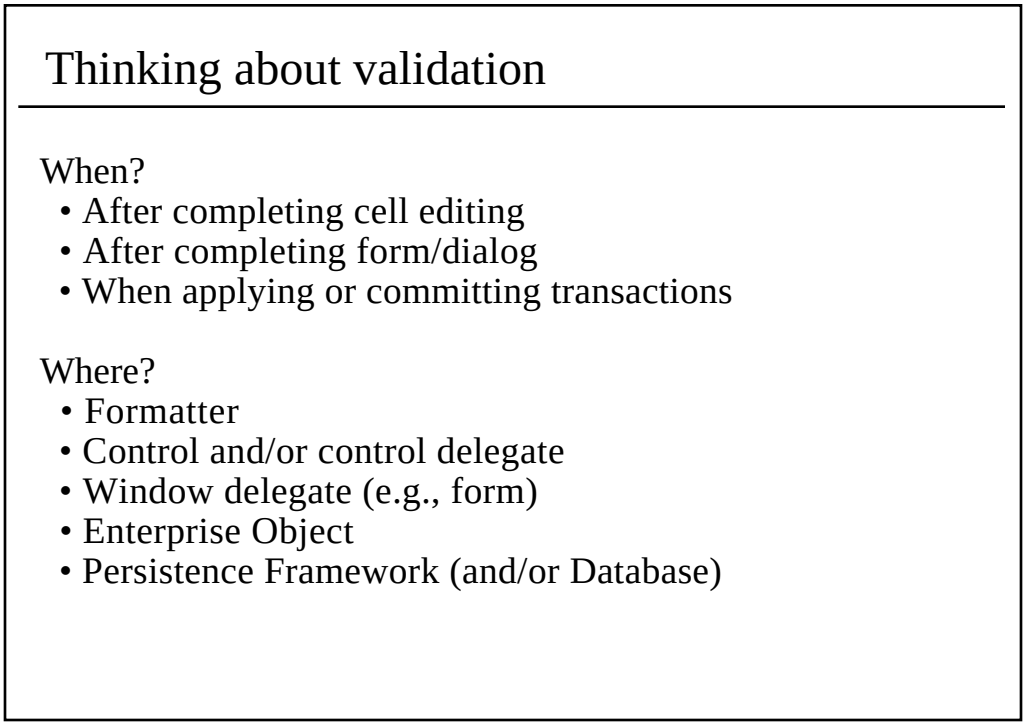
See the section on Interface Builder Palettes to explore providing a fully implemented NSFormatter subclass with the convenience of graphical instantiation and attribute inspection.



Validation vs. formatting

While a formatter is likely to perform some basic checking and even refusing certain edits or entire string representations, it is not necessarily the place for more profound and context-sensitive sorts of validation. A formatter handles basic aspects of a value type—a number cannot have the letter “a” and a date is likely to need a year specification. It is fundamentally concerned with the value object and its view only, not the larger context of the window, the application or the underlying enterprise data model. A formatter is not likely to worry about whether a number meets a budgetary limit, whether it makes sense in the context of the other numbers on the window nor whether the number works as an attribute in a particular model in a particular data base table. Likewise, error reporting, warnings, suggestions and interactive dialogs with the user are probably not in the domain of the formatter either.

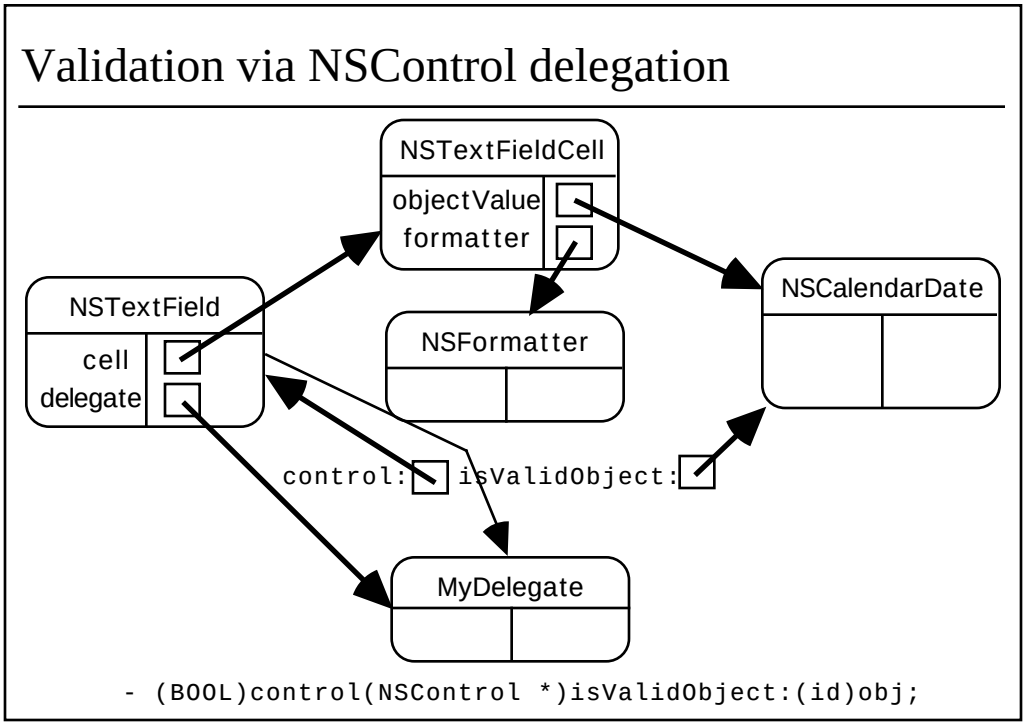
While there is great flexibility in design and the boundaries between objects are highly negotiable relative to a particular design situation, it is common to place this context sensitive and possibly interactive validation in a place other than the formatter.



Thinking about validation

When does such validation typically take place? Maybe just after the individual text value is edited, maybe not until several fields are completed and the user presses “OK”, and possibly not until a business object instance is written to a data base. While it is ideal to catch user input problems as early as possible and even avoid them with proper user interface control and feedback, it is not desirable to embed too much business logic in the user interface.

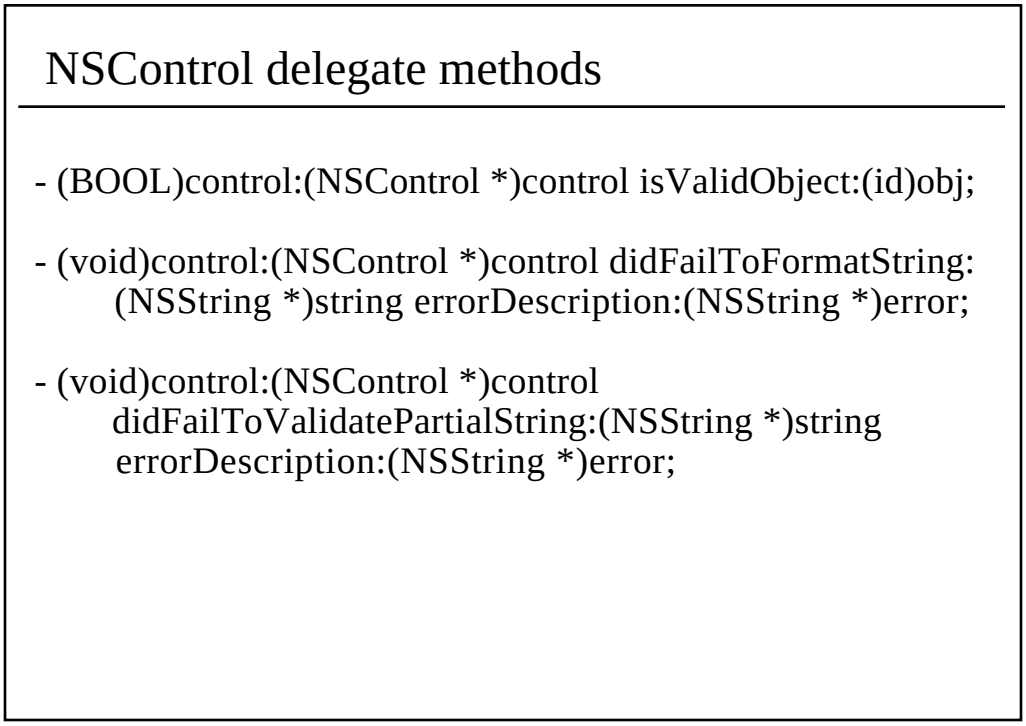
Validation can therefore belong in one or more of several different objects. How close to the user’s editing context, how reusable the design needs to be, how quickly and frequently will the rules change, how much logic is already implemented in the model and underlying data store—these questions should be considered when thinking about the right approach to validation. Your application and business context is likely to suggest a particular approach.



Validation via NSControl delegation

Many controls use delegates and provide API for validation there. Once a formatter has allowed a string to become a proper value object, the control's delegate is given a chance. The delegate can ask some of these deeper questions, using a larger application context such as taking other cell and control values into account, and it can provide the more complex user interface interactions required when the object is invalid.

The control delegate can also find out about strings that did not make it past the NSFormatter for one reason or another.The delegate can be notified and passed the error description strings provided by the formatter.



NSControl delegate methods

To fully participate, a control delegate can implement the following as necessary:

```
- (BOOL)control:(NSControl *)control isValidObject:(id)obj
```

The control's delegate gets the final chance to validate the object. If present, the formatter was able to represent the string as a value object. But does the object make sense in the current context of the application?

```
- (BOOL)control:(NSControl *)control
didFailToFormatString:(NSString *)string
errorDescription:(NSString *)error;

- (void)control:(NSControl *)control
didFailToValidatePartialString:(NSString *)string
errorDescription:(NSString *)error;
```

Both methods result from failures in the associated formatter. The control is now free to perform more elaborate user interface interaction to communicate and possibly rectify the unsuccessful edit operation.

These messages are available from any NSControl that uses one or more text-based cells and a delegate. Examples include NSTextField, NSMatrix and NSTableView.

Important ideas from this section

» NSFormatter subclasses can manage the mapping between value objects and their textual representation in the user interface.

» Your NSFormatter concrete subclass must implement two methods for mapping between the a cell's object value and the display string and may optionally provide special editing strings, attributed strings and perform partial string validation.

» NSString and NSCharacterSet provide a foundation for manipulating strings and making comparisons.

» More elaborate and context-sensitive validation of a user-edited value object is commonly placed in the control delegate, your custom class, leaving the formatter free to focus on its well-defined and more reusable role.

Classes featured in this section

» NSFormatter

» NSCharacterSet

» NSString

» NSAttributedString

» NSScanner

» NSControl

REVIEW FORMATTING AND VALIDATING TEXT

1. What are some of the differences between formatting and validation?

2. How does a formatter relate to a Model-View-Controller model?

3. How can a delegate help a control perform validation?

4. List some useful formatters you might want to implement?

EXERCISE 3.1 USING AND IMPLEMENTING FORMATTERS

There a few basic formatters that you will examine in this exercise. With NSFormatter, it also provides an easy way to create your own custom formatters. In the Expense Report application, each expense item contains a category, a date and an amount. The latter two can be dealt with by NSDateFormatter and NSNumberFormatter. You might require that the category be a non-empty capitalized string without whitespace—this is a perfect opportunity to write your own formatter. This exercise extends the previous chapter's Expense Report application by adding all three formatters.

Objectives

After completing this exercise, you will be able to:

- Use formatters to format dates and numbers for input and output
- Subclass NSFormatter to provide specialized formatters for other applications
- Programmatically set formatters for Application Kit objects

Exercise—Stage 1

1. The Expense Report application is a perfect context for testing and developing formatters. Make a backup copy of the previous exercise before moving on.

2. Open the Document nib. Adding formatters to the current expense report is straightforward for the Date and Amount columns, if you haven't already done so in an earlier exercise:

   - Select the Text palette from Interface Builder's palette window.
   - Drag a Date formatter onto the Date column header in your Document nib table view.
   - Drag a Money formatter on to the Amount column header.
   - If you have a Total field, drag a Money formatter on to that as well.
   - Configure the attributes for each formatter using Interface Builder's inspector. Once a formatter has been attached to a cell within a control, the inspector popup button contains a Formatter item.
   - Because the formatters are palettized, you can test the formatters in Interface Builder test mode. Build the application and check that the formatters work as expected.

Stage 2

1. Now consider how to provide a formatter for the Category column.There are skeleton files to assist you. Copy **AlphaFormatter.m** from the **SuppExercises/Provided/Supp_3.1_Files** area into the Classes suitcase of your Document subproject.

   » Implement a formatter that accepts only alphabetic characters and formatters the result to a capitalized string. The formatter should prohibit empty values—strings of length 0. A simple way to do this is to use an inverted character set to check the string for characters the formatter should not allow. Look over the NSFormatter, NSString and NSCharacterSet documentation. You may wish to look at the NSScanner class.

   » Remember to allow partial strings of zero length since the user can clear the cell while editing.

   » Decide whether you want to convert to a capitalized string as the user types or when they press return.

2. Your formatter isn't palettized yet so you will have to connect it to the table view category column programmatically in the DocController. You will find a useful code snippet in **SuppExercises/Provided/Supp_3.1_Files/ tableViewAttach.m**. You may have to customize it for your application.

3. Once you have made all your changes, build the application and try it out. Enter various characters and check that they produce the intended behavior. Think of a reasonable set of test cases to verify your formatter's functionality.

Enhancements

- Since you are using this formatter for the Category column of the table view, it makes sense to also use it for the Default Category text field, or form cell, on your preferences panel. Attach an instance of the formatter there as well.

- Make your formatter configurable. As a first step, make capitalization configurable. To implement this, add two accessor methods for getting and setting this boolean switch. This and any additional options have to be set programmatically as you cannot yet do it using Interface Builder. See the header file for a suggestion of a possible interface for this configurable alpha formatter.

- It's possible to have the control respond a formatter conversion failure. Check the NSFormatter and NSControl documentation to see how, and implement a control delegate with a method that calls NSBeep() if the user types an invalid character.