Data Formatting Guide

Data Management: Strings, Text, & Fonts

2009-08-06

Contents

Introduction to Data Formatting Programming Guide For Cocoa

This document describes how to use formatter objects to create and interpret formatted strings. The Foundation framework provides `NSDateFormatter` and `NSNumberFormatter` classes and the Core Foundation framework provides CFDateFormatter and CFNumberFormatter opaque types, but in contrast to many other similarly-named classes and types, they are not toll-free bridged. In Cocoa, you can use the formatters to interpret and create strings that represent other data types, and to validate the text in text fields and other cells. You can also extend an abstract class to create your own formatter.

Who Should Read This Document

You should read this document to gain an understanding of how to use formatters in Cocoa, and to understand the difference in the behavior of formatters between Mac OS X versions 10.0 to 10.3 and from 10.4 onwards.

Organization of This Document

The following articles describe the role of formatters, how they work, and how you can configure them:

- "Formatters" (page 7) discusses how formatters work in general.
- "Date Formatters" (page 13) describes how to use date formatters on Mac OS X version 10.4 and later.
- "Number Formatters" (page 9) describes how to use number formatters on Mac OS X version 10.4 and later.
- "Formatters and User Interface Elements" (page 17) describes how to set a formatter for a user interface element, and the interaction between an element and its formatter.
- "Date and Number Formatters on Mac OS X v10.0 to 10.3" (page 19) describes how to create and use date and number formatters using Mac OS X versions 10.0 to 10.3.
- "Creating a Custom Formatter" (page 27) outlines how to create custom formatter classes.

Formatters

Formatters define a common interface for creating, interpreting, and validating the textual representation of objects. The Foundation framework provides two concrete subclasses of `NSFormatter` to generate these objects: `NSNumberFormatter` and `NSDateFormatter`. The Core Foundation provides two equivalent opaque types: CFNumberFormatter and CFDateFormatter. The formatter objects in Foundation and Core Foundation are similar but are not toll-free bridged.

In Cocoa, user interface cells that display text but have an arbitrary object as their content can use formatters for both input and output. When a cell is displayed, the cell converts an arbitrary object to a textual representation. How a cell displays the object depends on whether or not the cell has an associated formatter. If a cell has no formatter, the cell displays its content by using the localized representation of the object. If the cell has a formatter, the cell obtains a formatted string from the formatter. When the user enters text into a cell, the cell converts the text to the underlying object using its formatter.

Formatters

Number Formatters

This article describes how you use number formatters using Mac OS X v10.4 and later and iOS. If you are using Mac OS X v10.3 or earlier, you should read "Date and Number Formatters on Mac OS X v10.0 to 10.3" (page 19).

Behavior Modes

On Mac OS X desktop version 10.4 and later, instances of an `NSNumberFormatter` can operate in two modes, 10.0 compatibility mode and 10.4 mode.

> **iOS:** The v10.0 compatibility mode is not available on iOS—only the 10.4 mode is available.

- In the v10.0 compatibility mode, `NSNumberFormatter` operates as it did in Mac OS X from version 10.0 to 10.3, including the limitations and any outstanding bugs—see "Date and Number Formatters on Mac OS X v10.0 to 10.3" (page 19).

- On Mac OS X v10.4 and later, the behavior of the `NSNumberFormatter` class is based on CFNumberFormatter, which is in turn based on the open-source ICU (International Components for Unicode) library. (`NSNumberFormatter` and CFNumberFormatter, however, are not toll-free bridged.) The v10.4 behavior mode allows more configurability and better localization.

The v10.4 behavior mode allows more configurability and better localization; you should use it for any new projects, and ideally upgrade any existing code that uses v10.0 behavior to use v10.4 behavior.

For backwards binary compatibility, the default behavior for `NSNumberFormatter` in Mac OS X v10.4 is the v10.0 behavior. On Mac OS X version 10.5 and later, however, the default is the v10.4 behavior. You can set the default behavior of all instances of `NSNumberFormatter` to the v10.4 behavior by invoking the class method, `setDefaultFormatterBehavior:` with the argument `NSNumberFormatterBehavior10_4`. You can also set the behavior for any instance individually by invoking `setFormatterBehavior:`.

You can set an application default (preference) to cause number formatters to be automatically converted to the new style. Set the `NSMakeNumberFormatters10_4` default to a Boolean `YES`/`true` value in the application's preferences. You must set the preference before any use is made of the `NSNumberFormatter` class. Note that the default has two effects:

1. Number formatters you initialize with `init` adopt the v10.4 formatter behavior.

2. Number formatters that are unarchived from either non-keyed or keyed archives are converted to v10.4-style if the archived formatter has an un-customized format string at unarchive time.

The object type for number formatters that use the v10.0 behavior mode is still `NSDecimalNumber`, but for 10.4-style formatters it is `NSNumber`. If you want, you can configure a 10.4-style formatter to generate instances of `NSDecimalNumber` using the method `setGeneratesDecimalNumbers:` (with the argument, `YES`). You are encouraged, however, to use `NSNumber` for new style formatters.

The v10.4 methods do not do anything when invoked on a v10.0-style formatter, and return a generic return value when required to return something. You should not invoke the v10.4 methods on a v10.0-style formatter. On a v10.4-style formatter, the old methods map approximately to one or more new methods/attributes, but the new methods should be used directly when possible.

Formatter Styles

There are many attributes you can get and set on a v10.4-style number formatter, including the number style, locale, negative-number and positive-number format strings, strings for special values, text attribute sets for created attributed strings, and various other configuration attributes. You are encouraged, however, not to change individual settings, but instead to use the `NSNumberFormatter` style constants to specify pre-defined sets of attributes that determine how a formatted number is displayed—`NSNumberFormatterNoStyle`, `NSNumberFormatterDecimalStyle`, `NSNumberFormatterCurrencyStyle`, `NSNumberFormatterPercentStyle`, `NSNumberFormatterScientificStyle`, or `NSNumberFormatterSpellOutStyle` (which generates a textual representation of a number). These are styles that the user can configure in the International preferences panel in System Preferences. If you specify your own format string, you lose user-configurability.

Parsing and Creating Strings

In addition to the methods inherited from `NSFormatter` (such as `getObjectValue:forString:errorDescription:)`, `NSNumberFormatter` adds two convenience methods—`stringFromNumber:` and `numberFromString:`—and a method to parse a string—`getObjectValue:forString:range:error:`. These methods make it easier for you to use an `NSNumberFormatter` object directly in code, and make it easier to format numbers into strings in more complex and more convenient ways than `NSString` formatting allows.

The `getObjectValue:forString:range:error:` method allows you to specify a subrange of the string to be parsed, and it returns the range of the string that was actually parsed (in the case of failure, it indicates where the failure occurred). It also returns an `NSError` object that can contain richer information than the failure string returned by the `getObjectValue:forString:errorDescription:` method inherited from `NSFormatter`.

> **Note:** Prior to Mac OS v10.6, the implementation of `getObjectValue:forString:errorDescription:` would return `YES` and an object value even if only part of the string could be parsed. This is problematic because you cannot be sure what portion of the string was parsed. For applications linked on or after Mac OS v10.6, this method instead returns an error if part of the string cannot be parsed. You can use `getObjectValue:forString:range:error:` to get the old behavior; this method returns the range of the substring that was successfully parsed.

Note that, since they work with general instances of `NSFormatter`, instances of `NSCell` only invoke the `NSFormatter getObjectValue:forString:errorDescription:` method in Mac OS X v10.4. For a v10.4-style number formatter, that method calls `getObjectValue:forString:range:error:`.

The new methods in v10.4 do not do anything when invoked on a v10.0-style formatter, and return a generic return value when required to return something. The new methods should not be invoked on a v10.0-style formatter. On a v10.4-style formatter, the old methods map approximately to one or more new methods/attributes, but the new methods should be used directly when possible.

Format Strings

The format string uses the format patterns from the Unicode Technical Standard #35 (this reference is to version tr35-6; formatters for Mac OS X v10.4 use version tr35-4). (When the formatter is in v10.0 mode, you must use the old-style format strings, described in "Number Format String Syntax (Mac OS X Versions 10.0 to 10.3)" (page ?).) Note with the Unicode format string format, you should enclose literal text in the format string inside single quotes (').

Percentages

If you use a format string with a "%" character to format percentages, the results may be confusing. Consider the following example:

```
NSNumberFormatter *numberFormatter = [[NSNumberFormatter alloc] init];
[numberFormatter setFormatterBehavior:NSNumberFormatterBehavior10_4];
[numberFormatter setFormat:@"0.00%;0.00%;-0.00%"];
NSNumber *four = [NSNumber numberWithFloat:4.0];
NSLog(@"%@", [numberFormatter stringFromNumber:four]);
// output: "400.00%"
```

Because the format string is specified to use percentages, `NSNumberFormatter` interprets the number four as a fraction (where 1 is 100%) and renders it as such (4 = 4/1 = 400%).

If you want to represent a number as a percentage, however, you should use the `NSNumberFormatterPercentStyle` style—this also ensures that percentages are formatted appropriately for the locale:

```
NSNumberFormatter *numberFormatter = [[NSNumberFormatter alloc] init];
[numberFormatter setFormatterBehavior:NSNumberFormatterBehavior10_4];
[numberFormatter setNumberStyle:NSNumberFormatterPercentStyle];
NSNumber *four = [NSNumber numberWithFloat:4.0];
```

```
NSLocale *usLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"en_US"];
[numberFormatter setLocale:usLocale];
NSLog(@"en_US: %@", [numberFormatter stringFromNumber:four]);
// output: "en_US: 400%"

NSLocale *faLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"fa_IR"];
[numberFormatter setLocale:faLocale];
NSLog(@"fa_IR: %@", [numberFormatter stringFromNumber:four]);
// output: "fa_IR:          "
```

Nomenclature

NSNumberFormatter provides several methods (such as setMaximumFractionDigits:) that allow you to manage the number of **fraction digits** allowed as input by an instance. "Fraction digits" are the numbers after the decimal separator (in English locales, the decimal separator is typically referred to as the "decimal point").

Date Formatters

This article describes how you use date formatters using Mac OS X v10.4 and later and iOS. If you are using Mac OS X v10.3 or earlier, you should read "Date and Number Formatters on Mac OS X v10.0 to 10.3" (page 19).

Behavior Modes

On Mac OS X desktop version 10.4 and later, instances of an `NSDateFormatter` can operate in two modes, 10.0 compatibility mode and 10.4 mode.

> **iOS:** The v10.0 compatibility mode is not available on iOS—only the 10.4 mode is available.

- In the v10.0 compatibility mode, `NSDateFormatter` operates as it did in Mac OS X from version 10.0 to 10.3, including the limitations and bugs—see "Date and Number Formatters on Mac OS X v10.0 to 10.3" (page 19).

- On Mac OS X v10.4 and later, the behavior of the `NSDateFormatter` class is based on CFDateFormatter, which is in turn based on the open-source ICU (International Components for Unicode) library. (`NSDateFormatter` and CFDateFormatter, however, are not toll-free bridged.)

The v10.4 behavior mode allows more configurability and better localization; you should use it for any new projects, and ideally upgrade any existing code that uses v10.0 behavior to use v10.4 behavior.

For backwards binary compatibility, the default behavior for `NSDateFormatter` on Mac OS X version 10.4 is the v10.0 behavior. On Mac OS X version 10.5 and later, however, the default is the v10.4 behavior. You can set the default behavior of all instances of `NSDateFormatter` to the v10.4 behavior by invoking the class method, `setDefaultFormatterBehavior:` with the argument `NSDateFormatterBehavior10_4`. You can also set the behavior for any instance individually by invoking `setFormatterBehavior:`.

You can set an application default (preference) to cause date formatters to be automatically converted to the new style. Set the `NSMakeDateFormatters10_4` default to a Boolean `YES`/`true` value in the application's preferences. You must set the preference before any use is made of the `NSDateFormatter` class. Note that the default has two effects:

1. Date formatters you initialize with `init` adopt the v10.4 formatter behavior.

2. Date formatters that are unarchived from either non-keyed or keyed archives are converted to v10.4-style if the archived formatter has an un-customized format string at unarchive time.

The object type for date formatters that use the v10.0 behavior mode is `NSCalendarDate`, but for 10.4-style formatters it is `NSDate`. If you want, you can configure a 10.4-style formatter to generate `NSCalendarDate` objects using `setGeneratesCalendarDates:`. You are encouraged, however, to switch to using `NSDate` for new style formatters.

The 10.4 methods do not do anything when invoked on a v10.0-style formatter, and return a generic return value when required to return something—you should therefore not invoke the new methods on a v10.0-style formatter. On a v10.4-style formatter, the old methods map approximately to one or more new methods or attributes, but you should use the new methods directly when possible.

Formatter Styles

There are many attributes you can get and set on a date formatter, including the date style, time style, locale, time zone, calendar, format string, the two-digit-year cross-over date, the default date which provides unspecified components, and there is also access to the various textual strings, like the month names. You are encouraged, however, not to change individual settings, but instead to use the `NSDateFormatter` style constants to specify pre-defined sets of attributes that determine how a formatted date is displayed—`NSDateFormatterNoStyle`, `NSDateFormatterShortStyle`, `NSDateFormatterMediumStyle`, `NSDateFormatterLongStyle`, or `NSDateFormatterFullStyle`. These are styles that the user can configure in the International preferences panel in System Preferences. If you specify your own format string, you lose user-configurability. The code sample below illustrates how you can set a date format using formatter styles.

```
// assume default behavior set for class using
// [NSDateFormatter setDefaultFormatterBehavior:NSDateFormatterBehavior10_4];
NSDateFormatter *dateFormatter =
        [[[NSDateFormatter alloc] init] autorelease];
[dateFormatter setDateStyle:NSDateFormatterMediumStyle];
[dateFormatter setTimeStyle:NSDateFormatterNoStyle];
NSDate *date =
        [NSDate dateWithTimeIntervalSinceReferenceDate:118800];
NSString *formattedDateString = [dateFormatter stringFromDate:date];
NSLog(@"formattedDateString for locale %@: %@",
        [[dateFormatter locale] localeIdentifier], formattedDateString);
// Output: formattedDateString for locale en_US: Jan 2, 2001
```

Parsing and Creating Strings

In addition to the methods inherited from `NSFormatter` (such as `getObjectValue:forString:errorDescription:`), `NSDateFormatter` adds two convenience methods—`stringFromDate:` and `dateFromString:`—and a method to parse a string—`getObjectValue:forString:range:error:`. These methods make it easier for you to use an `NSDateFormatter` object directly in code, and make it easier to format dates into strings more complex and more convenient ways than `NSString` formatting allows.

The `getObjectValue:forString:range:error:` method allows you to specify a subrange of the string to be parsed, and it returns the range of the string that was actually parsed (in the case of failure, it indicates where the failure occurred). It also returns an `NSError` object that can contain richer information than the failure string returned by the `getObjectValue:forString:errorDescription:` method inherited from `NSFormatter`.

> **Note:** Prior to Mac OS v10.6, the implementation of `getObjectValue:forString:errorDescription:` would return `YES` and an object value even if only part of the string could be parsed. This is problematic because you cannot be sure what portion of the string was parsed. For applications linked on or after Mac OS v10.6, this method instead returns an error if part of the string cannot be parsed. You can use `getObjectValue:forString:range:error:` to get the old behavior; this method returns the range of the substring that was successfully parsed.

Note that, since they work with general instances of `NSFormatter`, instances of `NSCell` only invoke the `NSFormatter getObjectValue:forString:errorDescription:` method on Mac OS X v10.4. For a v10.4-style date formatter, that method calls `getObjectValue:forString:range:error:`.

The v10.4-style date formatter's lenient parsing mode is not as forgiving as the "natural language" parsing of `NSDateFormatter` when the `allowsNaturalLanguage` option is enabled in the formatter. This has advantages and disadvantages: users will have to be more careful and perhaps thorough when typing in dates, but they are more likely to find that the value they were trying to input was correctly set to the value they wanted rather than what the "natural language" parsing guessed they meant.

Format Strings

The format string uses the format patterns from the Unicode standard (this reference is to version tr35-6 for Mac OS X v10.5; for Mac OS X v10.4 use version tr35-4). (When the formatter is in v10.0 mode, you must use the old-style format strings, described in "Date Format String Syntax (Mac OS X Versions 10.0 to 10.3)" (page 22).) Note with the Unicode format string format, you should enclose literal text in the format string inside single quotes ('), as illustrated in the following example:

```
NSDateFormatter *inputFormatter = [[NSDateFormatter alloc] init];
[inputFormatter setDateFormat:@"yyyy-MM-dd 'at' HH:mm"];

NSDate *formatterDate = [inputFormatter dateFromString:@"1999-07-11 at 10:30"];

NSDateFormatter *outputFormatter = [[NSDateFormatter alloc] init];
[outputFormatter setDateFormat:@"HH:mm 'on' EEEE MMMM d"];

NSString *newDateString = [outputFormatter stringFromDate:formatterDate];

NSLog(@"newDateString %@", newDateString);
// For US English, the output is:
// newDateString 10:30 on Sunday July 11
```

Formatters and User Interface Elements

This article describes how to associate a formatter with a cell in Cocoa. This article does not apply to iOS.

Associating a Formatter With a Cell

The easiest way to use a formatter is in Interface Builder to drag it from the palette onto a control such as a text field or a column in a table view. You can then configure the behavior you want using the inspector—typically you should use v10.4 behavior.

To create a formatter object programmatically and attach it to a cell, you allocate an instance of the formatter and set its format or style as you wish. You then use the `NSCell setFormatter:` method to associate the formatter instance with a cell. The following code example creates and configures an instance of `NSNumberFormatter`, and applies it to the cell of an `NSTextField` object using the `setFormatter:` method.

```
NSNumberFormatter *numberFormatter = [[NSNumberFormatter alloc] init];
[numberFormatter setNumberStyle:NSNumberFormatterCurrencyStyle];
[[textField cell] setFormatter:numberFormatter];
```

Similarly, you can create and configure an instance of `NSDateFormatter` object programmatically. The following example creates a date formatter then associates it with the cells of a form (`contactsForm`).

```
NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
[dateFormatter setDateStyle:NSDateFormatterMediumStyle];
[dateFormatter setTimeStyle:NSDateFormatterNoStyle];
[[contactsForm cells] makeObjectsPerformSelector:@selector(setFormatter:)
        withObject:dateFormatter]
```

Instances of formatter objects are immutable. In addition, when a cell with a formatter object is copied, the new cell retains the formatter object rather than copying it.

The type of object that you retrieve from a cell using the method `objectValue` depends upon the behavior of the formatter:

- For a number formatter, using Mac OS X version 10.0 behavior it would be a `NSDecimalNumber` object; using 10.4 behavior it would be by default a `NSNumber` object.

- For a date formatter, using Mac OS X version 10.0 behavior it would be a `NSCalendarDate` object, using v10.4 behavior it would be by default a `NSDate` object.

When the cell needs to display or edit its value, it passes its object to the formatter which returns the formatted string. When the user enters a string, or when a string is programmatically written in a cell (using `setStringValue`), the cell obtains the corresponding object from the formatter.

Delegation Methods for Error Handling

`NSControl` has delegation methods for handling errors returned in implementations of `NSFormatter`'s `getObjectValue:forString:errorDescription:`, `isPartialStringValid:proposedSelectedRange:originalString:originalSelectedRange:errorDescription:`, and `isPartialStringValid:newEditingString:errorDescription:` methods. These delegation methods are, respectively, `control:didFailToFormatString:errorDescription:` and `control:didFailToValidatePartialString:errorDescription:`.

Date and Number Formatters on Mac OS X v10.0 to 10.3

This article describes how to create and use formatters on Mac OS X 10.0 to 10.3, and the format string patterns available when you use a formatter in v10.0 mode. You are strongly encouraged to migrate to using v10.4 behavior.

Creating and Using Formatters Programmatically (Mac OS X 10.0 to 10.3)

The easiest way to use a formatter is in Interface Builder to drag it from the palette onto a control such as a text field or a column in a table view. You can also create and manipulate instances of the formatter programmatically. Do this if you're not using Interface Builder to create your user interface or if you simply want more fine-grained control over formatter object (for example, to change the text attributes of the values displayed).

Date Formatters

Creating a Date Formatter

To create a date formatter programmatically, simply alloc and init it, following normal rules for memory management.

```
NSDateFormatter *dateFormat = [[NSDateFormatter alloc]
        initWithDateFormat:@"%b %1d %Y" allowNaturalLanguage:NO];
```

Note the use of 1 in the date field to specify a width of 1—this ensures that single digit dates are output without a leading 0. See "Date Format String Syntax (Mac OS X Versions 10.0 to 10.3)" (page 22) for a complete description of the syntax of the corresponding date format strings.

You must specify a format string whenever you create a date formatter (In Java, a Gregorian date formatter is an object of the class NSGregorianDateFormatter; in Objective-C, it's an object of the class NSDateFormatter.) This format is a string that contains specifiers that are very similar to those used in the standard C library function strftime(). When a date formatter converts a date to a string, it uses this same format string. For example, a date format string of "%b %d %Y" would yield strings such as "Mar 15 1994". This code example creates a date formatter object that yields date strings such as "7/21/2003":

Number Formatters

Creating a Number Formatter

To create an NSNumberFormatter object programmatically, simply use alloc and init, and follow the normal rules for memory management.

```
NSNumberFormatter *numberFormatter =
    [[[NSNumberFormatter alloc] init] autorelease];
```

A common technique for assigning a format to an NSNumberFormatter object is to use the method setFormat,. You can also specify the format of positive, zero, and negative values in one format string as follows:

```
// specify just positive format
[numberFormatter setFormat:@"$#,##0.00"];

// specify positive and negative formats
[numberFormatter setFormat:@"$#,##0.00;($#,##0.00)"];

// specify positive, zero, and negative formats
[numberFormatter setFormat:@"$#,###.00;0.00;($#,##0.00)"];
```

As an alternative to using the setFormat: method, you can use the setPositiveFormat: and setNegativeFormat: methods.

Setting Text Attributes

In NSNumberFormatter, positive, negative, zero, nil, and "not a number" values are represented as attributed strings (NSAttributedString objects). With attributed strings, you can apply attributes such as color or font to a range of characters in a string. (For more information on NSAttributedString, see *Attributed String Programming Guide*.)

Because the values displayed by NSNumberFormatter are attributed strings, you can customize aspects of their appearance, such as their color and font. The NSNumberFormatter methods you use to do this are as follows:

```
    textAttributesForPositiveValues
    setTextAttributesForPositiveValues:
    textAttributesForNegativeValues
    setTextAttributesForNegativeValues:
    attributedStringForZero
    setAttributedStringForZero:
    attributedStringForNil
    setAttributedStringForNil:
    attributedStringForNotANumber
    setAttributedStringForNotANumber:
```

For example, in the output of this code example, negative values are displayed in red:

```
NSNumberFormatter *numberFormatter =
[[[NSNumberFormatter alloc] init] autorelease];
```

```
NSMutableDictionary *newAttrs = [NSMutableDictionary dictionary];

[numberFormatter setFormat:@"$#,##0.00;($#,##0.00)"];
[newAttrs setObject:[NSColor redColor] forKey:@"NSColor"];
[numberFormatter setTextAttributesForNegativeValues:newAttrs];
[[textField cell] setFormatter:numberFormatter];
```

Setting Separators

NSNumberFormatter supports two kinds of separators: thousand and decimal. By default these separators are represented by, respectively, the comma (,) and period (.) characters. By default, they're disabled.

All of the following Java statements have the effect of enabling thousand separators:

```
// use setFormat:
numberFormatter.setFormat("#,###");

// use setHasThousandSeparators:
numberFormatter.setHasThousandSeparators(true);

// use setThousandSeparator:
numberFormatter.setThousandSeparator("_");
```

And all of the following Objective-C statements also have the effect of enabling thousand separators:

```
// use setFormat:
[numberFormatter setFormat:@"#,###"];

// use setHasThousandSeparators:
[numberFormatter setHasThousandSeparators:YES];

// use setThousandSeparator:
[numberFormatter setThousandSeparator:@"_"];
```

If you use the method setHasThousandSeparators with an argument of no or false, it disables thousand separators, even if you've set them through another means.

Both of the following Java statements have the effect of enabling decimal separators:

```
// use setFormat:
numberFormatter.setFormat("0.00");

// use setDecimalSeparator:
numberFormatter.setDecimalSeparator("-");
```

And both of the following Objective-C statements also have the effect of enabling decimal separators:

```
// use setFormat:
[numberFormatter setFormat:@"0.00"];

// use setDecimalSeparator:
[numberFormatter setDecimalSeparator:@"-"];
```

When you enable or disable separators, it affects positive and negative formats. Consequently, both formats must use the same separator scheme.

Even though, you can use the `thousandSeparator` and `decimalSeparator` methods to return a string containing the character the receiver uses to represent each separator. These methods don't tell you whether separators are enabled. Even when separators are disabled, an NSNumberFormatter object still knows the characters it uses to represent separators.

Separators must be single characters. If you specify multiple characters in the arguments to `setThousandSeparator:` and `setDecimalSeparator:`, only the first character is used.

You can't use the same character to represent thousand and decimal separators.

Date Format String Syntax (Mac OS X Versions 10.0 to 10.3)

Date formatters format the textual representation of cells that contain date objects (including Gregorian dates), and convert textual representations of dates and times into date objects. You can express the representation of dates and times very flexibly: "Thu 22 Dec 1994" is just as acceptable as "12/22/94". In Cocoa, with natural-language processing for dates enabled, users can also express dates colloquially, such as "today," "day after tomorrow," and "a month from today."

Format String Syntax

You create date formatter objects by specifying a format string using `strftime`-style conversion specifiers. For example, the format string `"%m/%d/%y"` yields strings such as `"01/02/01"`, and `"%1m/%1d/%Y"` yields strings such as `"1/2/2001"`, as illustrated in the following example.

```
NSDateFormatter *dateFormatter = [[[NSDateFormatter alloc]
    initWithDateFormat:@"%1m/%1d/%Y" allowNaturalLanguage:NO] autorelease];
NSDate *date = [NSDate dateWithTimeIntervalSinceReferenceDate:118800];
NSString *formattedDateString = [dateFormatter stringFromDate:date];
NSLog(@"formattedDateString: %@", formattedDateString);

// Output: formattedDateString: 1/2/2001
```

You use the format string is used to specify both the input to and the output from date formatter objects. All the date formatter objects in both Foundation and Core Foundation use this syntax when specifying the format string. The date conversion specifiers in the table below cover a range of date conventions (differences from the format used by `strftime()` are noted in parentheses):

Specifier	Description
%%	A '%' character
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Shorthand for "%X %x", the locale format for date and time

Specifier	Description
%d	Day of the month as a decimal number (01-31)
%e	Same as `%d` but does not print the leading 0 for days 1 through 9 (unlike `strftime()`, does not print a leading space)
%F	Milliseconds as a decimal number (000-999)
%H	Hour based on a 24-hour clock as a decimal number (00-23)
%I	Hour based on a 12-hour clock as a decimal number (01-12)
%j	Day of the year as a decimal number (001-366)
%m	Month as a decimal number (01-12)
%M	Minute as a decimal number (00-59)
%p	AM/PM designation for the locale
%S	Second as a decimal number (00-59)
%w	Weekday as a decimal number (0-6), where Sunday is 0
%x	Date using the date representation for the locale, including the time zone (produces different results from `strftime()`)
%X	Time using the time representation for the locale (produces different results from `strftime()`)
%y	Year without century (00-99)
%Y	Year with century (such as 1990)
%Z	Time zone name (such as Pacific Daylight Time; produces different results from `strftime()`)
%z	Time zone offset in hours and minutes from GMT (HHMM)

Number Format String Syntax (Mac OS X Versions 10.0 to 10.3)

Number formatters format the textual representation of user interface cells that contain number objects and convert textual representations of numeric values into number objects. The representation encompasses integers, floats, and doubles. Floats and doubles can be formatted to a specified decimal position. Number formatters can also impose ranges on the numeric values cells can accept.

You create number formatter objects by specifying a number format string. This format string is used to specify both the input to and output from number formatter objects. The number formatter object in Cocoa uses this syntax when specifying the format string as described below. Note that you can specify different formats for positive, negative, and zero number values.

Format String Syntax

Number format strings can include the following five types of character:

- Numbers

  Format strings can include numeric characters. Wherever you include a number in a format string, the number is displayed unless an input character in the same relative position overwrites it. For example, if you have the positive format string @"9,990.00" and the value 53.88 is entered into a cell to which the format has been applied, the cell displays the value as 9,953.88.

- Separators

  Format strings can include the period character (.) as a decimal separator, and comma character (,) as a thousand separator. If you want to use other characters as separators, you can change them using the number formatter methods or functions. When you enable localization for a number formatter, separators are converted to characters appropriate to the environment in which the application is running.

- Placeholders

  You use the pound sign character (#) to represent numeric characters that will be input by the user. For example, suppose you have the positive format @"$#,##0.00". If the characters 76329 are entered into a cell to which the format has been applied, they are displayed as $76,329.00. Strictly speaking, however, you don't need to use placeholders. The format strings @",0.00", @"#,#0.00", and @"#,##0.00" are functionally equivalent. In other words, including separator characters in a format string signals a number formatter to use the separators, regardless of whether you use (or where you put) placeholders. The placeholder character's chief virtue lies in its ability to make format strings more human readable, which is especially useful if you're displaying formats in the user interface.

- Spaces

  To include a space in a format string, use the underscore character (_). This character inserts a space if no numeric character has been input to occupy that position.

- Currency

  The dollar sign character ($) is normally treated just like any other character that doesn't play a special role in a number formatter. However, when you enable localization for a number formatter object, the dollar sign character is converted to the currency symbol appropriate for the environment in which the application is running.

All other characters specified in a number format string are displayed as typed. The following table shows examples of the how the value 1019.55 is displayed for different positive formats:

Format String	Display
@"#,##0.00"	1,019.55
@"$#,##0.00"	$1,019.55
@"___,__0.00"	1,019.55

Specifying Positive, Negative, and Zero Formats

When creating a number formatter object, you can also specify a different format for positive, negative, and zero values, all using one format string. The format string can be one of the following:

- @"*positiveFormat*"

  For example, @"$###,##0.00" (a single number format string as described above).

- @"*positiveFormat*;*negativeFormat*"

  For example, @"###,##0.00;(###,##0.00)".

- @"*positiveFormat*;*zeroFormat*;*negativeFormat*"

  For example, @"$###,###.00;0.00;($###,##0.00)". Note that zero formats are treated as string constants.

No matter which option you choose, you're required to specify a format for positive values only. If you don't specify a format for negative and zero values, a default format based on the positive value format is used. For example, if your positive value format is "#,##0.00", an input value of "0" is displayed as "0.00".

If you don't specify a format for negative values, the format specified for positive values is used, preceded by a minus sign (-).

If you specify a separate format for negative values, its separators should be parallel to those specified in the positive format string. In number formatters, separators are either enabled or disabled for all formats—your negative and positive formats should therefore both use the same approach.

NSDateFormatter Format String Syntax

You create date formatter objects by specifying a format string using `strftime`-style conversion specifiers. For example, the format string "%m/%d/%y" yields strings such as "01/02/01", and "%1m/%1d/%Y" yields strings such as "1/2/2001", as illustrated in the following example.

```
NSDateFormatter *dateFormatter = [[[NSDateFormatter alloc]
    initWithDateFormat:@"%1m/%1d/%Y" allowNaturalLanguage:NO] autorelease];
NSDate *date = [NSDate dateWithTimeIntervalSinceReferenceDate:118800];
NSString *formattedDateString = [dateFormatter stringFromDate:date];
NSLog(@"formattedDateString: %@", formattedDateString);

// Output: formattedDateString: 1/2/2001
```

You use the format string is used to specify both the input to and the output from date formatter objects. All the date formatter objects in both Foundation and Core Foundation use this syntax when specifying the format string. The date conversion specifiers in the table below cover a range of date conventions (differences from the format used by `strftime()` are noted in parentheses):

Specifier	Description
%%	A '%' character
%a	Abbreviated weekday name

Specifier	Description
`%A`	Full weekday name
`%b`	Abbreviated month name
`%B`	Full month name
`%c`	Shorthand for "`%X %x`", the locale format for date and time
`%d`	Day of the month as a decimal number (01-31)
`%e`	Same as `%d` but does not print the leading 0 for days 1 through 9 (unlike `strftime()`, does not print a leading space)
`%F`	Milliseconds as a decimal number (000-999)
`%H`	Hour based on a 24-hour clock as a decimal number (00-23)
`%I`	Hour based on a 12-hour clock as a decimal number (01-12)
`%j`	Day of the year as a decimal number (001-366)
`%m`	Month as a decimal number (01-12)
`%M`	Minute as a decimal number (00-59)
`%p`	AM/PM designation for the locale
`%S`	Second as a decimal number (00-59)
`%w`	Weekday as a decimal number (0-6), where Sunday is 0
`%x`	Date using the date representation for the locale, including the time zone (produces different results from `strftime()`)
`%X`	Time using the time representation for the locale (produces different results from `strftime()`)
`%y`	Year without century (00-99)
`%Y`	Year with century (such as 1990)
`%Z`	Time zone name (such as Pacific Daylight Time; produces different results from `strftime()`)
`%z`	Time zone offset in hours and minutes from GMT (HHMM)

Creating a Custom Formatter

You can create various custom subclasses of `NSFormatter`. For example, you might want a custom formatter of telephone numbers, or a custom formatter of part numbers.

To subclass `NSFormatter`, you must, at the least, override the three primitive methods:

- `stringForObjectValue:`
- `getObjectValue:forString:errorDescription:`
- `attributedStringForObjectValue:withDefaultAttributes:`

In the first method you convert the cell's object to a string representation; in the second method you convert the string to the object associated with the cell.

In the third method, `attributedStringForObjectValue:withDefaultAttributes:`, you convert the object to a string that has attributes associated with it. For example, if you want negative financial amounts to appear in red, you have this method return a string with an attribute of red text. In `attributedStringForObjectValue:withDefaultAttributes:` get the non-attributed string by invoking `stringForObjectValue:` and then apply the proper attributes to that string.

If the string for editing must differ from the string for display—for example, the display version of a currency field shows a dollar sign but the editing version doesn't—implement `editingStringForObjectValue:` in addition to `stringForObjectValue:`.

You can edit the textual contents of a cell at each keypress and prevent the user from entering invalid characters using `isPartialStringValid:proposedSelectedRange:originalString:originalSelectedRange:errorDescription:` and `isPartialStringValid:newEditingString:errorDescription:`. You can apply this dynamic editing to things like telephone numbers or social security numbers; the person entering data enters the number only once, since the formatter automatically inserts the separator characters.

Document Revision History

This table describes the changes to *Data Formatting Guide*.

Date	Notes
2009-08-06	Added links to Cocoa Core Competencies.
2009-05-25	Corrected typographical errors.
2008-10-15	Corrected typographical errors.
2007-03-20	Updated links to Unicode format specifications.
2007-01-08	Updated links to Unicode format definitions; added section on formatting Percentages to "NSNumberFormatter on Mac OS X 10.4".
2006-05-23	Added notes about the use of formatters with Interface Builder.
	Moved discussion of string formatting and string format specifiers to *String Programming Guide*.
2005-11-09	Enhanced discussion of string formatting.
2005-08-11	Changed the title from "Data Formatting." Updated to describe new functionality in Mac OS X v10.4.
2004-08-31	Added descriptions of NSString format specifiers `%qx` and `%qX` to Formatting String Objects.
2003-08-07	Revised and updated content.
2003-01-15	Clarified how cell contents are displayed when no formatter is set.
2002-11-12	Revision history was added to existing document.