Data Formatting Guide

Developer

Contents

Listings

Introduction to Data Formatting Programming Guide For Cocoa

Important: This is a preliminary document for an API or technology in development. Apple is supplying this information to help you plan for the adoption of the technologies and programming interfaces described herein for use on Apple-branded products. This information is subject to change, and software implemented according to this document should be tested with final operating system software and final documentation. Newer versions of this document may be provided with future betas of the API or technology.

You use formatters to interpret and create strings that represent other data types, and to validate the text in text fields and other cells. Formatters are instances of subclasses of the abstract class, `NSFormatter`. The Foundation framework provides two concrete subclasses of `NSFormatter`: `NSNumberFormatter` and `NSDateFormatter`. (Core Foundation provides two equivalent opaque types: CFNumberFormatter and CFDateFormatter. These are similar but are not toll-free bridged.) You can create a subclass of `NSFormatter` for custom formatting.

You should read this document to gain a basic understanding of how to create and use date and number formatters, and how you can create a custom formatter object.

Organization of This Document

Date Formatters (page 5) describes how to use date formatters.

Number Formatters (page 13) describes how to use number formatters.

Formatters and User Interface Elements (page 16) describes how to set a formatter for a user interface element, and the interaction between an element and its formatter in OS X.

Creating a Custom Formatter (page 18) outlines how to create custom formatter classes.

Date Formatters

There are two basic methods you use to create a string representation of a date and to parse a string to get a date object using a date formatter—`dateFromString:` and `stringFromDate:` respectively. You can also use `getObjectValue:forString:range:error:` if you need more control over the range of a string you want to parse.

There are many attributes you can get and set on a date formatter. When you present information to the user, you should typically simply use the `NSDateFormatter` style constants to specify pre-defined sets of attributes that determine how a formatted date is displayed. If you need to generate a representation of a date in a precise format, however, you should use a format string.

If you need to parse a date string, the approach you take again depends on what you want to accomplish. If you want to parse input from the user, you should typically use the style constants so as to match their expectations. If you want to parse dates you get from a database or a web service, for example, you should use a format string.

In all cases, you should consider that formatters default to using the user's locale (`currentLocale`) superimposed with the user's preference settings. If you want to use the user's locale but without their individual settings, you can get the locale id from the current user locale (`localeIdentifier`) and make a new "standard" locale with that, then set the standard locale as the formatter's `locale`.

Use Formatter Styles to Present Dates and Times With the User's Preferences

`NSDateFormatter` makes it easy for you to format a date using the settings a user configured in the International preferences panel in System Preferences. The `NSDateFormatter` style constants—`NSDateFormatterNoStyle`, `NSDateFormatterShortStyle`, `NSDateFormatterMediumStyle`, `NSDateFormatterLongStyle`, and `NSDateFormatterFullStyle`—specify sets of attributes that determine how a date is displayed according to the user's preferences.

You specify the style of the date and time components of a date formatter independently using `setDateStyle:` and `setTimeStyle:` respectively. Listing 1 (page 6) illustrates how you can format a date using formatter styles. Notice the use of `NSDateFormatterNoStyle` to suppress the time component and yield a string containing just the date.

Listing 1 Formatting a date using formatter styles

```
NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
[dateFormatter setDateStyle:NSDateFormatterMediumStyle];
[dateFormatter setTimeStyle:NSDateFormatterNoStyle];

NSDate *date = [NSDate dateWithTimeIntervalSinceReferenceDate:162000];

NSString *formattedDateString = [dateFormatter stringFromDate:date];
NSLog(@"formattedDateString: %@", formattedDateString);
// Output for locale en_US: "formattedDateString: Jan 2, 2001".
```

Use Format Strings to Specify Custom Formats

There are broadly speaking two situations in which you need to use custom formats:

1. For fixed format strings, like Internet dates.
2. For user-visible elements that don’t match any of the existing styles

Fixed Formats

To specify a custom fixed format for a date formatter, you use `setDateFormat:`. The format string uses the format patterns from the Unicode Technical Standard #35. The version of the standard varies with release of the operating system:

- OS X v10.9 and iOS 7 use version tr35-31.
- OS X v10.8 and iOS 6 use version tr35-25.
- iOS 5 uses version tr35-19.
- OS X v10.7 and iOS 4.3 use version tr35-17.
- iOS 4.0, iOS 4.1, and iOS 4.2 use version tr35-15.
- iOS 3.2 uses version tr35-12.
- OS X v10.6, iOS 3.0, and iOS 3.1 use version tr35-10.
- OS X v10.5 uses version tr35-6.
- OS X v10.4 uses version tr35-4.

Although in principle a format string specifies a fixed format, by default `NSDateFormatter` still takes the user's preferences (including the locale setting) into account. You must consider the following points when using format strings:

- `NSDateFormatter` treats the numbers in a string you parse as if they were in the user's chosen calendar. For example, if the user selects the Buddhist calendar, parsing the year `2010` yields an `NSDate` object in `1467` in the Gregorian calendar. (For more about different calendrical systems and how to use them, see *Date and Time Programming Guide* .)
- In iOS, the user can override the default AM/PM versus 24-hour time setting. This may cause `NSDateFormatter` to rewrite the format string you set.

Note with the Unicode format string format, you should enclose literal text in the format string between apostrophes (`''`).

The following example illustrates using a format string to generate a string:

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

NSDate *date = [NSDate dateWithTimeIntervalSinceReferenceDate:162000];

NSString *formattedDateString = [dateFormatter stringFromDate:date];
NSLog(@"formattedDateString: %@", formattedDateString);
// For US English, the output may be:
// formattedDateString: 2001-01-02 at 13:00
```

There are two things to note about this example:

1. It uses `yyyy` to specify the year component. A common mistake is to use `YYYY`. `yyyy` specifies the calendar year whereas `YYYY` specifies the year (of "Week of Year"), used in the ISO year-week calendar. In most cases, `yyyy` and `YYYY` yield the same number, however they may be different. Typically you should use the calendar year.
2. The representation of the time may be `13:00`. In iOS, however, if the user has switched 24-Hour Time to Off, the time may be `1:00 pm`.

Custom Formats for User-Visible Dates

To display a date that contains a specific set of elements, you use `dateFormatFromTemplate:options:locale:`]. The method generates a format string with the date components you want to use, but with the correct punctuation and order appropriate for the user (that is, customized for the user's locale and preferences). You then use the format string to create a formatter.

For example, to create a formatter to display today's day name, day, and month using the current locale, you would write:

```
NSString *formatString = [NSDateFormatter dateFormatFromTemplate:@"EdMMM" options:0
                                          locale:[NSLocale currentLocale]];
NSDateFormatter *dateFormatter = [[NSDateFormatter alloc] init];
 [dateFormatter setDateFormat:formatString];

NSString *todayString = [dateFormatter stringFromDate:[NSDate date]];
NSLog(@"todayString: %@", todayString);
```

To understand the need for this, consider a situation where you want to display the day name, day, and month. You cannot create this representation of a date using formatter styles (there is no style that omits the year). Neither, though, can you *easily and consistently* create the representation correctly using format strings. Although at first glance it may seem straightforward, there's a complication: a user from the United States would typically expect dates in the form, "Mon, Jan 3", whereas a user from Great Britain would typically expect dates in the form "Mon 31 Jan".

The following example illustrates the point:

```
NSLocale *usLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"en_US"];
NSString *usFormatString = [NSDateFormatter dateFormatFromTemplate:@"EdMMM" options:0
 locale:usLocale];
NSLog(@"usFormatterString: %@", usFormatString);
// Output: usFormatterString: EEE, MMM d.

NSLocale *gbLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"en_GB"];
NSString *gbFormatString = [NSDateFormatter dateFormatFromTemplate:@"EdMMM" options:0
 locale:gbLocale];
NSLog(@"gbFormatterString: %@", gbFormatString);
// Output: gbFormatterString: EEE d MMM.
```

Parsing Date Strings

In addition to the methods inherited from NSFormatter (such as getObjectValue:forString:errorDescription:), NSDateFormatter adds dateFromString: and 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.

If you're working with fixed-format dates, you should first set the *locale* of the date formatter to something appropriate for your fixed format. In most cases the best locale to choose is en_US_POSIX, a locale that's specifically designed to yield US English results regardless of both user and system preferences. en_US_POSIX is also invariant in time (if the US, at some point in the future, changes the way it formats dates, en_US will change to reflect the new behavior, but en_US_POSIX will not), and between platforms (en_US_POSIX works the same on iPhone OS as it does on OS X, and as it does on other platforms).

Once you've set en_US_POSIX as the locale of the date formatter, you can then set the date format string and the date formatter will behave consistently for all users.

Listing 2 (page 9) shows how to use NSDateFormatter for both of the roles described above. First it creates a en_US_POSIX date formatter to parse the incoming RFC 3339 date string, using a fixed date format string and UTC as the time zone. Next, it creates a standard date formatter to render the date as a string to display to the user.

Listing 2 Parsing an RFC 3339 date-time

```
- (NSString *)userVisibleDateTimeStringForRFC3339DateTimeString:(NSString
*)rfc3339DateTimeString {
    /*
      Returns a user-visible date time string that corresponds to the specified
      RFC 3339 date time string. Note that this does not handle all possible
      RFC 3339 date time strings, just one of the most common styles.
     */

    NSDateFormatter *rfc3339DateFormatter = [[NSDateFormatter alloc] init];
    NSLocale *enUSPOSIXLocale = [[NSLocale alloc]
initWithLocaleIdentifier:@"en_US_POSIX"];

    [rfc3339DateFormatter setLocale:enUSPOSIXLocale];
    [rfc3339DateFormatter setDateFormat:@"yyyy'-'MM'-'dd'T'HH':'mm':'ss'Z'"];
    [rfc3339DateFormatter setTimeZone:[NSTimeZone timeZoneForSecondsFromGMT:0]];

    // Convert the RFC 3339 date time string to an NSDate.
    NSDate *date = [rfc3339DateFormatter dateFromString:rfc3339DateTimeString];

    NSString *userVisibleDateTimeString;
    if (date != nil) {
        // Convert the date object to a user-visible date string.
      NSDateFormatter *userVisibleDateFormatter = [[NSDateFormatter alloc] init];
        assert(userVisibleDateFormatter != nil);

        [userVisibleDateFormatter setDateStyle:NSDateFormatterShortStyle];
        [userVisibleDateFormatter setTimeStyle:NSDateFormatterShortStyle];

       userVisibleDateTimeString = [userVisibleDateFormatter stringFromDate:date];
    }
    return userVisibleDateTimeString;
}
```

Cache Formatters for Efficiency

Creating a date formatter is not a cheap operation. If you are likely to use a formatter frequently, it is typically more efficient to cache a single instance than to create and dispose of multiple instances. One approach is to use a `static` variable.

Listing 3 (page 10) re-implements the method shown in Listing 2 (page 9) to hold on to the date formatters for subsequent reuse.

Listing 3 Parsing an RFC 3339 date-time using a cached formatter

```
static NSDateFormatter *sUserVisibleDateFormatter = nil;
```

```
- (NSString *)userVisibleDateTimeStringForRFC3339DateTimeString:(NSString
*)rfc3339DateTimeString {
    /*
      Returns a user-visible date time string that corresponds to the specified
      RFC 3339 date time string. Note that this does not handle all possible
      RFC 3339 date time strings, just one of the most common styles.
     */

    // If the date formatters aren't already set up, create them and cache them
for reuse.
    static NSDateFormatter *sRFC3339DateFormatter = nil;
    if (sRFC3339DateFormatter == nil) {
        sRFC3339DateFormatter = [[NSDateFormatter alloc] init];
        NSLocale *enUSPOSIXLocale = [[NSLocale alloc]
initWithLocaleIdentifier:@"en_US_POSIX"];

        [sRFC3339DateFormatter setLocale:enUSPOSIXLocale];
       [sRFC3339DateFormatter setDateFormat:@"yyyy'-'MM'-'dd'T'HH':'mm':'ss'Z'"];
       [sRFC3339DateFormatter setTimeZone:[NSTimeZone timeZoneForSecondsFromGMT:0]];
    }

    // Convert the RFC 3339 date time string to an NSDate.
    NSDate *date = [rfc3339DateFormatter dateFromString:rfc3339DateTimeString];

    NSString *userVisibleDateTimeString;
    if (date != nil) {
        if (sUserVisibleDateFormatter == nil) {
            sUserVisibleDateFormatter = [[NSDateFormatter alloc] init];
            [sUserVisibleDateFormatter setDateStyle:NSDateFormatterShortStyle];
            [sUserVisibleDateFormatter setTimeStyle:NSDateFormatterShortStyle];
        }
        // Convert the date object to a user-visible date string.
       userVisibleDateTimeString = [sUserVisibleDateFormatter stringFromDate:date];
    }
    return userVisibleDateTimeString;
}
```

If you cache date formatters (or any other objects that depend on the user's current locale), you should subscribe to the NSCurrentLocaleDidChangeNotification notification and update your cached objects when the current locale changes. The code in Listing 3 (page 10) defines sUserVisibleDateFormatter outside of the method so that other code, not shown, can update it as necessary. In contrast, sRFC3339DateFormatter is defined inside the method because, by design, it is not dependent on the user's locale settings.

Note: In theory you could use an auto-updating locale (autoupdatingCurrentLocale) to create a locale that automatically accounts for changes in the user's locale settings. In practice this currently does not work with date formatters.

Consider Unix Functions for Fixed-Format, Unlocalized Dates

For date and times in a fixed, unlocalized format, that are always guaranteed to use the same calendar, it may sometimes be easier and more efficient to use the standard C library functions strptime_l and strftime_l.

Be aware that the C library also has the idea of a current locale. To guarantee a fixed date format, you should pass NULL as the loc parameter of these routines. This causes them to use the POSIX locale (also known as the C locale), which is equivalent to Cocoa's en_US_POSIX locale, as illustrated in this example.

```
struct tm  sometime;
const char *formatString = "%Y-%m-%d %H:%M:%S %z";
(void) strptime_l("2005-07-01 12:00:00 -0700", formatString, &sometime, NULL);
NSLog(@"NSDate is %@", [NSDate dateWithTimeIntervalSince1970: mktime(&sometime)]);
// Output: NSDate is 2005-07-01 12:00:00 -0700
```

Number Formatters

`NSNumberFormatter` provides two convenient methods—`stringFromNumber:` and `numberFromString:`—that you can use to create a string representation of a number and to create a number object from a string respectively. To create a localized string representation of a number without creating a formatter object, you can use the class method `localizedStringFromNumber:numberStyle:`.

If you have more sophisticated requirements when parsing a string, in addition to the methods inherited from `NSFormatter` (such as `getObjectValue:forString:errorDescription:`), 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 rich information about the problem.

There are many attributes you can get and set on a number formatter. When you present information to the user, you should typically simply use the `NSNumberFormatter` style constants to specify pre-defined sets of attributes that determine how a formatted number is displayed. If you need to generate a representation of a number in a precise format, however, you should use a format string.

Use Formatter Styles to Present Numbers With the User's Preferences

`NSNumberFormatter` makes it easy for you to format different sorts of number using the settings a user configured in the International preferences panel in System Preferences. The `NSNumberFormatter` style constants—`NSNumberFormatterDecimalStyle`, `NSNumberFormatterCurrencyStyle`, `NSNumberFormatterPercentStyle`, `NSNumberFormatterScientificStyle`, or `NSNumberFormatterSpellOutStyle` (which generates a textual representation of a number)—specify sets of attributes that determine how a number is displayed according to the user's preferences.

You specify the style of a number formatter using `setNumberStyle:`. Listing 1 (page 13) illustrates how you can format a date using formatter styles.

Listing 1 Formatting a number using a formatter style

```
NSNumberFormatter *numberFormatter = [[NSNumberFormatter alloc] init];
[numberFormatter setNumberStyle:NSNumberFormatterDecimalStyle];
NSString *formattedNumberString = [numberFormatter stringFromNumber:@122344.4563];
```

```
NSLog(@"formattedNumberString: %@", formattedNumberString);
// Output for locale en_US: "formattedNumberString: formattedNumberString:
122,344.453"
```

Use Format Strings to Specify Custom Formats

The format string uses the format patterns from the Unicode Technical Standard #35. The version of the standard varies with release of the operating system:

- OS X v10.9 and iOS 7 use version tr35-31.
- OS X v10.8 and iOS 6 use version tr35-25.
- iOS 5 uses version tr35-19.
- OS X v10.7 and iOS 4.3 use version tr35-17.
- iOS 4.0, iOS 4.1, and iOS 4.2 use version tr35-15.
- iOS 3.2 uses version tr35-12.
- OS X v10.6, iOS 3.0, and iOS 3.1 use version tr35-10.
- OS X v10.5 uses version tr35-6.
- OS X v10.4 uses version tr35-4.

Note with the Unicode format string format, you should enclose literal text in the format string between apostrophes (' ').

You specify the format string of a number formatter using `setPositiveFormat:` and `setNegativeFormat:`. Listing 2 (page 14) illustrates how you can format a date using formatter styles.

Listing 2 Formatting a number using a format string

```
NSNumberFormatter *numberFormatter = [[NSNumberFormatter alloc] init];
[numberFormatter setPositiveFormat:@"###0.##"];
NSString *formattedNumberString = [numberFormatter stringFromNumber:@122344.4563];
NSLog(@"formattedNumberString: %@", formattedNumberString);
// Output for locale en_US: "formattedNumberString: formattedNumberString:
122,344.45"
```

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 setPositiveFormat:@"0.00%;0.00%;-0.00%"];
NSLog(@"%@", [numberFormatter stringFromNumber:@4.0]);
// 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, you should use the `NSNumberFormatterPercentStyle` style—this also ensures that percentages are formatted appropriately for the locale:

```
NSNumberFormatter *numberFormatter = [[NSNumberFormatter alloc] init];
[numberFormatter setNumberStyle:NSNumberFormatterPercentStyle];

NSLocale *usLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"en_US"];
[numberFormatter setLocale:usLocale];
NSLog(@"en_US: %@", [numberFormatter stringFromNumber:@4.0]);
// Output: "en_US: 400%".

NSLocale *faLocale = [[NSLocale alloc] initWithLocaleIdentifier:@"fa_IR"];
[numberFormatter setLocale:faLocale];
NSLog(@"fa_IR: %@", [numberFormatter stringFromNumber:@4.0]);
// Output: "fa_IR: □□۰۰۴٪□.□"
```

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”).

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

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.

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.

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`'s `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]
```

When a cell with a formatter object is copied, the new cell makes a strong reference to the formatter object rather than copying it.

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 delegate 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:`.

Creating a Custom Formatter

You can create custom subclasses of `NSFormatter` to format representations of data other than dates and numbers.

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

- `stringForObjectValue:`
- `getObjectValue:forString:errorDescription:`

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.

You may also override `attributedStringForObjectValue:withDefaultAttributes:` to 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:`.

In OS X, you can edit the textual contents of a cell at each key press 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 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
2014-02-11	Updated links to ICU documentation.
2012-09-19	Updated links to TR35 for recent OS releases.
2011-08-01	Editorial updates.
2011-03-08	Significant update to the Date Formatting article.
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 OS X v10.4".
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 OS X v10.4.
2004-08-31	Added descriptions of NSString format specifiers `%qx` and `%qX` to Formatting String Objects.

Date	Notes
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.

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

APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS DOCUMENT, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS DOCUMENT IS PROVIDED "AS IS," AND YOU, THE READER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT, ERROR OR INACCURACY IN THIS DOCUMENT, even if advised of the possibility of such damages.

Some jurisdictions do not allow the exclusion of implied warranties or liability, so the above exclusion may not apply to you.