Dates and Times Programming Topics for Cocoa

Cocoa > Data Management

2007-09-04

Contents

Introduction to Dates and Times Programming Topics for Cocoa

This topic describes how to handle dates and times.

Who Should Read This Document

You should read this document to learn how to create and use dates and times with Cocoa.

Organization of This Document

These articles describe dates, times, and calendars in Cocoa:

- "Dates" (page 7)
- "Calendars" (page 11)
- "Using Time Zones" (page 13)
- "Converting Dates to Strings" (page 15)

Dates

This article describes the classes you use to represent dates in Cocoa, and the relationships between them.

Dates

Cocoa gives you two ways to represent dates and times.

- A date object is useful for representing a date users don't see. It has methods for creating dates, comparing dates, and computing intervals. This is implemented by `NSDate`.
- A Gregorian date object, a special type of date object, is useful for representing dates users do see. It adds methods for converting dates to strings, converting strings to dates, and retrieving elements from dates (such as hours, minutes, and the day of the week). This is implemented by `NSCalendarDate` in Objective-C and by `com.apple.cocoa.foundation.NSGregorianDate` in Java.

All kinds of date objects are immutable since they represent a invariant point in time.

Like various other Foundation classes, `NSDate` enables you to obtain operating-system functionality (dates and times) without depending on operating-system internals. It also provides a basis for the `NSRunLoop` and `NSTimer` classes, which use concrete date objects to implement local event loops and timers.

"Date" as used here implies clock time as well. Dates store their times as the number of seconds relative to an absolute reference time: the first instant of 1 January, 2001, Greenwich Mean Time (GMT). Dates before then are stored as negative numbers; dates after then are stored as positive numbers. The standard unit of time for date objects is a value typed as double in Java and `NSTimeInterval` in Objective-C and is expressed as seconds. These types makes possible a wide and fine-grained range of date and time values, giving precision within milliseconds for dates 10,000 years apart.

`NSDate` and its subclasses compute time as seconds relative to the absolute reference date. The sole primitive method of `NSDate`, `timeIntervalSinceReferenceDate`, provides the basis for all the other methods in the `NSDate` interface. `NSDate` converts all date and time representations to and from `NSTimeInterval` values that are relative to the absolute reference date. A positive interval relative to a date represents a point in the future, a negative interval represents a time in the past.

You can use the time zone associated with a date to change how the date prints its time interval. The time zone does not change how the time interval is stored. Because the value is stored independent of the time zone, you can accurately compare Gregorian dates with any other date objects or use them to create other date objects. It also means that you can track a date across different time zones; that is, you can create a new Gregorian date with a different time zone to see how the particular date is represented in that time zone.

Mac OS X implements time according to the Network Time Protocol (NTP) standard, which is based on Coordinated Universal Time. The current private implementations of `NSDate` follow the NTP standard. However, they do not account for leap seconds and therefore are not synchronized with International Atomic Time (the most accurate).

Creating Date Objects

If you want to store the current time, use the date class method `currentTimeIntervalSinceReferenceDate` to create the date object. If you want to store some time other than the current time, use one of the Java constructors or Objective-C `dateWithTimeInterval...` methods.

The constructors or `dateWithTimeInterval...` methods create date objects relative to a particular time, which the method name describes. You specify (in seconds) how much more recent or how much more in the past you want your date object to be. To specify a date that occurs earlier than the method's reference date, use a negative number of seconds.

The Objective-C code fragment below defines two date objects. tomorrow is exactly 24 hours from the current date and time, and yesterday is exactly 24 hours earlier than the current date and time.

```
NSTimeInterval secondsPerDay = 24 * 60 * 60;
NSDate *tomorrow = [NSDate
            dateWithTimeIntervalSinceNow:secondsPerDay];
NSDate *yesterday = [NSDate
            dateWithTimeIntervalSinceNow:-secondsPerDay];
```

To get new date objects with date-and-time values adjusted from existing date objects, use `addTimeInterval:`.

```
NSTimeInterval secondsPerDay = 24 * 60 * 60;
NSDate *today = [NSDate date];
NSDate *tomorrow, yesterday;

tomorrow = [today addTimeInterval:secondsPerDay];
yesterday = [today addTimeInterval:-secondsPerDay];
```

Basic Date Calculations

To compare dates, use the `isEqualToDate:`, `compare:`, `laterDate:`, and `earlierDate:` methods. These methods perform exact comparisons, which means they will detect subsecond differences between dates. You might want to compare dates with a less fine granularity. For example, you might want to consider two dates equal if they are within a minute of each other. If this is the case, use `timeIntervalSinceDate:` to compare the two dates or use a Gregorian date instead (`NSCalendarDate` in Objective-C, `NSGregorianDate` in Java). The following Objective-C code shows how to use `timeIntervalSinceDate:` to see if two dates are within one minute (60 seconds) of each other.

```
if (fabs([date2 timeIntervalSinceDate:date1]) < 60) ...
```

To obtain the difference between a date object and another point in time, you can send a `timeInterval...` message to the date object. For instance, `timeIntervalSinceNow` gives you the time, in seconds, between the current time and the receiving date object.

To retrieve conventional elements of an Gregorian date, use the ...`Of`... methods. For example, `dayOfWeek` returns a number that indicates the day of the week (0 is Sunday). The `monthOfYear` method returns a number between 1 and 12 that indicates the month. If you are using Mac OS X v10.4 or later, you can use a calendar object for more complicated calculations to determine, for example, how many weeks or months there are between two dates—see "Calendars" (page 11).

Calendars

Calendars encapsulate information about systems of reckoning time in which the beginning, length, and divisions of a year are defined. They provide information about the calendar and support for calendrical computations such as determining the range of a given calendrical unit and adding units to a given absolute time. This article describes the basic features of the `NSCalendar` class.

In Mac OS X version 10.4 and later, the `NSCalendar` class provides an Objective-C implementation of calendars for Cocoa. `NSCalendar` is closely associated with the `NSDateComponents` class, instances of which describe the components required for calendrical computations.

In a calendar, day, week, weekday, month, and year numbers are generally 1-based, but there may be calendar-specific exceptions. Ordinal numbers, where they occur, are 1-based. Some calendars represented by this API may have to map their basic unit concepts into year/month/week/day/… nomenclature. For example, a calendar composed of 4 quarters in a year instead of 12 months uses the "month" unit to represent quarters. The particular values of the unit are defined by each calendar, and are not necessarily "consistent with" or have a "correspondence with," values for that unit in another calendar.

Creating a Calendar

You create a calendar object by specifying an identifier for the calendar you want. Mac OS X provides data for several different calendars—Buddhist, Chinese, Gregorian, Hebrew, Islamic, and Japanese—specified by constants in `NSLocale`. You can get the calendar for the user's preferred locale (or in general from any `NSLocale` object) using the key `NSLocaleCalendar`, or most easily using the `NSCalendar` method `currentCalendar`. The following code fragment shows how to create a calendar object for the Japanese calendar, and for the current user:

```
NSCalendar *japaneseCalendar = [[NSCalendar alloc]
        initWithCalendarIdentifier:NSJapaneseCalendar];

NSCalendar *usersCalendar = [[NSCalendar alloc] initWithCalendarIdentifier:
        [[NSLocale currentLocale] objectForKey:NSLocaleCalendar]];

NSCalendar *currentCalendar = [NSCalendar currentCalendar];
```

Here, `usersCalendar` and `currentCalendar` are equal, although they are different objects.

Calendrical Calculations

To do calendar arithmetic, you use NSDate objects in conjunction with a calendar. For example, to convert between a decomposed date in one calendar and another calendar, you must first convert the decomposed elements to a date using the first calendar, then decompose it using the second. NSDate provides the absolute scale and epoch for dates and times, which can then be rendered into a particular calendar, for calendrical computations or user display.

Two NSCalendar methods—dateFromComponents: and dateByAddingComponents:toDate:options:—take as a parameter an NSDateComponents object that describes the calendrical components required for the computation. You can provide as many components as you need (or choose to). When there is incomplete information to compute an absolute time, default values similar to 0 and 1 are usually chosen by a calendar, but this is a calendar-specific choice. If you provide inconsistent information, calendar-specific disambiguation is performed (which may involve ignoring one or more of the parameters).

The methods (components:fromDate: and components:fromDate:toDate:options:) take a bit mask parameter that specifies which components to calculate when returning an NSDateComponents object. The bit mask is composed of NSCalendarUnit constants. You can use components:fromDate:toDate:options: to conveniently determine the temporal difference between two dates in units other than seconds (which you could calculate with the NSDate method, timeIntervalSinceDate:). The following code fragment shows how to get the approximate number of months and days between two dates using a Gregorian calendar:

```
NSDate *startDate = ...;
NSDate *endDate = ...;

NSCalendar *gregorian = [[NSCalendar alloc]
        initWithCalendarIdentifier:NSGregorianCalendar];

unsigned int unitFlags = NSMonthCalendarUnit | NSDayCalendarUnit;

NSDateComponents *components = [gregorian components:unitFlags
                                          fromDate:startDate
                                          toDate:endDate options:0];
int months = [components month];
int days = [components day];
```

It is important to note that an instance of NSDateComponents is meaningless in itself; you need to know what *calendar* it is interpreted against, and you need to know whether the values are absolute values of the units, or quantities of the units. Note also that there are differences in the way that NSCalendar's Gregorian calendar (NSGregorianCalendar) and NSCalendarDate interpret components—for example, NSCalendar's Gregorian calendar's week (interpreted using NSDateComponents's weekday) starts with Sunday = 1, whereas NSCalendarDate's week (see dayOfWeek) starts with Sunday = 0.

Using Time Zones

NSTimeZone is an abstract class that defines the behavior of time zone objects. Time zone objects represent geopolitical regions. Consequently, these objects have names for these regions. Time zone objects also represent a temporal offset, either plus or minus, from Greenwich Mean Time (GMT) and an abbreviation (such as "PST").

NSTimeZone provides several methods to make time zone objects. In Java, you use the constructors. In Objective-C, you use the class methods timeZoneWithName:, timeZoneWithAbbreviation:, timeZoneForSecondsFromGMT:. The most flexible method is timeZoneWithName:. The name may be in any of the formats understood by the system, for example "EST", "Etc/GMT-2", "America/Argentina/Buenos_Aires", "Europe/Monaco", "US/Pacific", or "posixrules", as shown in the following code fragment:

```
NSTimeZone *timeZoneEST = [NSTimeZone timeZoneWithName:@"EST"];
NSTimeZone *timeZoneBuenos_Aires =
        [NSTimeZone timeZoneWithName:@"America/Argentina/Buenos_Aires"];
NSTimeZone *timeZonePosix =
        [NSTimeZone timeZoneWithName:@"posixrules"];
```

If you use timeZoneWithAbbreviation:, you can use only abbreviations such as "EST". In the following code fragment, timeZoneEST will be initialized correctly, whereas timeZoneUSPacific will be nil.

```
NSTimeZone *timeZoneEST = [NSTimeZone timeZoneWithAbbreviation:@"EST"];
NSTimeZone *timeZoneUSPacific =
        [NSTimeZone timeZoneWithAbbreviation:@"US/Pacific"];
// timeZoneUSPacific = nil
```

For a complete list of time zone names and abbreviations known to the system, you can see the output of CFTimeZoneCopyKnownNames, as shown in the following code fragment:

```
#import <CoreFoundation/CoreFoundation.h>

NSString *timeZoneInformation = (NSString *)CFTimeZoneCopyKnownNames();
NSLog(@"timeZoneInformation: %@", timeZoneInformation);
```

The class also permits you to set the default time zone within your application (setDefaultTimeZone:). You can access this default time zone at any time with the defaultTimeZone class method, and with the localTimeZone class method, you can get a relative time zone object that decodes itself to become the default time zone for any locale in which it finds itself.

Some Gregorian date methods return date objects that are automatically bound to time zone objects. These date objects use the functionality of NSTimeZone to adjust dates for the proper locale. Unless you specify otherwise, objects returned from Gregorian date are bound to the default time zone for the current locale.

Converting Dates to Strings

The first section, "String Representations of NSDate Objects" (page 15), describes some formatting methods for `NSDate`. The last two sections, "The Calendar Format" (page 16) and "String Representations of NSCalendarDate Objects" (page 17), describes methods for formatting `NSCalendarDate`.

Important: The preferred way to convert a date to a string, or a string to a date, is to use a date formatter, as described in *Data Formatting Programming Guide for Cocoa*. For the sake of old code that was written before date formatters were available, the Objective-C interfaces for `NSDate` and `NSCalendarDate` still contain some methods for string conversions.

String Representations of NSDate Objects

To represent your date object as an `NSString` object, use the `description...` methods. The simplest method, `description`, prints out the date in the format YYYY-MM-DD HH:MM:SS ±HHMM, where ±HHMM represents the time zone offset in hours and minutes from GMT. (Adding the offset to the specific time yields the equivalent GMT.) To have a specific locale dictionary affect the representation of your `NSDate` object, use `descriptionWithLocale:` instead of `description`. The following keys in the locale dictionary affect `NSDate` objects:

Key	Description
`NSTimeDateFormatString`	Specifies how dates with times are printed. The default is to use full month names and days with a 24 hour clock, as in "Sunday, January 01, 2001 23:00:00 Pacific Standard Time."
`NSAMPMDesignation`	Specifies how the morning and afternoon designations are printed. The default is AM and PM.
`NSMonthNameArray`	Specifies the names for the months.
`NSShortMonthNameArray`	Specifies the abbreviations for the months.
`NSWeekDayNameArray`	Specifies the names for the days of the week.
`NSShortWeekDayNameArray`	Specifies the abbreviations for the days of the week.

Note that `NSDate`'s implementation of the `description` method uses `NSCalendarDate`. `NSCalendarDate` does not model the transition from the Julian to the Gregorian calendar, so using `NSDate`'s `description` method yields inaccurate results for dates earlier than October 1582. If you need to describe dates earlier than the transition, you should use `NSDateFormatter` (as described in *Data Formatting Programming Guide for Cocoa*).

The Calendar Format

Each `NSCalendarDate` object has a calendar format associated with it. This format is a string that contains date-conversion specifiers that are very similar to those used in the standard C library function `strftime()`. `NSCalendarDate` interprets dates that are represented as strings conforming to this format. You can set the default format for an `NSCalendarDate` object at initialization time or using the `setCalendarFormat:` method. Several methods allow you to specify formats other than the one bound to the object.

The date conversion specifiers cover a range of date conventions:

Conversion 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
`%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
`%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
`%X`	time using the time representation for the locale
`%y`	year without century (00-99)
`%Y`	year with century (such as 1990)

Conversion Specifier	Description
`%Z`	time zone name (such as Pacific Daylight Time)
`%z`	time zone offset in hours and minutes from GMT (HHMM)

Note that most of the formats that specify numeric values pad the width of the value with `0`s (for example, `%S` represents 6 seconds as `06`). You can suppress these `0`s using a width specifier, just as you would with, for example, `printf`, as illustrated by the following example.

```
NSCalendarDate *date = [NSCalendarDate dateWithYear:2006 month:2 day:2
                                hour:2 minute:2 second:2
                                timeZone:nil];
[date setCalendarFormat:@"%m (%1m), %d (%1d), %I (%1I), %j (%1j), %S (%1S)"];
NSLog(@"date: %@", date);

// Output: date: 02 (2), 02 (2), 02 (2), 033 (33), 02 (2)
```

String Representations of NSCalendarDate Objects

`NSCalendarDate` provides several `description...` methods for representing dates as strings. These methods—`description`, `descriptionWithLocale:`, `descriptionWithCalendarFormat:`, and `descriptionWithCalendarFormat:locale:`—take an implicit or explicit calendar format. The user's locale information affects the returned string. `NSCalendarDate` accesses the locale information as an `NSDictionary` object. If you use `descriptionWithLocale:` or `descriptionWithCalendarFormat:locale:`, you can specify a different locale dictionary. The following keys in the locale dictionary affect `NSCalendarDate`:

Locale Key	Description
`NSTimeDateFormatString`	Specifies how dates with times are printed, affecting strings that use the format specifiers `%c`, `%X`, or `%x`. The default is to use abbreviated months and days with a 24 hour clock, as in "Sun Jan 01 23:00:00 +6 2001".
`NSAMPMDesignation`	Specifies how the morning and afternoon designations are printed, affecting strings that use the `%p` format specifier. The default is AM and PM.
`NSMonthNameArray`	Specifies the names for the months, affecting strings that use the `%B` format specifier.
`NSShortMonthNameArray`	Specifies the abbreviations for the months, affecting strings that use the `%b` format specifier.
`NSWeekDayNameArray`	Specifies the names for the days of the week, affecting strings that use the `%A` format specifier.
`NSShortWeekDayNameArray`	Specifies the abbreviations for the days of the week, affecting strings that use the `%a` format specifier.

If you subclass `NSCalendarDate` and override `description`, you should also override `descriptionWithLocale:`. The `stringWithFormat:` method of `NSString` uses `descriptionWithLocale:` instead of description when you use the `%@` conversion specifier to print an `NSCalendarDate`. That is, this message:

```
[NSString stringWithFormat:@"The current date and time are %@",
        [MyNSCalendarDateSubclass date]]
```

invokes `descriptionWithLocale:`.

Document Revision History

This table describes the changes to *Dates and Times Programming Topics for Cocoa*.

Date	Notes
2007-09-04	Enhanced discussion of calendrical calculations using NSDateComponents.
2007-03-06	Added note regarding Julian and Gregorian calendars.
2006-05-23	Corrected typographical errors. Added a note about the use of width specifiers for calendar date format strings.
2006-02-07	Updated to include NSCalendar and NSDateFormatter changes introduced in Mac OS X v10.4.
2005-08-11	Changed title from "Dates and Times." Corrected minor typographic error.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.