Date and Time Programming Guide

Developer

Contents

Tables and Listings

About Dates and Times

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.

Date and time objects allow you to store references to particular instances in time. You can use date and time objects to perform calculations and comparisons that account for the corner cases of date and time calculations.

At a Glance

There are three main classes used for working with dates and times.

- `NSDate` allows you to represent an absolute point in time.
- `NSCalendar` allows you to represent a particular calendar, such as a Gregorian or Hebrew calendar. It provides the interface for most date-based calculations and allows you to convert between `NSDate` objects and `NSDateComponents` objects.
- `NSDateComponents` allows you to represent the components of a particular date, such as hour, minute, day, year, and so on.

In addition to these classes, `NSTimeZone` allows you to represent a geopolitical region's time zone information. It eases the task of working across different time zones and performing calculations that may be affected by daylight savings time transitions.

Creating and Using Date Objects to Represent Absolute Points in Time

Date objects represent dates and times in Cocoa. Date objects allow you to store absolute points in time which are meaningful across locales, calendars and timezones.

Relevant Chapters: Dates (page 8)

Working with Calendars and Date Components

Date components allow you to break a date down into the various parts that comprise it, such as day, month, year, hour, and so on. Calendars represent a particular form of reckoning time, such as the Gregorian calendar or the Chinese calendar. Calendar objects allow you to convert between date objects and date component objects, as well as from one calendar to another.

Relevant Chapters: Calendars, Date Components, and Calendar Units (page 11)

Performing Date and Time Calculations

Calendars and date components allow you to perform calculations such as the number of days or hours between two dates or finding the Sunday in the current week. You can also add components to a date or check when a date falls.

Relevant Chapters: Performing Calendar Calculations (page 16)

Working with Different Time Zones

Time zone objects allow you to present absolute times as local—that is, wall clock—time. In addition to time offsets, they also keep track of daylight saving time differences. Proper use of time zone objects can avoid issues such as miscalculation of elapsed time due to daylight saving time transitions or the user moving to a different time zone.

Relevant Chapters: Using Time Zones (page 23)

Special Considerations for Historical Dates

Dates in the past have a number of edge cases that do not exist for contemporary dates. These include issues such as dates that do not exist in a particular calendar—such as the lack of the year 0 in the Gregorian calendar—or calendar transitions—such as the Julian to Gregorian transition in the Middle Ages. There are also eras with seemingly backward time flow—such as BC dates in the Gregorian calendar.

Relevant Chapters: Historical Dates (page 26)

How to Use this Document

If your application keeps track of dates and times, read from Dates (page 8) to Using Time Zones (page 23). The `NSDate`, `NSCalendar`, `NSDateComponents`, and `NSTimeZone` classes described in these chapters work together to store, compare, and manipulate dates and times.

If your application deals with dates in the past—particularly prior to the early 1900s, also read Historical Dates (page 26) to learn about some of the issues that can arise when dealing with dates in the past.

See Also

If you display dates and times to users or create dates from user input, read:

- *Data Formatting Guide*, which explains how to create and format user-readable strings from date objects, and how to create date objects from formatted strings.

Dates

Date objects allow you to represent dates and times in a way that can be used for date calculations and conversions. As absolute points in time, date objects are meaningful across locales, timezones, and calendars.

Date Fundamentals

Cocoa represents dates and times as `NSDate` objects. `NSDate` is one of the fundamental Cocoa value objects. A date object represents an invariant point in time. Because a date is a point in time, it implies clock time as well as a day, so there is no way to define a date object to represent a day without a time.

To understand how Cocoa handles dates, you must consider `NSCalendar` and `NSDateComponents` objects as well. In a nontechnical context, a point in time is usually represented by a combination of a clock time and a day on a particular calendar (such as the Gregorian or Hebrew calendar). Supporting different calendars is important for localization. In Cocoa, you use a particular calendar to decompose a date object into its date components such as year, month, day, hour, and minute. Conversely, you can use a calendar to create a date object from date components. Calendar and date component objects are described in more detail in Calendars, Date Components, and Calendar Units (page 11).

`NSDate` provides methods for creating dates, comparing dates, and computing intervals. Date objects are immutable. The standard unit of time for date objects is floating point value typed as `NSTimeInterval` and is expressed in seconds. This type makes possible a wide and fine-grained range of date and time values, giving precision within milliseconds for dates 10,000 years apart.

`NSDate` computes time as seconds relative to an absolute reference time: the first instant of January 1, 2001, Greenwich Mean Time (GMT). Dates before then are stored as negative numbers; dates after then are stored as positive numbers. 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.

Cocoa implements time according to the Network Time Protocol (NTP) standard, which is based on Coordinated Universal Time.

Creating Date Objects

If you want a date that represents the current time, you allocate an `NSDate` object and initialize it with `init`:

```
NSDate *now = [[NSDate alloc] init];
```

or use the `NSDate` class method `date` to create the date object. If you want some time other than the current time, you can use one of `NSDate`'s `initWithTimeInterval...` or `dateWithTimeInterval...` methods; typically, however, you use a more sophisticated approach employing a calendar and date components as described in Calendar Basics (page 11).

The `initWithTimeInterval...` methods initialize 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.

Listing 1 defines two date objects. The `tomorrow` object is exactly 24 hours from the current date and time, and `yesterday` is exactly 24 hours earlier than the current date and time.

Listing 1 Creating dates with time intervals

```
NSTimeInterval secondsPerDay = 24 * 60 * 60;
NSDate *tomorrow = [[NSDate alloc]
            initWithTimeIntervalSinceNow:secondsPerDay];
NSDate *yesterday = [[NSDate alloc]
            initWithTimeIntervalSinceNow:-secondsPerDay];
[tomorrow release];
[yesterday release];
```

Listing 2 shows how to get new date objects with date-and-time values adjusted from existing date objects using `dateByAddingTimeInterval:`.

Listing 2 Creating dates by adding a time interval

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

tomorrow = [today dateByAddingTimeInterval: secondsPerDay];
```

```
yesterday = [today dateByAddingTimeInterval: -secondsPerDay];
[today release];
```

Basic Date Calculations

To compare dates, you can use the `isEqualToDate:`, `compare:`, `laterDate:`, and `earlierDate:` methods. These methods perform exact comparisons, which means they detect sub-second differences between dates. You may want to compare dates with a less fine granularity. For example, you may 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. The following code fragment 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, send a `timeIntervalSince...` message to the date object. For example, `timeIntervalSinceNow` gives you the time, in seconds, between the current time and the receiving date object.

To get the component elements of a date, such as the day of the week, use an `NSDateComponents` object in conjunction with an `NSCalendar` object. This technique is described in Calendar Basics (page 11).

Calendars, Date Components, and Calendar Units

Calendar objects encapsulate information about systems of reckoning time in which the beginning, length, and divisions of a year are defined. You use calendar objects to convert between absolute times and date components such as years, days, or minutes.

Calendar Basics

`NSCalendar` provides an implementation of various calendars. It provides data for several different calendars, including Buddhist, Gregorian, Hebrew, Islamic, and Japanese (which calendars are supported depends on the release of the operating system—check the `NSLocale` class to determine which are supported on a given release). `NSCalendar` is closely associated with the `NSDateComponents` class, instances of which describe the component elements of a date required for calendrical computations.

Calendars are specified by constants in `NSLocale`. You can get the calendar for the user's preferred locale most easily using the `NSCalendar` method `currentCalendar`; you can get the default calendar from any `NSLocale` object using the key `NSLocaleCalendar`. You can also create an arbitrary calendar object by specifying an identifier for the calendar you want. Listing 3 shows how to create a calendar object for the Japanese calendar and for the current user.

Listing 3 Creating calendar objects

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

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

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

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

Date Components and Calendar Units

You represent the component elements of a date—such as the year, day, and hour—using an `NSDateComponents` object. An `NSDateComponents` object can hold either absolute values or quantities of units (see Adding Components to a Date (page 16) for an example of using `NSDateComponents` to specify quantities of units). For date components objects to be meaningful, you need to know the associated calendar and purpose.

> **iOS Note:** In iOS 4.0 and later, `NSDateComponents` objects can contain a calendar, a timezone, and a date object. This allows date components to be passed to or returned from a method and retain their meaning.

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 may have to map their basic unit concepts into the year/month/week/day/… nomenclature. The particular values of the unit are defined by each calendar and are not necessarily consistent with values for that unit in another calendar.

Listing 4 shows how you can create a date components object that you can use to create the date where the year unit is 2004, the month unit is 5, and the day unit is 6 (in the Gregorian calendar this is May 6th, 2004). You can also use it to add 2004 year units, 5 month units, and 6 day units to an existing date. The value of `weekday` is undefined since it is not otherwise specified.

Listing 4 Creating a date components object

```
NSDateComponents *components = [[NSDateComponents alloc] init];
[components setDay:6];
[components setMonth:5];
[components setYear:2004];

NSInteger weekday = [components weekday]; // Undefined (== NSUndefinedDateComponent)
```

Converting between Dates and Date Components

To decompose a date into constituent components, you use the `NSCalendar` method `components:fromDate:`. In addition to the date itself, you need to specify the components to be returned in the `NSDateComponents` object. For this, the method takes a bit mask composed of `Calendar Units` constants. There is no need to specify any more components than those in which you are interested. Listing 5 shows how to calculate today's day and weekday.

Listing 5 Getting a date's components

```
NSDate *today = [NSDate date];
NSCalendar *gregorian = [[NSCalendar alloc]
                         initWithCalendarIdentifier:NSGregorianCalendar];
NSDateComponents *weekdayComponents =
                  [gregorian components:(NSDayCalendarUnit | NSWeekdayCalendarUnit)
 fromDate:today];
NSInteger day = [weekdayComponents day];
NSInteger weekday = [weekdayComponents weekday];
```

This gives you the absolute components for a date. For example, if you ask for the year and day components for November 7, 2010, you get 2010 for the year and 7 for the day. If you instead want to know what number day of the year it is you can use the `ordinalityOfUnit:inUnit:forDate:` method of the `NSCalendar` class.

It is also possible to create a date from components. You can configure an instance of `NSDateComponents` to specify the components of a date and then use the `NSCalendar` method `dateFromComponents:` to create the corresponding date object. You can provide as many components as you need (or choose to). When there is incomplete information to compute an absolute time, default values such as `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).

Listing 6 shows how to create a date object to represent (in the Gregorian calendar) the first Monday in May, 2008.

Listing 6 Creating a date from components

```
NSDateComponents *components = [[NSDateComponents alloc] init];
[components setWeekday:2]; // Monday
[components setWeekdayOrdinal:1]; // The first Monday in the month
[components setMonth:5]; // May
[components setYear:2008];
NSCalendar *gregorian = [[NSCalendar alloc]
                         initWithCalendarIdentifier:NSGregorianCalendar];
NSDate *date = [gregorian dateFromComponents:components];
```

To guarantee correct behavior you must make sure that the components used make sense for the calendar. Specifying "out of bounds" components—such as a day value of `-6` or February 30th in the Gregorian calendar—produce undefined behavior.

You may want to create a date object without components such as years—to store your friend's birthday, for instance. While it is not technically possible to create a yearless date, you can use date components to create a date object without a specified year, as in Listing 7.

Listing 7 Creating a yearless date

```
NSDateComponents *components = [[NSDateComponents alloc] init];
[components setMonth:11];
[components setDay:7];
NSCalendar *gregorian = [[NSCalendar alloc]
                         initWithCalendarIdentifier:NSGregorianCalendar];
NSDate *birthday = [gregorian dateFromComponents:components];
```

Note that `birthday` in this instance has the default value for the year, which in this case is 1 AD (though it is not guaranteed to always default to 1 AD). If you later convert this date back to components, or use an `NSDateFormatter` object to display it, make sure to not use the year value (as your friend may not appreciate being listed as that old). You can use the `NSDateFormatter` `dateFormatFromTemplate:options:locale:` method to create a yearless date formatter that adjusts to the users locale. For more information on date formatting see *Data Formatting Guide*.

Warning: Mixing and matching week-based calendar constants (`NSWeekOfMonthCalendarUnit`, `NSWeekOfYearCalendarUnit`, and `NSYearForWeekOfYearCalendarUnit`) with other week-oriented calendar constants (defined in the `Calendar Units` constants) will result in errors. Specifically, the calendar component `NSYearCalendarUnit` defines the ordinary calendar year, not the year in a week-based calendar. Using an `NSCalendar` with a calendar year and a weekday when using a calendar created using the week-based calendar constants results in ambiguous dates.

Converting from One Calendar to Another

To convert components of a date from one calendar to another—for example, from the Gregorian calendar to the Hebrew calendar—you first create a date object from the components using the first calendar, then you decompose the date into components using the second calendar. Listing 8 shows how to convert date components from one calendar to another.

Listing 8 Converting date components from one calendar to another

```
NSDateComponents *comps = [[NSDateComponents alloc] init];
[comps setDay:6];
[comps setMonth:5];
[comps setYear:2004];

NSCalendar *gregorian = [[NSCalendar alloc]
                         initWithCalendarIdentifier:NSGregorianCalendar];
NSDate *date = [gregorian dateFromComponents:comps];
[comps release];
[gregorian release];

NSCalendar *hebrew = [[NSCalendar alloc]
                        initWithCalendarIdentifier:NSHebrewCalendar];
NSUInteger unitFlags = NSDayCalendarUnit | NSMonthCalendarUnit |
                               NSYearCalendarUnit;
NSDateComponents *components = [hebrew components:unitFlags fromDate:date];

NSInteger day = [components day]; // 15
NSInteger month = [components month]; // 9
NSInteger year = [components year]; // 5764
```

Performing Calendar Calculations

`NSDate` provides the absolute scale and epoch for dates and times, which can then be rendered into a particular calendar for calendrical calculations or user display. To perform calendar calculations, you typically need to get the component elements of a date, such as the year, the month, and the day. You should use the provided methods for dealing with calendrical calculations because they take into account corner cases like daylight savings time starting or ending and leap years.

Adding Components to a Date

You use the `dateByAddingComponents:toDate:options:` method to add components of a date (such as hours or months) to an existing date. You can provide as many components as you wish. Listing 9 shows how to calculate a date an hour and a half in the future.

Listing 9 An hour and a half from now

```
NSDate *today = [[NSDate alloc] init];
NSCalendar *gregorian = [[NSCalendar alloc]
          initWithCalendarIdentifier:NSGregorianCalendar];
NSDateComponents *offsetComponents = [[NSDateComponents alloc] init];
[offsetComponents setHour:1];
[offsetComponents setMinute:30];
// Calculate when, according to Tom Lehrer, World War III will end
NSDate *endOfWorldWar3 = [gregorian dateByAddingComponents:offsetComponents
          toDate:today options:0];
```

Components to add can be negative. Listing 10 shows how you can get the Sunday in the current week (using a Gregorian calendar).

Listing 10 Getting the Sunday in the current week

```
NSDate *today = [[NSDate alloc] init];
NSCalendar *gregorian = [[NSCalendar alloc]
```

```
          initWithCalendarIdentifier:NSGregorianCalendar];


// Get the weekday component of the current date
NSDateComponents *weekdayComponents = [gregorian components:NSWeekdayCalendarUnit
          fromDate:today];


/*
Create a date components to represent the number of days to subtract from the
current date.
The weekday value for Sunday in the Gregorian calendar is 1, so subtract 1 from
the number of days to subtract from the date in question.  (If today is Sunday,
subtract 0 days.)
*/
NSDateComponents *componentsToSubtract = [[NSDateComponents alloc] init];
[componentsToSubtract setDay: 0 - ([weekdayComponents weekday] - 1)];


NSDate *beginningOfWeek = [gregorian dateByAddingComponents:componentsToSubtract
          toDate:today options:0];


/*
Optional step:
beginningOfWeek now has the same hour, minute, and second as the original date
(today).
To normalize to midnight, extract the year, month, and day components and create
a new date from those components.
*/
NSDateComponents *components =
     [gregorian components:(NSYearCalendarUnit | NSMonthCalendarUnit |
          NSDayCalendarUnit) fromDate: beginningOfWeek];
beginningOfWeek = [gregorian dateFromComponents:components];
```

Sunday is not the beginning of the week in all locales. Listing 11 illustrates how you can calculate the first moment of the week (as defined by the calendar's locale):

Listing 11 Getting the beginning of the week

```
NSDate *today = [[NSDate alloc] init];
```

```
NSDate *beginningOfWeek = nil;
BOOL ok = [gregorian rangeOfUnit:NSWeekCalendarUnit startDate:&beginningOfWeek
                    interval:NULL forDate: today];
```

Determining Temporal Differences

There are a few ways to calculate the amount of time between dates. Depending on the context in which the calculation is made, the user likely expects different behavior. Whichever calculation you use, it should be clear to the user how the calculation is being performed. Since Cocoa implements time according to the NTP standard, these methods ignore leap seconds in the calculation. You use `components:fromDate:toDate:options:` to determine the temporal difference between two dates in units other than seconds (which you can calculate with the `NSDate` method `timeIntervalSinceDate:`). Listing 12 shows how to get the number of months and days between two dates using a Gregorian calendar.

Listing 12 Getting the difference between two dates

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

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

NSUInteger unitFlags = NSMonthCalendarUnit | NSDayCalendarUnit;

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

This method handles overflow as you may expect. If the `fromDate:` and `toDate:` parameters are a year and 3 days apart and you ask for only the days between, it returns an `NSDateComponents` object with a value of 368 (or 369 in a leap year) for the day component. However, this method truncates the results of the calculation to the smallest unit supplied. For instance, if the `fromDate:` parameter corresponds to Jan 14, 2010 at 11:30 PM and the `toDate:` parameter corresponds to Jan 15, 2010 at 8:00 AM, there are only 8.5 hours between the two dates. If you ask for the number of days, you get 0, because 8.5 hours is less than 1 day. There may be

situations where this should be 1 day. You have to decide which behavior your users expect in a particular case. If you do need to have a calculation that returns the number of days, calculated by the number of midnights between the two dates, you can use a category to `NSCalendar` similar to the one in Listing 13.

Listing 13 Days between two dates, as the number of midnights between

```
@implementation NSCalendar (MySpecialCalculations)
-(NSInteger)daysWithinEraFromDate:(NSDate *) startDate toDate:(NSDate *) endDate
{
     NSInteger startDay=[self ordinalityOfUnit:NSDayCalendarUnit
          inUnit: NSEraCalendarUnit forDate:startDate];
     NSInteger endDay=[self ordinalityOfUnit:NSDayCalendarUnit
          inUnit: NSEraCalendarUnit forDate:endDate];
     return endDay-startDay;
}
@end
```

This approach works for other calendar units by specifying a different `NSCalendarUnit` value for the `ordinalityOfUnit:` parameter. For example, you can calculate the number of years based on the number of times Jan 1, 12:00 AM is present between.

Do not use this method for comparing second differences because it overflows `NSInteger` on 32-bit platforms. This method is only valid if you stay within the same era (in the Gregorian Calendar this means that both dates must be AD or both must be BC). If you do need to compare dates across an era boundary you can use something similar to the category in Listing 14.

Listing 14 Days between two dates in different eras

```
@implementation NSCalendar (MyOtherMethod)
-(NSInteger) daysFromDate:(NSDate *) startDate toDate:(NSDate *) endDate
{
     NSCalendarUnit units=NSEraCalendarUnit | NSYearCalendarUnit |
          NSMonthCalendarUnit | NSDayCalendarUnit;
     NSDateComponents *comp1=[self components:units fromDate:startDate];
     NSDateComponents *comp2=[self components:units fromDate endDate];
     [comp1 setHour:12];
     [comp2 setHour:12];
     NSDate *date1=[self dateFromComponents: comp1];
```

```
    NSDate *date2=[self dateFromComponents: comp2];
    return [[self components:NSDayCalendarUnit fromDate:date1 toDate:date2
options:0] day];
}
@end
```

This method creates components from the given dates, and then normalizes the time and compares the two dates. This calculation is more expensive than comparing dates within an era. If you do not need to cross era boundaries use the technique shown in Listing 13 (page 19) instead.

Checking When a Date Falls

If you need to determine if a date falls within the current week (or any unit for that matter) you can make use of the `NSCalendar` method `rangeOfUnit:startDate:interval:forDate:`. Listing 15 shows a method that determines if a given date falls within this week. The week in this case is defined as the period between Sunday at midnight to the following Saturday just before midnight (in the Gregorian calendar).

Listing 15 Determining whether a date is this week

```
-(BOOL)isDateThisWeek:(NSDate *)date {
    NSDate *start;
    NSTimeInterval extends;
    NSCalendar *cal=[NSCalendar autoupdatingCurrentCalendar];
    NSDate *today=[NSDate date];
    BOOL success= [cal rangeOfUnit:NSWeekCalendarUnit startDate:&start
        interval: &extends forDate:today];
    if(!success)return NO;
    NSTimeInterval dateInSecs = [date timeIntervalSinceReferenceDate];
    NSTimeInterval dayStartInSecs= [start timeIntervalSinceReferenceDate];
    if(dateInSecs > dayStartInSecs && dateInSecs < (dayStartInSecs+extends)){
        return YES;
    }
    else {
        return NO;
    }
```

```
}
```

This code uses `NSTimeInterval` values for the date to test and the start of the week and uses those to determine whether the date is this week.

Week-Based Calendars

A week-based calendar is defined by the weeks of a year. Instead of the year, month, and day of a date, a week-based calendar is defined by the week-year, the week number, and a weekday.

However, this can be complicated when the first week of the calendar overlaps the last week of the previous year's calendar. In this case there are two important properties of the calendar:

1. What is the first day of the week?
2. How many days does a week near the beginning of the year have to have within the ordinary calendar year for it to be considered the first week in the week-based calendar year?

A week-based calendar's first day of the year is on the first day of the week. The first week is preferred to be the week containing Jan 1 if that week satisfies the defined answer for the second point above.

For example, suppose the first day of the week is defined as Monday, in a week-based calendar interpretation of the Gregorian calendar. Consider the 2009/2010 transition shown in Table 1 and Table 2:

Table 1 December 2009 Calendar

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
20	21	22	23	24	25	26
27	28	29	30	31		

Table 2 January 2010 Calendar

Sunday	Monday	Tuesday	Wednesday	Thursday	Friday	Saturday
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16

Since the first day of the week is Monday, the 2010 week-based calendar year can begin either December 28 or January 4. That is, December 30, 2009 (ordinary) could be December 30, 2010 (week-based).

To choose between these two possibilities, there is the second criterion. Week Dec 28 - Jan 3 has 3 days in 2010. Week Jan 4-Jan 10 has 7 days in 2010.

If the minimum number of days in a first week is defined as 1 or 2 or 3, the week of Dec 28 satisfies the first week criteria and would be week 1 of the week-based calendar year 2010. Otherwise, the week of Jan 4 is the first week.

As another example, suppose you wanted to define a week-based calendar such that the first week of the calendar year begins with the first occurrence of a specific weekday.

In Table 2 (page 21) Monday January 4 is the first Monday of the ordinary year, so the week-based calendar begins on that day. What you are requesting then is that the first week of your week-based calendar is entirely within the new ordinary year or that the minimum number of days in first week is 7.

The `NSYearForWeekOfYearCalendarUnit` is the year number of a week-based calendar interpretation of the calendar you're working with, where the two properties of the week-based calendar discussed in above correspond to these two NSCalendar properties: `firstWeekday` and `minimumDaysInFirstWeek`.

Warning: Mixing and matching week-based calendar constants (`NSWeekOfMonthCalendarUnit`, `NSWeekOfYearCalendarUnit`, and `NSYearForWeekOfYearCalendarUnit`) with other week-oriented calendar constants (defined in the `Calendar Units` constants) will result in errors. Specifically, the calendar component `NSYearCalendarUnit` defines the ordinary calendar year, not the year in a week-based calendar. Using an `NSCalendar` with a calendar year and a weekday when using a calendar created using the week-based calendar constants results in ambiguous dates.

Using Time Zones

Time zones can create numerous problems for applications. Consider the following situation. You are in New York and it is 12:30 AM. You have an application that displays all of the Major League Baseball games that happen tomorrow. Because tomorrow is different depending on the time zone, situations like this must be carefully accounted for. Fortunately, a little planning and the assistance of the `NSTimeZone` class ease this task considerably.

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

Creating Time Zones

Time zones affect the values of date components that are calculated by calendar objects for a given `NSDate` object. You can create an `NSTimeZone` object and use it to set the time zone of an `NSCalendar` object. By default, `NSCalendar` uses the default time zone for the application—or process—when the calendar object is created. Unless the default time zone has been otherwise set, it is the time zone set in System Preferences.

In most cases, the user's default time zone should be used when creating date objects. There are cases when it may be necessary to use arbitrary time zones. For example, the user may want to specify that an appointment is in Greenwich Mean Time, because it is during her business trip to London next week. `NSTimeZone` provides several class methods to make time zone objects: `timeZoneWithName:`, `timeZoneWithAbbreviation:`, and `timeZoneForSecondsFromGMT:`. In most cases `timeZoneWithName:` provides the most accurate time zone, as it adjusts for daylight saving time, the trade-off is that you must know more precisely the location you are creating a time zone for.

For a complete list of time zone names known to the system, you can use the `knownTimeZoneNames` class method:

```
NSArray *timeZoneNames = [NSTimeZone knownTimeZoneNames];
```

Application Default Time Zone

You can set the default time zone within your application using `setDefaultTimeZone:`. You can access this default time zone at any time with the `defaultTimeZone` class method. With the `localTimeZone` class method you can get a time zone object that automatically updates itself to reflect changes to the default time zone.

Creating Dates with Time Zones

Time zones play an important part in determining when dates take place. Consider a simple calendar application that keeps track of appointments. For example, say you live in Chicago and you have a dentist appointment coming up at 10:00 AM on Tuesday. You will be in New York for Sunday and Monday, however. When you created that appointment it was done with the mindset of an absolute time. That time is 10:00 AM Central Time; when you go to New York, the time should be presented as 11:00 AM because you are in a different time zone, but it is the same absolute time. On the other hand, if you create an appointment to wake up and exercise every morning at 7:00 AM, you do not want your alarm to go off at 1:00 PM simply because you are on a business trip to Dublin—or at 5:00 AM because you are in Los Angeles.

`NSDate` objects store dates in absolute time. For example, the `date` object created in Listing 16 represents 4:00 PM CDT, 5:00 EDT, and so on.

Listing 16 Creating a date from components using a specific time zone

```
NSCalendar *gregorian=[[NSCalendar alloc] initWithCalendarIdentifier:
NSGregorianCalendar];
[gregorian setTimeZone:[NSTimeZone timeZoneWithAbbreviation:@"CDT"]];
NSDateComponents *timeZoneComps=[[NSDateComponents alloc] init];
[timeZoneComps setHour:16];
//specify whatever day, month, and year is appropriate
NSDate *date=[gregorian dateFromComponents:timeZoneComps];
```

If you need to create a date that is independent of timezone, you can store the date as an `NSDateComponents` object—as long as you store some reference to the corresponding calendar.

In iOS, `NSDateComponents` objects can contain a calendar, a timezone, and a date object. You can therefore store the calendar along with the components. If you use the `date` method of the `NSDateComponents` class to access the date, make sure that the associated timezone is up-to-date.

Time Zones and Daylight Saving Time

The `NSTimeZone` class also provides a number of instance methods to determine information about daylight saving time:

- `isDaylightSavingTime` determines whether daylight saving time is currently in effect.
- `daylightSavingTimeOffset` determines the current daylight saving time offset. For most time zones this is either zero or one.
- `nextDaylightSavingTimeTransition` determines when the next daylight saving time transition occurs.

There are also similarly named methods for determining this information for specific dates. If you are keeping track of events and appointments in your application, you can use this information to remind the user of upcoming daylight saving time transitions.

Historical Dates

There are a number of issues that can arise when dealing with dates in the past that do not exist for contemporary dates. These include dates that do not exist, previous eras where time flow moves from higher year numbers to lower ones (such as BC dates in the Gregorian calendar), and calendar transitions (such as the transition from the Julian calendar to the Gregorian calendar).

The Gregorian Calendar Has No Year 0

In the Julian and Gregorian calendars represented by the `NSGregorianCalendar`, there is no year 0. This means that the day following December 31, 1 BC is January 1, 1 AD. All of the provided methods for calendrical calculations take this into account, but you may need to account for it when you are creating dates from components. If you do attempt to create a date with year 0, it is instead 1 BC. In addition, if you create a date from components using a negative year value, it is created using astronomical year numbering in which 0 corresponds to 1 BC, -1 corresponds to 2 BC, and so on. For example, the two dates created in Listing 17equivalently represent May 7, 8 BC.

Listing 17 Using negative years to represent BC dates

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

NSDateComponents *bcDateComp=[[NSDateComponents alloc] init];
[bcDate setMonth: 5];
[bcDate setDay: 7];
[bcDate setYear: 8];
[bcDate setEra: 0];

NSDateComponents *astronDateComp=[[NSDateComponents alloc] init];
[bcDate setMonth: 5];
[bcDate setDay: 7];
[bcDate setYear: -7];
```

```
NSDate *bcDate=[gregorian dateFromComponents:bcDateComp];
NSDate *astronDate=[gregorian dateFromComponents:astronDateComp];
```

The Julian to Gregorian Transition

`NSCalendar` models the transition from the Julian to Gregorian calendar in October 1582. During this transition, 10 days were skipped. This means that October 15, 1582 follows October 4, 1582. All of the provided methods for calendrical calculations take this into account, but you may need to account for it when you are creating dates from components. Dates created in the gap are pushed forward by 10 days. For example October 8, 1582 is stored as October 18, 1582.

Some countries adopted the Gregorian calendar at various later times. Nevertheless, for consistency the change is modeled at the same time regardless of locale. If you need absolute historical accuracy for a particular locale, you can subtract the appropriate number of days from the date given by the Gregorian calendar. The number of days to subtract corresponds to the number of extra leap days in the Julian calendar. Thus for every 100th year, the Julian calendar falls behind a day if that year is not a multiple of 400. If you need to create a Julian date, you must subtract the correct number of days from a Gregorian date (10 in the 1500s and 1600s, 11 in the 1700s, 12 in the 1800s, 13 in the 1900s and 2000s, and so on). You must also take into account the existence of leap days that aren't in the Gregorian calendar.

Working with Eras with Backward Time Flow

In the Gregorian calendar, time is divided into two eras, the BC era and the AD era. In the BC era, time flows in a direction seemingly backwards, that is from higher year numbers to lower. However, days and months flow in the normal direction. For example February 1 follows January 31. This can be confusing if you ask what day follows December 31, 7 BC. The correct answer is January 1, 6 BC. This example is illustrated in Listing 18.

Listing 18 Tomorrow in the BC era

```
NSCalendar *gregorian=[[NSCalendar alloc]
     initWithCalendarIdentifier: NSGregorianCalendar];
NSDateComponents *dateBCComps=[[NSDateComponents alloc] init];
[dateBCComps setEra:0]; //Era 0 corresponds to BC
[dateBCComps setMonth:12];
[dateBCComps setDay:31];
[dateBCComps setYear:7];
```

```
NSDate *dateBC=[gregorian dateFromComponents:dateBCComps];
NSDateComponents *offsetDate=[[NSDateComponents alloc] init];
[offsetDate setDay:1];
NSDate *dateBC2=[gregorian dateByAddingComponents: offsetDate toDate:dateBC
     options:0];
```

After this code executes `dateBC2` corresponds to January 1, 6 BC.

Document Revision History

This table describes the changes to *Date and Time Programming Guide*.

Date	Notes
2013-04-23	Clarified week and year calendar date component usage.
2011-06-06	Expanded Calendrical Calculations section. Added Historical Dates Section and Week-Based Year Section.
2010-02-24	Corrected code snippet.
2009-07-21	Added links to Cocoa Core Competencies.
2008-07-03	Moved information about NSCalendarDate to an appendix, rewrote articles to replace references to NSCalendarDate, and expanded content. Added a section about how to get date components using NSCalendar and NSDateComponents and a section about how to convert from one calendar to another. Removed information about converting a date to a string. See *NSDateFormatter Class Reference* for that information.
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 OS X v10.4.
2005-08-11	Changed title from "Dates and Times." Corrected minor typographic error.

Date	Notes
2002-11-12	Revision history was added to existing document. It will be used to record changes to the content of the document.

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