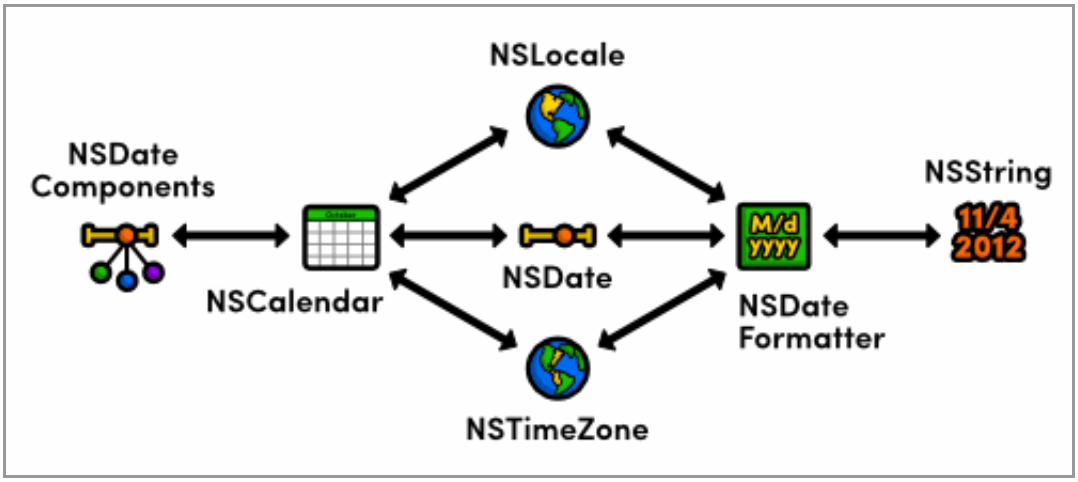
‹ Back to *Objective-C Data Types*

Date Programming

Date programming is a complex topic in any language, and Objective-C is no different. The Foundation Framework provides extensive support for working with dates in various calendrical systems. Unfortunately, the associated abstractions and dependencies give rise to a plethora of date-related classes that can be hard to navigate if you don't know where to start.



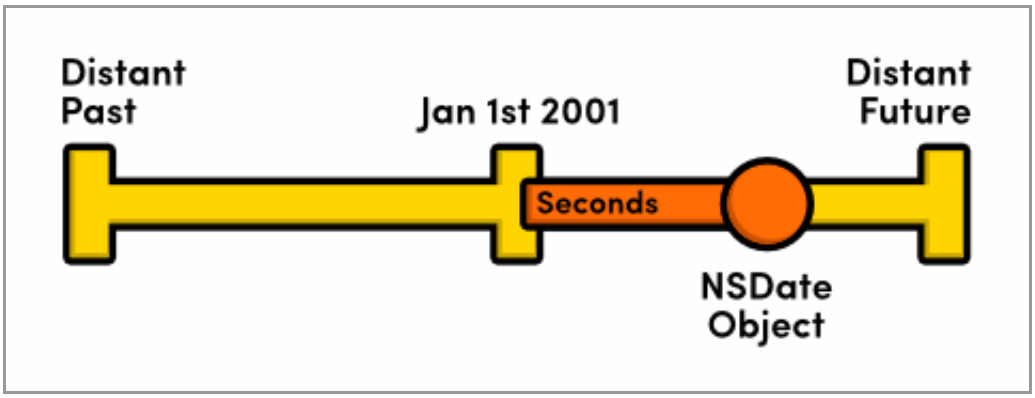
The essential classes for representing and manipulating dates

The above diagram provides a high-level overview of Objective-C's date-handling capabilities. The `NSDate` class represents a specific point in time, which can be reduced to `NSDateComponent`'s (e.g., days, weeks, years) by interpreting it in the context of an `NSCalendar` object. The `NSDateFormatter` class provides a human-readable version of an `NSDate`, and `NSLocale` and `NSTimeZone` encapsulate essential localization information for many calendrical operations.

This module explains all of this in much more detail using several hands-on examples. By the end of this module, you should be able to perform any kind of date operation that you'll ever need.

NSDate

An `NSDate` object represents a specific point in time, independent of any particular calendrical system, time zone, or locale. Internally, it just records the number of seconds from an arbitrary reference point (January 1st, 2001 GMT). For a date to be useful, it generally needs to be interpreted in the context of an `NSCalendar` object.



`NSDate` *recording the number of seconds from its reference point*

You'll typically want to create date objects using a calendar or a date formatter, as the `NSDate` class only provides a few low-level methods for creating dates from scratch. The `date` method returns an object representing the current date and time, and `dateWithTimeInterval:sinceDate:` generates a relative date using an `NSTimeInterval`, which is a `double` storing the number of seconds from the specified date. As you can see in the following example, a positive interval goes forward in time, and a negative one goes backwards.

```
NSDate *now = [NSDate date];
NSTimeInterval secondsInWeek = 7 * 24 * 60 * 60;
NSDate *lastWeek = [NSDate dateWithTimeInterval:-secondsInWeek
                                      sinceDate:now];
NSDate *nextWeek = [NSDate dateWithTimeInterval:secondsInWeek
                                      sinceDate:now];
NSLog(@"Last Week: %@", lastWeek);
NSLog(@"Right Now: %@", now);
NSLog(@"Next Week: %@", nextWeek);
```

This will output three datetime strings that look something like 2012-11-06 02:24:00 +0000. They contain the date (expressed as year-month-date), the time of day (expressed as hours:minutes:seconds) and the time zone (expressed as an offset

from Greenwich Mean Time (GMT)).

Aside from capturing an absolute point in time, the only real job of `NSDate` is to facilitate comparisons. It defines `isEqualToDate:` and `compare:` methods, which work just like the ones provided by `NSNumber`. In addition, the `earlierDate:` and `laterDate:` methods can be used as a convenient shortcut.

```
NSComparisonResult result = [now compare:nextWeek];
if (result == NSOrderedAscending) {
    NSLog(@"now < nextWeek");
} else if (result == NSOrderedSame) {
    NSLog(@"now == nextWeek");
} else if (result == NSOrderedDescending) {
    NSLog(@"now > nextWeek");
}
NSDate *earlierDate = [now earlierDate:lastWeek];
NSDate *laterDate = [now laterDate:lastWeek];
NSLog(@"%@ is earlier than %@", earlierDate, laterDate);
```

NSDateComponents

The `NSDateComponents` class is a simple data structure for representing the various time periods used by a calendrical system (days, weeks, months, years, etc). Unlike `NSDate`, the meaning of these components are entirely dependent on how it's used.

Consider an `NSDateComponents` object with its `year` property set to 2012. This could be interpreted as the year 2012 by a Gregorian calendar, the year 1469 by a Buddhist calendar, or two thousand and twelve years relative to some other date.

We'll work more with date components in the next section, but as a simple example, here's how you would create an `NSDateComponents` instance that could be interpreted as November 4th, 2012:

```
NSDateComponents *november4th2012 = [[NSDateComponents alloc] init];
[november4th2012 setYear:2012];
[november4th2012 setMonth:11];
[november4th2012 setDay:4];
NSLog(@"%@", november4th2012);
```

Again, it's important to understand that these components don't *actually* represent November 4th, 2012 until it is interpreted by a Gregorian calendar object as such.

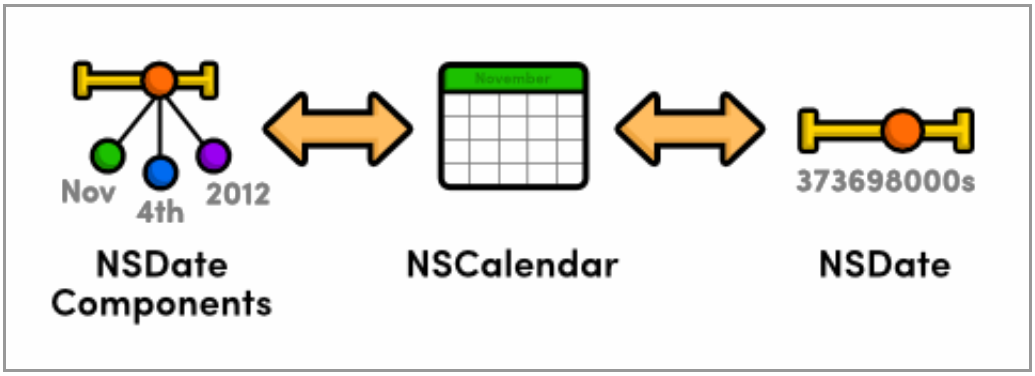
The complete list of component fields can be found below. Note that it's perfectly legal to set only the properties you need and leave the rest as `NSUndefinedDateComponent`, which is the default value for all fields.

`era`	`week`
`year`	`weekday`
`month`	`weekdayOrdinal`
`day`	`quarter`
`hour`	`weekOfMonth`
`minute`	`weekOfYear`
`second`	`yearForWeekOfYear`

NSCalendar

A calendrical system is a way of breaking down time into manageable units like years, months, weeks, etc. These units are represented as `NSDateComponents` objects; however, not all systems use the same units or interpret them in the same way. So, an `NSCalendar` object is required to give meaning to these components by defining the exact length of a year/month/week/etc.

This provides the necessary context for translating absolute `NSDate` instances into `NSDateComponents`. This is a very important ability, as it lets you work with dates on an intuitive, cultural level instead of a mathematical one. That is, it's much easier to say, "November 4th, 2012" than "373698000 seconds after January 1st, 2001."



Using `NSCalendar` *to convert between dates and date components*

This section explains how to create different types of calendars, then covers the three main responsibilities of `NSCalendar`: converting `NSDate` objects to components, creating `NSDate` objects from components, and performing calendrical calculations. We'll also take a brief look at `NSCalendarUnit`.

Creating Calendars

`NSCalendar`'s `initWithCalendarIdentifier:` initializer accepts an identifier that defines the calendrical system to use. In addition, the `currentCalendar` class method returns the user's preferred calendar. For most applications, you should opt for `currentCalendar` instead of manually defining one, since it reflects the user's device settings.

```
NSCalendar *gregorian = [[NSCalendar alloc]
                    initWithCalendarIdentifier:NSGregorianCalendar];
NSCalendar *buddhist = [[NSCalendar alloc]
                    initWithCalendarIdentifier:NSBuddhistCalendar];
NSCalendar *preferred = [NSCalendar currentCalendar];
NSLog(@"%@", gregorian.calendarIdentifier);
NSLog(@"%@", buddhist.calendarIdentifier);
NSLog(@"%@", preferred.calendarIdentifier);
```

The `calendarIdentifier` method lets you display the string-representation of the calendar ID, but it is read-only (you can't change the calendrical system after instantiation). The rest of Cocoa's built-in calendar identifiers can be accessed via the following constants:

`NSGregorianCalendar`	`NSBuddhistCalendar`
`NSChineseCalendar`	`NSHebrewCalendar`
`NSIslamicCalendar`	`NSIslamicCivilCalendar`
`NSJapaneseCalendar`	`NSRepublicOfChinaCalendar`
`NSPersianCalendar`	`NSIndianCalendar`
`NSISO8601Calendar`	

The next few sections walk through the basic usage of `NSCalendar`. If you change the identifier, you can see how different calendars have different interpretations of date components.

From Dates to Components

The following example shows you how to convert an NSDate into a culturally significant NSDateComponents object.

```
NSDate *now = [NSDate date];
NSCalendar *calendar = [[NSCalendar alloc]
                        initWithCalendarIdentifier:NSGregorianCalendar];
NSCalendarUnit units = NSYearCalendarUnit | NSMonthCalendarUnit | NSDayCalendarUnit;
NSDateComponents *components = [calendar components:units fromDate:now];

NSLog(@"Day: %ld", [components day]);
NSLog(@"Month: %ld", [components month]);
NSLog(@"Year: %ld", [components year]);
```

First, this creates an NSCalendar object that represents a Gregorian calendar. Then, it creates a bitmask defining the units to include in the conversion (see NSCalendarUnits for details). Finally, it passes these units and an NSDate to the components:fromDate: method.

From Components to Dates

A calendar can also convert in the other direction, which offers a much more intuitive way to create NSDate objects. It lets you define a date using the components of your native calendrical system:

```
NSCalendar *calendar = [[NSCalendar alloc]
                    initWithCalendarIdentifier:NSGregorianCalendar];
NSDateComponents *components = [[NSDateComponents alloc] init];
[components setYear:2012];
[components setMonth:11];
[components setDay:4];

NSDate *november4th2012 = [calendar dateFromComponents:components];
NSLog(@"%0.0f seconds between Jan 1st, 2001 and Nov 4th, 2012",
          [november4th2012 timeIntervalSinceReferenceDate]);
```

It's not necessary to define the units to include in the conversion, so this is a little more straightforward than translating dates to components. Simply instantiate an NSDateComponents object, populate it with the desired components, and pass it to the

`dateFromComponents:` method.

Remember that `NSDate` represents an absolute point in time, independent of any particular calendrical system. So, by calling `dateFromComponents:` on different `NSCalendar` objects, you can reliably compare dates from incompatible systems.

Calendrical Calculations

The third job of `NSCalendar` is to provide a high-level API for date-related calculations. First, we'll take a look at how calendars let you add components to a given `NSDate` instance. The following example generates a date object that is exactly one month away from the current date.

```
NSDate *now = [NSDate date];
NSCalendar *calendar = [[NSCalendar alloc]
                    initWithCalendarIdentifier:NSGregorianCalendar];
NSDateComponents *components = [[NSDateComponents alloc] init];
[components setMonth:1];
NSDate *oneMonthFromNow = [calendar dateByAddingComponents:components
                                                    toDate:now
                                                   options:0];
NSLog(@"%@", oneMonthFromNow);
```

All you have to do is create an `NSDateComponents` object, record the components you want to add, and pass it to `dateByAddingComponents:toDate:options:`. This is a good example of how the meaning of date components is entirely dependent on how it's used. Instead of representing a date, the above components represent a *duration*.

The `options` argument should either be `0` to let components overflow into higher units or `NSWrapCalendarComponents` to prevent this behavior. For example, if you set `month` to 14, passing `0` as an option tells the calendar to interpret it as 1 year and 2 months, whereas `NSWrapCalendarComponents` interprets it as 2 months and completely ignores the extra year.

Also note that `dateByAddingComponents:toDate:options:` is a calendar-aware operation, so it compensates for 30 vs. 31 day months (in the Gregorian calendar). This makes `NSCalendar` a more robust way of adding dates than `NSDate`'s low-level `dateWithTimeInterval:sinceDate:` method.

The other major calendrical operation provided by `NSCalendar` is

components:fromDate:toDate:options:, which calculates the interval between two NSDate objects. For example, you can determine the number of weeks since NSDate's internal reference point as follows:

```
NSDate *start = [NSDate dateWithTimeIntervalSinceReferenceDate:0];
NSDate *end = [NSDate date];
NSCalendar *calendar = [NSCalendar currentCalendar];
NSCalendarUnit units = NSWeekCalendarUnit;
NSDateComponents *components = [calendar components:units
                                           fromDate:start
                                             toDate:end
                                            options:0];
NSLog(@"It has been %ld weeks since January 1st, 2001",
      [components week]);
```

NSCalendarUnit

The first argument of components:fromDate: and components:fromDate:toDate:options: determine the properties that will be populated on the resulting NSDateComponents object. The possible values are defined by NSCalendarUnit, which enumerates the following constants:

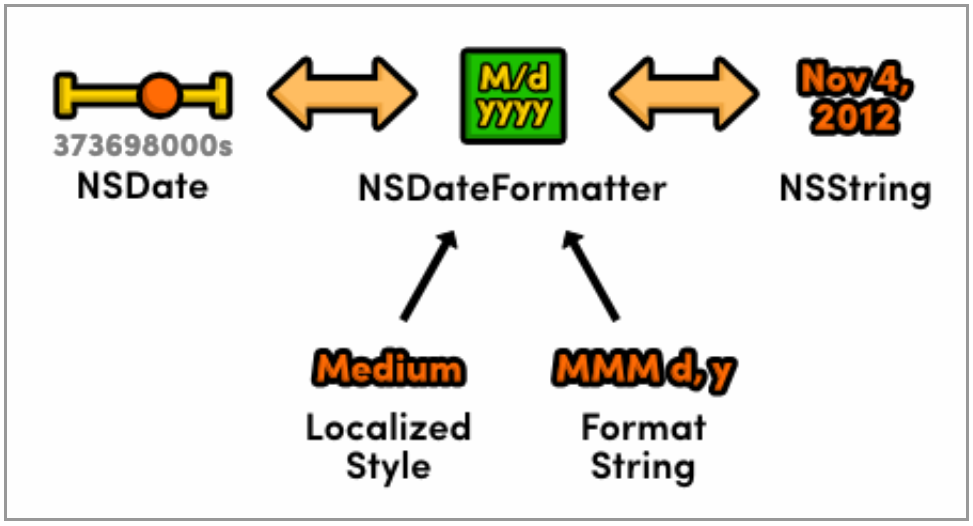
NSEraCalendarUnit	NSWeekdayCalendarUnit
NSYearCalendarUnit	NSWeekdayOrdinalCalendarUnit
NSMonthCalendarUnit	NSQuarterCalendarUnit
NSDayCalendarUnit	NSWeekOfMonthCalendarUnit
NSHourCalendarUnit	NSWeekOfYearCalendarUnit
NSMinuteCalendarUnit	NSYearForWeekOfYearCalendarUnit
NSSecondCalendarUnit	NSCalendarCalendarUnit
NSWeekCalendarUnit	NSTimeZoneCalendarUnit

When you need more than one of these units, you should combine them into a bitmask using the bitwise OR operator (|). For example, to include the day, hour, and minute, you would use the following value for the first parameter of components:fromDate:.

```
NSDayCalendarUnit | NSHourCalendarUnit | NSMinuteCalendarUnit
```

NSDateFormatter

The `NSDateFormatter` class makes it easy to work with the human-readable form of a date. Whereas calendars decompose a date into an `NSDateComponents` object, date formatters convert between `NSDate`'s and `NSString`'s.



Using a `NSDateFormatter` *to convert between dates and strings*

There are two ways to use a date formatter: 1) with localized styles or 2) with a custom format string. The former method is a better choice if you're displaying content to users, since it incorporates their preferences and device settings. The latter is useful when you need to know *exactly* what the resulting string representations will look like.

Localized Styles

Localized styles define how a date should look using abstract descriptions instead of specific date components. For example, instead of defining a date as `MM/dd/yy`, a style would say it should be displayed in its "short" form. This lets the system adapt the representation to the user's language, region, and preferences while still offering a level of control to the developer.

The following snippet formats a date using the "short" style. As you can see, separate styles are used for the date and the time. The `stringFromDate:` method formats an `NSDate` object according to the provided styles.

```
NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
[formatter setDateStyle:NSDateFormatterShortStyle];
[formatter setTimeStyle:NSDateFormatterShortStyle];
```

```
NSDate *now = [NSDate date];
NSString *prettyDate = [formatter stringFromDate:now];
NSLog(@"%@", prettyDate);
```

I'm an English speaker from the United States, so this will output `11/4/2012 8:09 PM`. If you have different default settings, you'll see the traditional date formatting used in your particular language/region. We'll take a closer look at this behavior in the NSLocale section. The complete list of formatting styles are included below.

`NSDateFormatterNoStyle`
`NSDateFormatterShortStyle`
`NSDateFormatterMediumStyle`
`NSDateFormatterLongStyle`
`NSDateFormatterFullStyle`

Note that the `NSDateFormatterNoStyle` constant can be used to omit the date or time from the output string.

Custom Format Strings

Again, the styles discussed above are the preferred way to define user-visible dates. However, if you're working with dates on a programmatic level, you may find the `setDateFormat:` method useful. This accepts a format string that defines precisely how the date will appear. For example, you can use `M.d.y` to output the month, date, and year separated by periods as follows:

```
// Formatter configuration
NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
NSLocale *posix = [[NSLocale alloc] initWithLocaleIdentifier:@"en_US_POSIX"];
[formatter setLocale:posix];
[formatter setDateFormat:@"M.d.y"];
// Date to string
NSDate *now = [NSDate date];
NSString *prettyDate = [formatter stringFromDate:now];
NSLog(@"%@", prettyDate);
```

As a best practice, you should always set the `locale` property of the `NSDateFormatter`

before using custom format strings. This ensures that the user's default locale won't affect the output, which would result in subtle, hard-to-reproduce bugs. In the above case, the POSIX locale ensures that the date is displayed as `11.4.2012`, regardless of the user's settings.

Custom date formats also present an alternative way to create date objects. Using the above configuration, it's possible to convert the string `11.4.2012` into an `NSDate` with the `dateFromString:` method. If the string cant't be parsed, it will return `nil`.

```
// String to date
NSString *dateString = @"11.4.2012";
NSDate *november4th2012 = [formatter dateFromString:dateString];
NSLog(@"%@", november4th2012);
```

In addition to `M`, `d`, and `y`, the Unicode Technical Standard #35 defines a plethora of other date format specifiers. This document contains a lot of information not really necessary to use `NSDateFormatter` correctly, so you may find the following list of sample format strings to be a more practical quick-reference. The date used to generate the output string is November 4th, 2012 8:09 PM Central Standard Time.

Format String	Output String
M/d/y	11/4/2012
MM/dd/yy	11/04/12
MMM d, ''yy	Nov 4, '12
MMMM	November
E	Sun
EEEE	Sunday
'Week' w 'of 52'	Week 45 of 52
'Day' D 'of 365'	Day 309 of 365
QQQ	Q4
QQQQ	4th quarter
m 'minutes past' h	9 minutes past 8
h:mm a	8:09 PM
HH:mm:ss's'	20:09:00s
HH:mm:ss:SS	20:09:00:00
h:mm a zz	8:09 PM CST
h:mm a zzzz	8:09 PM Central Standard Time

```
yyyy-MM-dd HH:mm:ss Z    2012-11-04 20:09:00 -0600
```

Notice that literal text needs to be wrapped in single quotes to prevent it from being parsed by the formatter.

NSLocale

An `NSLocale` object represents a set of conventions for a particular language, region, or culture. Both `NSCalendar` and `NSDateFormatter` rely on this information to localize many of their core operations.

The previous section already showed you how to create a custom locale via the `initWithLocaleIdentifier:` initializer. Let's take a look at the impact a locale change can have on an `NSDateFormatter` by running the same example using Egyptian Arabic instead of POSIX:

```
NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
NSLocale *egyptianArabic = [[NSLocale alloc] initWithLocaleIdentifier:@"ar_EG"];
[formatter setLocale:egyptianArabic];
[formatter setDateFormat:@"M.d.y"];

NSDate *now = [NSDate date];
NSString *prettyDate = [formatter stringFromDate:now];
NSLog(@"%@", prettyDate);
```

Instead of `11.4.2012`, the date is now displayed with Arabic digits: ١١.٤.٢٠١٢. This is why setting the locale before using custom format strings is so important—if you don't, `NSDateFormatter` will use the system default, and your format strings will return unexpected results for international users.

Locale identifiers are composed of a language abbreviation followed by a country abbreviation. In the above example, `ar` stands for Arabic, and `EG` represents the Egyptian dialect. You can obtain a complete list of available locale identifiers with the following (it's a very long list):

```
NSLog(@"%@", [NSLocale availableLocaleIdentifiers]);
```

But, whenever possible, you'll want to stick with the user's preferred locale. This makes sure that your app conforms to their expectations (you don't want to force an Arabic speaker to read English). You can access the preferred locale through the `currentLocale` class method, as shown below. Note that this is the default used by `NSCalendar` and `NSDateFormatter`.

```
NSLocale *preferredLocale = [NSLocale currentLocale]);
```

Custom `NSLocale` objects are usually only required for specialized applications or testing purposes, but it's still good to understand how iOS and OS X applications automatically translate your data for an international audience.

NSTimeZone

It's very important to understand that a string like `11.4.2012 8:09 PM` does *not* specify a precise point in time—8:09 PM in Chicago occurs 8 hours after 8:09 PM in Cairo. The time zone is a required piece of information for creating an absolute `NSDate` instance. Accordingly, `NSCalendar` and `NSDateFormatter` both require an `NSTimeZone` object to offset their calculations appropriately.

A time zone can be created explicitly with either its full name or its abbreviated one. The following example demonstrates both methods and shows how a time zone can alter the `NSDate` produced by a formatter. The generated dates are displayed in Greenwich Mean Time, and you can see that 8:09 PM in Chicago is indeed 8 hours after 8:09 PM in Cairo.

```
NSTimeZone *centralStandardTime = [NSTimeZone timeZoneWithAbbreviation:@"CST"];
NSTimeZone *cairoTime = [NSTimeZone timeZoneWithName:@"Africa/Cairo"];

NSDateFormatter *formatter = [[NSDateFormatter alloc] init];
NSLocale *posix = [[NSLocale alloc] initWithLocaleIdentifier:@"en_US_POSIX"];
[formatter setLocale:posix];
[formatter setDateFormat:@"M.d.y h:mm a"];
NSString *dateString = @"11.4.2012 8:09 PM";

[formatter setTimeZone:centralStandardTime];
NSDate *eightPMInChicago = [formatter dateFromString:dateString];
```

```
NSLog(@"%@", eightPMInChicago);      // 2012-11-05 02:09:00 +0000

[formatter setTimeZone:cairoTime];
NSDate *eightPMInCairo = [formatter dateFromString:dateString];
NSLog(@"%@", eightPMInCairo);        // 2012-11-04 18:09:00 +0000
```

Note that the `NSDateFormatter`'s `timeZone` property is a fallback, so format strings that contain one of the `z` specifiers will use the parsed value as expected. You can use `NSCalendar`'s `timeZone` property to the same effect.

Complete lists of time zone names and abbreviations can be accessed via `knownTimeZoneNames` and `abbreviationDictionary`, respectively. However, the latter does not necessarily provide values for *every* time zone.

```
NSLog(@"%@", [NSTimeZone knownTimeZoneNames]);
NSLog(@"%@", [NSTimeZone abbreviationDictionary]);
```

All users have a preferred time zone, whether it's configured explicitly or determined automatically based on their region. The `localTimeZone` class method is the best option for accessing the current time zone.

```
NSTimeZone *preferredTimeZone = [NSTimeZone localTimeZone];
```

As with locales, it's generally a better idea to stick with the default time zone. Custom `NSTimeZone` objects are typically only necessary for scheduling applications where time zones need to be explicitly associated with individual events.

Conclusion

We hope that you've enjoyed Ry's Objective-C Tutorial. Our goal was to walk you through the language behind iOS and OS X application development, and if you've been following along from the beginning of this tutorial, you should be feeling pretty comfortable with classes, properties, methods, protocols, categories, blocks, and the major data types in both C and Objective-C.

The next step is to take your fantastic Objective-C foundation and combine it with the Cocoa/Cocoa Touch frameworks to build some awesome OS X and iOS apps.

Apple's Start Developing iOS Apps Today guide is a great jumping-off point for new iOS developers, but if you're interested in game development, be sure to check back here in March 2013 for *Ry's Cocos2d Tutorial*.

Please feel free to contact us with any questions or comments about this tutorial or Objective-C in general. If you've created an app using the skills you learned here, we'd love to hear about that, too.

The End.