String Programming Guide

Developer

Contents

Tables

Introduction to String 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.

String Programming Guide for Cocoa describes how to create, search, concatenate, and draw strings. It also describes character sets, which let you search a string for characters in a group, and scanners, which convert numbers to strings and vice versa.

Who Should Read This Document

You should read this document if you need to work directly with strings or character sets.

Organization of This Document

This document contains the following articles:

- Strings (page 7) describes the characteristics of string objects in Cocoa.
- Creating and Converting String Objects (page 8) explains the ways in which `NSString` and its subclass `NSMutableString` create string objects and convert their contents to and from the various character encodings they support.
- Formatting String Objects (page 13) describes how to format `NSString` objects.
- String Format Specifiers (page 15) describes `printf`-style format specifiers supported by `NSString`.
- Reading Strings From and Writing Strings To Files and URLs (page 19) describes how to read strings from and write strings to files and URLs.
- Searching, Comparing, and Sorting Strings (page 22) describes methods for finding characters and substrings within strings and for comparing one string to another.
- Words, Paragraphs, and Line Breaks (page 28) describes how words, paragraphs, and line breaks are determined.

- Characters and Grapheme Clusters (page 30) describes how you can break strings down into user-perceived characters.
- Character Sets (page 33) explains how to use character set objects, and how to use `NSCharacterSet` methods to create standard and custom character sets.
- Scanners (page 36) describes `NSScanner` objects, which interpret and convert the characters of an `NSString` object into number and string values.
- String Representations of File Paths (page 40) describes the `NSString` methods that manipulate strings as file-system paths.
- Drawing Strings (page 44) discusses the methods of the `NSString` class that support drawing directly in an `NSView` object.

See Also

For more information, refer to the following documents:

- *Attributed String Programming Guide* is closely related to *String Programming Guide for Cocoa*. It provides information about `NSAttributedString` objects, which manage sets of attributes, such as font and kerning, that are associated with character strings or individual characters.
- *Data Formatting Guide* describes how to format data using objects that create, interpret, and validate text.
- *Internationalization and Localization Guide* provides information about localizing strings in your project, including information on how string formatting arguments can be ordered.
- *String Programming Guide for Core Foundation* in Core Foundation, discusses the Core Foundation opaque type CFString, which is toll-free bridged with the `NSString` class.

Strings

String objects represent character strings in Cocoa frameworks. Representing strings as objects allows you to use strings wherever you use other objects. It also provides the benefits of encapsulation, so that string objects can use whatever encoding and storage is needed for efficiency while simply appearing as arrays of characters.

A string object is implemented as an array of Unicode characters (in other words, a text string). An immutable string is a text string that is defined when it is created and subsequently cannot be changed. To create and manage an immutable string, use the `NSString` class. To construct and manage a string that can be changed after it has been created, use `NSMutableString`.

The objects you create using `NSString` and `NSMutableString` are referred to as string objects (or, when no confusion will result, merely as strings). The term *C string* refers to the standard C `char *` type.

A string object presents itself as an array of Unicode characters. You can determine how many characters it contains with the `length` method and can retrieve a specific character with the `characterAtIndex:` method. These two "primitive" methods provide basic access to a string object. Most use of strings, however, is at a higher level, with the strings being treated as single entities: You compare strings against one another, search them for substrings, combine them into new strings, and so on. If you need to access string objects character-by-character, you must understand the Unicode character encoding—specifically, issues related to composed character sequences. For details see:

- *The Unicode Standard, Version 4.0*. The Unicode Consortium. Boston: Addison-Wesley, 2003. ISBN 0-321-18578-1.
- The Unicode Consortium web site: http://www.unicode.org/.

Creating and Converting String Objects

`NSString` and its subclass `NSMutableString` provide several ways to create string objects, most based around the various character encodings it supports. Although string objects always present their own contents as Unicode characters, they can convert their contents to and from many other encodings, such as 7-bit ASCII, ISO Latin 1, EUC, and Shift-JIS. The `availableStringEncodings` class method returns the encodings supported. You can specify an encoding explicitly when converting a C string to or from a string object, or use the default C string encoding, which varies from platform to platform and is returned by the `defaultCStringEncoding` class method.

Creating Strings

The simplest way to create a string object in source code is to use the Objective-C `@"..."` construct:

```
NSString *temp = @"Contrafibularity";
```

Note, when creating a string constant in this fashion, you should use UTF-8 characters. You can put the actual Unicode data in the string for localized text, as in `NSString *hello = @"□□□□□";`. You can also insert `\u` or `\U` in the string for non-graphic characters.

Objective-C string constant is created at compile time and exists throughout your program's execution. The compiler makes such object constants unique on a per-module basis, and they're never deallocated. You can also send messages directly to a string constant as you do any other string:

```
BOOL same = [@"comparison" isEqualToString:myString];
```

NSString from C Strings and Data

To create an `NSString` object from a C string, you use methods such as `initWithCString:encoding:`. You must correctly specify the character encoding of the C string. Similar methods allow you to create string objects from characters in a variety of encodings. The method `initWithData:encoding:` allows you to convert string data stored in an `NSData` object into an `NSString` object.

```
char *utf8String = /* Assume this exists. */ ;
```

```
NSString *stringFromUTFString = [[NSString alloc] initWithUTF8String:utf8String];


char *macOSRomanEncodedString = /* assume this exists */ ;
NSString *stringFromMORString =
            [[NSString alloc] initWithCString:macOSRomanEncodedString
                              encoding:NSMacOSRomanStringEncoding];


NSData *shiftJISData =  /* assume this exists */ ;
NSString *stringFromShiftJISData =
            [[NSString alloc] initWithData:shiftJISData
                              encoding:NSShiftJISStringEncoding];
```

The following example converts an `NSString` object containing a UTF-8 character to ASCII data then back to an `NSString` object.

```
unichar ellipsis = 0x2026;
NSString *theString = [NSString stringWithFormat:@"To be continued%C", ellipsis];


NSData *asciiData = [theString dataUsingEncoding:NSASCIIStringEncoding
allowLossyConversion:YES];


NSString *asciiString = [[NSString alloc] initWithData:asciiData
encoding:NSASCIIStringEncoding];


NSLog(@"Original: %@ (length %d)", theString, [theString length]);
NSLog(@"Converted: %@ (length %d)", asciiString, [asciiString length]);


// output:
// Original: To be continued… (length 16)
// Converted: To be continued... (length 18)
```

Note: `NSString` is not intended as a data storage container for arbitrary sequences of bytes. For this type of functionality, refer to `NSData`.

Variable Strings

To create a variable string, you typically use `stringWithFormat::` or `initWithFormat:` (or for localized strings, `localizedStringWithFormat:`). These methods and their siblings use a format string as a template into which the values you provide (string and other objects, numerics values, and so on) are inserted. They and the supported format specifiers are described in Formatting String Objects (page 13).

You can build a string from existing string objects using the methods `stringByAppendingString:` and `stringByAppendingFormat:` to create a new string by adding one string after another, in the second case using a format string.

```
NSString *hString = @"Hello";
NSString *hwString = [hString stringByAppendingString:@", world!"];
```

Strings to Present to the User

When creating strings to present to the user, you should consider the importance of localizing your application. In general, you should avoid creating user-visible strings directly in code. Instead you should use strings in your code as a key to a localization dictionary that will supply the user-visible string in the user's preferred language. Typically this involves using `NSLocalizedString` and similar macros, as illustrated in the following example.

```
NSString *greeting = NSLocalizedStringFromTable
    (@"Hello", @"greeting to present in first launch panel", @"greetings");
```

For more about internationalizing your application, see *Internationalization and Localization Guide* . Localizing String Resources describes how to work with and reorder variable arguments in localized strings.

Combining and Extracting Strings

You can combine and extract strings in various ways. The simplest way to combine two strings is to append one to the other. The `stringByAppendingString:` method returns a string object formed from the receiver and the given argument.

```
NSString *beginning = @"beginning";
NSString *alphaAndOmega = [beginning stringByAppendingString:@" and end"];
// alphaAndOmega is @"beginning and end"
```

You can also combine several strings according to a template with the `initWithFormat:`, `stringWithFormat:`, and `stringByAppendingFormat:` methods; these are described in more detail in Formatting String Objects (page 13).

You can extract substrings from the beginning or end of a string to a particular index, or from a specific range, with the `substringToIndex:`, `substringFromIndex:`, and `substringWithRange:` methods. You can also split a string into substrings (based on a separator string) with the `componentsSeparatedByString:` method. These methods are illustrated in the following examples—notice that the index of the index-based methods starts at `0`:

```
NSString *source = @"0123456789";
NSString *firstFour = [source substringToIndex:4];
// firstFour is @"0123"

NSString *allButFirstThree = [source substringFromIndex:3];
// allButFirstThree is @"3456789"

NSRange twoToSixRange = NSMakeRange(2, 4);
NSString *twoToSix = [source substringWithRange:twoToSixRange];
// twoToSix is @"2345"

NSArray *split = [source componentsSeparatedByString:@"45"];
// split contains { @"0123", @"6789" }
```

If you need to extract strings using pattern-matching rather than an index, you should use a scanner—see Scanners (page 36).

Getting C Strings

To get a C string from a string object, you are recommended to use `UTF8String`. This returns a `const char *` using UTF8 string encoding.

```
const char *cString = [@"Hello, world" UTF8String];
```

The C string you receive is owned by a temporary object, and will become invalid when automatic deallocation takes place. If you want to get a permanent C string, you must create a buffer and copy the contents of the `const char *` returned by the method.

Similar methods allow you to create string objects from characters in the Unicode encoding or an arbitrary encoding, and to extract data in these encodings. `initWithData:encoding:` and `dataUsingEncoding:` perform these conversions from and to `NSData` objects.

Conversion Summary

This table summarizes the most common means of creating and converting string objects:

Source	Creation method	Extraction method
In code	`@"..."` compiler construct	N/A
UTF8 encoding	`stringWithUTF8String:`	`UTF8String`

Unicode encoding	`stringWithCharacters: length:`	`getCharacters:`
		`getCharacters:range:`

Arbitrary encoding	`initWithData: encoding:`	`dataUsingEncoding:`

Existing strings	`stringByAppendingString:`
	`stringByAppendingFormat:`
		N/A

Format string	`localizedStringWithFormat:`
	`initWithFormat: locale:`
		Use `NSScanner`

Localized strings	`NSLocalizedString` and similar	N/A

Formatting String Objects

This article describes how to create a string using a format string, how to use non-ASCII characters in a format string, and a common error that developers make when using NSLog or NSLogv.

Formatting Basics

NSString uses a format string whose syntax is similar to that used by other formatter objects. It supports the format characters defined for the ANSI C function printf(), plus %@ for any object (see String Format Specifiers (page 15) and the IEEE printf specification). If the object responds to descriptionWithLocale: messages, NSString sends such a message to retrieve the text representation. Otherwise, it sends a description message. Localizing String Resources describes how to work with and reorder variable arguments in localized strings.

In format strings, a '%' character announces a placeholder for a value, with the characters that follow determining the kind of value expected and how to format it. For example, a format string of "%d houses" expects an integer value to be substituted for the format expression '%d'. NSString supports the format characters defined for the ANSI C functionprintf(), plus '@' for any object. If the object responds to the descriptionWithLocale: message, NSString sends that message to retrieve the text representation, otherwise, it sends a description message.

Value formatting is affected by the user's current locale, which is an NSDictionary object that specifies number, date, and other kinds of formats. NSString uses only the locale's definition for the decimal separator (given by the key named NSDecimalSeparator). If you use a method that doesn't specify a locale, the string assumes the default locale.

You can use NSString's stringWithFormat: method and other related methods to create strings with printf-style format specifiers and argument lists, as described in Creating and Converting String Objects (page 8). The examples below illustrate how you can create a string using a variety of format specifiers and arguments.

```
NSString *string1 = [NSString stringWithFormat:@"A string: %@, a float: %1.2f",
                                               @"string", 31415.9265];
// string1 is "A string: string, a float: 31415.93"

NSNumber *number = @1234;
```

```
NSDictionary *dictionary = @{@"date": [NSDate date]};
NSString *baseString = @"Base string.";
NSString *string2 = [baseString stringByAppendingFormat:
        @" A number: %@, a dictionary: %@", number, dictionary];
// string2 is "Base string. A number: 1234, a dictionary: {date = 2005-10-17
09:02:01 -0700; }"
```

Strings and Non-ASCII Characters

You can include non-ASCII characters (including Unicode) in strings using methods such as `stringWithFormat:` and `stringWithUTF8String:`.

```
NSString *s = [NSString stringWithFormat:@"Long %C dash", 0x2014];
```

Since `\xe2\x80\x94` is the 3-byte UTF-8 string for `0x2014`, you could also write:

```
NSString *s = [NSString stringWithUTF8String:"Long \xe2\x80\x94   dash"];
```

NSLog and NSLogv

The utility functions `NSLog()` and `NSLogv()` use the `NSString` string formatting services to log error messages. Note that as a consequence of this, you should take care when specifying the argument for these functions. A common mistake is to specify a string that includes formatting characters, as shown in the following example.

```
NSString *string = @"A contrived string %@";
NSLog(string);
// The application will probably crash here due to signal 10 (SIGBUS)
```

It is better (safer) to use a format string to output another string, as shown in the following example.

```
NSString *string = @"A contrived string %@";
NSLog(@"%@", string);
// Output: A contrived string %@
```

String Format Specifiers

This article summarizes the format specifiers supported by string formatting methods and functions.

Format Specifiers

The format specifiers supported by the `NSString` formatting methods and CFString formatting functions follow the IEEE printf specification; the specifiers are summarized in Table 1 (page 15). Note that you can also use the "`n$`" positional specifiers such as `%1$@ %2$s`. For more details, see the IEEE printf specification. You can also use these format specifiers with the `NSLog` function.

Table 1 Format specifiers supported by the `NSString` formatting methods and CFString formatting functions

Specifier	Description
`%@`	Objective-C object, printed as the string returned by `descriptionWithLocale:` if available, or `description` otherwise. Also works with `CFTypeRef` objects, returning the result of the `CFCopyDescription` function.
`%%`	`'%'` character.
`%d`, `%D`	Signed 32-bit integer (`int`).
`%u`, `%U`	Unsigned 32-bit integer (`unsigned int`).
`%x`	Unsigned 32-bit integer (`unsigned int`), printed in hexadecimal using the digits 0–9 and lowercase a–f.
`%X`	Unsigned 32-bit integer (`unsigned int`), printed in hexadecimal using the digits 0–9 and uppercase A–F.
`%o`, `%O`	Unsigned 32-bit integer (`unsigned int`), printed in octal.
`%f`	64-bit floating-point number (`double`).
`%e`	64-bit floating-point number (`double`), printed in scientific notation using a lowercase e to introduce the exponent.
`%E`	64-bit floating-point number (`double`), printed in scientific notation using an uppercase E to introduce the exponent.

Specifier	Description
%g	64-bit floating-point number (double), printed in the style of %e if the exponent is less than –4 or greater than or equal to the precision, in the style of %f otherwise.
%G	64-bit floating-point number (double), printed in the style of %E if the exponent is less than –4 or greater than or equal to the precision, in the style of %f otherwise.
%c	8-bit unsigned character (unsigned char), printed by NSLog() as an ASCII character, or, if not an ASCII character, in the octal format \\ddd or the Unicode hexadecimal format \\udddd, where d is a digit.
%C	16-bit Unicode character (unichar), printed by NSLog() as an ASCII character, or, if not an ASCII character, in the octal format \\ddd or the Unicode hexadecimal format \\udddd, where d is a digit.
%s	Null-terminated array of 8-bit unsigned characters. Because the %s specifier causes the characters to be interpreted in the system default encoding, the results can be variable, especially with right-to-left languages. For example, with RTL, %s inserts direction markers when the characters are not strongly directional. For this reason, it's best to avoid %s and specify encodings explicitly.
%S	Null-terminated array of 16-bit Unicode characters.
%p	Void pointer (void *), printed in hexadecimal with the digits 0–9 and lowercase a–f, with a leading 0x.
%a	64-bit floating-point number (double), printed in scientific notation with a leading 0x and one hexadecimal digit before the decimal point using a lowercase p to introduce the exponent.
%A	64-bit floating-point number (double), printed in scientific notation with a leading 0X and one hexadecimal digit before the decimal point using a uppercase P to introduce the exponent.
%F	64-bit floating-point number (double), printed in decimal notation.

Table 2 Length modifiers supported by the NSString formatting methods and CFString formatting functions

Length modifier	Description
h	Length modifier specifying that a following d, o, u, x, or X conversion specifier applies to a short or unsigned short argument.

Length modifier	Description
`hh`	Length modifier specifying that a following `d`, `o`, `u`, `x`, or `X` conversion specifier applies to a `signed char` or `unsigned char` argument.
`l`	Length modifier specifying that a following `d`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long` or `unsigned long` argument.
`ll`, `q`	Length modifiers specifying that a following `d`, `o`, `u`, `x`, or `X` conversion specifier applies to a `long long` or `unsigned long long` argument.
`L`	Length modifier specifying that a following `a`, `A`, `e`, `E`, `f`, `F`, `g`, or `G` conversion specifier applies to a `long double` argument.
`z`	Length modifier specifying that a following `d`, `o`, `u`, `x`, or `X` conversion specifier applies to a `size_t` or the corresponding signed integer type argument.
`t`	Length modifier specifying that a following `d`, `o`, `u`, `x`, or `X` conversion specifier applies to a `ptrdiff_t` or the corresponding unsigned integer type argument.
`j`	Length modifier specifying that a following `d`, `o`, `u`, `x`, or `X` conversion specifier applies to a `intmax_t` or `uintmax_t` argument.

Platform Dependencies

OS X uses several data types—`NSInteger`, `NSUInteger`,`CGFloat`, and `CFIndex`—to provide a consistent means of representing values in 32- and 64-bit environments. In a 32-bit environment, `NSInteger` and `NSUInteger` are defined as `int` and `unsigned int`, respectively. In 64-bit environments, `NSInteger` and `NSUInteger` are defined as `long` and `unsigned long`, respectively. To avoid the need to use different printf-style type specifiers depending on the platform, you can use the specifiers shown in Table 3. Note that in some cases you may have to cast the value.

Table 3 Format specifiers for data types

Type	Format specifier	Considerations
`NSInteger`	`%ld` or `%lx`	Cast the value to `long`.
`NSUInteger`	`%lu` or `%lx`	Cast the value to `unsigned long`.
`CGFloat`	`%f` or `%g`	`%f` works for floats and doubles when formatting; but note the technique described below for scanning.

Type	Format specifier	Considerations
`CFIndex`	`%ld` or `%lx`	The same as `NSInteger`.
pointer	`%p` or `%zx`	`%p` adds `0x` to the beginning of the output. If you don't want that, use `%zx` and no typecast.

The following example illustrates the use of `%ld` to format an `NSInteger` and the use of a cast.

```
NSInteger i = 42;
printf("%ld\n", (long)i);
```

In addition to the considerations mentioned in Table 3, there is one extra case with scanning: you must distinguish the types for `float` and `double`. You should use `%f` for float, `%lf` for double. If you need to use `scanf` (or a variant thereof) with `CGFloat`, switch to `double` instead, and copy the `double` to `CGFloat`.

```
CGFloat imageWidth;
double tmp;
sscanf (str, "%lf", &tmp);
imageWidth = tmp;
```

It is important to remember that `%lf` does not represent `CGFloat` correctly on either 32- or 64-bit platforms. This is unlike `%ld`, which works for `long` in all cases.

Reading Strings From and Writing Strings To Files and URLs

Reading files or URLs using `NSString` is straightforward provided that you know what encoding the resource uses—if you don't know the encoding, reading a resource is more challenging. When you write to a file or URL, you must specify the encoding to use. (Where possible, you should use URLs because these are more efficient.)

Reading From Files and URLs

`NSString` provides a variety of methods to read data from files and URLs. In general, it is much easier to read data if you know its encoding. If you have plain text and no knowledge of the encoding, you are already in a difficult position. You should avoid placing yourself in this position if at all possible—anything that calls for the use of plain text files should specify the encoding (preferably UTF-8 or UTF-16+BOM).

Reading data with a known encoding

To read from a file or URL for which you know the encoding, you use `stringWithContentsOfFile:encoding:error:` or `stringWithContentsOfURL:encoding:error:`, or the corresponding `init...` method, as illustrated in the following example.

```
NSURL *URL = ...;
NSError *error;
NSString *stringFromFileAtURL = [[NSString alloc]
                                      initWithContentsOfURL:URL
                                      encoding:NSUTF8StringEncoding
                                      error:&error];
if (stringFromFileAtURL == nil) {
    // an error occurred
    NSLog(@"Error reading file at %@\n%@",
              URL, [error localizedFailureReason]);
    // implementation continues ...
```

You can also initialize a string using a data object, as illustrated in the following examples. Again, you must specify the correct encoding.

```
NSURL *URL = ...;
NSData *data = [NSData dataWithContentsOfURL:URL];

// Assuming data is in UTF8.
NSString *string = [NSString stringWithUTF8String:[data bytes]];

// if data is in another encoding, for example ISO-8859-1
NSString *string = [[NSString alloc]
            initWithData:data encoding: NSISOLatin1StringEncoding];
```

Reading data with an unknown encoding

If you find yourself with text of unknown encoding, it is best to make sure that there is a mechanism for correcting the inevitable errors. For example, Apple's Mail and Safari applications have encoding menus, and TextEdit allows the user to reopen the file with an explicitly specified encoding.

If you are forced to guess the encoding (and note that in the absence of explicit information, it is a *guess*):

1. Try `stringWithContentsOfFile:usedEncoding:error:` or `initWithContentsOfFile:usedEncoding:error:` (or the URL-based equivalents).

   These methods try to determine the encoding of the resource, and if successful return by reference the encoding used.

2. If (1) fails, try to read the resource by specifying UTF-8 as the encoding.

3. If (2) fails, try an appropriate legacy encoding.

   "Appropriate" here depends a bit on circumstances; it might be the default C string encoding, it might be ISO or Windows Latin 1, or something else, depending on where your data are coming from.

4. Finally, you can try `NSAttributedString`'s loading methods from the Application Kit (such as `initWithURL:options:documentAttributes:error:`).

   These methods attempt to load plain text files, and return the encoding used. They can be used on more-or-less arbitrary text documents, and are worth considering if your application has no special expertise in text. They might not be as appropriate for Foundation-level tools or documents that are not natural-language text.

Writing to Files and URLs

Compared with reading data from a file or URL, writing is straightforward—`NSString` provides two convenient methods, `writeToFile:atomically:encoding:error:` and `writeToURL:atomically:encoding:error:`. You must specify the encoding that should be used, and choose whether to write the resource atomically or not. If you do not choose to write atomically, the string is written directly to the path you specify. If you choose to write it atomically, it is written first to an auxiliary file, and then the auxiliary file is renamed to the path. This option guarantees that the file, if it exists at all, won't be corrupted even if the system should crash during writing. If you write to an URL, the atomicity option is ignored if the destination is not of a type that can be accessed atomically.

```
NSURL *URL = ...;
NSString *string = ...;
NSError *error;
BOOL ok = [string writeToURL:URL atomically:YES
                  encoding:NSUnicodeStringEncoding error:&error];
if (!ok) {
    // an error occurred
    NSLog(@"Error writing file at %@\n%@",
              path, [error localizedFailureReason]);
    // implementation continues ...
```

Summary

This table summarizes the most common means of reading and writing string objects to and from files and URLs:

Source	Creation method	Extraction method

URL contents	`stringWithContentsOfURL:encoding:error:`
	`stringWithContentsOfURL:usedEncoding:error:`
		`writeToURL:atomically:encoding:error:`

File contents	`stringWithContentsOfFile:encoding:error:`
	`stringWithContentsOfFile:usedEncoding:error:`
		`writeToFile:atomically:encoding:error:`

Searching, Comparing, and Sorting Strings

The string classes provide methods for finding characters and substrings within strings and for comparing one string to another. These methods conform to the Unicode standard for determining whether two character sequences are equivalent. The string classes provide comparison methods that handle composed character sequences properly, though you do have the option of specifying a literal search when efficiency is important and you can guarantee some canonical form for composed character sequences.

Search and Comparison Methods

The search and comparison methods each come in several variants. The simplest version of each searches or compares entire strings. Other variants allow you to alter the way comparison of composed character sequences is performed and to specify a specific range of characters within a string to be searched or compared; you can also search and compare strings in the context of a given locale.

These are the basic search and comparison methods:

Search methods	Comparison methods
`rangeOfString:`	`compare:`
`rangeOfString: options:`	`compare:options:`
`rangeOfString: options:range:`	`compare:options: range:`
`rangeOfString: options:range: locale:`	`compare:options: range:locale:`
`rangeOfCharacterFromSet:`	
`rangeOfCharacterFromSet: options:`	
`rangeOfCharacterFromSet: options:range:`	

Searching strings

You use the `rangeOfString:...` methods to search for a substring within the receiver. The `rangeOfCharacterFromSet:...` methods search for individual characters from a supplied set of characters.

Substrings are found only if completely contained within the specified range. If you specify a range for a search or comparison method and don't request `NSLiteralSearch` (see below), the range must not break composed character sequences on either end; if it does, you could get an incorrect result. (See the method description for `rangeOfComposedCharacterSequenceAtIndex:` for a code sample that adjusts a range to lie on character sequence boundaries.)

You can also scan a string object for numeric and string values using an instance of `NSScanner`. For more about scanners, see Scanners (page 36). Both the `NSString` and the `NSScanner` class clusters use the `NSCharacterSet` class cluster for search operations. For more about character sets, see Character Sets (page 33).

If you simply want to determine whether a string contains a given pattern, you can use a predicate:

```
BOOL match = [myPredicate evaluateWithObject:myString];
```

For more about predicates, see *Predicate Programming Guide*.

Comparing and sorting strings

The `compare:...` methods return the lexical ordering of the receiver and the supplied string. Several other methods allow you to determine whether two strings are equal or whether one is the prefix or suffix of another, but they don't have variants that allow you to specify search options or ranges.

The simplest method you can use to compare strings is `compare:`—this is the same as invoking `compare:options:range:` with no options and the receiver's full extent as the range. If you want to specify comparison options (`NSCaseInsensitiveSearch`, `NSLiteralSearch`, or `NSNumericSearch`) you can use `compare:options:`; if you want to specify a locale you can use `compare:options:range:locale:`. `NSString` also provides various convenience methods to allow you to perform common comparisons without the need to specify ranges and options directly, for example `caseInsensitiveCompare:` and `localizedCompare:`.

Important: For user-visible sorted lists, you should *always* use localized comparisons. Thus typically instead of `compare:` or `caseInsensitiveCompare:` you should use `localizedCompare:` or `localizedCaseInsensitiveCompare:`.

If you want to compare strings to order them in the same way as they're presented in Finder, you should use `compare:options:range:locale:` with the user's locale and the following options: `NSCaseInsensitiveSearch`, `NSNumericSearch`, `NSWidthInsensitiveSearch`, and `NSForcedOrderingSearch`. For an example, see Sorting strings like Finder (page 26).

Search and Comparison Options

Several of the search and comparison methods take an "options" argument. This is a bit mask that adds further constraints to the operation. You create the mask by combining the following options (not all options are available for every method):

Search option	Effect
`NSCaseInsensitive-Search`	Ignores case distinctions among characters.
`NSLiteralSearch`	Performs a byte-for-byte comparison. Differing literal sequences (such as composed character sequences) that would otherwise be considered equivalent are considered not to match. Using this option can speed some operations dramatically.
`NSBackwardsSearch`	Performs searching from the end of the range toward the beginning.
`NSAnchoredSearch`	Performs searching only on characters at the beginning or, if `NSBackwardsSearch` is also specified, the end of the range. No match at the beginning or end means nothing is found, even if a matching sequence of characters occurs elsewhere in the string.
`NSNumericSearch`	When used with the `compare:options:` methods, groups of numbers are treated as a numeric value for the purpose of comparison. For example, `Filename9.txt` < `Filename20.txt` < `Filename100.txt`.

Search and comparison are currently performed as if the `NSLiteralSearch` option were specified.

Examples

Case-Insensitive Search for Prefix and Suffix

`NSString` provides the methods `hasPrefix:` and `hasSuffix:` that you can use to find an *exact* match for a prefix or suffix. The following example illustrates how you can use `rangeOfString:options:` with a combination of options to perform *case insensitive* searches.

```
NSString *searchString = @"age";


NSString *beginsTest = @"Agencies";
```

```
NSRange prefixRange = [beginsTest rangeOfString:searchString
    options:(NSAnchoredSearch | NSCaseInsensitiveSearch)];

// prefixRange = {0, 3}

NSString *endsTest = @"BRICOLAGE";
NSRange suffixRange = [endsTest rangeOfString:searchString
    options:(NSAnchoredSearch | NSCaseInsensitiveSearch | NSBackwardsSearch)];

// suffixRange = {6, 3}
```

Comparing Strings

The following examples illustrate the use of various string comparison methods and associated options. The first shows the simplest comparison method.

```
NSString *string1 = @"string1";
NSString *string2 = @"string2";
NSComparisonResult result;
result = [string1 compare:string2];
// result = -1 (NSOrderedAscending)
```

You can compare strings numerically using the `NSNumericSearch` option:

```
NSString *string10 = @"string10";
NSString *string2 = @"string2";
NSComparisonResult result;

result = [string10 compare:string2];
// result = -1 (NSOrderedAscending)

result = [string10 compare:string2 options:NSNumericSearch];
// result = 1 (NSOrderedDescending)
```

You can use convenience methods (`caseInsensitiveCompare:` and `localizedCaseInsensitiveCompare:`) to perform case-insensitive comparisons:

```
NSString *string_a = @"Aardvark";
NSString *string_A = @"AARDVARK";

result = [string_a compare:string_A];
// result = 1 (NSOrderedDescending)

result = [string_a caseInsensitiveCompare:string_A];
// result = 0 (NSOrderedSame)
// equivalent to [string_a compare:string_A options:NSCaseInsensitiveSearch]
```

Sorting strings like Finder

To sort strings the way Finder does in OS X v10.6 and later, use the `localizedStandardCompare:` method. It should be used whenever file names or other strings are presented in lists and tables where Finder-like sorting is appropriate. The exact behavior of this method is different under different localizations, so clients should not depend on the exact sorting order of the strings.

The following example shows another implementation of similar functionality, comparing strings to order them in the same way as they're presented in Finder, and it also shows how to sort the array of strings. First, define a sorting function that includes the relevant comparison options (for efficiency, pass the user's locale as the context—this way it's only looked up once).

```
int finderSortWithLocale(id string1, id string2, void *locale)
{
    static NSStringCompareOptions comparisonOptions =
        NSCaseInsensitiveSearch | NSNumericSearch |
        NSWidthInsensitiveSearch | NSForcedOrderingSearch;

    NSRange string1Range = NSMakeRange(0, [string1 length]);

    return [string1 compare:string2
                    options:comparisonOptions
                    range:string1Range
                    locale:(NSLocale *)locale];
}
```

You pass the function as a parameter to `sortedArrayUsingFunction:context:` with the user's current locale as the context:

```
NSArray *stringsArray = @[@"string 1",
                          @"String 21",
                          @"string 12",
                          @"String 11",
                          @"String 02"];


NSArray *sortedArray = [stringsArray sortedArrayUsingFunction:finderSortWithLocale
                                     context:[NSLocale currentLocale]];


// sortedArray contains { "string 1", "String 02", "String 11", "string 12", "String
 21" }
```

Words, Paragraphs, and Line Breaks

This article describes how word and paragraph boundaries are defined, how line breaks are represented, and how you can separate a string by paragraph.

Word Boundaries

The text system determines word boundaries in a language-specific manner according to Unicode Standard Annex #29 with additional customization for locale as described in that document. On OS X, Cocoa presents APIs related to word boundaries, such as the `NSAttributedString` methods `doubleClickAtIndex:` and `nextWordFromIndex:forward:`, but you cannot modify the way the word-boundary algorithms themselves work.

Line and Paragraph Separator Characters

There are a number of ways in which a line or paragraph break can be represented. Historically, `\n`, `\r`, and `\r\n` have been used. Unicode defines an unambiguous paragraph separator, `U+2029` (for which Cocoa provides the constant `NSParagraphSeparatorCharacter`), and an unambiguous line separator, `U+2028` (for which Cocoa provides the constant `NSLineSeparatorCharacter`).

In the Cocoa text system, the `NSParagraphSeparatorCharacter` is treated consistently as a paragraph break, and `NSLineSeparatorCharacter` is treated consistently as a line break that is not a paragraph break—that is, a line break within a paragraph. However, in other contexts, there are few guarantees as to how these characters will be treated. POSIX-level software, for example, often recognizes only `\n` as a break. Some older Macintosh software recognizes only `\r`, and some Windows software recognizes only `\r\n`. Often there is no distinction between line and paragraph breaks.

Which line or paragraph break character you should use depends on how your data may be used and on what platforms. The Cocoa text system recognizes `\n`, `\r`, or `\r\n` all as paragraph breaks—equivalent to `NSParagraphSeparatorCharacter`. When it inserts paragraph breaks, for example with `insertNewline:`, it uses `\n`. Ordinarily `NSLineSeparatorCharacter` is used only for breaks that are specifically line breaks and not paragraph breaks, for example in `insertLineBreak:`, or for representing HTML `<br>` elements.

If your breaks are specifically intended as line breaks and not paragraph breaks, then you should typically use `NSLineSeparatorCharacter`. Otherwise, you may use `\n`, `\r`, or `\r\n` depending on what other software is likely to process your text. The default choice for Cocoa is usually `\n`.

Separating a String "by Paragraph"

A common approach to separating a string "by paragraph" is simply to use:

```
NSArray *arr = [myString componentsSeparatedByString:@"\n"];
```

This, however, ignores the fact that there are a number of other ways in which a paragraph or line break may be represented in a string—`\r`, `\r\n`, or Unicode separators. Instead you can use methods—such as `lineRangeForRange:` or `getParagraphStart:end:contentsEnd:forRange:`—that take into account the variety of possible line terminations, as illustrated in the following example.

```
NSString *string = /* assume this exists */;
unsigned length = [string length];
unsigned paraStart = 0, paraEnd = 0, contentsEnd = 0;
NSMutableArray *array = [NSMutableArray array];
NSRange currentRange;
while (paraEnd < length) {
    [string getParagraphStart:&paraStart end:&paraEnd
    contentsEnd:&contentsEnd forRange:NSMakeRange(paraEnd, 0)];
    currentRange = NSMakeRange(paraStart, contentsEnd - paraStart);
    [array addObject:[string substringWithRange:currentRange]];
}
```

Characters and Grapheme Clusters

It's common to think of a string as a sequence of characters, but when working with `NSString` objects, or with Unicode strings in general, in most cases it is better to deal with substrings rather than with individual characters. The reason for this is that what the user perceives as a character in text may in many cases be represented by multiple characters in the string. `NSString` has a large inventory of methods for properly handling Unicode strings, which in general make Unicode compliance easy, but there are a few precautions you should observe.

`NSString` objects are conceptually UTF-16 with platform endianness. That doesn't necessarily imply anything about their internal storage mechanism; what it means is that `NSString` lengths, character indexes, and ranges are expressed in terms of UTF-16 units, and that the term "character" in `NSString` method names refers to 16-bit platform-endian UTF-16 units. This is a common convention for string objects. In most cases, clients don't need to be overly concerned with this; as long as you are dealing with substrings, the precise interpretation of the range indexes is not necessarily significant.

The vast majority of Unicode code points used for writing living languages are represented by single UTF-16 units. However, some less common Unicode code points are represented in UTF-16 by surrogate pairs. A surrogate pair is a sequence of two UTF-16 units, taken from specific reserved ranges, that together represent a single Unicode code point. CFString has functions for converting between surrogate pairs and the UTF-32 representation of the corresponding Unicode code point. When dealing with `NSString` objects, one constraint is that substring boundaries usually should not separate the two halves of a surrogate pair. This is generally automatic for ranges returned from most Cocoa methods, but if you are constructing substring ranges yourself you should keep this in mind. However, this is not the only constraint you should consider.

In many writing systems, a single character may be composed of a base letter plus an accent or other decoration. The number of possible letters and accents precludes Unicode from representing each combination as a single code point, so in general such combinations are represented by a base character followed by one or more combining marks. For compatibility reasons, Unicode does have single code points for a number of the most common combinations; these are referred to as precomposed forms, and Unicode normalization transformations can be used to convert between precomposed and decomposed representations. However, even if a string is fully precomposed, there are still many combinations that must be represented using a base character and combining marks. For most text processing, substring ranges should be arranged so that their boundaries do not separate a base character from its associated combining marks.

In addition, there are writing systems in which characters represent a combination of parts that are more complicated than accent marks. In Korean, for example, a single Hangul syllable can be composed of two or three subparts known as jamo. In the Indic and Indic-influenced writing systems common throughout South and Southeast Asia, single written characters often represent combinations of consonants, vowels, and marks such as viramas, and the Unicode representations of these writing systems often use code points for these individual parts, so that a single character may be composed of multiple code points. For most text processing, substring ranges should also be arranged so that their boundaries do not separate the jamo in a single Hangul syllable, or the components of an Indic consonant cluster.

In general, these combinations—surrogate pairs, base characters plus combining marks, Hangul jamo, and Indic consonant clusters—are referred to as grapheme clusters. In order to take them into account, you can use `NSString`'s `rangeOfComposedCharacterSequencesForRange:` or `rangeOfComposedCharacterSequenceAtIndex:` methods, or `CFStringGetRangeOfComposedCharactersAtIndex`. These can be used to adjust string indexes or substring ranges so that they fall on grapheme cluster boundaries, taking into account all of the constraints mentioned above. These methods should be the default choice for programmatically determining the boundaries of user-perceived characters.:

In some cases, Unicode algorithms deal with multiple characters in ways that go beyond even grapheme cluster boundaries. Unicode casing algorithms may convert a single character into multiple characters when going from lowercase to uppercase; for example, the standard uppercase equivalent of the German character "ß" is the two-letter sequence "SS". Localized collation algorithms in many languages consider multiple-character sequences as single units; for example, the sequence "ch" is treated as a single letter for sorting purposes in some European languages. In order to deal properly with cases like these, it is important to use standard `NSString` methods for such operations as casing, sorting, and searching, and to use them on the entire string to which they are to apply. Use `NSString` methods such as `lowercaseString`, `uppercaseString`, `capitalizedString`, `compare:` and its variants, `rangeOfString:` and its variants, and `rangeOfCharacterFromSet:` and its variants, or their CFString equivalents. These all take into account the complexities of Unicode string processing, and the searching and sorting methods in particular have many options to control the types of equivalences they are to recognize.

In some less common cases, it may be necessary to tailor the definition of grapheme clusters to a particular need. The issues involved in determining and tailoring grapheme cluster boundaries are covered in detail in Unicode Standard Annex #29, which gives a number of examples and some algorithms. The Unicode standard in general is the best source for information about Unicode algorithms and the considerations involved in processing Unicode strings.

If you are interested in grapheme cluster boundaries from the point of view of cursor movement and insertion point positioning, and you are using the Cocoa text system, you should know that on OS X v10.5 and later, `NSLayoutManager` has API support for determining insertion point positions within a line of text as it is laid out. Note that insertion point boundaries are not identical to glyph boundaries; a ligature glyph in some cases, such as an “fi” ligature in Latin script, may require an internal insertion point on a user-perceived character boundary. See *Cocoa Text Architecture Guide* for more information.

Character Sets

An NSCharacterSet object represents a set of Unicode characters. NSString and NSScanner objects use NSCharacterSet objects to group characters together for searching operations, so that they can find any of a particular set of characters during a search.

Character Set Basics

A character set object represents a set of Unicode characters. Character sets are represented by instances of a class cluster. The cluster's two public classes, NSCharacterSet and NSMutableCharacterSet, declare the programmatic interface for immutable and mutable character sets, respectively. An immutable character set is defined when it is created and subsequently cannot be changed. A mutable character set can be changed after it's created.

A character set object doesn't perform any tasks; it simply holds a set of character values to limit operations on strings. The NSString and NSScanner classes define methods that take NSCharacterSet objects as arguments to find any of several characters. For example, this code excerpt finds the range of the first uppercase letter in myString:.

```
NSString *myString = @"some text in an NSString...";
NSCharacterSet *characterSet = [NSCharacterSet uppercaseLetterCharacterSet];
NSRange letterRange = [myString rangeOfCharacterFromSet:characterSet];
```

After this fragment executes, letterRange.location is equal to the index of the first "N" in "NSString" after rangeOfCharacterFromSet: is invoked. If the first letter of the string were "S", then letterRange.location would be 0.

Creating Character Sets

NSCharacterSet defines class methods that return commonly used character sets, such as letters (uppercase or lowercase), decimal digits, whitespace, and so on. These "standard" character sets are always immutable, even if created by sending a message to NSMutableCharacterSet. See Standard Character Sets and Unicode Definitions (page 35) for more information on standard character sets.

You can use a standard character set as a starting point for building a custom set by making a mutable copy of it and changing that. (You can also start from scratch by creating a mutable character set with `alloc` and `init` and adding characters to it.) For example, this fragment creates a character set containing letters, digits, and basic punctuation:

```
NSMutableCharacterSet *workingSet = [[NSCharacterSet alphanumericCharacterSet]
mutableCopy];
[workingSet addCharactersInString:@";:,."];
NSCharacterSet *finalCharacterSet = [workingSet copy];
```

To define a custom character set using Unicode code points, use code similar to the following fragment (which creates a character set including the form feed and line separator characters):

```
UniChar chars[] = {0x000C, 0x2028};
NSString *string = [[NSString alloc] initWithCharacters:chars
                           length:sizeof(chars) / sizeof(UniChar)];
NSCharacterSet *characterSet = [NSCharacterSet
characterSetWithCharactersInString:string];
```

Performance considerations

Because character sets often participate in performance-critical code, you should be aware of the aspects of their use that can affect the performance of your application. Mutable character sets are generally much more expensive than immutable character sets. They consume more memory and are costly to invert (an operation often performed in scanning a string). Because of this, you should follow these guidelines:

- Create as few mutable character sets as possible.
- Cache character sets (in a global dictionary, perhaps) instead of continually recreating them.
- When creating a custom set that doesn't need to change after creation, make an immutable copy of the final character set for actual use, and dispose of the working mutable character set. Alternatively, create a character set file as described in Creating a character set file (page 35) and store it in your application's main bundle.
- Similarly, avoid archiving character set objects; store them in character set files instead. Archiving can result in a character set being duplicated in different archive files, resulting in wasted disk space and duplicates in memory for each separate archive read.

Creating a character set file

If your application frequently uses a custom character set, you should save its definition in a resource file and load that instead of explicitly adding individual characters each time you need to create the set. You can save a character set by getting its bitmap representation (an `NSData` object) and saving that object to a file:

```
NSData *charSetRep = [finalCharacterSet bitmapRepresentation];
NSURL *dataURL = <#URL for character set#>;
NSError *error;
BOOL result = [charSetRep writeToURL:dataURL options:NSDataWritingAtomic
error:&error];
```

By convention, character set filenames use the extension `.bitmap`. If you intend for others to use your character set files, you should follow this convention. To read a character set file with a `.bitmap` extension, simply use the `characterSetWithContentsOfFile:` method.

Standard Character Sets and Unicode Definitions

The standard character sets, such as that returned by `letterCharacterSet`, are formally defined in terms of the normative and informative categories established by the Unicode standard, such as Uppercase Letter, Combining Mark, and so on. The formal definition of a standard character set is in most cases given as one or more of the categories defined in the standard. For example, the set returned by `lowercaseLetterCharacterSet` include all characters in normative category Lowercase Letters, while the set returned by `letterCharacterSet` includes the characters in all of the Letter categories.

Note that the definitions of the categories themselves may change with new versions of the Unicode standard. You can download the files that define category membership from http://www.unicode.org/.

Scanners

An NSScanner object scans the characters of an NSString object, typically interpreting the characters and converting them into number and string values. You assign the scanner's string on creation, and the scanner progresses through the characters of that string from beginning to end as you request items.

Creating a Scanner

NSScanner is a class cluster with a single public class, NSScanner. Generally, you instantiate a scanner object by invoking the class method scannerWithString: or localizedScannerWithString:. Either method returns a scanner object initialized with the string you pass to it. The newly created scanner starts at the beginning of its string. You scan components using the scan... methods such as scanInt:, scanDouble:, and scanString:intoString:. If you are scanning multiple lines, you typically create a while loop that continues until the scanner is at the end of the string, as illustrated in the following code fragment:

```
float aFloat;
NSScanner *theScanner = [NSScanner scannerWithString:aString];
while ([theScanner isAtEnd] == NO) {

    [theScanner scanFloat:&aFloat];
    // implementation continues...
}
```

You can configure a scanner to consider or ignore case using the setCaseSensitive: method. By default a scanner ignores case.

Using a Scanner

Scan operations start at the scan location and advance the scanner to just past the last character in the scanned value representation (if any). For example, after scanning an integer from the string "137 small cases of bananas", a scanner's location will be 3, indicating the space immediately after the number. Often you need to advance the scan location to skip characters in which you are not interested. You can change the implicit scan location with the `setScanLocation:` method to skip ahead a certain number of characters (you can also use the method to rescan a portion of the string after an error). Typically, however, you either want to skip characters from a particular character set, scan past a specific string, or scan up to a specific string.

You can configure a scanner to skip a set of characters with the `setCharactersToBeSkipped:` method. A scanner ignores characters to be skipped at the beginning of any scan operation. Once it finds a scannable character, however, it includes all characters matching the request. Scanners skip whitespace and newline characters by default. Note that case is always considered with regard to characters to be skipped. To skip all English vowels, for example, you must set the characters to be skipped to those in the string "AEIOUaeiou".

If you want to read content from the current location up to a particular string, you can use `scanUpToString:intoString:` (you can pass `NULL` as the second argument if you simply want to skip the intervening characters). For example, given the following string:

```
137 small cases of bananas
```

you can find the type of container and number of containers using `scanUpToString:intoString:` as shown in the following example.

```
NSString *bananas = @"137 small cases of bananas";
NSString *separatorString = @" of";

NSScanner *aScanner = [NSScanner scannerWithString:bananas];

NSInteger anInteger;
[aScanner scanInteger:&anInteger];
NSString *container;
[aScanner scanUpToString:separatorString intoString:&container];
```

It is important to note that the search string (`separatorString`) is `" of"`. By default a scanner ignores whitespace, so the space character after the integer is ignored. Once the scanner begins to accumulate characters, however, all characters are added to the output string until the search string is reached. Thus if the search string is `"of"` (no space before), the first value of `container` is "small cases " (includes the space following); if the search string is `" of"` (with a space before), the first value of `container` is "small cases" (no space following).

After scanning up to a given string, the scan location is the beginning of that string. If you want to scan past that string, you must therefore first scan in the string you scanned up to. The following code fragment illustrates how to skip past the search string in the previous example and determine the type of product in the container. Note the use of `substringFromIndex:` to in effect scan up to the end of a string.

```
[aScanner scanString:separatorString intoString:NULL];
NSString *product;
product = [[aScanner string] substringFromIndex:[aScanner scanLocation]];
// could also use:
// product = [bananas substringFromIndex:[aScanner scanLocation]];
```

Example

Suppose you have a string containing lines such as:

Product: Acme Potato Peeler; Cost: 0.98 73

Product: Chef Pierre Pasta Fork; Cost: 0.75 19

Product: Chef Pierre Colander; Cost: 1.27 2

The following example uses alternating scan operations to extract the product names and costs (costs are read as a `float` for simplicity's sake), skipping the expected substrings "Product:" and "Cost:", as well as the semicolon. Note that because a scanner skips whitespace and newlines by default, the loop does no special processing for them (in particular there is no need to do additional whitespace processing to retrieve the final integer).

```
NSString *string = @"Product: Acme Potato Peeler; Cost: 0.98 73\n\
Product: Chef Pierre Pasta Fork; Cost: 0.75 19\n\
Product: Chef Pierre Colander; Cost: 1.27 2\n";

NSCharacterSet *semicolonSet;
NSScanner *theScanner;

NSString *PRODUCT = @"Product:";
NSString *COST = @"Cost:";

NSString *productName;
float productCost;
```

```
NSInteger productSold;

semicolonSet = [NSCharacterSet characterSetWithCharactersInString:@";"];
theScanner = [NSScanner scannerWithString:string];

while ([theScanner isAtEnd] == NO)
{
    if ([theScanner scanString:PRODUCT intoString:NULL] &&
        [theScanner scanUpToCharactersFromSet:semicolonSet
            intoString:&productName] &&
        [theScanner scanString:@";" intoString:NULL] &&
        [theScanner scanString:COST intoString:NULL] &&
        [theScanner scanFloat:&productCost] &&
        [theScanner scanInteger:&productSold])
    {
        NSLog(@"Sales of %@: $%1.2f", productName, productCost * productSold);
    }
}
```

Localization

A scanner bases some of its scanning behavior on a locale, which specifies a language and conventions for value representations. `NSScanner` uses only the locale's definition for the decimal separator (given by the key named `NSDecimalSeparator`). You can create a scanner with the user's locale by using `localizedScannerWithString:`, or set the locale explicitly using `setLocale:`. If you use a method that doesn't specify a locale, the scanner assumes the default locale values.

String Representations of File Paths

NSString provides a rich set of methods for manipulating strings as file-system paths. You can extract a path's directory, filename, and extension, expand a tilde expression (such as "~me") or create one for the user's home directory, and clean up paths containing symbolic links, redundant slashes, and references to "." (current directory) and ".." (parent directory).

Note: Where possible, you should use instances of NSURL to represent paths—the operating system deals with URLs more efficiently than with string representations of paths.

Representing a Path

NSString represents paths generically with '/' as the path separator and '.' as the extension separator. Methods that accept strings as path arguments convert these generic representations to the proper system-specific form as needed. On systems with an implicit root directory, absolute paths begin with a path separator or with a tilde expression ("~/..." or "~user/..."). Where a device must be specified, you can do that yourself—introducing a system dependency—or allow the string object to add a default device.

You can create a standardized representation of a path using stringByStandardizingPath. This performs a number of tasks including:

- Expansion of an initial tilde expression;
- Reduction of empty components and references to the current directory ("//" and "/./") to single path separators;
- In absolute paths, resolution of references to the parent directory ("..") to the real parent directory;

for example:

```
NSString *path = @"/usr/bin/./grep";
NSString *standardizedPath = [path stringByStandardizingPath];
// standardizedPath: /usr/bin/grep


path = @"~me";
```

```
standardizedPath = [path stringByStandardizingPath];
// standardizedPath (assuming conventional naming scheme): /Users/Me

path = @"/usr/include/objc/..";
standardizedPath = [path stringByStandardizingPath];
// standardizedPath: /usr/include

path = @"/private/usr/include";
standardizedPath = [path stringByStandardizingPath];
// standardizedPath: /usr/include
```

User Directories

The following examples illustrate how you can use `NSString`'s path utilities and other Cocoa functions to get the user directories.

```
// Assuming that users' home directories are stored in /Users

NSString *meHome = [@"~me" stringByExpandingTildeInPath];
// meHome = @"/Users/me"

NSString *mePublic = [@"~me/Public" stringByExpandingTildeInPath];
// mePublic = @"/Users/me/Public"
```

You can find the home directory for the current user and for a given user with `NSHomeDirectory` and `NSHomeDirectoryForUser` respectively:

```
NSString *currentUserHomeDirectory = NSHomeDirectory();
NSString *meHomeDirectory = NSHomeDirectoryForUser(@"me");
```

Note that you should typically use the function `NSSearchPathForDirectoriesInDomains` to locate standard directories for the current user. For example, instead of:

```
NSString *documentsDirectory =
                [NSHomeDirectory() stringByAppendingPathComponent:@"Documents"];
```

you should use:

```
NSString *documentsDirectory;
NSArray *paths = NSSearchPathForDirectoriesInDomains(NSDocumentDirectory,
NSUserDomainMask, YES);
if ([paths count] > 0) {
    documentsDirectory = [paths objectAtIndex:0];
}
```

Path Components

`NSString` provides a rich set of methods for manipulating strings as file-system paths, for example:

`pathExtension`	The path extension, if any, of the string as interpreted as a path. (read-only)
`stringByDeletingPathExtension`	A new string made by deleting the extension (if any, and only the last) from the receiver. (read-only)
`stringByDeletingLastPathComponent`	A new string made by deleting the last path component from the receiver, along with any final path separator. (read-only)

Using these and related methods described in *NSString Class Reference*, you can extract a path's directory, filename, and extension, as illustrated by the following examples.

```
NSString *documentPath = @"~me/Public/Demo/readme.txt";


NSString *documentDirectory = [documentPath stringByDeletingLastPathComponent];
// documentDirectory = @"~me/Public/Demo"


NSString *documentFilename = [documentPath lastPathComponent];
// documentFilename = @"readme.txt"


NSString *documentExtension = [documentPath pathExtension];
// documentExtension = @"txt"
```

File Name Completion

You can find possible expansions of file names using `completePathIntoString:caseSensitive:matchesIntoArray:filterTypes:`. For example, given a directory `~/Demo` that contains the following files:

```
ReadMe.txt readme.html readme.rtf recondite.txt test.txt
```

you can find all possible completions for the path `~/Demo/r` as follows:

```
NSString *partialPath = @"~/Demo/r";
NSString *longestCompletion;
NSArray *outputArray;

unsigned allMatches = [partialPath completePathIntoString:&longestCompletion
    caseSensitive:NO
    matchesIntoArray:&outputArray
    filterTypes:nil];

// allMatches = 3
// longestCompletion = @"~/Demo/re"
// outputArray = (@"~/Demo/readme.html", "~/Demo/readme.rtf", "~/Demo/recondite.txt")
```

You can find possible completions for the path `~/Demo/r` that have an extension ".txt" or ".rtf" as follows:

```
NSArray *filterTypes = @[@"txt", @"rtf"];

unsigned textMatches = [partialPath completePathIntoString:&outputName
    caseSensitive:NO
    matchesIntoArray:&outputArray
    filterTypes:filterTypes];
// allMatches = 2
// longestCompletion = @"~/Demo/re"
// outputArray = (@"~/Demo/readme.rtf", @"~/Demo/recondite.txt")
```

Drawing Strings

You can draw string objects directly in a focused `NSView` using methods such as `drawAtPoint:withAttributes:` (to draw a string with multiple attributes, such as multiple text fonts, you must use an `NSAttributedString` object). These methods are described briefly in Text in *Cocoa Drawing Guide* .

The simple methods, however, are designed for drawing small amounts of text or text that is only drawn rarely—they create and dispose of various supporting objects every time you call them. To draw strings repeatedly, it is more efficient to use `NSLayoutManager`, as described in Drawing Strings in *Text Layout Programming Guide* . For an overview of the Cocoa text system, of which `NSLayoutManager` is a part, see *Cocoa Text Architecture Guide* .

Document Revision History

This table describes the changes to *String Programming Guide* .

Date	Notes
2014-02-11	Added information about word boundary delineation and changed section title to "Words, Paragraphs, and Line Breaks."
2013-09-17	Fixed typo in code snippet in "Formatting String Objects" article.
2012-12-13	Clarified description of NSAnchoredSearch option.
2012-07-17	Updated code snippets to adopt new Objective-C features.
2012-06-11	Corrected string constant character set to UTF-8. Added guidance about using localizedStandardCompare: for Finder-like sorting. Added caveat to avoid using %s with RTL languages. Revised "String Format Specifiers" article.
2009-10-15	Added links to Cocoa Core Competencies.
2008-10-15	Added new aricle on character clusters; updated list of string format specifiers.
2007-10-18	Corrected minor typographical errors.
2007-07-10	Added notes regarding NSInteger and NSUInteger to "String Format Specifiers".
2007-03-06	Corrected minor typographical errors.
2007-02-08	Corrected sentence fragments and improved the example in "Scanners."
2006-12-05	Added code samples to illustrate searching and path manipulation.
2006-11-07	Made minor revisions to "Scanners" article.

Date	Notes
2006-10-03	Added links to path manipulation methods.
2006-06-28	Corrected typographical errors.
2006-05-23	Added a new article, "Reading Strings From and Writing Strings To Files and URLs"; significantly updated "Creating and Converting Strings." Included "Creating a Character Set" into Character Sets (page 33).
2006-01-10	Changed title from "Strings" to conform to reference consistency guidelines.
2004-06-28	Added Formatting String Objects (page 13) article. Added *Data Formatting* and the Core Foundation *Strings* programming topics to the introduction.
2004-02-06	Added information about custom Unicode character sets and retrieved missing code fragments in "Creating a Character Set". Added information and cross-reference to Drawing Strings (page 44). Rewrote introduction and added an index.
2003-09-09	Added `NSNumericSearch` description to Searching, Comparing, and Sorting Strings (page 22).
2003-03-17	Reinstated the sample code that was missing from Scanners (page 36).
2003-01-17	Updated Creating and Converting String Objects (page 8) to recommend the use of UTF8 encoding, and noted the pending deprecation of the `cString...` methods.
2002-11-12	Revision history was added to existing topic.

Index