NSString Class Cluster

Class Cluster Description

NSString objects represent character strings in OpenStep 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. The cluster's two public classes, NSString and NSMutableString, declare the programmatic interface for noneditable and editable strings, respectively.

Note: An immutable string is a text string that is defined when it is created and subsequently cannot be changed. An immutable string is implemented as array of Unicode characters (in other words, a text string). 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 NSMutableStringare referred to as *string objects* (or, when no confusion will result, merely as *strings*). The term *C string* refers to the standard **char *** type. Because of the nature of class clusters, string objects aren't actual instances of the NSString or NSMutableString classes but of one of their private subclasses. Although a string object's class is private, its interface is public, as declared by these abstract superclasses, NSString and NSMutableString. (See "Class Clusters" in the introduction to the Foundation Kit for more information on class clusters and on creating subclasses within a cluster.) The string classes adopt the NSCopying and NSMutableCopying protocols, making it convenient to convert a string of one type to the other.

A string object presents itself as an array of Unicode® characters (Unicode is a registered trademark of Unicode, Inc.). 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: Worldwide Character Encoding*, Version 1.0, Volumes 1 and 2. The Unicode Consortium. Addison-Wesley. ISBN 0–201–56788–1 (Volume 1) and 0–201–60845–6 (Volume 2).
>
> Version 2.0, also published by Addison-Wesley, is forthcoming. ISBN 0–201–48345–9.

Creating and Converting String Objects

NSString provides several means for creating instances, 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.

The simplest way to create a string object in source code is to use either the **stringWithCString:** class method or the **initWithCString:** instance method. Each takes a standard null-terminated C string in the default C string encoding and produces a string object. As a convenience, the Objective-C language also supports the **@"..."** construct to create a string object constant from 7-bit ASCII encoding:

```
NSString *temp = @"/tmp/scratch";
```

Such an object 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 (though you can retain and release them as you do any other object).

To get a C string from a string object, use the **cString** message. This returns a **char *** in the system's default string encoding, or raises an exception if it can't convert its contents to that encoding. The C string you receive is owned by a temporary object, though, so it will become invalid when automatic deallocation takes place (see "Object Ownership and Automatic Disposal" in the introduction to the Foundation Kit for further information). If you want to get a permanent C string, you must create a buffer and use one of the **getCString:...** methods to fill it. You can find out how large the buffer needs to be with the **cStringLength** 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. You can also read a string directly from a file in the Unicode or default C string encoding using the **stringWithContentsOfFile:** class method, and write a string using **writeToFile:atomically:**.

Finally, two types of method allow you to build a string from existing string objects. **localizedStringWithFormat:** and its 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. The methods **stringByAppendingString:** and **stringByAppendingFormat:** create a new string by adding one string after another, in the second case using a format string.

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 function **printf()**, 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.

Note: Many compilers perform typecasting of arguments to **printf()**, so that printing an integer into a **%f** (floating-point) placeholder works as expected. This typecasting doesn't occur with NSString's formatted methods, so be sure to cast your values explicitly.

Value formatting is affected by the user's current locale, which is an NSDictionary specifying 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. See "Locales" in the "Other Features" section of the Foundation Kit documentation for more information on locales.

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

Source	Creation Method	Extraction Method
Default C string encoding	stringWithCString:	getCString: (or cString)
In code	@"..." compiler construct	
Unicode encoding	stringWithCharacters:length:	getCharacters:length:
Arbitrary encoding	initWithData:encoding:	dataUsingEncoding:
File contents	stringWithContentsOfFile:	writeToFile:atomically:
Format string	localizedStringWithFormat:	initWithFormat:locale:
Existing strings	stringByAppendingString:	stringByAppendingFormat:

Working with String Objects

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.

The search and comparison methods each come in three 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 specify these options (not all options are available for every method):

Search Option	Effect
NSCaseInsensitiveSearch	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 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.

Note: Search and comparison are currently performed as if the NSLiteralSearch option were specified. As the Unicode encoding becomes more widely used, and the need for more flexible comparison increases, the default behavior will be changed accordingly.

Substrings are only found if completely contained within the specified range. If you specify a range for a search or comparison method and don't request NSLiteralSearch, 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.)

The basic search and comparison methods are these:

rangeOfString:	compare:
rangeOfString:options:	compare:options:
rangeOfStrings:options:range:	compare:options:range:

rangeOfCharacterFromSet:
rangeOfCharacterFromSet:options:
rangeOfCharacterFromSet:options:range:

The **rangeOfString:...** methods search for a substring within the receiver. The **rangeOfCharacterFromSet:...** methods search for individual characters from a supplied set of characters. 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.

In addition to searching and comparing strings, you can combine and divide them in various ways. The simplest way to put two strings together is to append one to the other. The **stringByAppendingString:** method returns a string object formed from the receiver and the argument supplied. You can also combine several strings according to a template with the **initWithFormat:**, **stringWithFormat:** and **stringByAppendingFormat:** methods. See ""Creating and Converting String Objects"" for more information.

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.

Most of the NSString classes' remaining methods are for conveniences such as changing case, quickly extracting numeric values, and working with encodings. There's also a set of methods for treating strings as file system paths, described below in ""Manipulating Paths"." An additional class cluster, NSScanner, allows you to scan a string object for numeric and string values. Both the NSString and the NSScanner class clusters use the NSCharacterSet class cluster for search operations. See the appropriate class specifications for more information.

Manipulating Paths

In addition to all the basic methods for working with character strings merely as strings, NSString also provides a rich set of methods for manipulating strings as file system paths. A string can extract a path's directory, file name, 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). These methods are listed under "Working with paths" in the "MethodTypes"""" section.

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*/..."). On systems that require explicit expression of root directories for different devices, such as Microsoft Windows 95, absolute paths begin with the name of the device (for example, "C:/Documents/Paper.doc" to represent the actual path "C:\Documents\Paper.doc"). 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.

NSString

Inherits From:	NSObject
Conforms To:	NSCoding NSCopying NSMutableCopying NSObject (NSObject)
Declared In:	Foundation/NSString.h Foundation/NSPathUtilities.h

Class Description

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

The NSString class has two primitive methods—**length** and **characterAtIndex:**—that provide the basis for all other methods in its interface. The **length** method returns the total number of Unicode characters in the string. **characterAtIndex:** gives access to each character in the string by index, with index values starting at 0.

NSString declares methods for finding and comparing strings. It also declares methods for reading numeric values from strings, for combining strings in various ways, and for converting a string to different forms (such as encoding and case changes). General use of these methods is presented in the class cluster description under ““Working with String Objects”.”

Adopted Protocols

NSCoding	– encodeWithCoder: – initWithCoder:
NSCopying	– copyWithZone:
NSMutableCopying	– mutableCopyWithZone:

Method Types

Creating temporary strings	+ stringWithCString: + stringWithCString:length: + stringWithFormat: + localizedStringWithFormat: + stringWithContentsOfFile: + stringWithCharacters:length: + string + stringWithString:
Initializing newly allocated strings	– initWithCharactersNoCopy:length:freeWhenDone: – initWithCharacters:length: – initWithCStringNoCopy:length:freeWhenDone: – initWithCString:length: – initWithCString: – initWithString: – initWithFormat: – initWithFormat:arguments: – initWithFormat:locale: – initWithFormat:locale:arguments: – initWithData:encoding: – initWithContentsOfFile: – init
Getting a string's length	– length
Accessing characters	– characterAtIndex: – getCharacters: – getCharacters:range:
Combining strings	– stringByAppendingFormat: – stringByAppendingString:
Dividing strings	– componentsSeparatedByString: – substringFromIndex: – substringWithRange: – substringToIndex:
Finding characters and substrings	– rangeOfCharacterFromSet: – rangeOfCharacterFromSet:options: – rangeOfCharacterFromSet:options:range: – rangeOfString: – rangeOfString:options: – rangeOfString:options:range:
Determining line ranges	– getLineStart:end:contentsEnd:forRange: – lineRangeForRange:

Determining composed character sequences
– rangeOfComposedCharacterSequenceAtIndex:

Converting string contents into a property list
– propertyList
– propertyListFromStringsFileFormat

Identifying and comparing strings
– caseInsensitiveCompare:
– compare:
– compare:options:
– compare:options:range:
– hasPrefix:
– hasSuffix:
– isEqualToString:
– hash

Getting a shared prefix – commonPrefixWithString:options:

Changing case
– capitalizedString
– lowercaseString
– uppercaseString

Getting C strings
– cString
– lossyCString
– cStringLength
– getCString:
– getCString:maxLength:
– getCString:maxLength:range:remainingRange:

Getting numeric values
– doubleValue
– floatValue
– intValue

Working with encodings
+ availableStringEncodings
+ defaultCStringEncoding
+ localizedNameOfStringEncoding:
– canBeConvertedToEncoding:
– dataUsingEncoding:
– dataUsingEncoding:allowLossyConversion:
– description
– fastestEncoding
– smallestEncoding

Working with paths	+ pathWithComponents: – pathComponents – completePathIntoString:caseSensitive: matchesIntoArray:filterTypes: – fileSystemRepresentation – getFileSystemRepresentation:maxLength: – isAbsolutePath – lastPathComponent – pathExtension – stringByAbbreviatingWithTildeInPath – stringByAppendingPathComponent: – stringByAppendingPathExtension: – stringByDeletingLastPathComponent – stringByDeletingPathExtension – stringByExpandingTildeInPath – stringByResolvingSymlinksInPath – stringByStandardizingPath – stringsByAppendingPaths:
Writing to a file	– writeToFile:atomically:

Class Methods

availableStringEncodings

+ (const NSStringEncoding *)**availableStringEncodings**

Returns a zero-terminated list of the encodings that string objects support in the application's environment. Among the more commonly used are:

NSASCIIStringEncoding
NSNEXTSTEPStringEncoding
NSUnicodeStringEncoding
NSISOLatin1StringEncoding
NSISOLatin2StringEncoding
NSSymbolStringEncoding

See the "Types and Constants" section of the Foundation Kit documentation for a complete list and descriptions of supported encodings.

See also: + **localizedNameOfStringEncoding:**

defaultCStringEncoding

+ (NSStringEncoding)**defaultCStringEncoding**

Returns the C string encoding assumed for any method accepting a C string as an argument (these methods use **CString** in the keywords for such arguments; for example, **stringWithCString:**). The default C string encoding is determined from system information, and can't be changed programmatically for an individual process. See the description of NSStringEncoding in the "Types and Constants" section for a full list of supported encodings.

localizedNameOfStringEncoding:

+ (NSString *)**localizedNameOfStringEncoding:**(NSStringEncoding)*encoding*

Returns a human-readable string giving the name of *encoding* in the current locale's language. See "Locales" in the "Other Features" section of the Foundation Kit documentation for more information on locales.

localizedStringWithFormat:

+ (NSString *)**localizedStringWithFormat:**(NSString *)*format, ...*

Returns a string created by using *format* as a template into which the following argument values are substituted according to the formatting information of the current locale. For example, this code excerpt creates a string from another string and an **int**:

```
NSString *myString = [NSString localizedStringWithFormat:@"%@: %d\n",
    @"Cost", 32];
```

The resulting string has the value "Cost: 32\n".

See ""Creating and Converting String Objects"" in the class cluster description for more information.

pathWithComponents:

+ (NSString *)**pathWithComponents:**(NSArray *)*components*

Returns a string built from the strings in *components*, by concatenating them with a path separator between each pair. To create an absolute path, use a slash mark "/" as the first component. To include a trailing path divider, use an empty string as the last component. This method doesn't clean up the path created; use **stringByStandardizingPath** to resolve empty components, references to the parent directory, and so on.

On systems that require an explicit root device for an absolute path, this method add a default device specifier (such as "C:" on Windows systems).

See also: – **pathComponents**

string

+ (id)**string**

Returns an empty string.

See also: – **init**

stringWithCharacters:length:

+ (id)**stringWithCharacters:**(const unichar *)*chars* **length:**(unsigned int)*length*

Returns a string containing *length* characters taken from *chars*, which may not be NULL. This method doesn't stop short at a zero character.

See also: – **initWithCharacters:length:**

stringWithContentsOfFile:

+ (id)**stringWithContentsOfFile:**(NSString *)*path*

Returns a string created by reading characters from the file named by *path*. If the contents begin with a byte-order mark (U+FEFF or U+FFFE), interprets the contents as Unicode characters; otherwise interprets the contents as characters in the default C string encoding. Returns **nil** if the file can't be opened.

See also: – **initWithContentsOfFile:**, + **defaultCStringEncoding**

stringWithCString:

+ (id)**stringWithCString:**(const char *)*cString*

Returns a string containing derived from the characters in *cString*, which must end with a zero character and which may not be NULL. *cString* should contain characters in the default C string encoding. If the argument passed to **stringWithCString** is not a NULL-terminated C string, the results are undefined.

See also: – **initWithCString:**, + **defaultCStringEncoding**

stringWithCString:length:

+ (id)**stringWithCString:**(const char *)*cString* **length:**(unsigned int)*length*

Returns a string containing characters derived from *cString*, which may not be NULL. *cString* should contain characters in the default C string encoding. This method converts *length* * **sizeof(char)** bytes from *cString*, and doesn't stop short at a zero character.

See also: – **initWithCString:length:**, + **defaultCStringEncoding**

stringWithFormat:

+ (id)**stringWithFormat:**(NSString *)*format*, ...

Returns a string created in the manner of **localizedStringWithFormat:**, but using the default locale to format numbers. See ““Creating and Converting String Objects”” in the class cluster description for more information.

See also: – **initWithFormat:**

stringWithString:

+ (id)**stringWithString:**(NSString *)*aString*

Returns a string created by copying the characters from *aString*.

See also: – **initWithString:**

nstance Methods

canBeConvertedToEncoding:

– (BOOL)**canBeConvertedToEncoding:**(NSStringEncoding)*encoding*

Returns YES if the receiver can be converted to *encoding* without loss of information. Returns NO if characters would have to be changed or deleted in the process of changing encodings.

If you plan to actually convert a string, the **dataUsingEncoding:...** methods simply return **nil** on failure, so you can avoid the overhead of invoking this method yourself by simply trying to convert the string.

See also: – **dataUsingEncoding:allowLossyConversion:**

capitalizedString

– (NSString *)**capitalizedString**

Returns a string with the first character from each word in the receiver changed to its corresponding uppercase value, and all remaining characters set to their corresponding lowercase values. A “word” here is any sequence of characters delimited by spaces, tabs, or line terminators (listed under **getLineStart:end:contentsEnd:forRange:**). Other common word delimiters such as hyphens and other punctuation aren’t considered, so this method may not generally produce the desired results for multi-word strings.

Case transformations aren’t guaranteed to be symmetrical or to produce strings of the same lengths as the originals. See **lowercaseString** for an example.

See also: – **lowercaseString**, – **uppercaseString**

caseInsensitiveCompare:

– (NSComparisonResult)**caseInsensitiveCompare:**(NSString *)*aString*

This convenience method invokes **compare:options:** with NSCaseInsensitiveSearch as the only option.

characterAtIndex:

– (unichar)**characterAtIndex:**(unsigned int)*index*

Returns the character at the array position given by *index*. Raises an NSRangeException if *index* lies beyond the end of the string.

See also: – **getCharacters:**, – **getCharacters:range:**

commonPrefixWithString:options:

– (NSString *)**commonPrefixWithString:**(NSString *)*aString*
options:(unsigned int)*mask*

Returns a string containing characters that the receiver and *aString* have in common, starting from the beginning of each up to the first characters that aren't equivalent. The returned string is based on the characters of the receiver. For example, if the receiver is "Ma¨dchen" and *aString* is "Mädchenschule", the string returned is "Ma¨dchen", not "Mädchen". The following search options may be specified in *mask* by combining them with the C bitwise OR operator:

NSCaseInsensitiveSearch
NSLiteralSearch

See ""Working with String Objects"" in the class cluster description for details on these options.

See also: – **hasPrefix**

compare:

– (NSComparisonResult)**compare:**(NSString *)*aString*

Invokes **compare:options:** with no options.

See also: – **compare:options:range:**, – **caseInsensitiveCompare:**, – **isEqualToString:**

compare:options:

– (NSComparisonResult)**compare:**(NSString *)*aString* **options:**(unsigned int)*mask*

Invokes **compare:options:range:** with *mask* as the options and the receiver's full extent as the range.

See also: – **caseInsensitiveCompare:**, – **isEqualToString:**

compare:options:range:

– (NSComparisonResult)**compare:**(NSString *)*aString*
options:(unsigned int)*mask*
range:(NSRange)*aRange*

Returns NSOrderedAscending if the substring of the receiver given by *aRange* precedes *aString* in lexical ordering, NSOrderedSame if the substring of the receiver and *aString* are equivalent in lexical value, and NSOrderedDescending if the substring of the receiver follows *aString*. You can specify the following options in *mask* by combining them with the C bitwise OR operator:

NSCaseInsensitiveSearch
NSLiteralSearch

See ""Working with String Objects"" in the class cluster description for details on these options.

Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

See also: – **caseInsensitiveCompare:**, – **isEqualToString:**

completePathIntoString:caseSensitive:matchesIntoArray:filterTypes:

– (unsigned int)**completePathIntoString:**(NSString **)*outputName*
caseSensitive:(BOOL)*flag*
matchesIntoArray:(NSArray **)*outputArray*
filterTypes:(NSArray *)*filterTypes*

Attempts to perform file- name completion on the receiver, interpreting it as a path in the file system and returning by reference in *outputName* the longest path that matches the receiver. Considers case if *flag* is YES. If *outputArray* is non-NULL, returns all matching file names in an NSArray given by *outputArray*. If an array of strings is provided in *filterTypes*, considers only paths whose extensions (not including the extension separator) match one of those strings.

Returns 0 if no matches are found and 1 if exactly one match is found. In the case of multiple matches, returns the actual number of matching paths if *outputArray* is provided, or simply a positive value if *outputArray* is NULL. Hence, you can check for the existence of matches without retrieving by passing NULL as *outputArray*

componentsSeparatedByString:

– (NSArray *)**componentsSeparatedByString:**(NSString *)*separator*

Returns an NSArray containing substrings from the receiver that have been divided by *separator*. The substrings in the array appear in the order they did in the receiver. If the string begins or ends with the separator, the first or last substring, respectively, is empty. For example, this code excerpt:

```
NSString *list = @"wrenches, hammers, saws";
NSArray *listItems = [list componentsSeparatedByString:@", "];
```

produces an array with these contents:

Index	Substring
0	wrenches
1	hammers
2	saws

If **list** begins with a comma and space—for example, ", wrenches, hammers, saws"—the array has these contents:

Index	Substring
0	*(empty string)*
1	wrenches
2	hammers
3	saws

If **list** has no separators—for example, "wrenches"—the array contains the string itself, in this case "wrenches".

See also: – **componentsJoinedByString:** (NSArray class cluster), – **pathComponents**

cString

– (const char *)**cString**

Returns a representation of the receiver as a C string in the default C string encoding. The returned C string will be automatically freed just as a returned object would be released; your code should copy the C string or use **getCString:** if it needs to store the C string outside of the autorelease context in which the C string is created.

Raises an NSCharacterConversionException if the receiver can't be represented in the default C string encoding without loss of information. Use **canBeConvertedToEncoding:** if necessary to check whether a string can be losslessly converted to the default C string encoding. If it can't, use **lossyCString** or

dataUsingEncoding:allowLossyConversion: to get a C string representation with some loss of information.

See also: – **getCString:**, – **canBeConvertedToEncoding:**, + **defaultCStringEncoding**, – **cStringLength**, – **getCharacters:**

cStringLength

– (unsigned int)**cStringLength**

Returns the length in **char**-sized units of the receiver's C string representation in the default C string encoding.

Raises 0 if the receiver can't be represented in the default C string encoding without loss of information. You can also use **canBeConvertedToEncoding:** to check whether a string can be losslessly converted to the default C string encoding. If it can't, use **lossyCString** to get a C string representation with some loss of information, then check its length explicitly using the ANSI function **strlen()**.

See also: – **cString**, – **canBeConvertedToEncoding:**, + **defaultCStringEncoding**, – **length**

dataUsingEncoding:

– (NSData *)**dataUsingEncoding:**(NSStringEncoding)*encoding*

Invokes **dataUsingEncoding:allowLossyConversion:** with NO as the argument for allowing lossy conversion.

dataUsingEncoding:allowLossyConversion:

– (NSData *)**dataUsingEncoding:**(NSStringEncoding)*encoding*
allowLossyConversion:(BOOL)*flag*

Returns an NSData object containing a representation of the receiver in *encoding*. Returns **nil** if *flag* is NO and the receiver can't be converted without losing some information (such as accents or case). If *flag* is YES and the receiver can't be converted without losing some information, some characters may be removed or altered in conversion. For example, in converting a character from NSUnicodeStringEncoding to NSASCIIStringEncoding, the character 'Á' becomes 'A', losing the accent.

The result of this method, when lossless conversion is made, is the default "plain text" format for *encoding* and is the recommended way to save or transmit a string object.

See also: – **canBeConvertedToEncoding:**

description

@protocol NSObject
– (NSString *)**description**

Returns **self**.

doubleValue

– (double)**doubleValue**

Returns the floating-point value of the receiver's text as a **double**, skipping whitespace at the beginning of the string. Returns HUGE_VAL or –HUGE_VAL on overflow, 0.0 on underflow. Also returns 0.0 if the receiver doesn't begin with a valid text representation of a floating-point number.

This method uses formatting information stored in the default locale; use an NSScanner for localized scanning of numeric values from a string.

See also: – **intValue**, – **floatValue**, – **scanDouble:** (NSScanner)

fastestEncoding

– (NSStringEncoding)**fastestEncoding**

Returns the fastest encoding to which the receiver may be converted without loss of information. "Fastest" applies to retrieval of characters from the string. This encoding may not be space efficient.

See also: – **smallestEncoding**, – **getCharacters:range:**

fileSystemRepresentation

– (const char *)**fileSystemRepresentation**

Returns a file system specific representation of the receiver, as described for **getFileSystemRepresentation:maxLength:**. The returned C string will be automatically freed just as a returned object would be released; your code should copy the representation or use **getFileSystemRepresentation:maxLength:** if it needs to store the representation outside of the autorelease context in which the representation is created.

Raises an NSCharacterConversionException if the receiver can't be represented in the file system's encoding.

floatValue

– (float)**floatValue**

Returns the floating-point value of the receiver's text as a **float**, skipping whitespace at the beginning of the string. Returns HUGE_VAL or –HUGE_VAL on overflow, 0.0 on underflow. Also returns 0.0 if the receiver doesn't begin with a valid text representation of a floating-point number.

This method uses formatting information stored in the default locale; use an NSScanner for localized scanning of numeric values from a string.

See also: – **doubleValue**, – **intValue**, – **scanFloat:** (NSScanner)

getCharacters:

– (void)**getCharacters:**(unichar *)*buffer*

Invokes **getCharacters:range:** with *buffer* and the entire extent of the receiver as the range. *buffer* must be large enough to contain all the characters in the string.

See also: – **length**

getCharacters:range:

– (void)**getCharacters:**(unichar *)*buffer* **range:**(NSRange)*aRange*

Copies characters from *aRange* in the receiver into *buffer*, which must be large enough to contain them. Does *not* add a zero character. Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

The abstract implementation of this method uses **characterAtIndex:** repeatedly, correctly extracting the characters, though very inefficiently. Subclasses should override it to provide a fast implementation.

getCString:

– (void)**getCString:**(char *)*buffer*

Invokes **getCString:maxLength:range:remainingRange:** with NSMaximumStringLength as the maximum length, the receiver's entire extent as the range, and NULL for the remaining range. *buffer* must be large enough to contain the resulting C string plus a terminating zero character (which this method adds).

Raises an NSCharacterConversionException if the receiver can't be represented in the default C string encoding without loss of information. Use **canBeConvertedToEncoding:** if necessary to check whether a string can be losslessly converted to the default C string encoding. If it can't, use **lossyCString** or

dataUsingEncoding:allowLossyConversion: to get a C string representation with some loss of information.

See also: – **cStringLength**, – **canBeConvertedToEncoding:**, + **defaultCStringEncoding**, – **getCharacters:**

getCString:maxLength:

– (void)**getCString:**(char *)*buffer* **maxLength:**(unsigned int)*maxLength*

Invokes **getCString:maxLength:range:remainingRange:** with *maxLength* as the maximum length in **char**-sized units, the receiver's entire extent as the range, and NULL for the remaining range. *buffer* must be large enough to contain *maxLength* **char**s plus a terminating zero **char** (which this method adds).

Raises an NSCharacterConversionException if the receiver can't be represented in the default C string encoding without loss of information. Use **canBeConvertedToEncoding:** if necessary to check whether a string can be losslessly converted to the default C string encoding. If it can't, use **lossyCString** or **dataUsingEncoding:allowLossyConversion:** to get a C string representation with some loss of information.

See also: – **cStringLength**, – **canBeConvertedToEncoding:**, + **defaultCStringEncoding**, – **getCharacters:**

getCString:maxLength:range:remainingRange:

– (void)**getCString:**(char *)*buffer*
maxLength:(unsigned int)*maxLength*
range:(NSRange)*aRange*
remainingRange:(NSRange *)*leftoverRange*

Converts the receiver's characters to the default C string encoding and stores them in *buffer*. *buffer* must be large enough to contain *maxLength* **char**s plus a terminating zero **char** (which this method adds). Copies and converts as many character as possible from *aRange*, and stores the range of those not converted in the NSRange given by *leftoverRange* (if it's non-NULL). Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

Raises an NSCharacterConversionException if the receiver can't be represented in the default C string encoding without loss of information. Use **canBeConvertedToEncoding:** if necessary to check whether a string can be losslessly converted to the default C string encoding. If it can't, use **lossyCString** or **dataUsingEncoding:allowLossyConversion:** to get a C string representation with some loss of information.

See also: – **cStringLength**, – **canBeConvertedToEncoding:**, + **defaultCStringEncoding**, – **getCharacters:**

getFileSystemRepresentation:maxLength:

– (BOOL)**getFileSystemRepresentation:**(char *)*buffer* **maxLength:**(unsigned int)*maxLength*

Interprets the receiver as a system-independent path, filling *buffer* with a C string in a format and encoding suitable for use with file system calls. This is done by replacing the abstract path and extension separator characters (‘/’ and ‘.’ respectively) with their equivalents for the operating system. For example, on Microsoft Windows 95 the receiver “C:/Working/Sample.tiff” is returned as the C string “C:\Working\Sample.tiff”.

buffer must be large enough to contain *maxLength* **char**s plus a terminating zero **char** (which this method adds). Returns YES if *buffer* is successfully filled with a file system representation, NO if not (for example, if *maxLength* would be exceeded). Also returns NO if the receiver can’t be represented in the file system’s encoding.

If the system-specific path or extension separator appear in the abstract representation, the characters they’re converted to depend on the system (unless they’re identical to the abstract separators). On Windows 95, for example, a ‘\’ character is converted to ‘/’.

See also: – **fileSystemRepresentation**

getLineStart:end:contentsEnd:forRange:

+ (void)**getLineStart:**(unsigned int *)*startIndex*
end:(unsigned int *)*lineEndIndex*
contentsEnd:(unsigned int *)*contentsEndIndex*
forRange:(NSRange)*aRange*

Returns by reference the indexes of the smallest range of lines containing *aRange*. A line is delimited by any of these characters, the longest possible sequence being preferred to any shorter:

U+000D (**\r** or CR)	U+2028 (Unicode line separator)
U+000A (**\n** or LF)	U+2029 (Unicode paragraph separator)

\r\n, in that order (also known as CRLF)

When this method returns, *startIndex* contains the index of the first character of the line, which is at or before the location of *aRange; lineEndIndex* contains the index of the first character past the line terminator; and *contentsEndIndex* contains the index of the first character of the line terminator itself. You may pass a NULL pointer for any of these arguments, in which case the work to compute the value isn’t performed.

You can use the results of this method to construct ranges for lines by using the start index as the range’s location and the difference between the end index and the start index as the range’s length.

See also: – **lineRangeForRange:**, – **substringFromRange:**

hash

– (unsigned int)**hash**

Returns an unsigned integer that can be used as a hash table address. If two string objects are equal (as determined by the **isEqualToString:** method), they must have the same hash value. The abstract implementation of this method fulfills this requirement, so subclasses of NSString shouldn't override it.

hasPrefix:

– (BOOL)**hasPrefix:**(NSString *)*aString*

Returns YES if *aString* matches the beginning characters of the receiver, NO otherwise. Returns NO if *aString* is the null string. This method is a convenience for comparing strings using the NSAnchoredSearch option. See ""Working with String Objects"" in the class cluster description for more information.

See also: – **hasSuffix:**, – **compare:options:range:**

hasSuffix:

– (BOOL)**hasSuffix:**(NSString *)*aString*

Returns YES if *aString* matches the ending characters of the receiver, NO otherwise. Returns NO if *aString* is the null string. This method is a convenience for comparing strings using the NSAnchoredSearch and NSBackwardsSearch options. See ""Working with String Objects"" in the class cluster description for more information.

See also: – **hasPrefix:**, – **compare:options:range:**

init

– (id)**init**

Initializes the receiver, a newly allocated NSString, to contain no characters. Returns **self**.

See also: + **string**

initWithCharacters:length:

– (id)**initWithCharacters:**(const unichar *)*characters* **length:**(unsigned int)*length*

Initializes the receiver, a newly allocated NSString, by copying *length* characters from *characters*, which may not be NULL. This method doesn't stop short at a zero character. Returns **self**.

See also: – **stringWithCharacters:length:**

initWithCharactersNoCopy:length:freeWhenDone:

– (id)**initWithCharactersNoCopy:**(unichar *)*characters*
length:(unsigned int)*length*
freeWhenDone:(BOOL)*flag*

Initializes the receiver, a newly allocated NSString, to contain *length* characters from *characters*, which may not be NULL. This method doesn't stop short at a zero character. The receiver becomes the owner of *characters;* if *flag* is YES the receiver will free the memory when it no longer needs them, but if *flag* is NO it won't. Returns **self**.

See also: + **stringWithCharacters:length:**

initWithContentsOfFile:

– (id)**initWithContentsOfFile:**(NSString *)*path*

Initializes the receiver, a newly allocated NSString, by reading characters from the file named by *path*. If the contents begin with a byte-order mark (U+FEFF or U+FFFE), interprets the contents as Unicode characters; otherwise interprets the contents as characters in the default C string encoding. Returns **self**, or **nil** if the file can't be opened.

See also: + **stringWithContentsOfFile:**, + **defaultCStringEncoding**

initWithCString:

– (id)**initWithCString:**(const char *)*cString*

Initializes the receiver, a newly allocated NSString, by converting the characters in *cString* from the default C string encoding into the Unicode character encoding. *cString* must be a zero-terminated C string in the default C string encoding, and may not be NULL. Returns **self**.

Note: To create an immutable string from an immutable CString buffer, do not attempt to use this method. Instead, use **initWithCStringNoCopy**.

See also: – **stringWithCString:**, **-initWithCStringNoCopy**,+ **defaultCStringEncoding**

initWithCString:length:

– (id)**initWithCString:**(const char *)*cString* **length:**(unsigned int)*length*

Initializes the receiver, a newly allocated NSString, by converting the characters of *cString* from the default C string encoding into the Unicode character encoding. This method converts *length* * **sizeof(char)** bytes

from *cString*, and doesn't stop short at a zero character. *cString* must contain bytes in the default C string encoding, and may not be NULL. Returns **self**.

See also: + **stringWithCString:length:**, + **defaultCStringEncoding**

initWithCStringNoCopy:length:freeWhenDone:

– (id)**initWithCStringNoCopy:**(char *)*cString*
length:(unsigned int)*length*
freeWhenDone:(BOOL)*flag*

Initializes the receiver, a newly allocated NSString, by converting the characters of *cString* from the default C string encoding into the Unicode character encoding. This method converts *length* * **sizeof(char)** bytes from *cString*, and doesn't stop short at a zero character. *cString* must be contain characters in the default C string encoding, and may not be NULL. The receiver becomes the owner of *cString;* if *flag* is YES it will free the memory when it no longer needs it, but if *flag* is NO it won't. Returns **self**.

Note: You can use this method to create an immutable string from an immutable (**const char***) C-string buffer. If you receive a warning message, you can disregard it; its purpose is simply to warn you that the C string passed as the method's first argument may be modified. If you make certain that the **freeWhenDone** argument to **initWithStringNoCopy** is NO, the C string passed as the method's first argument cannot be modified, so you can safely use **initWithStringNoCopy** to create an immutable string from an immutable (**const char***) C-string buffer.

See also: + **stringWithCString:length:**, + **defaultCStringEncoding**

initWithData:encoding:

– (id)**initWithData:**(NSData *)*data* **encoding:**(NSStringEncoding)*encoding*

Initializes the receiver, a newly allocated NSString, by converting the bytes in *data* into Unicode characters. *data* must be an NSData object containing bytes in *encoding* and in the default plain text format (that is, pure content with no attribute or other markup) for that encoding. Returns **self**.

initWithFormat:

– (id)**initWithFormat:**(NSString *)*format, ...*

Invokes **initWithFormat:locale:arguments:** with **nil** as the locale.

See also: + **stringWithFormat:**

initWithFormat:arguments:

– (id)**initWithFormat:**(NSString *)*format* **arguments:**(va_list)*argList*

Invokes **initWithFormat:locale:arguments:** with **nil** as the locale.

See also: + **stringWithFormat:**

initWithFormat:locale:

– (id)**initWithFormat:**(NSString *)*format*
locale:(NSDictionary *)*dictionary, ...*

Invokes **initWithFormat:locale:arguments:** with *dictionary* as the locale.

See also: + **localizedStringWithFormat:**

initWithFormat:locale:arguments:

– (id)**initWithFormat:**(NSString *)*format*
locale:(NSDictionary *)*dictionary*
arguments:(va_list)*argList*

Initializes a newly allocated string object, using *format* as a template into which the following argument values are substituted according to the formatting information of the current locale. For example, this code excerpt creates a string from **myArgs**, which is derived from a string object with the value “Cost:” and an **int** with the value 32:

```
va_list myArgs;
NSDictionary *myLocale;    /* Assume this exists. */

NSString *myString = [[NSString alloc] initWithFormat:@"%@: %d\n",
    locale:[[NSUserDefaults standardUserDefaults] dictionaryRepresentation]
    arguments:myArgs];
```

(Note the message construct for retrieving the user’s locale.). The resulting string has the value “Cost: 32\n”.

See ““Creating and Converting String Objects”” in the class cluster description for more information. Returns **self**.

See also: – **initWithFormat:arguments:**

initWithString:

– (id)**initWithString:**(NSString *)*aString*

Initializes the receiver, a newly allocated NSString, by copying the characters from *aString*. Returns **self**.

See also: + **stringWithString:**

intValue

– (int)**intValue**

Returns the integer value of the string's text, assuming a decimal representation and skipping whitespace at the beginning of the string. Returns INT_MAX or INT_MIN on overflow. Returns 0 if the receiver doesn't begin with a valid decimal text representation of a number.

This method uses formatting information stored in the default locale; use an NSScanner for localized scanning of numeric values from a string.

See also: – **doubleValue**, – **floatValue**, – **scanInt:** (NSScanner)

isAbsolutePath

– (BOOL)**isAbsolutePath**

Interprets the receiver as a path, returning YES if it represents an absolute path, NO if it represents a relative path. See ""Manipulating Paths"" in the class description for more information on paths.

isEqualToString:

– (BOOL)**isEqualToString:**(NSString *)*aString*

Returns YES if *aString* is equivalent to the receiver (if they have the same **id** or if they compare as NSOrderedSame), NO otherwise. When you know both objects are strings, this method is a faster way to check equality than **isEqual:**.

See also: – **compare:options:range:**

lastPathComponent

– (NSString *)**lastPathComponent**

Returns the last path component of the receiver. The following table illustrates the effect of **lastPathComponent** on a variety of different paths:

Receiver's String Value	String Returned
"/tmp/scratch.tiff"	"scratch.tiff"
"/tmp/scratch"	"scratch"
"/tmp/"	"tmp"
"scratch"	"scratch"
"/"	"" *(an empty string)*

length

– (unsigned int)**length**

Returns the number of Unicode characters in the receiver. This includes the individual characters of composed character sequences, so you can't use this method to determine if a string will be visible when printed, or how long it will appear.

See also: – **cStringLength**, – **sizeWithAttributes:** (NSString Additions in the Application Kit)

lineRangeForRange:

+ (NSRange)**lineRangeForRange:**(NSRange)*aRange*

Returns the smallest range of lines containing *aRange*, including the characters that terminate the line.

See also: – **getLineStart:end:contentsEnd:forRange:**, – **substringFromRange:**

lossyCString

– (const char *)**lossyCString**

Returns a representation of the receiver as a C string in the default C string encoding, possibly losing information in converting to that encoding (and not raising an exception as **cString** does). The returned C string will be automatically freed just as a returned object would be released; your code should copy the C string or use **getCString:** if it needs to store the C string outside of the autorelease context in which the C string is created.

See also: – **getCString:**, – **canBeConvertedToEncoding**, + **defaultCStringEncoding**, – **cStringLength**, – **getCharacters:**

lowercaseString

– (NSString *)**lowercaseString**

Returns a string with each character from the receiver changed to its corresponding lowercase value. Case transformations aren't guaranteed to be symmetrical or to produce strings of the same lengths as the originals. The result of this statement:

```
lcString = [myString lowercaseString];
```

might not be equal to this:

```
lcString = [[myString uppercaseString] lowercaseString];
```

For example, the uppercase form of "ß" in German is "SS", so converting "eßen" to uppercase then lowercase produces this sequence of strings:

"eßen"
"ESSEN"
"essen"

See also: – **capitalizedString**, – **uppercaseString**

pathComponents

– (NSArray *)**pathComponents**

Interprets the receiver as a path, returning an array of strings containing, in order, each path component of the receiver. The strings in the array appear in the order they did in the receiver. If the string begins or ends with the path separator then the first or last component, respectively, is empty. Empty components (caused by consecutive path separators) are deleted. For example, this code excerpt:

```
NSString *path = @"tmp/scratch";
NSArray *pathComponents = [path componentsSeparatedByString:@"/"];
```

produces an array with these contents:

Index	Path Component
0	tmp
1	scratch

If the receiver begins with a slash—for example, "/tmp/scratch"—the array has these contents:

Index	Path Component
0	"/"
1	"tmp"
2	"scratch"

If the receiver has no separators—for example, “scratch”—the array contains the string itself, in this case “scratch”.

See also: + **pathWithComponents:**, – **stringByStandardizingPath:**, – **componentsSeparatedByString:**

pathExtension

– (NSString *)**pathExtension**

Interprets the receiver as a path, returning the receiver’s extension, if any (not including the extension divider). The following table illustrates the effect of **pathExtension** on a variety of different paths:

Receiver’s String Value	String Returned
“/tmp/scratch.tiff”	“tiff”
“/tmp/scratch”	“” *(an empty string)*
“/tmp/”	“” *(an empty string)*
“/tmp/scratch..tiff”	“tiff”

propertyList

– (id)**propertyList**

Parses the receiver as a text representation of a property list, returning an NSString, NSData, NSArray, or NSDictionary object according to the topmost element. Arrays are delimited by parentheses, with individual elements separated by commas and optional spaces. Dictionaries are delimited by curly braces, with key-value pairs separated by semicolons, the key and value separated by an equals sign. Strings appear as plain text if they contain no whitespace, or enclosed in straight quotation marks if they do. Data items are delimited by angle brackets and encoded as hexadecimal digits. Here’s a short example of a text-format property list:

```
{
    Title = "Star Wars";
    Director = "Lucas, George";
    Cast = (
        "Hamill, Mark",
        "Fisher, Carrie",
        "Ford, Harrison"
    );
    "Thumbnail Image" = <040b7479 70656473 (many more sets of digits) 8484074e>
}
```

See also: – **propertyListFromStringsFileFormat**, + **stringWithContentsOfFile:**

propertyListFromStringsFileFormat

– (NSDictionary *)**propertyListFromStringsFileFormat**

Returns a dictionary object initialized with the keys and values found in the receiver. The receiver must contain text in the format used for **.strings** files. In this format, keys and values are separated by an equals sign, and each key-value pair is terminated with a semicolon. The value is optional, however; if not present, the equals sign is also omitted. The keys and values themselves are always strings enclosed in straight quotation marks. Comments may be included, delimited by **/*** and ***/** as for ANSI C comments. Here's a short example of a strings file:

```
/* Question in confirmation panel for quitting. */
"Confirm Quit" = "Are you sure you want to quit?";

/* Message when user tries to close unsaved document */
"Close or Save" = "Save changes before closing?";

/* Word for Cancel */
"Cancel";
```

See also: – **propertyList**, + **stringWithContentsOfFile:**

rangeOfCharacterFromSet:

– (NSRange)**rangeOfCharacterFromSet:**(NSCharacterSet *)*aSet*

Invokes **rangeOfCharacterFromSet:options:** with no options.

rangeOfCharacterFromSet:options:

– (NSRange)**rangeOfCharacterFromSet:**(NSCharacterSet *)*aSet*
options:(unsigned int)*mask*

Invokes **rangeOfCharacterFromSet:options:range:** with *mask* for the options and the entire extent of the receiver for the range.

rangeOfCharacterFromSet:options:range:

– (NSRange)**rangeOfCharacterFromSet:**(NSCharacterSet *)*aSet*
options:(unsigned int)*mask*
range:(NSRange)*aRange*

Returns the range in the receiver of the first character found from *aSet*. The search is restricted to characters in the receiver within *aRange*. The following options may be specified in *mask* by combining them with the C bitwise OR operator:

NSCaseInsensitiveSearch
NSLiteralSearch
NSBackwardsSearch

See ““Working with String Objects”” in the class cluster description for details on these options. Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

Since precomposed characters in *aSet* can match composed characters sequences in the receiver, the length of the returned range can be greater than one. For example, if you search for “ü” in the string “stru¨del”, the returned range is {3,2}.

rangeOfComposedCharacterSequenceAtIndex:

– (NSRange)**rangeOfComposedCharacterSequenceAtIndex:**(unsigned int)*anIndex*

Returns the range in the receiver of the composed character sequence located at *anIndex*. The composed character sequence includes the first base character found at or before *anIndex*, and its length includes the base character and all non-base characters following the base character.

Raises an NSRangeException if *anIndex* lies beyond the end of the string.

If you want to write a method to adjust an arbitrary range so that it includes the composed character sequences on its boundaries, you can create a method such as this:

```
- (NSRange)adjustRange:(NSRange)aRange
{
    unsigned int index, endIndex;
    NSRange newRange, endRange;

    index = aRange.location;
    newRange = [self rangeOfComposedCharacterSequenceAtIndex:index];

    index = aRange.location + aRange.length;
    endRange = [self rangeOfComposedCharacterSequenceAtIndex:index];
    endIndex = endRange.location + endRange.length;

    newRange.length = endIndex - newRange.location;

    return newRange;
}
```

adjustRange: begins by correcting the location for the beginning of *aRange*, storing it in **newRange**. It then works at the end of *aRange*, correcting the location and storing it in **endIndex**. Finally, it sets the length of **newRange** to the difference between **endIndex** and the new range’s location.

rangeOfString:

– (NSRange)**rangeOfString:**(NSString *)*aString*

Invokes **rangeOfString:options:** with no options.

rangeOfString:options:

– (NSRange)**rangeOfString:**(NSString *)*aString* **options:**(unsigned int)*mask*

Invokes **rangeOfString:options:range:** with the options specified by *mask* and the entire extent of the receiver as the range.

rangeOfString:options:range:

– (NSRange)**rangeOfString:**(NSString *)*subString*
options:(unsigned int)*mask*
range:(NSRange)*aRange*

Returns an NSRange giving the location and length of the first occurrence of *subString* within *aRange* in the receiver. If *subString* isn't found, the length of the returned NSRange is zero. The length of the returned range and that of *subString* may differ if equivalent composed character sequences are matched. The following options may be specified in *mask* by combining them with the C bitwise OR operator:

NSCaseInsensitiveSearch
NSLiteralSearch
NSBackwardsSearch
NSAnchoredSearch

See ""Working with String Objects"" in the class cluster description for details on these options. Raises an NSRangeException if any part of *aRange* lies beyond the end of the string. Returns a range of {NSNotFound, 0} if *subString* is the null string.

smallestEncoding

– (NSStringEncoding)**smallestEncoding**

Returns the smallest encoding to which the receiver can be converted without loss of information. This encoding may not be the fastest for accessing characters, but is very space-efficient. This method itself may take some time to execute.

See also: – **fastestEncoding**, – **getCharacters:range:**

stringByAbbreviatingWithTildeInPath

– (NSString *)**stringByAbbreviatingWithTildeInPath**

Returns a string representing the receiver as a path, with a tilde, "~", substituted for the full path to the current user's home directory, or "*~user*" for a user other than the current user. Returns the receiver unaltered if it doesn't begin with the user's home directory.

See also: – **stringByExpandingTildeInPath**

stringByAppendingFormat:

– (NSString *)**stringByAppendingFormat:**(NSString *)*format*, ...

Returns a string made by appending to the receiver a string constructed from *format* and following arguments in the manner of **stringWithFormat:**.

See also: – **stringByAppendingString:**

stringByAppendingPathComponent:

– (NSString *)**stringByAppendingPathComponent:**(NSString *)*aString*

Returns a string made by appending *aString*, preceded by if necessary by a path separator. The following table illustrates the effect of this method on a variety of different paths, assuming that *aString* is supplied as "scratch.tiff":

Receiver's String Value	Resulting String
"/tmp"	"/tmp/scratch.tiff"
"/tmp/"	"/tmp/scratch.tiff"
"/"	"/scratch.tiff"
"" *(an empty string)*	"scratch.tiff"

See also: – **stringsByAppendingPaths:**, – **stringByAppendingPathExtension:**, – **stringByDeletingLastPathComponent**

stringByAppendingPathExtension:

– (NSString *)**stringByAppendingPathExtension:**(NSString *)*string*

Returns a string made by appending to the receiver an extension separator followed by *aString*. The following table illustrates the effect of this method on a variety of different paths, assuming that *aString* is supplied as @"tiff":

Receiver's String Value	Resulting String
"/tmp/scratch.old"	"/tmp/scratch.old.tiff"
"/tmp/scratch."	"/tmp/scratch..tiff"
"/tmp/"	"/tmp/.tiff"
"scratch"	"scratch.tiff"

See also: – **stringByAppendingPathComponent:**, – **stringByDeletingPathExtension**

stringByAppendingString:

– (NSString *)s**tringByAppendingString:**(NSString *)*aString*

Returns a string object made by appending *aString* to the receiver. This code excerpt, for example:

```
NSString *errorTag = @"Error: ";
NSString *errorString = @"premature end of file.";
NSString *errorMessage = [errorTag
    stringByAppendingString:errorString];
```

produces the string "Error: premature end of file.".

See also: – **stringByAppendingFormat:**

stringByDeletingLastPathComponent

– (NSString *)**stringByDeletingLastPathComponent**

Returns a string made by deleting the last path component from the receiver, along with any final path separator. If the receiver represents the root path, however, it's returned unaltered. The following table illustrates the effect of this method on a variety of different paths:

Receiver's String Value	Resulting String
"/tmp/scratch.tiff"	"/tmp"
"/tmp/lock/"	"/tmp"
"/tmp/"	"/"
"/tmp"	"/"
"/"	"/"
"scratch.tiff"	"" *(an empty string)*

See also: – **stringByDeletingPathExtension**, – **stringByAppendingPathComponent:**

stringByDeletingPathExtension

– (NSString *)**stringByDeletingPathExtension**

Returns a string made by deleting the extension (if any, and only the last) from the receiver. Strips any trailing path separator before checking for an extension. If the receiver represents the root path, however, it's returned unaltered. The following table illustrates the effect of this method on a variety of different paths:

Receiver's String Value	Resulting String
"/tmp/scratch.tiff"	"/tmp/scratch"
"/tmp/"	"/tmp"
"scratch.bundle/"	"scratch"
"scratch..tiff"	"scratch."
".tiff"	"" *(an empty string)*
"/"	"/"

See also: – **pathExtension**, – **stringByDeletingLastPathComponent**

stringByExpandingTildeInPath

– (NSString *)**stringByExpandingTildeInPath**

Returns a string made by expanding the initial component, if it begins with "~" or "~*user*", to its full path value. Returns the receiver unaltered if that component can't be expanded.

See also: – **stringByAbbreviatingWithTildeInPath**

stringByResolvingSymlinksInPath

– (NSString *)stringByResolvingSymlinksInPath

On Microsoft Windows: Returns **self**.

On UNIX platforms: Expands an initial tilde expression in the receiving path, then resolves all symbolic links and references to current or parent directories if possible, returning a standardized path. If the original path is absolute, all symbolic links are guaranteed to be removed; if it's a relative path, symbolic links that can't be resolved are left unresolved in the returned string. Returns **self** if an error occurs.

Note: If the name of the receiving path begins with **/private**, the **stringByResolvingSymlinksInPath** method strips off the **/private** designator, provided the result is the name of an existing file.

See also: – **stringByStandardizingPath**, – **stringByExpandingTildeInPath**

stringByStandardizingPath

– **(NSString *)stringByStandardizingPath**

Returns a string representing the receiving path, with extraneous path components removed. If stringByStandardizingPath detects symbolic links in a path name, the stringByResolvingSymlinksInPath method is called to resolve them. If an invalid path name is provided, stringByStandardizingPath may attempt to resolve it by calling **stringByResolvingSymlinksInPath, and the results are undefined. If any other kind error is encountered (such as a path component not existing), self is returned.**

The changes that this method can make in the provided string are:

- An initial tilde expression is expanded using **stringByExpandingTildeInPath**.
- Empty components and references to the current directory (that is, the sequences "//" and "/./") are reduced to single path separators.
- In absolute paths only, references to the parent directory (that is, the component "..") are resolved to the real parent directory if possible using **stringByResolvingSymlinksInPath**, which consults the file system to resolve each potential symbolic link.
- In relative paths, because symbolic links can't be resolved, references to the parent directory are left in place.
- On Mach, an initial component of "/private" is removed from the path if the result still indicates an existing file or directory (checked by consulting the file system).

See also: – **stringByExpandingTildeInPath**, – **stringByResolvingSymlinksInPath**

stringsByAppendingPaths:

– (NSArray *)**stringsByAppendingPaths:**(NSArray *)*paths*

Returns an array of strings made by separately appending each string in *paths* to the receiver, preceded by if necessary by a path separator. See **stringByAppendingPathComponent:** for an individual example.

substringFromIndex:

– (NSString *)**substringFromIndex:**(unsigned int)*anIndex*

Returns a string object containing the characters of the receiver from the one at *anIndex* to the end. Raises an NSRangeException if *anIndex* lies beyond the end of the string.

See also: – **substringWithRange:**, – **substringToIndex:**

substringWithRange:

– (NSString *)**substringWithRange:**(NSRange)*aRange*

Returns a string object containing the characters of the receiver that lie within *aRange*. Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

See also: – **substringFromIndex:**, – **substringToIndex:**

substringToIndex:

– (NSString *)**substringToIndex:**(unsigned int)*anIndex*

Returns a string object containing the characters of the receiver up to, but not including, the one at *anIndex*. Raises an NSRangeException if *anIndex* lies beyond the end of the string.

See also: – **substringFromIndex:**, – **substringWithRange:**

uppercaseString

– (NSString *)**uppercaseString**

Returns a string with each character from the receiver changed to its corresponding uppercase value. Case transformations aren't guaranteed to be symmetrical or to produce strings of the same lengths as the originals. See **lowercaseString** for an example.

See also: – **capitalizedString**, – **lowercaseString**

writeToFile:atomically:

– (BOOL)**writeToFile:**(NSString *)*path* **atomically:**(BOOL)*flag*

Writes the string's characters to the file named by *path*, returning YES on success and NO on failure. If *flag* is YES, attempts to write the file safely so that an existing file named by *path* is not overwritten, nor does a new file at *path* actually get created, unless the write is successful. The string is written in the default C string encoding if possible (that is, if no information would be lost), in the Unicode encoding otherwise.

If *path* contains a tilde (~) character, you must expand it with **stringByExpandingTildeInPath:** before invoking this method.

See also: + **defaultCStringEncoding**

NSMutableString

Inherits From: NSString : NSObject

Conforms To: NSCoding (NSString)
NSCopying (NSString)
NSMutableCopying (NSString)
NSObject (NSObject)

Declared In: Foundation/NSString.h

Class Description

The NSMutableString class declares the programmatic interface to an object that manages a mutable string—that is, a string whose contents can be edited. To construct and manage an immutable string—or a string that cannot be changed after it has been created—use an object of the NSString class.

An immutable string is implemented as array of Unicode characters (in other words, as a text string). The NSMutableString class adds one primitive method—**replaceCharactersInRange:withString:**—to the basic string-handling behavior inherited from NSString. All other methods that modify a string work through this method. For example, **insertString:atIndex:** simply replaces the characters in a range of zero length, while **deleteCharactersInRange:** replaces the characters in a given range with no characters.

Method Types

Creating temporary strings	+ stringWithCapacity:
Initializing an NSMutableString	– initWithCapacity:
Modifying a string	**– appendFormat:** **– appendString:** **– deleteCharactersInRange:** **– insertString:atIndex:** **– replaceCharactersInRange:withString:** **– setString:**

Class Methods

stringWithCapacity:

+ (NSMutableString *)**stringWithCapacity:**(unsigned int)*capacity*

Returns an empty mutable string, using *capacity* as a hint for how much initial storage to reserve.

Instance Methods

appendFormat:

– (void)**appendFormat:**(NSString *)*format*, ...

Adds a constructed string to the receiver. Creates the new string by using NSString's **stringWithFormat:** method with the arguments listed.

See also: – **appendString:**

appendString:

– (void)**appendString:**(NSString *)*aString*

Adds the characters of *aString* to end of the receiver.

See also: – **appendFormat:**

deleteCharactersInRange:

– (void)**deleteCharactersInRange:**(NSRange)*aRange*

Removes the characters in *aRange* from the receiver. Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

initWithCapacity:

– (id)**initWithCapacity:**(unsigned int)*capacity*

Initializes a newly allocated NSMutableString, using *capacity* as a hint for how much memory to allocate. Returns **self**.

insertString:atIndex:

– (void)**insertString:**(NSString *)*aString* **atIndex:**(unsigned int)*anIndex*

Inserts the characters of *aString* into the receiver, so that the new characters begin at *anIndex* and the existing characters from *anIndex* to the end are shifted by the length of *aString*. Raises an NSRangeException if *anIndex* lies beyond the end of the string.

replaceCharactersInRange:withString:

– (void)**replaceCharactersInRange:**(NSRange)*aRange*
withString:(NSString *)*aString*

Replaces the characters from *aRange* with those in *aString*. Raises an NSRangeException if any part of *aRange* lies beyond the end of the string.

setString:

– (void)**setString:**(NSString *)*aString*

Replaces the characters of the receiver with those in *aString*.