NSString Class Cluster

Class Cluster Description

NSString objects represent character strings in the Foundation Kit, which uses them in place of the C language's **char *** data type. 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 static and dynamic strings, respectively.

The objects you create using these classes are referred to as *string objects* (and when no confusion will result, merely as *strings*). Because of the nature of class clusters, string objects are not 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 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. You can determine how many characters it contains with the **length** method and can retrieve a specific Unicode character with the **characterAtIndex:** method. These two 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: strings are compared against one another, search for substrings, combined into new strings, and so on. If you need to access string objects character-by-character, you must understand the Unicode encoding; see *The Unicode Standard: Worldwide Character Encoding* for details:

> *The Unicode Standard: Worldwide Character Encoding*, Version 1.0, Volume 1. The Unicode Consortium. Addison-Wesley, 1990, 1991. ISBN 0–201–56788–1

Using String Objects with C String API

Although string objects fit nicely in the Objective C realm, there remains a lot of API that expects C strings: ANSI C and other library functions, your own code, and even some object-oriented API. You can both create string objects from ANSI C strings and extract C string representations from string objects. The simplest way to create a string object in source code is to use either the **stringWithCString:** or the **initWithCString:** method. Each takes a standard null-terminated C string in the system's default encoding (a superset

of ASCII for most, EUC for Japanese systems) and produces a Unicode string object. The Objective C compiler also supports the **@"..."** construction to create a string object constant:

```
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 like you would any other object).

To get a C string from a string object, you use the **cString** message. This returns a **char *** in the system's default string 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, and if you need to know the encoding that will be used, use the **defaultCStringEncoding** method.

To convert a string object to or from a C string in a specific encoding, use the **initWithData:encoding:** and **dataUsingEncoding:...** methods. The string classes support the encodings listed under the **NSStringEncoding** data type in the Types and Constants section of this chapter.

Working with Composed Character Sequences

A *composed character sequence* is any sequence of Unicode characters that is to be considered a single logical unit—a "letter" in many writing systems. Composed character sequences are made of *base characters* and *non-spacing* (or diacritical) *characters*. For example, you could encode the letter "ü" with the base character "lowercase u" and the non-spacing character "umlaut." It's possible to have any number of non-spacing characters associated with a base character: You can have a lowercase "u" with an umlaut and a tilde above, and with an underscore below. Non-spacing characters that don't affect each other can appear in any order after the base character, and all such permutations must be considered equivalent for comparison. For example, the pre-composed character "lowercase u umlaut" is equivalent to the composed character sequence "lowercase u" + "umlaut"; likewise, the following three combinations are equivalent to each other:

"lowercase u umlaut" + "underscore"
"lowercase u" + "umlaut" + "underscore"
"lowercase u" + "underscore" + "umlaut"

Beyond considerations of equivalence for comparison of composed character sequences, note that it's an error to unintentionally break a composed character sequence, whether by only examining part of it (unless you mean specifically to examine its parts), by inserting

characters anywhere inside it, or by removing one of the characters that compose it. Whenever you perform some arbitrary computation of an index into a Unicode string, you must adjust that index to fall between composed character sequences for many operations, such as searching and comparing, to be meaningful. The **rangeOfComposedCharacterSequenceAtIndex:** method gives you the beginning index and length of the composed character sequence that lies across a given character index. For example, suppose you need to divide a string in half. You could use the following sequence to move the break back so that it contains a full composed character sequence:

```
unsigned int halfway;
NSRange midCharSequence;

halfway = [stringObj length] / 2;
midCharSequence = [stringObj
    rangeOfComposedCharacterSequenceAtIndex:halfway];
halfway = midCharSequence.location;
```

Working with String Objects

The string classes provide methods for finding characters and strings within strings and for comparing one string against another. These methods conform to the Unicode standard for determining what 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. The options you can specify are given in the table below (not all options are available for every method):

Search Option	Effect
NSCaseInsensitiveSearch	Case distinctions among characters, when they would normally be made, are ignored.
NSLiteralSearch	A byte-for-byte comparison is made. Differing sequences of Unicode characters that would otherwise be considered equivalent are considered not to match.
NSBackwardsSearch	Searching is performed from the end of the range toward the beginning.
NSAnchoredSearch	Searching is performed only on the range of 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 that the default search behavior is case-sensitive from the beginning of the string, with composed character sequences compared according to the Unicode standard. 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 Unicode 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 equality of strings or whether one is the prefix or suffix of another, but these 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 in the manner of **printf()** with the **initWithFormat:**, **stringWithFormat:** and **stringByAppendingFormat:** methods.

The string classes allow you to extract substrings from the beginning or end to a particular index, or from a specific range, with the **substringToIndex:**, **substringFromIndex:**, and **substringFromRange:** methods. You can also ask for an array object containing all substrings divided by a separator string with the **componentsSeparatedByString:** method.

Most of the NSString classes' remaining methods are for conveniences like changing case, quickly extracting numeric values, and working with encodings. 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 auxiliary NSCharacterSet class cluster. See the appropriate class specifications for more information.

Notes on Unicode Support

The NSString classes internally support Unicode, but the associated NSCharacterSet classes don't fully support Unicode. See the NSCharacterSet class cluster specification for details.

▶ NSString

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

Class Description

The NSString class declares the programmatic interface for an object that manages an immutable array of Unicode characters (in other words, a text string). NSString's two primitive methods—**length** and **characterAtIndex:**—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."

Instance Variables

None declared in this class.

Adopted Protocols

NSCoding	– encodeUsingCoder: – decodeUsingCoder:
NSCopying	– copyWithZone: – copy

NSMutableCopying	– mutableCopyWithZone: – mutableCopy

Method Types

Creating temporary strings	+ stringWithCharacters:length: + stringWithCString:length: + stringWithCString: + stringWithFormat:
Initializing newly allocated strings	– initWithCharactersNoCopy:length:freeWhenDone: – initWithCharacters:length: – initWithCStringNoCopy:length:freeWhenDone: – initWithCString:length: – initWithCString: – initWithString: – initWithFormat: – initWithFormat:arguments: – initWithData:encoding: – initWithContentsOfFile: – init
Getting a string's length	– length
Accessing characters	– characterAtIndex: – getCharacters: – getCharacters:range:
Combining strings	– stringByAppendingFormat: – stringByAppendingString:
Dividing strings into pieces	– componentsSeparatedByString: – substringFromIndex: – substringFromRange: – substringToIndex:
Finding characters and substrings	– rangeOfCharacterFromSet: – rangeOfCharacterFromSet:options: – rangeOfCharacterFromSet:options:range: – rangeOfString: – rangeOfString:options: – rangeOfString:options:range:

Determining composed character sequences	– rangeOfComposedCharacterSequenceAtIndex:
Converting String Contents into a Property List	– propertyList – propertyListFromStringsFileFormat
Identifying and Comparing strings	– compare: – compare:options: – compare:options:range: – hasPrefix: – hasSuffix: – isEqual: – isEqualToString: – hash
Getting a shared prefix	– commonPrefixWithString:options:
Changing case	– capitalizedString – lowercaseString – uppercaseString
Getting ANSI C strings	– cString – cStringLength – getCString: – getCString:maxLength: – getCString:maxLength:range:remainingRange:
Getting numeric values	– floatValue – intValue
Working with encodings	– canBeConvertedToEncoding: – dataUsingEncoding: – dataUsingEncoding:allowLossyConversion: + defaultCStringEncoding – description – fastestEncoding – smallestEncoding

Working with paths	– completePathIntoString:caseSensitive: matchesIntoArray:filterTypes: – lastPathComponent – pathExtension – stringByAbbreviatingWithTildeInPath – stringByAppendingPathComponent – stringByAppendingPathExtension: – stringByDeletingLastPathComponent – stringByDeletingPathExtension – stringByExpandingTildeInPath – stringByResolvingSymlinksInPath – stringByStandardizingPath

Class Methods

defaultCStringEncoding

+ (NSStringEncoding)**defaultCStringEncoding**

Returns the C string encoding assumed for any method accepting a C string as an argument (such methods use **CString** in the keywords for such arguments; for example, **stringWithCString:**). See the description of **NSStringEncoding** in the Types and Constants section for a full list of supported encodings.

stringWithCharacters:length:

+ (NSString *)**stringWithCharacters:**(const unichar *)*chars* **length:**(unsigned)*length*

Returns a string containing *chars*. *length* characters are copied into the string, regardless of whether a null character exists in *chars*.

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

stringWithCString:length:

+ (NSString *)**stringWithCString:**(const char *)*byteString* **length:**(unsigned)*length*

Returns a string containing characters from *byteString*, which need not be null-terminated. *byteString* should contain characters in the default C string encoding. *length* bytes are copied into the string, regardless of whether a null byte exists in *byteString*.

See also: – **initWithCString:length:**

stringWithCString:

+ (NSString *)**stringWithCString:**(const char *)*byteString*

Returns a string containing the characters in *byteString*, which must be null-terminated. *byteString* should contain characters in the default C string encoding.

See also: – **initWithCString:**

stringWithFormat:

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

Returns a string created by using *format* as a **printf()** style format string, and the following arguments as values to be substituted into the format string. For example, this code excerpt creates a string from another string and an **int**:

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

The result string has the value “Cost: 32\n”.

See also: – **initWithFormat:**

Instance 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.

Note that 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 of each word changed to its corresponding uppercase value, and all remaining characters set to their corresponding lowercase values. A word is any sequence of characters delimited by spaces, tabs, or newline characters.

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 capitalizedString] lowercaseString];
```

See **lowercaseString** for an example.

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

characterAtIndex:

– (unichar)**characterAtIndex:**(unsigned)*anIndex*

Returns the character at the array position given by *anIndex*. This method raises an **NSRangeException** exception if *anIndex* lies beyond the end of the string.

commonPrefixWithString:options:

– (NSString *)**commonPrefixWithString:**(NSString *)*aString*
options:(unsigned)*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 "Stru¨delhaus" and *aString* is "Strüdel", the returned string will be "Stru¨del", not "Strüdel". 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:**

compare:options:

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

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

compare:options:range:

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

Returns **NSOrderedAscending** if the substring given by *aRange* in the receiver precedes the corresponding substring of *aString* in lexical ordering, **NSOrderedSame** if the two substrings are equivalent in lexical value, and **NSOrderedDescending** if the receiver's substring follows *aString*'s. The following 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.

This method raises an **NSRangeException** exception if any part of *aRange* lies beyond the end of the string.

completePathIntoString:caseSensitive:matchesIntoArray:filterTypes:

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

Regards the receiver as containing a partial filename and returns in *outputName* the longest matching path name. Case is considered if *flag* is YES. If *outputArray* is given, all matching file names are returned in *outputArray*. If *filterTypes* is provided, this method considers only those paths that match one of the types. Returns 0 if no matches are found; otherwise, the return value is positive.

componentsSeparatedByString:

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

Constructs and returns an NSArray containing substrings from the receiver that have been divided by *separator*. The strings in the array appear in the order they did in the receiver. For example, this code excerpt:

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

produces an array containing these strings:

“tmp” at index 0
“gigo” at index 1

If **path** begins with a slash—for example, “/tmp/gigo”—the array contains these strings:

“” at index 0 (because there are no characters before the first slash)
“tmp” at index 1
“gigo” at index 2

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

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

cString

– (const char *)**cString**

Returns a representation of the receiver as a C string in the default C string encoding. The returned string will be automatically freed just as a returned object would be; your code

should copy the string or use **getCString:** if it needs to store it outside of the method in which the string is created.

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

cStringLength

– (unsigned)**cStringLength**

Returns the length in bytes of the C string representation of the receiver.

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

dataUsingEncoding:

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

Invokes **dataUsingEncoding:allowLossyConversion:** with NO as the argument to allow lossy conversion.

dataUsingEncoding:allowLossyConversion:

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

Returns an NSData object containing a representation of the receiver in *encoding*. If *flag* is NO and the receiver can't be converted without losing some information (such as accents or case) this method returns **nil**. 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" would become "A", losing the accent.

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

See also: – **canBeConvertedToEncoding:**

description

– (NSString *)**description**

Returns a quoted version of the string. The string may always be quoted; it will certainly be quoted if it contains backslashes or quotes. If quoted, then internal quotes and backslashes are also backslashed.

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:**

floatValue

– (float)**floatValue**

Returns the floating-point value of the receiver’s text. Whitespace at the beginning of the string is skipped. If the receiver begins with a valid text representation of a floating-point number, that number’s value is returned, otherwise 0.0 is returned. HUGE_VAL or –HUGE_VAL is returned on overflow. 0.0 is returned on underflow.

See also: – **intValue**

getCharacters:

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

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

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. This method does *not* add a null character. This method raises an **NSRangeException** exception if any part of *aRange* lies beyond the end of the string.

If you subclass NSString, this method works properly using **characterAtIndex:**. Your subclass should override this method 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 null character (which this method adds).

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

getCString:maxLength:

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

Invokes **getCString:maxLength:range:remainingRange:** with *maxLength* 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 *maxLength* bytes plus a terminating null character (which this method adds).

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

getCString:maxLength:range:remainingRange:

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

Copies up to *maxLength* of the receiver's characters as bytes in the default C string encoding into *buffer*. *buffer* must be large enough to contain *maxLength* bytes plus a terminating null character (which this method adds). Characters are copied from *aRange*; if not all characters can be copied, the range of those not copied is put into *leftoverRange*.

This method raises an **NSRangeException** exception if any part of *aRange* lies beyond the end of the string.

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

hash

– (unsigned)**hash**

Returns an unsigned integer that can be used as a table address in a hash table structure. If two string objects are equal (as determined by the **isEqual:** method), they must have the same hash value. Because of this, your subclass of NSString shouldn't override **hash**.

hasPrefix:

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

Returns YES if *aString* matches the beginning characters of the receiver, NO otherwise. The default search options are used (see "Working with String Objects" in the class cluster description for further information).

See also: – **hasSuffix:**

hasSuffix:

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

Returns YES if *aString* matches the ending characters of the receiver, NO otherwise. The default search options are used (see "Working with String Objects" in the class cluster description for further information).

See also: – **hasPrefix:**

init

– **init**

Initializes the receiver, a newly allocated NSString, to contain no characters. This is the only initialization method that a subclass of NSString should override.

initWithCharacters:length:

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

Initializes the receiver, a newly allocated NSString, by copying *length* characters from *characters*. This method doesn't stop at a null character.

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

initWithCharactersNoCopy:length:freeWhenDone:

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

Initializes the receiver, a newly allocated NSString, to contain *length* characters from *characters*. This method doesn't stop at a null 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.

initWithContentsOfFile:

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

Initializes the receiver, a newly allocated NSString, by reading NEXTSTEP-encoded characters from the file whose name is given by *path*.

initWithCString:

– **initWithCString:**(const char *)*aString*

Initializes the receiver, a newly allocated NSString, by converting the one-byte characters in *aString* into Unicode characters. *aString* must be a null-terminated C string in the default C string encoding.

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

initWithCString:length:

– **initWithCString:**(const char *)*aString* **length:**(unsigned)*length*

Initializes the receiver, a newly allocated NSString, by converting *length* one-byte characters in *aString* into Unicode characters. This method doesn't stop at a null byte. *aString* must be contain bytes in the default C string encoding.

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

initWithCStringNoCopy:length:freeWhenDone:

– **initWithCStringNoCopy:**(char *)*aString*
length:(unsigned)*length*
freeWhenDone:(BOOL)*flag*

Initializes the receiver, a newly allocated NSString, by converting *length* one-byte characters in *aString* into Unicode characters. This method doesn't stop at a null byte. *aString* must be contain bytes in the default C string encoding. The receiver becomes the owner of *aString;* if *flag* is YES it will free the memory when it no longer needs it, but if *flag* is NO it won't.

See also: + **defaultCStringEncoding**

initWithData:encoding:

– **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 for that encoding.

initWithFormat:

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

Initializes the receiver, a newly allocated NSString, by constructing a string from *format* and following string objects in the manner of **printf()**.

See also: – **stringWithFormat:**

initWithFormat:arguments:

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

Initializes the receiver, a newly allocated NSString, by constructing a string from *format* and *argList* in the manner of **vprintf()**.

initWithString:

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

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

intValue

– (int)**intValue**

Returns the integer value of the receiver's text. Whitespace at the beginning of the string is skipped. If the receiver begins with a valid representation of an integer, that number's value is returned, otherwise 0 is returned. INT_MAX or INT_MIN is returned on overflow.

See also: – **intValue**

isEqual:

– (BOOL)**isEqual:***anObject*

Returns YES if both the receiver and *anObject* have the same **id** or if they're both NSStrings that compare as **NSOrderedSame**, NO otherwise.

See also: – **compare:**

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:**

lastPathComponent

– (NSString *)**lastPathComponent**

Returns the last component in the receiving path. If the receiving path consists solely of a slash, **lastPathComponent** returns the empty string. The following table illustrates the effect of **lastPathComponent** on a variety of different paths:

Receiving Path	Resulting String
/Foo/Bar.tiff	Bar.tiff
/Foo/Bar	Bar
/Foo/Bar/	Bar
Foo	Foo
/	“”

length

– (unsigned int)**length**

Returns the number of 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**

lowercaseString

– (NSString *)**lowercaseString**

Returns a string with each character 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” is “SS”, so converting “eßen” to uppercase then lowercase would produce this sequence of strings:

eßen
ESSEN
essen

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

pathExtension

– (NSString *)**pathExtension**

Returns a string consisting only of the receiving path's extension. If the receiving path does not have an extension ("/Foo/Bar" or "/Foo/Bar", for example), **pathExtension** returns the empty string. The following table illustrates the effect of **pathExtension** on a variety of different paths:

Receiving Path	Resulting String
/Foo/Bar.tiff	tiff
/Foo/Bar	""
/Foo/Bar/	""

propertyList

– (id)**propertyList**

Depending on the format of the receiver's contents, returns a string, data, array, or dictionary object represention of those contents.

propertyListFromStringsFileFormat

– (NSDictionary *)**propertyListFromStringsFileFormat**

Returns a dictionary object initialized with the keys and values found in the receiver. The receiver's format must be that used for "**.string**" files.

rangeOfCharacterFromSet:

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

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

Finds the first occurrence of a character from the specified set and returns its range. Note that the range covers only the first found character, not a sequence of characters. If not found, returned length is 0.

rangeOfCharacterFromSet:options:

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

Finds the first occurrence of a character from the specified set and returns its range. Note that the range covers only the first found character, not a sequence of characters. If not found, returned length is 0. Possible options are: NSLiteralSearch, NSBackwardsSearch, NSAnchoredSearch.

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 those 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. This method raises an **NSRangeException** exception 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, it’s possible that the length of the returned range be greater than one. For example, if you search for “ü” in the string “stru¨del”, the returned range will be {3,2}.

rangeOfComposedCharacterSequenceAtIndex:

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

Returns an NSRange giving the location and length in the receiver of the composed character sequence located at *anIndex*. The composed character sequence includes the first base character found at or past *anIndex*, and its length includes the base character and all zero-width or non-base characters following the base character.

This method raises an **NSRangeException** exception 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 index, endIndex;
    NSRange newRange, endRange;

   /*
    * Calculate the beginning location for the range.
    */
    index = aRange.location;
    newRange = [self rangeOfComposedCharacterSequenceAtIndex:index];

   /*
    * Calculate the ending index for the range.
    */
    index = aRange.location + aRange.length;
    endRange = [self rangeOfComposedCharacterSequenceAtIndex:index];
    endIndex = endRange.location + endRange.length;

   /*
    * Set the length of the adjusted range.
    */
    newRange.length = endIndex - newRange.location;

    return newRange;
}
```

rangeOfString:

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

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

See also: – rangeOfString:options:, – rangeOfString:options:range:

rangeOfString:options:

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

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

See also: – rangeOfString:, – rangeOfString:options:range:

rangeOfString:options:range:

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

Returns an NSRange giving the location and length in the receiver of *aString*. If *aString* isn't found, the length of the returned NSRange is zero. The length of the returned range and that of *aString* may differ if equivalent composed character sequences are matched. The search is restricted to the substring of the receiver given by *aRange*. 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. This method raises an **NSRangeException** exception if any part of *aRange* lies beyond the end of the string.

See also: – rangeOfString:, – rangeOfString:options:

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.

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

stringByAbbreviatingWithTildeInPath

– (NSString *)**stringByAbbreviatingWithTildeInPath**

Returns the receiving path with a "~" character substituted for the first occurrence of the user's home directory. If the receiving path does not start with the user's home directory, the receiving path is returned unaltered.

See also: – **stringByExpandingTildeInPath**

stringByAppendingFormat:

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

Returns a string made by using *format* as a **printf()** style format string, and the following arguments as values to be substituted into the format string. This code excerpt, for example:

```
NSString *errorTag = @"Error: ";
NSString *errorLocation = @"Filename.m";
NSString *errorText = [errorTag
    stringByAppendingFormat:@"Undefined value in %@\n",
    errorLocation];
```

produces the string “Error: Undefined value in Filename.m\n”.

This method is equivalent to invoking **stringWithFormat:** with the arguments listed, and passing the resulting string to **stringByAppendingString:**.

stringByAppendingString:

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

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

```
NSString *errorTag = @"This is where ";
NSString *insertTabA = @"you insert tab A.";
NSString *errorText = [errorTag stringByAppendingString:insertTabA];
```

produces the string “This is where you insert tab A.”.

See also: – **stringByAppendingFormat:**

stringByAppendingPathComponent:

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

Returns the receiving path with the path component specified by *aString* appended. The following table illustrates the effect of **stringByAppendingPathComponent:** on a variety of different paths, assuming that *aString* is supplied as @“Bar.tiff”:

Receiving Path	Resulting String
/Foo	/Foo/Bar.tiff
/Foo/	/Foo/Bar.tiff
/	/Bar.tiff
“”	Bar.tiff

See also: – **stringByAppendingPathExtension:**

stringByAppendingPathExtension:

– (NSString *)**stringByAppendingPathExtension:**(NSString *)*aString*

Returns the receiving path with a period and the path extension specified by *aString* appended. The following table illustrates the effect of **stringByAppendingPathExtension:** on a variety of different paths, assuming that *aString* is supplied as @“tiff”:

Receiving Path	Resulting String
/Foo/Bar.x	/Foo/Bar.x.tiff
/Foo/	/Foo/.tiff
Foo	Foo.tiff

See also: – **stringByAppendingPathComponent:**

stringByDeletingLastPathComponent

– (NSString *)**stringByDeletingLastPathComponent**

Returns the receiving path with the last path component removed. The following table illustrates the effect of **stringByDeletingLastPathComponent** on a variety of different paths:

Receiving Path	Resulting String
/Foo/Bar.tiff	/Foo
/Foo/	/
/	/
Foo	“”

See also: – **stringByDeletingPathExtension**

stringByDeletingPathExtension

– (NSString *)**stringByDeletingPathExtension**

Returns the receiving path with the path extension removed. The following table illustrates the effect of **stringByDeletingPathExtension** on a variety of different paths:

Receiving Path	Resulting String
/Foo/Bar.tiff	/Foo/Bar
/Foo/	/Foo
/	/

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

stringByExpandingTildeInPath

– (NSString *)**stringByExpandingTildeInPath**

Returns the receiving path with the first "~" character (or "~*user*" expression) expanded. If the user does not exist, or the receiving path does not begin with a tilde, the receiving path is returned unaltered.

See also: – **stringByAbbreviatingWithTildeInPath**

stringByResolvingSymlinksInPath

- (NSString *)**stringByResolvingSymlinksInPath**

Resolves all symbolic links in the receiving path, returning a path with no symbolic links and cleaned up as described in **stringByStandardizingPath**. The empty string is returned if an error occurs.

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

stringByStandardizingPath

- (NSString *)**stringByStandardizingPath**

Expands "~" in the receiving path, removes "/private" (if possible), and condenses "//" and "/./". **stringByStandardizingPath** replaces any use of ".." in absolute paths by taking the real parent directory (if possible; otherwise, it just trims the previous component). **stringByStandardizingPath** does not stat the path; it just cleans up the syntax.

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

substringFromIndex:

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

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

See also: – **substringFromRange:**, – **substringToIndex:**

substringFromRange:

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

Returns a string object containing the characters of the receiver which lie within *aRange*. This method raises an **NSRangeException** exception if any part of *aRange* lies beyond the end of the string.

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

substringToIndex:

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

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

See also: – **substringFromIndex:**, – **substringFromRange:**

uppercaseString

– (NSString *)**uppercaseString**

Returns a string with each character 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. The result of this statement:

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

might not be equal to this:

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

See **lowercaseString** for an example.

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

▶ NSMutableString

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

Class Description

The NSMutableString class declares the programmatic interface to objects that manage a modifiable array of Unicode characters (text strings). This class adds a method for replacing characters—**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.

Instance Variables

None declared in this class.

Method Types

Creating temporary strings	+ stringWithCapacity: + stringWithCharacters:length: + stringWithCString: + stringWithCString:length: + stringWithFormat:
Initializing a Mutable String	– initWithCapacity:

Modifying a string	– appendFormat: – appendString: – deleteCharactersInRange: – insertString:atIndex: – replaceCharactersInRange:withString: – setString:

Class Methods

stringWithCapacity:

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

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

stringWithCharacters:length:

+ (NSMutableString *)**stringWithCharacters:**(const unichar *)*chars*
length:(unsigned)*length*

Returns a mutable string containing *chars*. The first *length* characters are copied into the string. This method doesn't stop at a null character.

stringWithCString:length:

+ (NSMutableString *)**stringWithCString:**(const char *)*byteString*
length:(unsigned)*length*

Returns a mutable string containing *length* characters made from *byteString*. This method doesn't stop at a null byte. *byteString* should contain bytes in the default C string encoding.

See also: + **defaultCStringEncoding** (NSString)

stringWithCString:

+ (NSMutableString *)**stringWithCString:**(const char *)*byteString*

Returns a mutable string containing the characters in *byteString*, which must be null-terminated. *bytes* should contain bytes in the default C string encoding.

See also: + **defaultCStringEncoding** (NSString)

stringWithFormat:

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

Returns a mutable string created by using *format* as a **printf()** style format string, and the subsequent arguments as values to be substituted into the format string.

Instance Methods

appendFormat:

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

Adds a constructed string to the receiver. The new characters are created by using *format* as a **printf()** style format string, and the following arguments as values to be substituted into the format string.

This method is equivalent to invoking **stringWithFormat:** with the arguments listed, and passing the resulting string to **appendString:**.

appendString:

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

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

deleteCharactersInRange:

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

Removes from the receiver the characters in *aRange*. This method raises an **NSRangeException** exception if any part of *aRange* lies beyond the end of the string.

initWithCapacity:

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

Initializes a newly allocated mutable string object, giving it enough allocated memory to hold *capacity* characters.

insertString:atIndex:

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

Inserts the characters of *aString* into the receiver, such that the new characters begin at *anIndex* and the existing character from *anIndex* to the end are shifted by the length of *aString*. This method raises an **NSRangeException** exception if *anIndex* lies beyond the end of the string.

replaceCharactersInRange:withString:

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

Inserts the characters of *otherString* into the receiver, such that they replace the characters in *aRange*. This method raises an **NSRangeException** exception 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*.