SearchableText

Protocol Description

The SearchableText protocol defines a set of methods for an object to implement if it wants to provide searching functionality. This need not apply to only Text or TextField objects±an NXBrowser, for example, could provide the text of its cells. There are no set requirements to be able to provide text searching via the SearchableText protocol, but if an object has a user interface representation, it should have some form of *selection* and be able to interpret its text in a linear way (what this means for a non-Text object, a Matrix for instance, will hopefully be made clear in the following discussion).

The SearchableText models

Searchable text, that is, text that can be operated on by an object conforming to the SearchableText protocol, is modeled in two ways: as in C, a *linear* sequence of characters; and, particular to the SearchableText protocol, as a *circular* sequence of characters.

figure.eps ¬

In the linear model, a sequence of text has one beginning and one ending point; it has two *edges*. Moving from the beginning to the end is moving in a *forward* direction; similarly, moving from the end to the beginning is moving in a *reverse* direction. There is no text before the beginning, in this model, and no text after the end. Two other important points, the *start of selection* and the *end of selection* lie between the beginning and end points (if they exist at all±see the section **Selection states**). The start of selection is defined to be between the beginning of the text and the end of selection, and the end of selection is between the start of selection and the end of the text; they may also be coincident with each other, and/or coincident with the beginning or end points.

Under the circular model, on the other hand, text has only one edge. *One edge*, you say. *Why is there an edge at all if the text is circular?* Well, a similar situation exists with polar coordinates±there is an arbitrary ray which acts as a reference point from which angles are measured. The edge in the circular text model acts as a reference point for searches of the text. It is also, most commonly, the beginning *and* ending point of the text; that is, it is a point just before the beginning of the text and just after the end. This means that under this model there is text before the beginning of the text±namely, the text at the end (as it were)±and text after the end. (The concept of a beginning and end of the text are borrowed from the linear model, and only make sense in the presence of a text edge, which itself is an artificial construct introduced to ease the mapping from the linear model, and provide a fixed point of reference in the model to make some other things easier. However, for most of this document I will assume that the reader desires to use the circular model with linear text, and ignore these issues.)

But what does it mean to move in a forward direction in the circular text model? We adopt the convention that it should mean the same thing it means in the linear model; namely, that in a forward direction, the start of the selection is between the text edge and the end of the selection, and as one moves from the text edge to the text edge, the start of the selection is passed through before the end of the selection is passed through (unless the points are coincident). The beginning of the text is taken to be just before the first character closest to the text edge, moving in a forward direction (coincident with the text edge, note). In the reverse direction, the end of the selection is reached before the start of the selection, again moving from text edge to text edge, and the end of text is just before the first character closest to the text edge (in a reverse direction). If the circular text model is to be applied to the text of an object that has no concept of a selection, then some other definitions of forward and reverse must be constructed.

Searching under the circular text model

So, under this strange, yet wonderful, circular model, there are three points of reference on which searches can be based±nine permutations of starting and ending positions (see **Constants and Defined Types** for the names of constants that define each of these). Notice that the three points also define three regions in the text. Searching from the start of selection to the end of selection in a forward direction, for instance, seems to be a straight-forward operation (no pun intended). But searching from the end of selection to the start of selection? The figure above makes this clear: a forward search will cross the edge of the text, searching two of three regions; a reverse search will search one region, the region not searched by the same forward search. This illustrates another property of searching under the circular text model: reversed text searching not only reverses the direction of a search, but also complements the set of text to search (except for the three permutations that search the entire text).

Now, interpretation of the circular text model is somewhat implementation dependant, with respect to the "boundaries" formed by the text edge, start of selection, and end of selection points. In *theory*, there are no seams in circular text, and the three special points each exist *between* a pair of characters. The character "on one side" of the start of selection is adjacent to the character "on the other side". However, in practice users will find that an implementation that includes part of the previous match in a "next-found" match, or which finds a match that crosses the text edge, to be unintuitive. Therefore, it is suggested that implementations use the following two guidelines:

- When mapping from a linear text model (say, that of the AppKit Text class) to the circular model, always treat the text edge as a hard boundary, i.e., successful searches will never result in a match that crosses the text edge.
- Treat the start of selection and end of selection points as hard boundaries only when they (either or both) are an end point of a search. For instance, in a TextEdgeToSelEnd search, the end of selection should be a hard boundary, but the start of selection point should not (in other words, a successful match could cross the start of selection point that existed before the search began).

There may also be other implemenation considerations. For instance, in a matrix of cells, it may not be appropriate to select only part of the text of a specific cell, but all the text of the cell should be selected.

Also, using the same example, it may not be appropriate to begin a text match in the text of one cell, and complete the match in another cell.   Implementors must consider these issues carefully, and compensate for them in their actions and the return values from the methods.

Selection states

There are three "abstract" states that a selection can be in:

- Non-existent.   There is no selection at all; the start of selection and end of selection are undefined. This is the only state possible for objects providing searchable text, but have no concept of a "selection".   Methods or search modes which operate on a selection do nothing (or return an error indication) when the selection is in this state.
- Existent, but empty.   A selection exists, but the length of the selection is zero (the start and end of selection points are coincident, for example).
- Existent, and non-empty.   A selection exists, and has length greater than zero.

The state of the selection also encompasses exactly which set of characters is selected.   Objects that can have selections must be able to map to and from the circular text model used by this protocol and their own text representation. For instance, a Matrix of TextFieldCells must be able to return appropriate values from the **searchFor:mode:reverse:regexpr:cases:position:size:** method and translate the parameters of **selectTextFrom:to:** into a selection of the appropriate cell(s).

These states are described only for completeness. No concrete reference to them is required by the protocol.

Instance Methods

makeSelectionVisible
  - (oneway void)**makeSelectionVisible**

Adjusts the view that contains the text to make the selection visible. The state of the selection is not changed. If there is no selection, or this action is otherwise inappropriate, this method does nothing.

replaceAll:with:mode:regexpr:cases:

- (int)**replaceAll:**(const char *)*pattern*
  with:(const char *)*replacement*
  mode:(SearchMode)*mode*
  regexpr:(BOOL)*regexpr*
  cases:(BOOL)*cases*

Replaces all instances of matches of the null-terminated string *pattern* in the searchable text with the null-terminated string *replacement*, and returns the number of replacements made. *replacement* is a literal string that should be substituted for each instance of text matching *pattern*. If *regexpr* is NO, *pattern* is treated as a literal string, otherwise *pattern* is interpreted as a regular expression (the regular expression syntax is not defined, and is chosen by an implementation). If *cases* is YES, the search is case sensitive, otherwise it is not. See the section **Constants and Defined Types** for the values of the *mode* parameter. Upon error (for instance, if the state of the text or selection does not support the operation requested), a number less than 0 (a value of type SearchErr) is returned. The state of the selection after this operation is not defined (an implementation may choose any behavior).

replaceSelection:

- (oneway void)**replaceSelection:**(const char *)*replacement*

Replaces the current selection in the text with the null-terminated string *replacement*. If the selection is empty, the string *replacement* is inserted in the text at that point. If there is no selection, or this action is otherwise not appropriate, this method does nothing. The state of the selection after the operation is not defined (an implementation may choose any behavior).

searchFor:mode:reverse:regexpr:cases:position:size:

- (int)**searchFor:**(const char *)*pattern*
  mode:(SearchMode)*mode*
  reverse:(BOOL)*rev*
  regexpr:(BOOL)*regexpr*
  cases:(BOOL)*cases*
  position:(out int *)*pos*
  size:(out int *)*size*

Searches for matches to the null-terminated string *pattern* in the searchable text. The text to be searched is indicated by the *mode* and *rev* parameters. See the section **Constants and Defined Types** for the values of the *mode* parameter. If *rev* is YES, a "backwards" search is performed. If *regexpr* is NO, *pattern* is treated as a literal string, otherwise *pattern* is interpreted as a regular expression (the regular expression syntax is not defined, and is chosen by an implementation). If *cases* is YES, the search is case sensitive, otherwise it is not. If text matching *pattern* is found in the text, the position in the text from the text edge in a forward direction to the match (indexed from zero) is returned by reference in *pos*, the length of the matched text in *size*, and 1 is returned by the method. If text matching *pattern* is not found, this method returns 0. Upon error (for instance, if the state of the text or selection does not support the operation requested), a number less than 0 (a value of type SearchErr) is returned. The state of the selection after the operation is not defined (an implementation may choose any behavior).

selectTextFrom:to:

- (oneway void)**selectTextFrom:**(int)start **to:**(int)end

Sets the selected characters to be those from position *start* to position *end*-1 inclusive, relative to the text edge and indexed from zero. The previous selection state is discarded. Both values must be non-negative, and *start*<=*end*. If *start* and *end* are equal, the selection is set to be empty, at the position "between" positions *start*-1 and *start*. *start* must be less than or equal to *end*; otherwise, the behavior of this method is not defined. After this operation, the selection need not be visible to the user, but an implementation may choose to make it so. If this action is not appropriate, this method does nothing.

writeSelectionToPasteboard:asType:

- (void)**writeSelectionToPasteboard:**(in Pasteboard *)*pboard* **asType:**(in NXAtom)*type*

Writes the characters currently in the selection to the pasteboard *pboard* as the type *type*. If there is no selection, or the selection is empty, this method does nothing. The state of the selection is not changed.

Constants and Defined Types

SearchMode

SYNOPSIS

typedef enum {
TextEdgeToSelStart,
TextEdgeToSelEnd,
TextEdgeToTextEdge,
SelStartToSelEnd,
SelStartToTextEdge,
SelStartToSelStart,
SelEndToTextEdge,
SelEndToSelStart,
SelEndToSelEnd
} **SearchMode**;

DESCRIPTION

These values are used as an argument to the **replaceAll:with:mode:regexpr:cases:** and **searchFor:mode:reverse:regexpr:cases:position:size:** methods, to specify the extent of the search and replace operations. See the **Protocol Description** for more information.

Value	Direction	Search from the first character after the...	through to the last character before the...
TextEdgeToSelStart	Forward	beginning of text	start of selection
	Reverse	end of text	start of selection
TextEdgeToSelEnd	Forward	beginning of text	end of selection
	Reverse	end of text	end of selection
TextEdgeToTextEdge	Forward	beginning of text	end of text
	Reverse	end of text	beginning of text
SelStartToSelEnd	Forward	start of selection	end of selection
	Reverse	start of selection	end of selection
SelStartToTextEdge	Forward	start of selection	end of text
	Reverse	start of selection	beginning of text

SelStartToSelStart	Forward	start of selection	start of selection	Reverse	start of selection	start of selection
SelEndToTextEdge	Forward	end of selection	end of text	Reverse	end of selection	beginning of text
SelEndToSelStart	Forward	end of selection	start of selection	Reverse	end of selection	start of selection
SelEndToSelEnd	Forward	end of selection	end of selection	Reverse	end of selection	end of selection

SearchErr

SYNOPSIS

```
typedef enum {
    SEARCH_INVALID_OPERATION = -1,
    SEARCH_INVALID_ARGUMENT = -2,
    SEARCH_INVALID_REGEXPR = -3,
    SEARCH_NO_SELECTION = -4,
    SEARCH_CANNOT_WRITE = -5,
    SEARCH_UNIMPLEMENTED = -6,
    SEARCH_ABORTED = -7,
    SEARCH_INTERNAL_ERROR = -8
} SearchErr;
```

DESCRIPTION

These values are returned from the **replaceAll:with:mode:regexpr:cases:** and **searchFor:mode:reverse:regexpr:cases:position:size:** methods in the event of an error. An implementation of the SearchableText protocol need not use any value other than SEARCH_INVALID_OPERATION, but it is recommended that an implementation be as specific as possible.

SEARCH_INVALID_OPERATION	A generic, unspecified failure.
SEARCH_INVALID_ARGUMENT	An invalid argument was passed to the method. For

	example, *pattern* was NULL.
SEARCH_INVALID_REGEXPR	The *pattern* was not a valid regular expression, when regular expression matching was requested.
SEARCH_NO_SELECTION	There is no selection (not even an empty one) and the mode of the search or replace operation requested was relative to the selection.
SEARCH_CANNOT_WRITE	The text of the object is read-only.
SEARCH_UNIMPLEMENTED	A feature or combination of parameters has not been implemented.
SEARCH_ABORTED	The operation was aborted by the user before it could be completed.
SEARCH_INTERNAL_ERROR	An unspecified internal error occured. For example, memory allocation failure.