ABSTRACT

This specification describes a flexible and versatile text wrapping JavaScript function that prioritizes existing line breaks within the text, supports custom break points, with the option of removing them or not from the text, and permits adding custom gutters or line prefixes.

SIGNATURE, ARGUMENTS AND RETURN VALUE

The signature of the function is:

`wrapText (text, colsNum = 80, gutters = null, breakers = null)`

where:

- `text` is an arbitrary UTF-8 String, in any language. It is acceptable and expected that this String contains newline sequences, such as `\n`, `\r\n`, `\r`.

- `colsNum` is an optional integer that guides the number of chars to be fitted on each line of text; defaults to `80`.

- `gutters` is an optional Array of strings; if provided, they will be prepended to each line of text, respectively, based on availability rules (see next).

- `breakers` is an optional Array of Objects, each having two keys: `value` (a String or Regex) and `shouldConsume` (a Boolean). They provide the break opportunities needed to wrap the string into lines; defaults to: [{ value: /\s+/, shouldConsume: true }].

The `wrapText` function should return an Array of Strings, each element in the Array representing a line of text.

FLOW AND FUNCTIONALITY NOTES

1. The received `text` should first be split into substrings based on its containing newline sequences.

2. The resulting substrings will be further split into tokens, based on the `breakers` Array. This is a rather complex step, and is described below:

2.1. In the scope of this document, a `token` is an Object with three keys: `str`, `delim` and `size`.

2.2. Substrings are split by any of the items in the `breakers` Array, and results are stored in the `str` keys of dedicated token Objects (a suggested way of doing the split is described in the APENDIX of this specification).

2.3. If the "breaker" item a particular split was done by had its `shouldConsume` property set to `false`, then the `delim` property of relevant tokens should contain the chars that would otherwise be lost from the substring as a result of the split (or, chars that would be "consumed" by the split, if you will).

For instance, the implied/default `breakers` Array is [{ value: /\s+/, shouldConsume: true }], which causes a String to be split by any dense span of whitespace chars, with that whitespace also being removed from the string. Since `shouldConsume` is true here, the respective dense spans of whitespace chars shall NOT be stored to the `delim` properties of relevant resulting tokens. Were it `false`, they should have been.

2.4. The `size` key holds the accumulated char count of `str` and `delim` if `delim` is not empty, or the `str` char count plus `1`, otherwise.

3. The capacity (or maximum char allowance) of each line is computed, based on `colsNum` and `gutters`. Each line will have a capacity of `colsNum` subtracted the length, in chars, of the gutter of that line. The gutter of that line is the [<line index>th] gutter in the `gutters` Array, if one exists in-there, or the last gutter available in the Array otherwise. Lacking any `gutters`, the line capacity is simply `colsNum`.

4. Virtual lines of text are built, namely by "fitting" tokens within the available capacity/char allowance of each line. Each line starts with its "gutter", where applicable, with the gutter being determined as per the rules given at #3. The gutter string, should it exist, is to be immediately followed by the `str` of the first token fitted on that line, with no intervening space.

IMPORTANT:

The delimitations effected at #1, where we split the original string into substrings based on original string's intrinsic new line sequences MUST BE KEPT. Under no circumstances will, e.g., a token resulting from the first substring be fitted on a virtual line pertaining to the second substring. Essentially, this means that we will always seal (close) a line when running out of tokens from a specific substring, even if that line is underfilled.

The logic for fitting tokens to lines goes as follows:

4.1. While the number of chars currently on the line plus the `size` of a candidate token gives less than or equal to the line capacity, the candidate token is fitted on the line.

4.2. Also, if the comparison above yields a `+1` delta (i.e., adding a candidate token's `size` would outweigh the line capacity by `1`) AND that candidate token's `delim` is empty, then the candidate token is fitted on the line as well.

4.3. Otherwise, the line is sealed, and a new one is opened to accept tokens.

NOTE:

"Fitting" a token implies adding the value of its `str` property to the line. If the token's `delim` property is not empty, the chars within its `delim` will also be added to the line, immediately following those of `str`. Otherwise, a single whitespace char will be added to the line, UNLESS we are in the 4.2 situation described above.

APENDIX

5. THE SUGGESTED mechanism to be used for extracting tokens from substrings is:

5.1. from the current char position of the substring we are extracting tokens from, search in turn for each of the items in the `breakers` Array and take note of their respective match indexes. Act upon the most nearby match only, by doing the following:

5.1.1. slice the string from the current char index up to but not including the index of the match; create an Object and set its `str` property to that.

5.1.2. if the breaker item pertaining to that match has its `shouldConsume` property set to `false`, then also place the chars matched inside the `delim` property of the Object created above.

5.1.3. compute the sum of char lengths of `str` and `delim` (the later shall be `0` if not applicable) and store that as `size` within that Object.

5.2. store the Object, being cautious not to mix tokens obtained from one substring with tokens from another.