savetheclocktower Ensure `String#extractScripts` and `String#evalScripts` ignore `SCRIP... ... History

6 contributors

927 lines (883 sloc) | 30.2 KB

```
/** section: Language
 * class String
 *
 *  Extensions to the built-in `String` class.
 *
 *  Prototype enhances the [[String]] object with a series of useful methods for
 *  ranging from the trivial to the complex. Tired of stripping trailing
 *  whitespace? Try [[String#strip]]. Want to replace `replace`? Have a look at
 *  [[String#sub]] and [[String#gsub]]. Need to parse a query string? We have
 *  [[String#toQueryParams what you need]].
**/
Object.extend(String, {
  /**
   *  String.interpret(value) -> String
   *
   *  Coerces `value` into a string. Returns an empty string for `null`.
  **/
  interpret: function(value) {
    return value == null ? '' : String(value);
  },
  specialChar: {
    '\b': '\\b',
    '\t': '\\t',
    '\n': '\\n',
    '\f': '\\f',
    '\r': '\\r',
    '\\': '\\\\'
  }
});

Object.extend(String.prototype, (function() {

  function prepareReplacement(replacement) {
    if (Object.isFunction(replacement)) return replacement;
    var template = new Template(replacement);
    return function(match) { return template.evaluate(match) };
  }

  // In some versions of Chrome, an empty RegExp has "(?:)" as a `source`
  // property instead of an empty string.
  function isNonEmptyRegExp(regexp) {
    return regexp.source && regexp.source !== '(?:)';
  }


  /**
   *  String#gsub(pattern, replacement) -> String
   *
   *  Returns the string with _every_ occurence of a given pattern replaced by either a
   *  regular string, the returned value of a function or a [[Template]] string.
   *  The pattern can be a string or a regular expression.
   *
   *  If its second argument is a string [[String#gsub]] works just like the native JavaScript
   *  method `replace()` set to global match.
   *
   *      var mouseEvents = 'click dblclick mousedown mouseup mouseover mousemove mouseout';
   *
   *      mouseEvents.gsub(' ', ', ');
   *      // -> 'click, dblclick, mousedown, mouseup, mouseover, mousemove, mouseout'
   *
   *      mouseEvents.gsub(/\s+/, ', ');
   *      // -> 'click, dblclick, mousedown, mouseup, mouseover, mousemove, mouseout'
   *
   *  If you pass it a function, it will be invoked for every occurrence of the pattern
   *  with the match of the current pattern as its unique argument. Note that this argument
   *  is the returned value of the `match()` method called on the current pattern. It is
   *  in the form of an array where the first element is the entire match and every subsequent
   *  one corresponds to a parenthesis group in the regex.
   *
   *      mouseEvents.gsub(/\w+/, function(match){ return 'on' + match[0].capitalize() });
   *      // -> 'onClick onDblclick onMousedown onMouseup onMouseover onMousemove onMouseout'
   *
   *      var markdown = '![a pear](/img/pear.jpg) ![an orange](/img/orange.jpg)';
   *
   *      markdown.gsub(/!\[(.*?)\]\((.*?)\)/, function(match) {
   *        return '<img alt="' + match[1] + '" src="' + match[2] + '" />';
   *      });
   *      // -> '<img alt="a pear" src="/img/pear.jpg" /> <img alt="an orange" src="/img/orange.jpg" />'
 *
 *  Lastly, you can pass [[String#gsub]] a [[Template]] string in which you can also access
 *  the returned value of the `match()` method using the ruby inspired notation: `#{0}`
 *  for the first element of the array, `#{1}` for the second one, and so on.
 *  So our last example could be easily re-written as:
 *
 *      markdown.gsub(/!\[(.*?)\]\((.*?)\)/, '<img alt="#{1}" src="#{2}" />');
 *      // -> '<img alt="a pear" src="/img/pear.jpg" /> <img alt="an orange" src="/img/orange.jpg" />'
 *
 *  If you need an equivalent to [[String#gsub]] but without global match set on, try [[String#sub]].
 *
 *  ##### Note
 *
 *  Do _not_ use the `"g"` flag on the regex as this will create an infinite loop.
**/
function gsub(pattern, replacement) {
  var result = '', source = this, match;
  replacement = prepareReplacement(replacement);

  if (Object.isString(pattern))
    pattern = RegExp.escape(pattern);

  if (!(pattern.length || isNonEmptyRegExp(pattern))) {
    replacement = replacement('');
    return replacement + source.split('').join(replacement) + replacement;
  }

  while (source.length > 0) {
    match = source.match(pattern)
    if (match && match[0].length > 0) {
      result += source.slice(0, match.index);
      result += String.interpret(replacement(match));
      source  = source.slice(match.index + match[0].length);
    } else {
      result += source, source = '';
    }
  }
  return result;
}

/**
 *  String#sub(pattern, replacement[, count = 1]) -> String
 *
 *  Returns a string with the _first_ `count` occurrences of `pattern` replaced by either
 *  a regular string, the returned value of a function or a [[Template]] string.
 *  `pattern` can be a string or a regular expression.
 *
 *  Unlike [[String#gsub]], [[String#sub]] takes a third optional parameter which specifies
 *  the number of occurrences of the pattern which will be replaced.
 *  If not specified, it will default to 1.
 *
 *  Apart from that, [[String#sub]] works just like [[String#gsub]].
 *  Please refer to it for a complete explanation.
 *
 *  ##### Examples
 *
 *      var fruits = 'apple pear orange';
 *
 *      fruits.sub(' ', ', ');
 *      // -> 'apple, pear orange'
 *
 *      fruits.sub(' ', ', ', 1);
 *      // -> 'apple, pear orange'
 *
 *      fruits.sub(' ', ', ', 2);
 *      // -> 'apple, pear, orange'
 *
 *      fruits.sub(/\w+/, function(match){ return match[0].capitalize() + ',' }, 2);
 *      // -> 'Apple, Pear, orange'
 *
 *      var markdown = '![a pear](/img/pear.jpg) ![an orange](/img/orange.jpg)';
 *
 *      markdown.sub(/!\[(.*?)\]\((.*?)\)/, function(match) {
 *        return '<img alt="' + match[1] + '" src="' + match[2] + '" />';
 *      });
 *      // -> '<img alt="a pear" src="/img/pear.jpg" /> ![an orange](/img/orange.jpg)'
 *
 *      markdown.sub(/!\[(.*?)\]\((.*?)\)/, '<img alt="#{1}" src="#{2}" />');
 *      // -> '<img alt="a pear" src="/img/pear.jpg" /> ![an orange](/img/orange.jpg)'
 *
 *  ##### Note
 *
 *  Do _not_ use the `"g"` flag on the regex as this will create an infinite loop.
**/
function sub(pattern, replacement, count) {
  replacement = prepareReplacement(replacement);
  count = Object.isUndefined(count) ? 1 : count;

  return this.gsub(pattern, function(match) {
    if (--count < 0) return match[0];
    return replacement(match);
  });
}

/** related to: String#gsub
 *  String#scan(pattern, iterator) -> String
 *
 *  Allows iterating over every occurrence of the given pattern (which can be a
   *  string or a regular expression).
   *  Returns the original string.
   *
   *  Internally just calls [[String#gsub]] passing it `pattern` and `iterator` as arguments.
   *
   *  ##### Examples
   *
   *      'apple, pear & orange'.scan(/\w+/, alert);
   *      // -> 'apple pear & orange' (and displays 'apple', 'pear' and 'orange' in three successive alert dialogs)
   *
   *  Can be used to populate an array:
   *
   *      var fruits = [];
   *      'apple, pear & orange'.scan(/\w+/, function(match) { fruits.push(match[0]) });
   *      fruits.inspect()
   *      // -> ['apple', 'pear', 'orange']
   *
   *  or even to work on the DOM:
   *
   *      'failure-message, success-message & spinner'.scan(/(\w|-)+/, Element.toggle)
   *      // -> 'failure-message, success-message & spinner' (and toggles the visibility of each DOM element)
   *
   *  ##### Note
   *
   *  Do _not_ use the `"g"` flag on the regex as this will create an infinite loop.
  **/
  function scan(pattern, iterator) {
    this.gsub(pattern, iterator);
    return String(this);
  }

  /**
   *  String#truncate([length = 30[, suffix = '...']]) -> String
   *
   *  Truncates a string to given `length` and appends `suffix` to it (indicating
   *  that it is only an excerpt).
   *
   *  ##### Examples
   *
   *      'A random sentence whose length exceeds 30 characters.'.truncate();
   *      // -> 'A random sentence whose len...'
   *
   *      'Some random text'.truncate();
   *      // -> 'Some random text.'
   *
   *      'Some random text'.truncate(10);
   *      // -> 'Some ra...'
   *
   *      'Some random text'.truncate(10, ' [...]');
   *      // -> 'Some [...]'
  **/
  function truncate(length, truncation) {
    length = length || 30;
    truncation = Object.isUndefined(truncation) ? '...' : truncation;
    return this.length > length ?
      this.slice(0, length - truncation.length) + truncation : String(this);
  }

  /**
   *  String#strip() -> String
   *
   *  Strips all leading and trailing whitespace from a string.
   *
   *  ##### Example
   *
   *      '    hello world!    '.strip();
   *      // -> 'hello world!'
  **/
  function strip() {
    return this.replace(/^\s+/, '').replace(/\s+$/, '');
  }

  /**
   *  String#stripTags() -> String
   *
   *  Strips a string of any HTML tags.
   *
   *  Note that [[String#stripTags]] will only strip HTML 4.01 tags &mdash; like
   *  `div`, `span`, and `abbr`. It _will not_ strip namespace-prefixed tags
   *  such as `h:table` or `xsl:template`.
   *
   *  Watch out for `<script>` tags in your string, as [[String#stripTags]] will
   *  _not_ remove their content. Use [[String#stripScripts]] to do so.
   *
   *  ##### Caveat User
   *
   *  Note that the processing [[String#stripTags]] does is good enough for most
   *  purposes, but you cannot rely on it for security purposes. If you're
   *  processing end-user-supplied content, [[String#stripTags]] is _not_
   *  sufficiently robust to ensure that the content is completely devoid of
   *  HTML tags in the case of a user intentionally trying to circumvent tag
   *  restrictions. But then, you'll be running them through
   *  [[String#escapeHTML]] anyway, won't you?
   *
   *  ##### Examples
   *
   *      'a <a href="#">link</a>'.stripTags();
   *       // -> 'a link'
   *
   *      'a <a href="#">link</a><script>alert("hello world!");</script>'.stripTags();
   *      // -> 'a linkalert("hello world!");'
   *
   *      'a <a href="#">link</a><script>alert("hello world!");</script>'.stripScripts().stripTags();
   *      // -> 'a link'
  **/
  function stripTags() {
    return this.replace(/<\w+(\s+("[^"]*"|'[^']*'|[^>])+)?(\/)?>|<\/\w+>/gi, '');
  }

  /**
   *  String#stripScripts() -> String
   *
   *  Strips a string of things that look like HTML script blocks.
   *
   *  ##### Example
   *
   *      "<p>This is a test.<script>alert("Look, a test!");</script>End of test</p>".stripScripts();
   *      // => "<p>This is a test.End of test</p>"
   *
   *  ##### Caveat User
   *
   *  Note that the processing [[String#stripScripts]] does is good enough for
   *  most purposes, but you cannot rely on it for security purposes. If you're
   *  processing end-user-supplied content, [[String#stripScripts]] is probably
   *  not sufficiently robust to prevent hack attacks.
  **/
  function stripScripts() {
    return this.replace(new RegExp(Prototype.ScriptFragment, 'img'), '');
  }

  /**
   *  String#extractScripts() -> Array
   *
   *  Extracts the content of any `<script>` blocks present in the string and
   *  returns them as an array of strings.
   *
   *  This method is used internally by [[String#evalScripts]]. It does _not_
   *  evaluate the scripts (use [[String#evalScripts]] to do that), but can be
   *  usefull if you need to evaluate the scripts at a later date.
   *
   *  ##### Examples
   *
   *      'lorem... <script>2 + 2</script>'.extractScripts();
   *      // -> ['2 + 2']
   *
   *      '<script>2 + 2</script><script>alert("hello world!")</script>'.extractScripts();
   *      // -> ['2 + 2', 'alert("hello world!")']
   *
   *  ##### Notes
   *
   *  To evaluate the scripts later on, you can use the following:
   *
   *      var myScripts = '<script>2 + 2</script><script>alert("hello world!")</script>'.extractScripts();
   *      // -> ['2 + 2', 'alert("hello world!")']
   *
   *      var myReturnedValues = myScripts.map(function(script) {
   *        return eval(script);
   *      });
   *      // -> [4, undefined] (and displays 'hello world!' in the alert dialog)
  **/
  function extractScripts() {
    var matchAll = new RegExp(Prototype.ScriptFragment, 'img'),
        matchOne = new RegExp(Prototype.ScriptFragment, 'im');
    var matchMimeType = new RegExp(Prototype.ExecutableScriptFragment, 'im');
    var matchTypeAttribute = /type=/i;

    var results = [];
    (this.match(matchAll) || []).each(function(scriptTag) {
      var match = scriptTag.match(matchOne);
      var attributes = match[1];
      if (attributes !== '') {
        // If the script has a `type` attribute, make sure it has a
        // JavaScript MIME-type. If not, ignore it.
        attributes = attributes.strip();
        var hasTypeAttribute = (matchTypeAttribute).test(attributes);
        var hasMimeType = (matchMimeType).test(attributes);
        if (hasTypeAttribute && !hasMimeType) return;
      }
      results.push(match ? match[2] : '');
    });

    return results;
  }

  /**
   *  String#evalScripts() -> Array
   *
   *  Evaluates the content of any inline `<script>` block present in the string.
   *  Returns an array containing the value returned by each script.
   *  `<script>`  blocks referencing external files will be treated as though
   *  they were empty (the result for that position in the array will be `undefined`);
   *  external files are _not_ loaded and processed by [[String#evalScripts]].
   *
   *  ##### Examples
   *
   *      'lorem... <script>2 + 2</script>'.evalScripts();
```

```
 *      // -> [4]
 *
 *      '<script>2 + 2<script><script>alert("hello world!")</script>'.evalScripts();
 *      // -> [4, undefined] (and displays 'hello world!' in the alert dialog)
 *
 *  ##### About `evalScripts`, `var`s, and defining functions
 *
 *  [[String#evalScripts]] evaluates script blocks, but this **does not** mean
 *  they are evaluated in the global scope. They aren't, they're evaluated in
 *  the scope of the [[String#evalScripts]] method. This has important
 *  ramifications for your scripts:
 *
 *  * Anything in your script declared with the `var` keyword will be
 *    discarded momentarily after evaluation, and will be invisible to any
 *    other scope.
 *  * If any `<script>` blocks _define functions_, they will need to be
 *    assigned to properties of the `window` object.
 *
 *  For example, this won't work:
 *
 *      // This kind of script won't work if processed by evalScripts:
 *      function coolFunc() {
 *        // Amazing stuff!
 *      }
 *
 *  Instead, use the following syntax:
 *
 *      // This kind of script WILL work if processed by evalScripts:
 *      window.coolFunc = function() {
 *        // Amazing stuff!
 *      }
 *
 *  (You can leave off the `window.` part of that, but it's bad form.)
**/
function evalScripts() {
  return this.extractScripts().map(function(script) { return eval(script); });
}

/** related to: String#unescapeHTML
 *  String#escapeHTML() -> String
 *
 *  Converts HTML special characters to their entity equivalents.
 *
 *  ##### Example
 *
 *      '<div class="article">This is an article</div>'.escapeHTML();
 *      // -> "&lt;div class="article"&gt;This is an article&lt;/div&gt;"
**/
function escapeHTML() {
  return this.replace(/&/g,'&amp;').replace(/</g,'&lt;').replace(/>/g,'&gt;');
}

/** related to: String#escapeHTML
 *  String#unescapeHTML() -> String
 *
 *  Strips tags and converts the entity forms of special HTML characters
 *  to their normal form.
 *
 *  ##### Examples
 *
 *      'x &gt; 10'.unescapeHTML()
 *      // -> 'x > 10'
 *
 *      '<h1>Pride &amp; Prejudice</h1>;'.unescapeHTML()
 *      // -> '<h1>Pride & Prejudice</h1>'
**/
function unescapeHTML() {
  // Warning: In 1.7 String#unescapeHTML will no longer call String#stripTags.
  return this.stripTags().replace(/&lt;/g,'<').replace(/&gt;/g,'>').replace(/&amp;/g,'&');
}

/**
 *  String#parseQuery([separator = '&']) -> Object
**/

/** alias of: String#parseQuery, related to: Hash#toQueryString
 *  String#toQueryParams([separator = '&']) -> Object
 *
 *  Parses a URI-like query string and returns an object composed of
 *  parameter/value pairs.
 *
 *  This method is realy targeted at parsing query strings (hence the default
 *  value of`"&"` for the `separator` argument).
 *
 *  For this reason, it does _not_ consider anything that is either before a
 *  question  mark (which signals the beginning of a query string) or beyond
 *  the hash symbol (`"#"`), and runs `decodeURIComponent()` on each
 *  parameter/value pair.
 *
 *  [[String#toQueryParams]] also aggregates the values of identical keys into
 *  an array of values.
 *
 *  Note that parameters which do not have a specified value will be set to
 *  `undefined`.
 *
 *  ##### Examples
 *
 *      'section=blog&id=45'.toQueryParams();
```

```javascript
   *      // -> {section: 'blog', id: '45'}
   *
   *      'section=blog;id=45'.toQueryParams(';');
   *      // -> {section: 'blog', id: '45'}
   *
   *      'http://www.example.com?section=blog&id=45#comments'.toQueryParams();
   *      // -> {section: 'blog', id: '45'}
   *
   *      'section=blog&tag=javascript&tag=prototype&tag=doc'.toQueryParams();
   *      // -> {section: 'blog', tag: ['javascript', 'prototype', 'doc']}
   *
   *      'tag=ruby%20on%20rails'.toQueryParams();
   *      // -> {tag: 'ruby on rails'}
   *
   *      'id=45&raw'.toQueryParams();
   *      // -> {id: '45', raw: undefined}
  **/
  function toQueryParams(separator) {
    var match = this.strip().match(/([^?#]*)(#.*)?$/);
    if (!match) return { };

    return match[1].split(separator || '&').inject({ }, function(hash, pair) {
      if ((pair = pair.split('='))[0]) {
        var key = decodeURIComponent(pair.shift()),
            value = pair.length > 1 ? pair.join('=') : pair[0];

        if (value != undefined) {
          value = value.gsub('+', ' ');
          value = decodeURIComponent(value);
        }

        if (key in hash) {
          if (!Object.isArray(hash[key])) hash[key] = [hash[key]];
          hash[key].push(value);
        }
        else hash[key] = value;
      }
      return hash;
    });
  }

  /**
   *  String#toArray() -> Array
   *
   *  Splits the string character-by-character and returns an array with
   *  the result.
   *
   *  ##### Examples
   *
   *      'a'.toArray();
   *      // -> ['a']
   *
   *      'hello world!'.toArray();
   *      // -> ['h', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r', 'l', 'd', '!']
  **/
  function toArray() {
    return this.split('');
  }

  /**
   *  String#succ() -> String
   *
   *  Used internally by ObjectRange.
   *
   *  Converts the last character of the string to the following character in
   *  the Unicode alphabet.
   *
   *  ##### Examples
   *
   *      'a'.succ();
   *      // -> 'b'
   *
   *      'aaaa'.succ();
   *      // -> 'aaab'
  **/
  function succ() {
    return this.slice(0, this.length - 1) +
      String.fromCharCode(this.charCodeAt(this.length - 1) + 1);
  }

  /**
   *  String#times(count) -> String
   *
   *  Concatenates the string `count` times.
   *
   *  ##### Example
   *
   *      "echo ".times(3);
   *      // -> "echo echo echo "
  **/
  function times(count) {
    return count < 1 ? '' : new Array(count + 1).join(this);
  }

  /**
   *  String#camelize() -> String
   *
   *  Converts a string separated by dashes into a camelCase equivalent. For
   *  instance, `'foo-bar'` would be converted to `'fooBar'`.
   *
   *  Prototype uses this internally for translating CSS properties into their
   *  DOM `style` property equivalents.
   *
   *  ##### Examples
   *
   *      'background-color'.camelize();
   *      // -> 'backgroundColor'
   *
   *      '-moz-binding'.camelize();
   *      // -> 'MozBinding'
  **/
  function camelize() {
    return this.replace(/-+(.)?/g, function(match, chr) {
      return chr ? chr.toUpperCase() : '';
    });
  }

  /**
   *  String#capitalize() -> String
   *
   *  Capitalizes the first letter of a string and downcases all the others.
   *
   *  ##### Examples
   *
   *      'hello'.capitalize();
   *      // -> 'Hello'
   *
   *      'HELLO WORLD!'.capitalize();
   *      // -> 'Hello world!'
  **/
  function capitalize() {
    return this.charAt(0).toUpperCase() + this.substring(1).toLowerCase();
  }

  /**
   *  String#underscore() -> String
   *
   *  Converts a camelized string into a series of words separated by an
   *  underscore (`_`).
   *
   *  ##### Example
   *
   *      'borderBottomWidth'.underscore();
   *      // -> 'border_bottom_width'
   *
   *  ##### Note
   *
   *  Used in conjunction with [[String#dasherize]], [[String#underscore]]
   *  converts a DOM style into its CSS equivalent.
   *
   *      'borderBottomWidth'.underscore().dasherize();
   *      // -> 'border-bottom-width'
  **/
  function underscore() {
    return this.replace(/::/g, '/')
               .replace(/([A-Z]+)([A-Z][a-z])/g, '$1_$2')
               .replace(/([a-z\d])([A-Z])/g, '$1_$2')
               .replace(/-/g, '_')
               .toLowerCase();
  }

  /**
   *  String#dasherize() -> String
   *
   *  Replaces every instance of the underscore character `"_"` by a dash `"-"`.
   *
   *  ##### Example
   *
   *      'border_bottom_width'.dasherize();
   *      // -> 'border-bottom-width'
   *
   *  ##### Note
   *
   *  Used in conjunction with [[String#underscore]], [[String#dasherize]]
   *  converts a DOM style into its CSS equivalent.
   *
   *      'borderBottomWidth'.underscore().dasherize();
   *      // -> 'border-bottom-width'
  **/
  function dasherize() {
    return this.replace(/_/g, '-');
  }

  /** related to: Object.inspect
   *  String#inspect([useDoubleQuotes = false]) -> String
   *
   *  Returns a debug-oriented version of the string (i.e. wrapped in single or
   *  double quotes, with backslashes and quotes escaped).
   *
   *  For more information on `inspect` methods, see [[Object.inspect]].
   *
   *  #### Examples
   *
   *      'I\'m so happy.'.inspect();
   *      // -> '\'I\\\'m so happy.\''
   *      // (displayed as 'I\'m so happy.' in an alert dialog or the console)
   *
   *      'I\'m so happy.'.inspect(true);
   *      // -> '"I'm so happy."'
   *      // (displayed as "I'm so happy." in an alert dialog or the console)
  **/
  function inspect(useDoubleQuotes) {
    var escapedString = this.replace(/[\x00-\x1f\\]/g, function(character) {
      if (character in String.specialChar) {
        return String.specialChar[character];
      }
      return '\\u00' + character.charCodeAt().toPaddedString(2, 16);
    });
    if (useDoubleQuotes) return '"' + escapedString.replace(/"/g, '\\"') + '"';
    return "'" + escapedString.replace(/'/g, '\\\'') + "'";
  }

  /**
   *  String#unfilterJSON([filter = Prototype.JSONFilter]) -> String
   *
   *  Strips comment delimiters around Ajax JSON or JavaScript responses.
   *  This security method is called internally.
   *
   *  ##### Example
   *
   *      '/*-secure-\n{"name": "Violet", "occupation": "character", "age": 25}\n*\/'.unfilterJSON()
   *      // -> '{"name": "Violet", "occupation": "character", "age": 25}'
  **/
  function unfilterJSON(filter) {
    return this.replace(filter || Prototype.JSONFilter, '$1');
  }

  /**
   *  String#isJSON() -> Boolean
   *
   *  Check if the string is valid JSON by the use of regular expressions.
   *  This security method is called internally.
   *
   *  ##### Examples
   *
   *      "something".isJSON();
   *      // -> false
   *      "\"something\"".isJSON();
   *      // -> true
   *      "{ foo: 42 }".isJSON();
   *      // -> false
   *      "{ \"foo\": 42 }".isJSON();
   *      // -> true
  **/
  function isJSON() {
    var str = this;
    if (str.blank()) return false;
    str = str.replace(/\\(?:["\\\/bfnrt]|u[0-9a-fA-F]{4})/g, '@');
    str = str.replace(/"[^"\\\n\r]*"|true|false|null|-?\d+(?:\.\d*)?(?:[eE][+\-]?\d+)?/g, ']');
    str = str.replace(/(?:^|:|,)(?:\s*\[)+/g, '');
    return (/^[\],:{}\s]*$/).test(str);
  }

  /**
   *  String#evalJSON([sanitize = false]) -> object
   *
   *  Evaluates the JSON in the string and returns the resulting object.
   *
   *  If the optional `sanitize` parameter is set to `true`, the string is
   *  checked for possible malicious attempts; if one is detected, `eval`
   *  is _not called_.
   *
   *  ##### Warning
   *
   *  If the JSON string is not well formated or if a malicious attempt is
   *  detected a `SyntaxError` is thrown.
   *
   *  ##### Examples
   *
   *      var person = '{ "name": "Violet", "occupation": "character" }'.evalJSON();
   *      person.name;
   *      //-> "Violet"
   *
   *      person = 'grabUserPassword()'.evalJSON(true);
   *      //-> SyntaxError: Badly formed JSON string: 'grabUserPassword()'
   *
   *      person = '/*-secure-\n{"name": "Violet", "occupation": "character"}\n*\/'.evalJSON()
   *      person.name;
   *      //-> "Violet"
   *
   *  ##### Note
   *
   *  Always set the `sanitize` parameter to `true` for data coming from
   *  externals sources to prevent XSS attacks.
   *
   *  As [[String#evalJSON]] internally calls [[String#unfilterJSON]], optional
   *  security comment delimiters (defined in [[Prototype.JSONFilter]]) are
   *  automatically removed.
  **/
  function evalJSON() {
    var json = this.unfilterJSON();
    return JSON.parse(json);
  }

  /**
   *  String#include(substring) -> Boolean
   *
   *  Checks if the string contains `substring`.
   *
   *  ##### Example
   *
   *      'Prototype framework'.include('frame');
   *      //-> true
   *      'Prototype framework'.include('frameset');
   *      //-> false
  **/
  function include(pattern) {
    return this.indexOf(pattern) > -1;
  }

  /**
   *  String#startsWith(substring[, position]) -> Boolean
   *  - substring (String): The characters to be searched for at the start of
   *    this string.
   *  - [position] (Number): The position in this string at which to begin
   *    searching for `substring`; defaults to 0.
   *
   *  Checks if the string starts with `substring`.
   *
   *  `String#startsWith` acts as an ECMAScript 6 [polyfill](http://remysharp.com/2010/10/08/what-is-a-polyfill/).
   *  It is only defined if not already present in the user's browser, and it
   *  is meant to behave like the native version as much as possible. Consult
   *  the [ES6 specification](http://wiki.ecmascript.org/doku.php?id=harmony%3Aspecification_drafts) for more
   *  information.
   *
   *  ##### Example
   *
   *      'Prototype JavaScript'.startsWith('Pro');
   *      //-> true
   *      'Prototype JavaScript'.startsWith('Java', 10);
   *      //-> true
  **/
  function startsWith(pattern, position) {
    position = Object.isNumber(position) ? position : 0;
    // We use `lastIndexOf` instead of `indexOf` to avoid tying execution
    // time to string length when string doesn't start with pattern.
    return this.lastIndexOf(pattern, position) === position;
  }

  /**
   *  String#endsWith(substring[, position]) -> Boolean
   *  - substring (String): The characters to be searched for at the end of
   *    this string.
   *  - [position] (Number): Search within this string as if this string were
   *    only this long; defaults to this string's actual length, clamped
   *    within the range established by this string's length.
   *
   *  Checks if the string ends with `substring`.
   *
   *  `String#endsWith` acts as an ECMAScript 6 [polyfill](http://remysharp.com/2010/10/08/what-is-a-polyfill/).
   *  It is only defined if not already present in the user's browser, and it
   *  is meant to behave like the native version as much as possible. Consult
   *  the [ES6 specification](http://wiki.ecmascript.org/doku.php?id=harmony%3Aspecification_drafts) for more
   *  information.
   *
   *  ##### Example
   *
   *      'slaughter'.endsWith('laughter')
   *      // -> true
   *      'slaughter'.endsWith('laugh', 6)
   *      // -> true
  **/
  function endsWith(pattern, position) {
    pattern = String(pattern);
    position = Object.isNumber(position) ? position : this.length;
    if (position < 0) position = 0;
    if (position > this.length) position = this.length;
    var d = position - pattern.length;
    // We use `indexOf` instead of `lastIndexOf` to avoid tying execution
    // time to string length when string doesn't end with pattern.
    return d >= 0 && this.indexOf(pattern, d) === d;
  }

  /**
   *  String#empty() -> Boolean
   *
   *  Checks if the string is empty.
   *
   *  ##### Example
   *
   *      ''.empty();
   *      //-> true
   *
   *      '  '.empty();
   *      //-> false
  **/
  function empty() {
    return this == '';
  }

  /**
   *  String#blank() -> Boolean
   *
   *  Check if the string is "blank" &mdash; either empty (length of `0`) or
   *  containing only whitespace.
   *
   *  ##### Example
   *
   *      ''.blank();
   *      //-> true
   *
   *      '  '.blank();
   *      //-> true
   *
   *      ' a '.blank();
   *      //-> false
  **/
  function blank() {
    return /^\s*$/.test(this);
  }

  /**
   *  String#interpolate(object[, pattern]) -> String
   *
   *  Treats the string as a [[Template]] and fills it with `object`'s
   *  properties.
  **/
  function interpolate(object, pattern) {
    return new Template(this, pattern).evaluate(object);
  }

  return {
    gsub:           gsub,
    sub:            sub,
    scan:           scan,
    truncate:       truncate,
    // Firefox 3.5+ supports String.prototype.trim
    // (`trim` is ~ 5x faster than `strip` in FF3.5)
    strip:          String.prototype.trim || strip,
    stripTags:      stripTags,
    stripScripts:   stripScripts,
    extractScripts: extractScripts,
    evalScripts:    evalScripts,
    escapeHTML:     escapeHTML,
    unescapeHTML:   unescapeHTML,
    toQueryParams:  toQueryParams,
    parseQuery:     toQueryParams,
    toArray:        toArray,
    succ:           succ,
    times:          times,
    camelize:       camelize,
    capitalize:     capitalize,
    underscore:     underscore,
    dasherize:      dasherize,
    inspect:        inspect,
    unfilterJSON:   unfilterJSON,
    isJSON:         isJSON,
    evalJSON:       evalJSON,
    include:        include,
    // Firefox 18+ supports String.prototype.startsWith, String.prototype.endsWith
    startsWith:     String.prototype.startsWith || startsWith,
    endsWith:       String.prototype.endsWith || endsWith,
    empty:          empty,
    blank:          blank,
    interpolate:    interpolate
  };
})());
```