Strings and Regular Expressions

Relevant Links

- Flanagan's book, sections 3.2, 10.1, 10.2, (10.3 optional)
- MDN's guide on RegExps[1]
- MDN's RegExp reference[2]
- MDN's String object reference[3]

Strings in Javascript

- String literals are formed by surrounding the text in single or double quotes: "a string", 'another string'.
- Escape characters are combinations that hold special meaning. Refer to table 3.1 in the book, or in MDN's String reference.
- The plus operator is overloaded to cause string concatenation. Other values will be coersed to strings if necessary.
- Strings can be accessed via array indexing location: "hi there"[4] == "h", "hello".length == 5. There is no separate type for single characters.
- Strings are immutable: You cannot change their value, you can only create a new string.

String methods

charAt[4] An alternative for accessing a specific location in the string
charCodeAt[5] Returns the numeric Unicode value of the character at a specific location.
concat[6] Concatenates strings
indexOf[7] Search within the string for the first occurence of a substring.
lastIndexOf[8] Similar to indexOf, but finds the last occurence instead.
split[9] Split a string into an array of strings based on a separator parameter.
substr[10] Extract a section of a string
substring[11] Very similar to substr, but slightly different arguments
toLowerCase[12] Converts to lower case
toUpperCase[13] Converts to upper case

Regular Expressions in Javascript

- Regular expressions are patterns that allow us to express very intelligent search patterns in strings.

[1] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_Expressions
[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/RegExp
[3] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String

- The full language of regular expressions is quite rich and will take some time to fully digest. We will only scratch the surface here.
- A regular expression literal is marked by two forward slashes: `/stuffHere/`. The second slash may optionally be followed by the letters "i", "g", and/or "m", which affect how the expression behaves.

  i the regular expression ignores case, so an "a" would much both lower case and upper case A's.
  g the regular expression becomes "global", so it will keep on searching for more matches after the first one.
  m the regular expression possibly matches across multiple lines.

- Between the two forward slashes, a number of elements can be present, each with its meaning. Here are the most important ones:

  c Any character except for a few special that we will mention furhter down matches itself. This includes numbers, letters, some punctuation etc.
  \t Matches a tab character.
  \n Matches a newline character.
  \r Matches a carriage return. Often appears before the newline character to denote end of line.
  . A dot matches any character at all, except for newlines. Precede the dot with a backslash if you want to match a literal dot.
  \d Matches a digit.
  \D Matches any non-digit.
  \w Matches an alphanumeric character or underscore.
  \W Matches any character that is not alphanumeric or underscore.
  \s Matches a whitespace character (spaces, tabs, newlines and a couple more).
  \S Matches a non-whitespace character.
  **** Matches an actual backslash.
  \c For a character that has a special meaning by itself, escape it with a backslash if you want its literal meaning. e.g. `\\.`, `\\?`, `\\*`, `\\[`.
  [xyz] Matches any of the characters inside the brackets. You can also indicate ranges, like `[a-z.0-9]`, which matches any lowercase character or digit or dot.
  [^xyz] Matches anything BUT what follows the caret.
  ^ Doesn't match an actual character, but it must match the beginning of a line.
  $ Doesn't match an actual character, but it must match the end of a line.
  \b Doesn't match an actual character, but it must match a word boundary. Useful for matching whole words. For instance `/\\bcat\\b/` will match in "12cat" but not in "cats".
  (...) Treats the contents as a group. In many regular expression searches, you would be able to access the matched pieces. Also called *capturing parentheses*.
  \n where n is a number. Matches a previously matched group.

(?:...) *Non-capturing parentheses.* Useful when you simply want to group some elements together, but do not care for capturing the result separately.

x* Matches the expression x zero or more times.

x+ Matches the expression one or more times.

x? Matches the expression zero or one time.

x*? Matches zero or more times, but in a "non-greedy" fashion, meaning it will try to match the smallest possible number of times.

x|y Matches either x or y.

- Some examples would be in order:

```
/([\w\-]+\d+)@hanover\.edu/
```

This will match all student emails at Hanover (faculty/staff emails don't end in digits). It looks for a word character or dash, one or more times, followed by one or more digits, then a literal at sign, "hanover", then a literal dot, then "edu".

```
/(.)(.)(.)\3\2\1/
```

This regular expression will match palindromes with total length 6, e.g. something like "abccba". The parenthesized dots match one arbitrary element each, then the backticked numbers refer to those matches in reverse order.

```
/^\s*[+-]?(?:\d+\.?|\.\d+)\d*(?:[eE][+-]?\d+)?\s*$/
```

This is a fairly complicated regular expression that matches numbers in decimal or scientific format. It starts with a caret, and ends in a dollar sign, meaning that the contents much match the entire line/string. It allows for an arbitrary number of whitespace characters on either end. After that, it has an optional sign, followed by: either at least one digit followed by an optional dot, or a dot following by at least one digit, these form a non-capturing group, and they are followed by zero or more digits. Finally, there is another optional non-capturing group, that matches an optional exponent. It consists of an uppercase or lowercase e, followed by an optional sign, followed by at least one digit.

```
/"((?:\\\\|\\"|[^"])*)"/
```

This is a complicated regular expression used to capture a "double-quoted string" within a string. If you read in a data file or some javascript code, you may find in it some quoted strings. The quoted string ends when we find the next unescaped quote. A quote can be escaped by using a backslash, unless that backslash itself was escaped by another backslash. That is why the above pattern has 3 parts separated by vertical lines to indicate any of the 3 alternatives at any given time. The first part matches two backslashes back to back, the next matches a backslash followed by a quote, the third part matches any character except for a quote. Make sure you think about this and understand why it is necessary to do it this way.

Using regular expressions

- If `s` is a string and `r` is a regular expression, there are a number of different ways we can try to "apply" the regular expression to the string to see if it matches:

Returns a result array or null.
Returns a boolean indicating whether there was a match or not.
Returns an array of the matched results, or null.
Returns the index of the first successful match, or -1 if there is no match.
Replaces a match with s2. Look at the documentation for details on this second argument.
Use the matches of r as separators to split the string at.

r.exec(s) **r.test(s)** **s.match(r)** **s.search(r)** **s.replace(r, s2)** **s.split(r)** methods operate on one item at a time, unless the "g" flag has been added to the regular expression.

- The array returned by these methods varies by method, and often contains extra properties. Consult the documentation for details.
- When working with regular expressions, it is always a good idea to create a list of test strings first, then see how the regular expression behaves on those strings.