JavaScript Regular Expressions

CSC 310 - Programming Languages

JavaScript Regular Expressions

- A regular expression (also called a regex or regexp) specifies a pattern for searching in text.
- JavaScript regular expressions can be creating in two ways:
  - `regexp = new RegExp("pattern", "flags")`
  - `regexp = /pattern/gmi;  // flags are optional`
- The \ is the escape character for all special characters within a regexp, so to match / within a regexp we need to use \/

RegExp Flags

- `i`: the search is case-insensitive
- `g`: search for all matches (not only the first match)
- `m`: multiline mode
- `s`: enable "dotall" mode which allows dot (.) to match the newline character
- `u`: enable Unicode support
- `y`: enable "sticky" mode which searches the exact position in the text

String Methods and RegExp

- `str.match(regexp)`: finds all matches of `regexp` in the string `str`
  - the `g` flag returns an array of all matches instead of the first match
  - if there is no match, then `null` is returned
- `str.replace(regexp, replacement)`: replaces `regexp` matches with `replacement` in `str`
  - can specify replacement options in the replacement string using a `$` prefix

RegExp String Method Examples

- **`str.match(regexp)`**:

```
> "AA bb aa cc".match(/aa/gi);
[ 'AA', 'aa' ]
```

- **`str.replace(regexp, pattern)`**:

```
> "AA bb aa cc".replace(/aa/gi, "d");
'd bb d cc'
```

Character Classes

- A character class is notation to specify matches from a set delimited by square brackets, for example

```
> "cat hat mat".match(/[chm]at/g);
['cat', 'hat', 'mat']
```

- A character class can contain character ranges, for example
  - `/[a-z]/`: matches any lower case letter
  - `/[1-5]/`: matches any digit from 1 to 5
- The caret character in the beginning of a character class denotes complement of the class, for example
  - `/[^abc]/`: match any character except `a`, `b`, or `c`.

Special Character Classes

- `\d`: digit from 0 to 9
- `\s`: (white)space (including tabs, newlines, etc.)
- `\w`: word containing alphabetic characters, digits, or underscore
- `\D`: non-digit
- `\S`: non-space
- `\W`: non-word
- `.`: any character except newline

Additional Matches

- `^`: match the beginning of the text
- `$`: match the end of the text
- `\b`: match a word boundary
  - At the beginning of a string, if the first string character is a word character `\w`
  - Between two characters in the string, where one is a word character `\w` and the other is not
  - At the end of the string, if the last string character is a word character `\w`

Multiline Mode

- The `m` flag enables multiline mode which changes the behavior of `^` and `$`
  - `^`: match the beginning of a line
  - `$`: match the end of a line

Repeating Patterns (Quantifiers)

- `+`: match the pattern before the `+` one or more times
- `*`: match the pattern before the `*` zero or more times
- `?`: match the pattern before the `?` zero or one times
- `{n}`: match the pattern before the `{n}` exactly `n` times
- `{n,m}`: match the pattern before the `{n,m}` at least `n` times and at most `m` times; the `m` can be omitted which means match `n` or more times

Choice Patterns

- The vertical bar (|) denotes a choice (also called alternation) between the pattern on the left and the pattern on the right
- Example:

```
> "the cat and the dog".match(/dog|cat/g);
[ 'cat', 'dog' ]
```

Capturing Groups

- A fragment of a pattern can be enclosed in parentheses which is called a capturing group; this does two things:
  - capture the match as a separate item in the result array
  - apply quantifiers to groups
- Examples:

```
> "Hahaha".match(/(ha)+/g);
[ 'haha' ]
```

Capturing Group Content

- The capturing groups are numbered from left to right
- A `str.match` with no g flag returns an array where
  - index 0: full match
  - index 1: content of first group
  - index 2: content of second group
  - ...
- Example

```
let p  = /\((\d{3})\)-(\d{3})-(\d{4}))/;
let phone_num = "(123)-456-7890".match(p);
let area_code = phone_num[1];
```

Nested Groups

- Capturing groups can be nested; the numbering of the groups is still left to right
- Example:

```
> let h  = "Hello, World!".match(/((Hello), (World))!/)
undefined
> h[1]
'Hello, World'
> h[2]
'Hello'
> h[3]
'World'
```

Optional Groups

- Groups that are optional and do not exist in the match still have a corresponding element in the array; this element is set to undefined
- Example:

```
> let m = "ac".match(/a(b)?(c)?/);
> m[1]
undefined
> m[2]
'c'
```

Searching for All Matches with Groups

- The `str.match` method does not return the contents of groups when the `g` flag is set:

```
> '<h1> <h2>`.match(/<(.*?)>/g);
[ '<h1>', '<h2>' ]
```

- The `str.matchAll` method returns an iterable object; this object returns every match as an array with groups when the `g` flag is specified:

```
> let results = '<h1> <h2>'.matchAll(/<(.*?)>/g);
> results = Array.from(results); // iterator to array
> results[0][1]
'h1'
> results[1][1]
'h2'
```

Named Groups

- A group can be referred to by name by prefixing the group with ?<name> where name is the is the name of the reference
- Example:

```
> let m = "abc".match(/(?<first>\w)/)
> m.groups.first
'a'
```

Excluding Groups

- A group may be excluded by adding `?:` to the beginning of the group
- Example:

```
> let m = "abc".match(/(a)(?:b)(c)/)
> m[1]
> m[1]
'a'
> m[2]
'c'
```

Capturing Groups and Replacement

- The `str.replace(regexp, replacement)` can refer to capture group content in the `replacement` string by index or name.
- Example by index:

```
> let name = "Jane Doe";
> name.replace(/(\w+) (\w+)/, "$2, $1");
'Doe, Jane'
```

- Example by name:

```
> let name = "Jane Doe";
> name.replace(/(?<F>\w+) (?<L>\w+)/, "$<L>, $<F>")
'Doe, Jane'
```

Replacement String Options

- The replacement string argument of the `str.repace(regexp, replacement)` method can include additional options related to the match:
  - `$$`: insert the `$`
  - `$&`: insert the matched substring
  - `` $` ``: insert the portion of the string that precedes the matched substring
  - `$'`: insert the portion of the string that follows the matched substring

Backreferences

- The content of capturing groups can be used in the pattern itself.
- A group can be referenced by number using `\N` where `N` is the group number or by name with `\k<name>` where `name` is the name of the group.
- Example:

```
>let str = `string with "quoted" 'words'`;
> str.match(/(['"])(.*?)\1/g);
[ '"quoted"', "'words'" ]
```

Lookahead and Lookbehind

- The syntax `X(?=Y)` means "look for `X` followed by `Y`
- The syntax `X(?!Y)` means "look for `X` not followed by `Y`
- The syntax `X(?<=Y)` means "look for `X` after `Y`
- The syntax `X(?<!Y)` means "look for `X` not after `Y`

Regular Expressions

Some people, when confronted with a problem, think "I know, I'll use regular expressions." Now they have two problems.
– Jamie Zawinski