JavaScript Regular Expressions and Exception Handling

Prof . N. Nalini
AP(Sr)
SCOPE
VIT
VELLORE

What is a regular expression?

```
/[a-zA-Z_\-]+@(([a-zA-Z_\-])+\.)+[a-zA-Z]{2,4}/
```

- **regular expression** ("regex"): describes a pattern of text
  - can test whether a string matches the expr's pattern
  - can use a regex to search/replace characters in a string
  - very powerful, but tough to read

- regular expressions occur in many places:
  - text editors (TextPad) allow regexes in search/replace
  - languages: JavaScript; Java `Scanner`, `String split`
  - Unix/Linux/Mac shell commands (`grep`, `sed`, `find`, etc.)

String regexp methods

`.match(`***`regexp`***`)`	returns first match for this string against the given regular expression; if global /g flag is used, returns array of all matches
`.replace(`***`regexp, text`***`)`	replaces first occurrence of the regular expression with the given text; if global /g flag is used, replaces all occurrences
`.search(`***`regexp`***`)`	returns first index where the given regular expression occurs
`.split(`***`delimiter[,limit]`***`)`	breaks apart a string into an array of strings using the given regular as the delimiter; returns the array of tokens

Regex flags / modifiers

`/pattern/g`	global; match/replace all occurrences
`/pattern/i`	case-insensitive
`/pattern/m`	multi-line mode
`/pattern/y`	"sticky" search, starts from a given index

- flags can be combined:

  `/abc/gi` matches *all* occurrences of abc, AbC, aBc, ABC, ...

Basic regexes

```
/abc/
```

- a regular expression literal in JS is written `/pattern/`
- the simplest regexes simply match a given substring

- the above regex matches any line containing "abc"
  - *YES*: `"abc"`, `"abcdef"`, `"defabc"`, `".=.abc.=."`
  - *NO*: `"fedcba"`, `"ab c"`, `"AbC"`, `"Bash"`, …

String match

***string*.match(*regex*)**

- if string fits pattern, returns matching text; else `null`
  - can be used as a Boolean truthy/falsey test:

```
if (name.match(/[a-z]+/)) { ... }
```

- **`g`** after regex for array of *global* matches
  - `"obama".match(/.a/g)` returns `["ba", "ma"]`

- **`i`** after regex for case-*insensitive* match
  - `name.match(/Mary/i)` matches `"mary"`, `"MaRY"`

String replace

string`.replace(`***regex***`, "`***text***`")`

- replaces *first occurrence* of pattern with the given text
  - `var state = "Mississippi";`
    `state.replace(/s/, "x")` returns `"Mixsissippi"`
- **`g`** after regex to replace *all occurrences*
  - `state.replace(/s/`**`g`**`, "x")` returns `"Mixxixxippi"`
- *returns* the modified string as its result; must be stored
  - **`state = state`**`.replace(/s/g, "x");`

Special characters

| means OR

- `/abc|def|g/` matches lines with `"abc"`, `"def"`, or `"g"`
- precedence: `^Subject|Date:` vs. `^(Subject|Date):`
- There's no AND & symbol.

`()` are for grouping

- `/(Homer|Marge) Simpson/` matches lines containing `"Homer Simpson"` or `"Marge Simpson"`

`\` starts an escape sequence

- many characters must be escaped: `/\$.[](^*+?`
- `"\.\\n"` matches lines containing `".\n"`

Wildcards and anchors

- `.` (a dot) matches any character except `\n`
  - `/.oo.y/` matches `"Doocy"`, `"goofy"`, `"LooPy"`, …
  - use `\.` to literally match a dot `.` character

`^` matches the beginning of a line; `$` the end

- `/^if$/` matches lines that consist entirely of `if`

`\<` demands that pattern is the beginning of a *word*;
`\>` demands that pattern is the end of a word

- `/\<for\>/` matches lines that contain the word `"for"`

Quantifiers

Quantifier	Description
n+	Matches any string that contains at least one *n*
n*	Matches any string that contains zero or more occurrences of *n*
n?	Matches any string that contains zero or one occurrences of *n*
n{X}	Matches any string that contains a sequence of *X* *n*'s
n{X,Y}	Matches any string that contains a sequence of X to Y *n*'s
n{X,}	Matches any string that contains a sequence of at least X *n*'s
n$	Matches any string with *n* at the end of it
^n	Matches any string with *n* at the beginning of it
?=n	Matches any string that is followed by a specific string *n*
?!n	Matches any string that is not followed by a specific string *n*

Quantifiers: * + ?

***** means 0 or more occurrences

- `/abc*/` matches "ab", "abc", "abcc", "abccc", ...
- `/a(bc)/`" matches "a", "abc", "abcbc", "abcbcbc", ...
- `/a.*a/` matches "aa", "aba", "a8qa", "a!?_a", ...

+ means 1 or more occurrences

- `/a(bc)+/` matches "abc", "abcbc", "abcbcbc", ...
- `/Goo+gle/` matches "Google", "Gooogle", "Goooogle", ...

? means 0 or 1 occurrences

- `/Martina?/` matches lines with "Martin" or "Martina"
- `/Dan(iel)?/` matches lines with "Dan" or "Daniel"

More quantifiers

`{min,max}` means between ***min*** and ***max*** occurrences

- `/a(bc){2,4}/` matches lines that contain "abcbc", "abcbcbc", or "abcbcbcbc"

- ***min*** or ***max*** may be omitted to specify any number
  - `{2,}` 2 or more
  - `{,6}` up to 6
  - `{3}` exactly 3

Character sets

`[ ]` group characters into a *character set*;
will match any single character from the set

- `/[bcd]art/` matches lines with "bart", "cart", and "dart"
- equivalent to `/(b|c|d)art/` but shorter

- inside `[ ]`, most modifier keys act as normal characters
  - `/what[.!*?]*/` matches "what", "what.", "what!", "what?**!", …

  – *Exercise* : Match letter grades e.g. A+, B-, D.

Character ranges

- inside a character set, specify a range of chars with `-`
  - `/[a-z]/` matches any lowercase letter
  - `/[a-zA-Z0-9]/` matches any letter or digit

- an initial `^` inside a character set negates it
  - `/[^abcd]/` matches any character but not a, b, c, or d

- inside a character set, `-` must be escaped to be matched
  - `/[\-+]?[0-9]+/` matches optional - or +, followed by at least one digit
    - *Exercise* : Match phone numbers, e.g. 206-685-2181 .

Built-in character ranges

- `\b` word boundary (e.g. spaces between words)
- `\B` non-word boundary
- `\d` any digit; equivalent to `[0-9]`
- `\D` any non-digit; equivalent to `[^0-9]`
- `\s` any whitespace character; `[ \f\n\r\t\v...]`
- `\S` any non-whitespace character
- `\w` any word character; `[A-Za-z0-9_]`
- `\W` any non-word character
- `\xhh, \uhhhh` the given hex/Unicode character

  - `/\w+\s+\w+/` matches two space-separated words

```
<!DOCTYPE html>
<html>
<body>

<p>Click the button to do a global search for "h.t" in a string.</p>

<button onclick="myFunction()">Try it</button>

<p id="demo"></p>

<script>
function myFunction() {
  var str = "That's hot!";
  var patt1 = /h.t/g;
  var result = str.match(patt1);
  document.getElementById("demo").innerHTML = result;
}
</script>

</body>
</html>
```

```
<script>
function myFunction() {
  var str = "Visit W3Schools.\rLearn JavaScript.";
  var patt1 = /\r/;
  var result = str.search(patt1);
  document.getElementById("demo").innerHTML = result;
}
</script>
```

```
<script>
function myFunction() {
  var str = "\nis th\nis is it?";
  var patt1 = /is/ig;
  var result = str.match(patt1);
  document.getElementById("demo").innerHTML = result;
}
</script>
```

```
<script>
function myFunction() {
  var str = "\nIs th\nis it?";
  var patt1 = /^is/igm;
  var result = str.match(patt1);
  document.getElementById("demo").innerHTML = result;
}
</script>
```

Back-references

- text "captured" in `()` is given an internal number; use `\number` to refer to it elsewhere in the pattern
  - `\0` is the overall pattern,
  - `\1` is the first parenthetical capture, `\2` the second, ...
  - Example: "A" surrounded by same character: `/(.)A\1/`

  - variations
    - `(?:text)` match ***text*** but don't capture
    - ***a***`(?=`***b***`)` capture pattern ***b*** but only if preceded by ***a***
    - ***a***`(?!`***b***`)` capture pattern ***b*** but only if not preceded by ***a***

Replacing with back-references

- you can use back-references when replacing text:
  - refer to captures as ***$number*** in the replacement string
  - Example: to swap a last name with a first name:

```
var name = "Durden,    Tyler";
name = name.replace(/(\w+),\s+(\w+)/, "$2 $1");
// "Tyler Durden"
```

  - *Exercise* : Reformat phone numbers from 206-685-2181 format to (206) 685.2181 format.

The RegExp object

```
new RegExp(string)
new RegExp(string, flags)
```

- constructs a regex dynamically based on a given string

```
var r = /ab+c/gi;                    is equivalent to
var r = new RegExp("ab+c", "gi");
```

  - useful when you don't know regex's pattern until runtime
    - Example: Prompt user for his/her name, then search for it.
    - Example: The empty regex (think about it).

exec() and test()

- exec() - Tests for a match in a string. Returns the first match

Example

Search a string for the character "e":

```
var str = "The best things in life are free";
var patt = new RegExp("e");
var res = patt.exec(str);
```

- Test()-Tests for a match in a string. Returns true or false

Example

Search a string for the character "e":

```
var str = "The best things in life are free";
var patt = new RegExp("e");
var res = patt.test(str);
```

Working with RegExp

- in a regex literal, forward slashes must be \ escaped:

```
/http[s]?:\/\/\w+\.com/
```

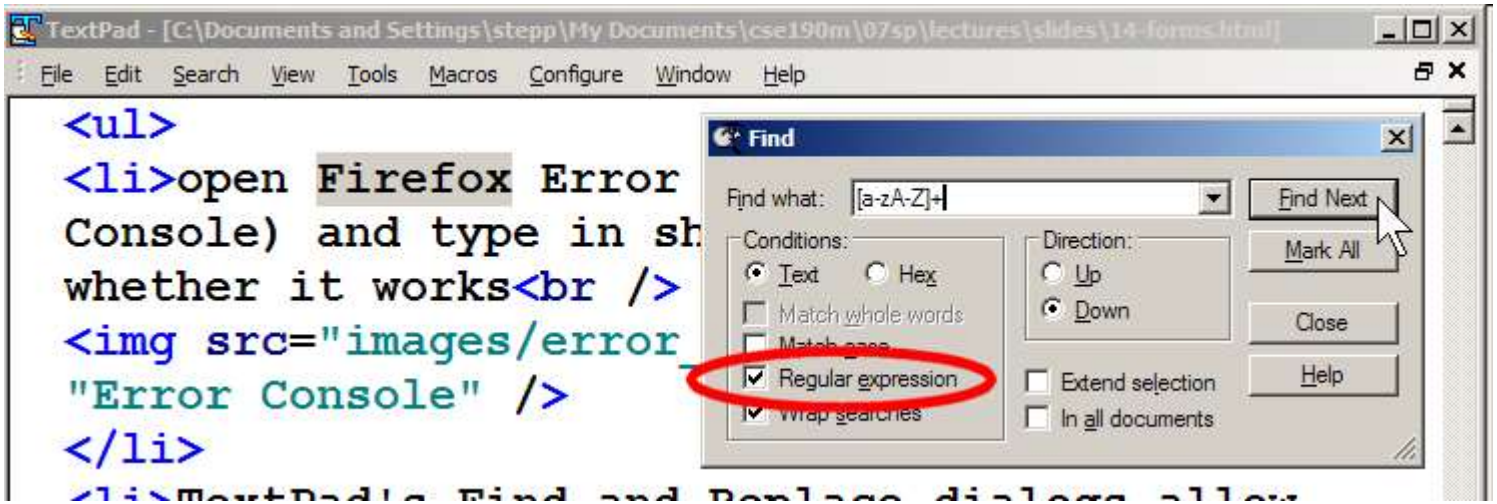
- in a `new RegExp` object, the pattern is a string, so the usual escapes are necessary (quotes, backslashes, etc.):

```
new RegExp("http[s]?://\\w+\\.com")
```

- a `RegExp` object has various properties/methods:
  - properties: `global`, `ignoreCase`, `lastIndex`, `multiline`, `source`, `sticky`; methods: `exec`, `test`

Regexes in editors and tools

- Many editors allow regexes in their Find/Replace feature



- many command-line Linux/Mac tools support regexes

```
grep -e "[pP]hone.*206[0-9]{7}" contacts.txt
```

JavaScript Exception

- When an error occurs, JavaScript will normally stop and generate an error message.
- The try statement allow to define a block of code to be tested for errors while it is being executed.
- The catch statement allows to define a block of code to be executed, if an error occurs in the try block.
- The JavaScript statements try and catch come in pairs:

```
try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
```

JavaScript Error Types

- **The Six JavaScript Error Types**
  - EvalError: Raised when the eval() functions is used in an incorrect manner.
  - RangeError: Raised when a numeric variable exceeds its allowed range.
  - ReferenceError: Raised when an invalid reference is used.
  - SyntaxError: Raised when a syntax error occurs while parsing JavaScript code.
  - TypeError: Raised when the type of a variable is not as expected.
  - URIError: Raised when the encodeURI() or decodeURI() functions are used in an incorrect manner.
- JavaScript will actually create an Error object with two properties: name and message.
  - **Name : Error name.**
    - **For instance, for an undefined variable that's "ReferenceError".**
  - **Message : Textual message about error details.**

Error message and name

```
<p id="demo"></p>

<script>
try {
  adddlert("Welcome guest!");
}
catch(err) {
  document.getElementById("demo").innerHTML = err.message;
}
</script>
```

EX:1(adddlert is not defined)

EX:2

```
try {
        execute this block
} catch (error) {
        if (error.name === 'RangeError')
  {
        do this
  }
  else if (error.name === 'ReferenceError')
  {
        do this
  }
  else
  {
        do this for more generic errors
  }
}
```

EX:3 (Type Error)

```
var num = 1;
try {
  num.toUpperCase();   // You cannot convert a number to upper case
}
catch(err) {
  document.getElementById("demo").innerHTML = err.name;
}
```

JavaScript Exception

The throw Statement

- The throw statement allows you to create a custom error.
- Technically you can throw an exception (throw an error).
- The exception can be a JavaScript String, a Number, a Boolean or an Object:

```
throw "Too big";    // throw a text
throw 500;          // throw a number
```

```
<!DOCTYPE html>
<html>
<body>

<p>Please input a number between 5 and 10:</p>

<input id="demo" type="text">
<button type="button" onclick="myFunction()">Test Input</button>
<p id="p01"></p>

<script>
function myFunction() {
  var message, x;
  message = document.getElementById("p01");
  message.innerHTML = "";
  x = document.getElementById("demo").value;
  try {
    if(x == "")  throw "empty";
    if(isNaN(x)) throw "not a number";
    x = Number(x);
    if(x < 5)  throw "too low";
    if(x > 10)   throw "too high";
  }
  catch(err) {
    message.innerHTML = "Input is " + err;
  }
}
</script>

</body>
</html>
```

The finally Statement

The `finally` statement lets you execute code, after try and catch, regardless of the result:

Syntax

```
try {
  Block of code to try
}
catch(err) {
  Block of code to handle errors
}
finally {
  Block of code to be executed regardless of the try / catch result
}
```

Example

```
function myFunction() {
  var message, x;
  message = document.getElementById("p01");
  message.innerHTML = "";
  x = document.getElementById("demo").value;
  try {
    if(x == "") throw "is empty";
    if(isNaN(x)) throw "is not a number";
    x = Number(x);
    if(x > 10) throw "is too high";
    if(x < 5) throw "is too low";
  }
  catch(err) {
    message.innerHTML = "Error: " + err + ".";
  }
  finally {
    document.getElementById("demo").value = "";
  }
}
```