CSE 154: Web Programming

Exam 2 “Cheat Sheet”

Note that this is not a comprehensive cheat sheet for JS/SQL, but provides a quick reference for common functions, modules, syntax, etc. you may find helpful during a CSE 154 Exam 2.

JavaScript Language Reference (client- and server-side)

`string` Methods and Properties

Method/Property	Description
`length`	Returns the length of a string
`charAt(index)`	Returns the character at the specified index
`indexOf(str)`	Returns the position of the first occurrence of a specified value in a string (-1 if not found)
`split(delimiter)`	Splits a string into an array of substrings based on delimiter (e.g. “\n” for new lines).
`substring(start, end)`	Extracts the characters from a string between two specified indices (start is inclusive, end is exclusive)
`trim()`	Removes whitespace from both ends of a string
`toLowerCase()`	Returns a lowercase version of a string
`toUpperCase()`	Returns an uppercase version of a string

Array Methods and Properties

Method/Property	Description
`length`	Sets or returns the number of elements in an array
`push(el)`	Adds new elements to the end of an array and returns the new length
`pop()`	Removes and returns the last element of an array
`unshift(el)`	Adds new elements to the beginning of an array and returns the new length
`shift()`	Removes and returns the first element in an array
`sort()`	Sorts the elements of an array
`indexOf(el)`	Returns the index of the element in the array, or -1 if not found

JavaScript Methods Useful with JSON/Objects

```
{ key : value, key : value, … }
```

Function	Description
`JSON.parse(string)`	Returns the JSON object given a valid string representation.
`JSON.stringify(object)`	Returns a string representation of the given object.
`Object.keys(data)`	Returns an array of keys of the given object
`Object.values(data)`	Returns an array of values of the given object.

Other handy JavaScript Methods

Function	Description
`parseInt(value)`	Returns the integer representation of the given value, if it starts with a Number-like type Examples: `parseInt("12px")` evaluates to 12, `parseInt(10.5)` evaluates to 10
`console.log(data)`	Outputs the `data` to the JavaScript console

JavaScript Timer Functions

Method	Description
`setTimeout(fn, ms)`	Executes a function `fn` after a delay of `ms` milliseconds. Returns a Number value representing the ID of the timeout being set.
`setInterval(fn, ms)`	Executes a function `fn` at every given time-interval (in milliseconds). Returns a Number value representing the ID of the interval being set.
`clearTimeout(timerId)`	Stops the execution of the delay timer specified by timerId
`clearInterval(timerId)`	Stops the execution of the interval timer specified by timerId

JavaScript Math Functions

Method	Description
`Math.random()`	Returns a double between 0 (inclusive) and 1 (exclusive)
`Math.abs(n)`	Returns the absolute value of `n`
`Math.min(a, b, ...)`	Returns the smallest of 0 or more numbers
`Math.max(a, b, ...)`	Returns the largest of 0 or more numbers
`Math.round(n)`	Returns the value of `n` rounded to the nearest integer
`Math.ceil(n)`	Returns the smallest integer greater than or equal to `n`
`Math.floor(n)`	Returns the largest integer less than or equal to `n`

Client-Side JavaScript Reference

`document` Methods

Method/Property	Description
`getElementById(id)`	Returns a DOM object whose id property matches the specified string. If no matches are found, null is returned.
`querySelector(sel)`	Returns the first DOM element that matches the specified selector, or group of selectors. If no matches are found, null is returned.
`querySelectorAll(sel)`	Returns a list of the document's elements that match the specified group of selectors. If no matches are found, null is returned.
`createElement(tagName)`	Creates and returns an Element node with the given tag name

DOM Element Methods and Properties

Method/Property	Description
`el.id`	Sets or returns the value of the id attribute of an element
`el.getAttribute(attr)`	Returns the specified attribute value `attr` of `el`
`el.textContent`	Sets or returns the text content of the specified node
`el.innerHTML`	Sets or returns the HTML content of an element
`el.classList`	Returns the class name(s) of `el`
`el.className`	Sets or returns the value of the class attribute of `el`
`el.addEventListener(event, fn)`	Attaches an event handler function `fn` to the specified element `el` to listen to `event`
`el.removeEventListener(event, fn)`	Removes the event handler `fn` to the specified `el` listening to `event`
`el.children`	Returns a collection of the child elements of `el`
`el.parentNode`	Returns the parent node of `el`
`el.appendChild(child)`	Adds a new child node to `el` as the last child node
`el.insertBefore(newNode, refNode)`	Adds `newNode` to parent `el` before `el`'s child `refNode` position
`el.removeChild(child)`	Removes a child node from a parent element

Accessing DOM Element Attributes

Recall that if you have an HTML element on your page that has attributes, you can set those properties through JavaScript as well. For instance:

```
<img id="dog-tag" src="img/doggie.jpg" alt="My Cute Dog" />
```

Your could do the following in your JavaScript code (using the `id` alias for `document.getElementById`):

```
id("dogtag").alt = "My really cute dog";
```

Example DOM Element attributes include (other than `src`, and `alt` above) are:

Property	**Description**
`disabled`	Whether or not this DOM element is disabled on the page (boolean)
`value`	The current value of form elements (`input`, `textarea`, `checkbox` `radio`, `select`, etc.)
`name`	The value of the name attribute of a form element

DOM Element `.classList` Methods and Properties

Method/Property	**Description**
`add(class)`	Adds specified class values. These values are ignored if they already exist in the list
`remove(class)`	Removes the specified class value if it exists
`toggle(class)`	Toggles the listed class value. If the class exists, then removes it and returns false, otherwise adds it to the list and returns true
`contains(class)`	Returns true if the specified class value exists in the classList

Common Event Types

`load`	`mouseout`	`mouseup`	`keydown`	`change`
`click`	`mouseover`	`mouseenter`	`keyup`	`error`
`dblclick`	`mousedown`	`submit`	`select`	`success`

The Module Pattern

Whenever writing JavaScript, you should use the module pattern, wrapping the content of the code (`window load` event handler and other functions) in an anonymous function. Below is a template for reference:

```
"use strict";
(function() {

  // any module-globals (limit the use of these when possible)
  window.addEventListener("load", init);

  function init() {
    ...
  }
  // other functions
})();
```

Helper Alias Functions

You may use any of the following alias functions in your exam without defining them:

```
function id(idName) {
  return document.getElementById(idName);
}

function qs(selector) {
  return document.querySelector(selector);
}

function qsa(selector) {
  return document.querySelectorAll(selector);
}

function gen(tagName) {
   return document.createElement(tagName);
}
```

```
// you can assume the following checkStatus is defined in your file,
// you do not need to write this on your exam
function checkStatus(response) {
  if (!response.ok) {
    throw Error("Error in request: " + response.statusText)
  }
  return response;
}
```

Server-side JavaScript (Node.js) Reference

Basic Node.js Project Structure

```
example-node-project/
  .gitignore
  APIDOC.md
  app.js
  node_modules/
    ...
  data/
    ...
  package.json
  public/
    img/
      ...
    index.html
    index.js
    styles.css
```

Example Express app Template

```
"use strict";

/* File comment */

const express = require("express");
// other modules you use
// program constants

const app = express();
// if serving front-end files in public/
app.use(express.static(“public”));

// if handling different POST formats
app.use(express.urlencoded({ extended: true }));
app.use(express.json());
app.use(multer().none());

// app.get/app.post endpoints

// helper functions

const PORT = process.env.PORT || 8000;
app.listen(PORT);
```

Useful Core Modules

Module	Description
`fs`	The “file system” module with various functions to process data in the file system.
`path`	Provides functions to process path strings.
`util`	Provides various “utility” functions, such as util.promisify.

Other Useful Modules

The following modules must be installed for each new project using **`npm install <module-name>`**.

Module	Description
`express`	A module for simplifying the http-server core module in Node to implement APIs.
`glob`	Allows for quick traversal and filtering of files in a complex directory.
`multer`	Used to support FormData POST requests on the server-side so we can access the req.body parameters.
`mysql`	Provides functionality for interacting with a database and tables.
`promise-mysql`	Promisified wrapper over mysql module - each function in the mysql module returns a promise instead of taking a callback as the last argument (recommended).
`cookie-parser`	A module to access cookies with req/res objects in Express.

Express Route Functions

Function	Description

`app.get(“path”, middlewareFn(s));`
 	
`app.get(“/”, (req, res) => {`
`  ...`
`});`
 	
`app.get(“/:city”, (req, res) => {`
`  let city = req.params.city;`
`  ...`
`});`
 	
`app.get(“/cityData”, (req, res) => {`
`  let city = req.query.city;`
`  ...`
`});`
 	
`// Example with multiple middleware functions`
`app.get(“/”, validateInput, (req, res) => {`
`  ...`
`}, handleErr);`
	Defines a server endpoint which accepts a valid **`GET`** request. Request and response objects are passed as **`req`** and **`res`** respectively. Path parameters can be specified in **`path`** with “:varname” and accessed via **`req.params`**. Query parameters can be accessed via **`req.query`**.

`app.post(“path”, middlewareFn(s));`
 	
`app.post(“/addItem”, (req, res) => {`
`  let itemName = req.body.name;`
`  ...`
`}`
	Defines a server endpoint which accepts a valid **`POST`** request. Request and response objects are passed as **`req`** and **`res`** respectively. POST body is accessible via **`req.body`**. Requires POST middleware and multer module for FormData POST requests.

Request Object Properties/Functions

Property/Function	Description
`req.params`	An object containing properties mapped to the named route "parameters". For example, if you have the route /user/:name, then the "name" property is available as req.params.name.
`req.query`	An object containing a property for each query string parameter in the request URL, depending on a *`?key1=value1&key2=value2& ...`* pattern. If there is no query string, is { }..
`req.body`	Holds a dictionary of POST parameters as key/value pairs. Requires multer module for multipart form requests (e.g. FormData) and using middleware functions (see Express template) for other POST request types.
`req.cookies`	Retrieves all cookies sent in the request. Requires cookie-parser module.

Response Object Properties/Functions

Property/Function	Description

`res.set(headerName, value);`
 	
`res.set("Content-Type", "text/plain");`
`res.set("Content-Type", "application/json");`
	Used to set different response headers - commonly the "Content-type" (though there are others we don't cover).

`res.type("text");`
`res.type("json");`
	Shorthand for setting the "Content-Type" header.

`res.send(data);`
 	
`res.send("Hello");`
`res.send({ "msg" : "Hello" });`
	Sends the data back to the client, signaling an end to the response (does not terminate your JS program).

`res.end();`	Ends the request/response cycle without any data (does not terminate your JS program).
`res.json(data);`	Shorthand for setting the content type to JSON and sending JSON.

`res.status(statusCode)`
 	
`res.status(400).send("client-side error message");`
`res.status(500).send("server-side error message");`
	Specifies the HTTP status code of the response to communicate success/failure to a client.

Useful fs Module Functions

Note: You will often see the following "promisified" for use with async/await. The promisified versions will return a Promise that resolves callback's contents or rejects if there was an error. Remember that you should always handle potential fs function errors (try/catch if async/await, if/else if standard callback).

Example of promisifying callback-version of `fs.readFile`:

```
fs.readFile("file.txt", "utf8", (err, contents) => {
  ...
});

// Promisified:
const util = require("util");
const readFile = util.promisify(fs.readFile);

async function example() {
  try {
    let contents = await readFile("file.txt");
    ...
  } catch(err) {
    ...
  }
}
```

Function	Description

`fs.readFile(filename, "utf8", callback);`
 	
`fs.readFile("file.txt", "utf8",`
`  (err, contents) => { ... }`
`);`
	Reads the contents of the file located at relative directory **`filename`**. If successful, passes the file contents to the callback as **`contents`** parameter. Otherwise, passes error info to callback as **`error`** parameter.

`fs.writeFile(filename, data, "utf8", callback);`
 	
`fs.writeFile("file.txt", "new contents", "utf8",`
`  (err) => {  ...  }`
`);`
	Writes **`data`** string to the file specified by **`filename`**, overwriting its contents if it already exists. If an error occurs, **`error`** is passed to the callback function.

`fs.appendFile(filename, data, "utf8", callback);`
 	
`fs.appendFile("file.txt", "added contents", "utf8",`
`  (err) => { ... }`
`);`
	Writes **`data`** to the file specified by **`filename`**, appending to its contents. Creates a new file if the filename does not exist. If an error occurs, **`error`** is passed to the callback function.

`fs.existsSync(filename);`	Returns true if the given filename exists. <u>This is the only synchronous fs function you may use in CSE154</u> (the asynchronous function is deprecated due to race conditions).

`fs.readdir(path, callback);`
 	
`fs.readdir("dir/path", (err, contents) => {`
`  ...`
`});`
	Retrieves all files within a directory located at **`path`.** If successful, passes the array of directory content paths (as strings) to the callback as **`contents`** parameter. Otherwise, passes error info to callback as **`error`** parameter.

Useful path Module Functions

Function	Description
`path.basename(pathStr);`	Returns the filename of the **pathStr**. Ex. “picture.png” for “img/picture.png”
`path.extname(pathStr);`	Returns the file type/file extension of the **pathStr**. Ex. “.png” for “img/picture.png”
`path.dirname(pathStr);`	Returns the directory name of the **pathStr**. Ex: “img/” for “img/picture.png”

The glob module

Function	Description

`glob(pattern, callback);`
 	
`glob(“img/*”, (err, paths) => {`
`  ...`
`});`
 	
`// promisified`
`paths = await glob(“img/*”);`
	Globs all path strings matching a provided **`pattern`**. The pattern will generally follow the structure of a file path, along with any wildcards. If no paths match, the result array will be empty. If successful, passes the array of directory content paths (as strings) to the callback as **`contents`** parameter. Otherwise, passes error info to callback as **`error`** parameter.
	
	Common selectors:
	`*` - A wildcard selector that will contextually match a single filename, suffix, or directory.
	`**` - A recursive selector that will search into any subdirectories for the pattern that follows it.

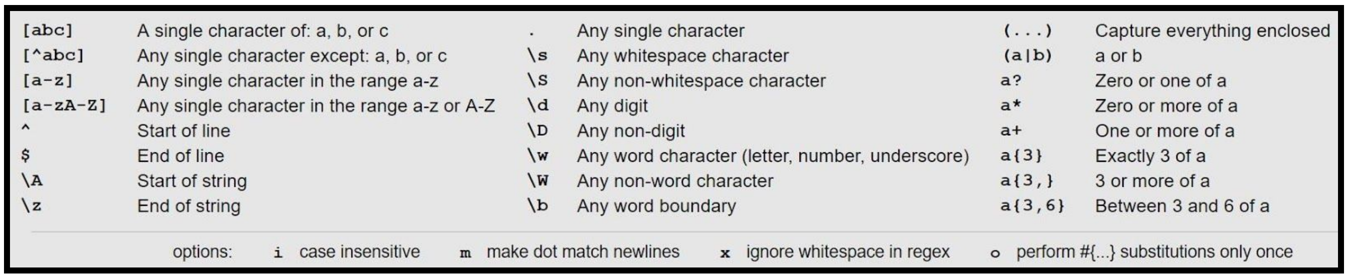
The promise-mysql module

Function	Description

`mysql.createConnection({`
`  host : hostname, // default localhost`
`  port : port,`
`  user : username,`
`  password : pw,`
`  database : dbname`
`});`
	Returns a connected database object using config variables. If an error occurs during connection (e.g. the SQL server is not running), does not return a defined database object.

`db.query(qryString);`
 	
`db.query(qryString, [placeholders]);`
	Executes the SQL query. If the query is a SELECT statement, returns a Promise that resolves to an array of RowDataPackets with the records matching the **qryString** passed. If the query is an INSERT statement, the Promise resolves to an OkPacket. Throws an error if something goes wrong during the query.
	
	When using variables in your query string, you should use ? placeholders in the string and populate [placeholders] with the variable names to sanitize the input against SQL injection

Regex Reference



`[abc]`	A single character of: a, b, or c	`.`	Any single character	`(...)`	Capture everything enclosed
`[^abc]`	Any single character except: a, b, or c	`\s`	Any whitespace character	`(a|b)`	a or b
`[a-z]`	Any single character in the range a-z	`\S`	Any non-whitespace character	`a?`	Zero or one of a
`[a-zA-Z]`	Any single character in the range a-z or A-Z	`\d`	Any digit	`a*`	Zero or more of a
`^`	Start of line	`\D`	Any non-digit	`a+`	One or more of a
`$`	End of line	`\w`	Any word character (letter, number, underscore)	`a{3}`	Exactly 3 of a
`\A`	Start of string	`\W`	Any non-word character	`a{3,}`	3 or more of a
`\z`	End of string	`\b`	Any word boundary	`a{3,6}`	Between 3 and 6 of a

options: `i` case insensitive `m` make dot match newlines `x` ignore whitespace in regex `o` perform #{...} substitutions only once

Special characters that need to be escaped to match as literals: `[]^$.|?*+(){}\`

SQL

`SELECT`

Description: Used to select data from a database table. If `DISTINCT` is used, no duplicate rows are returned.

Syntax (without `DISTINCT`):

```
SELECT column(s)
FROM table;
```

Syntax (with `DISTINCT`):

```
SELECT DISTINCT column(s)
FROM table;
```

`WHERE`

Description: Used to filter records, returning only those which meet provided conditions.

Syntax:

```
SELECT column(s)
FROM table
WHERE condition(s);
```

Condition types:

- `=, >, >=, <, <=`
- `!=, <>` (not equal)
- `BETWEEN min AND max`
- `LIKE %pattern` (where `%` is a wildcard)
- `LIKE pattern%`
- `LIKE %pattern%`

`ORDER BY`

Description: Used to sort the result set in ascending (default) or descending order.

Syntax:

```
SELECT column(s)
FROM table
ORDER BY column(s) ASC|DESC;
```

LIMIT

Description: Used to give the top-n elements of a given category.

Syntax:

```
SELECT column(s)
FROM table
LIMIT n;
```

CREATE TABLE

Description: Used to create a new table (optionally with PRIMARY KEY and/or FOREIGN KEY(s)). If using FOREIGN KEY, must reference a PRIMARY KEY or UNIQUE value of an existing table.

Syntax:

```
CREATE TABLE tablename(
  column1 datatype PRIMARY KEY,
  column2 datatype,
  ...
  columnN datatype,
  FOREIGN KEY (col) REFERENCES othertable(other_col)
);
```

Common Column Data Types:

`VARCHAR(N)` - strings of up to N characters (e.g., ’Whitaker’)

`INT` - integers (e.g., 10)

`DECIMAL(d,s)` – Has at most d digits, and s digits after the decimal place

`DATETIME` - date/time representation (e.g., ’2017-05-25 18:20:32’)

Alias for tables names

Description: You can use an alias to give a table a different name. You assign a table an alias by using the AS keyword as the following syntax for a partial query:

```
tablename AS tablealias
```

Examples:

```
SELECT g.name, g.platform FROM Games AS g
WHERE g.name LIKE "%Pokemon%"
```

As a shortcut you can also write this same query without the `AS`

```
SELECT g.name, g.platform FROM Games g
WHERE g.name LIKE "%Pokemon%"
```

INSERT INTO

Description: Used to insert a new record (row) into an existing table, where the listed values correspond to the listed columns.

Syntax:

```
INSERT INTO tablename (column1, column2, ..., columnN)
VALUES (value1, value2, ..., valueN);
```

DELETE

Description: Used to remove a record (row) which matches condition(s) from an existing table. If WHERE condition is omitted deletes all records in the table (but table is not deleted).

Syntax:

```
DELETE FROM tablename
WHERE condition(s);
```

UPDATE

Description: Used to modify the existing records in a table. If WHERE condition is omitted, updates all records.

Syntax:

```
UPDATE table_name
SET column1 = value1, column2 = value2, ... WHERE condition(s);
```

JOIN

Description: Used to select values from more than one table. Note that you can write multi-table queries either with equivalent WHERE clauses, or using the JOIN keyword.

Syntax/Examples:

```
SELECT col(s)
FROM table1, table2, ...
WHERE table1.a = table2.b
AND table2.c > '42';
```

OR

```
SELECT col(s)
FROM table1
JOIN table2 on table1.a = table2.b
JOIN ...
WHERE table2.c > '42';
```