CPEN400A – Term 1: Final Exam (2019)

Total Time: 2 hours and 30 minutes

Name: **Student Number:**

Instructions

1. This exam has **ten** questions, and is printed on a total of **fifteen** pages. Please ensure that you have a complete copy before starting the exam.
2. Write your answers in the space provided, except for the multiple choice questions. If you need more space, use the back of the paper, or ask us for more paper if you need.
3. Please use the Scantrans sheets for the multiple choice questions.
4. **You are allowed two "cheat" sheets of A4 size paper (can be handwritten or printed on both sides). These cheat sheets must be turned in with the exam.**

Points Distribution Table

Question	Points	Out of
1		15
2		5
3		5
4		5
5		5
6		5
7		5
8		5
9		5
10		5
Total		60

1. **Multiple choice questions (Please use the Scantrans sheets to answer the multiple choice questions – we will not grade any answers that are not on the Scantrans).**

   A. Which of the following components of a web application is essential for its working (i.e., without it, the application cannot be loaded in the browser)?
      a. HTML
      b. CSS
      c. JavaScript
      d. None of the above

   B. Which of the following statements is NOT correct about the DOM (Document Object Model) of a web application?
      a. DOM is created when the web application is loaded
      b. Events on a DOM node are propagated to other DOM nodes
      c. JavaScript code can modify individual DOM nodes
      d. CSS rules do not consider the DOM in their evaluation

   C. Which of the following statements is NOT true about prototypal inheritance (using Object.create) in JavaScript?
      a. The new object's prototype is set to the object passed to Object.create
      b. The constructor property of the new object is set to the constructor function
      c. The new object can be initialized during the call to Object.create
      d. None of the above

   D. When using the function.call to call a function *foo* in JavaScript, the *this* value inside the function body is set to
      a. The function *foo* object
      b. The .prototype field of *foo*
      c. The global context
      d. The first argument of *call*

   E. Which of the following statements is true about functions in JavaScript ?
      a. Every function has a *.prototype* field associated with it, AND this field is initialized to a new object when the function is created
      b. Every function has a *.prototype* field associated with it, BUT this field is NOT initialized to a new object when the function is created
      c. Only those functions that are used as constructors have a *.prototype* field associated with them, AND this field is initialized to a new object when the function is created
      d. Only those functions that are used as constructors have a *.prototype* field associated with them, AND this field is initialized to a new object when the function is used as a constructor

F. Two DOM nodes n and p are such that n is an ancestor of node p. Assume that we have defined both bubble and capture handlers on nodes n and p. A mouse click event occurs on node p. Which of the following statements is true?

   a. The capture handler of node n is invoked after the capture handler of p
   b. The capture handler of node n is invoked before the capture handler of p
   c. The capture handler of node n is invoked after the bubble handler of p
   d. None of the above

G. When using getElementByID to retrieve a node from the DOM, which of the following statements is NOT true

   a. getElementByID will return null if there is no node with the given ID
   b. getElementByID will return a list of all nodes with a given ID (if they exist)
   c. getElementByID will return a single node with the given ID (if it exists)
   d. None of the above

H. When the onload handler of an ajax request is invoked after the request is sent, which of the following statements is NOT true ?

   a. The request did not time out (time exceeded the request's timeout)
   b. The request did not receive a network error (e.g., server not found)
   c. The request did not receive a server error (e.g., 404 not found)
   d. None of the above

I. The fs.readFileSync function in node.js differs from the fs.readFile function in that

   a. It reads the entire file in one go rather than in chunks or blobs
   b. It blocks the main thread until the request completes
   c. It can return an error without causing termination of the program
   d. None of the above

J. When using *Promises,* adding a .catch as the last handler of the promise catches which of the following exceptions

   a. Errors thrown in the resolution of the original promise
   b. Errors thrown in the .then blocks of the promise handlers
   c. Both (a) and (b)
   d. Neither (a) nor (b)

K. The main disadvantage of NoSQL databases over traditional SQL databases is

   a. They are often slower, and hence need more powerful hardware to run queries
   b. They cannot support partition tolerance, and are hence less tolerant of failures
   c. They do not allow the administrator to easily modify the schema
   d. They require programmers to explicitly write code for table joins

L. A database that follows the BASE semantics compromises on which of the following properties of the CAP theorem

a. Consistency
b. Availability
c. Partition Tolerance
d. All of the above

M. Which of the following is NOT a prerequisite for an application to conform to REST principles?

a. Statelessness
b. Cacheability
c. Layered system
d. Authentication

N. Other than syntax, the main difference between arrow functions used in ES6 and regular functions used in ES5 is

a. Regular functions can take no parameters, while arrow functions must take at least one parameter
b. Regular functions don't have to return any values, while arrow functions must return a value
c. Regular functions have their *this* value be set to the global context in standalone mode, while arrow functions have it to set to the *this* value of the calling function
d. Regular functions have their *this* value be set to the global context in standalone, while arrow functions have it to set to the *this* value of the enclosing scope

O. Which of the following statements is something that can be done using the *class* keyword in ES6 compared to constructor functions in ES5.

a. The *class* keyword allows you to define multiple parents of an object
b. The *class* keyword does not allow you to get the prototype of an object
c. The *class* keyword prevents you from invoking the constructor as a standalone function
d. The *class* keyword prevents you from overwriting the members of an object after its creation

-------------------- End of Multiple Choice Questions ------------------

2. **HTML and CSS**

Consider the following HTML file describing a simple webpage. You can make reasonable assumptions about the web browser used to render the page.

```
<head>
    <link rel="stylesheet" type="text/css" href="rules.css">
</head>
<body>
    <div id="one" class="A">
        <div id="two" class="B"> <p>This</p>
            <div id="three" class="B"> <p>is a</p>
                <div id="four" class="A"><p>test</p>
                </div>
            </div>
        </div>
    </div>
</body>
```

A. Draw the DOM tree corresponding to the HTML page. No partial points will be given. (2 points)

```
html
  |-------body
             |---------[div id="one" class="A"]
                                 |--------p------This
                                 |---------[div id="two" class="B"]
                                             |------------p------is
                                             |------------[div id="three" class="B"]
                                                          |-------p-------a
                                                          |------[div id="four" class="A"]
                                                                  |-------p
```

B. Consider the following rules.css file applied to the page. What are the colors of the three <p> elements in the page (in order)? No explanation is needed.(**3 points)**

```
div { color: blue; }
div.B { color:orange; }
div.A { color:red; }
div #three { color:green; }
```

1. Orange
2. Greed
3. Red

Each of the above carries 1 point.

3. **JavaScript Objects**

   Write a function *inherits* that takes a constructor function *foo,* and an object 'o1' as arguments, and returns an object 'o2' that is identical to `o1' except that o2 is created as a sub-type of *foo*. In other words, inherits should make the "parent" object of o2 be of type 'foo', but everything else should be the same as object o1. For example, *inherits(Person, e),* should return an object with the same fields as 'e', except that it looks like it was created by inheriting from objects of type Person (i.e., objects created using the Person constructor function). Remember, you cannot change an object's __proto__ property after its creation. You may not use Object.setPrototypeOf for this question or no points will be given. (**5 points)**

   NOTE: only the properties defined in the object o1 itself should be copied to o2.

   function *inherits*(constructor, obj) {

```
    var res = Object.create( new constructor() );    // 1 pt.

    for (field in obj) {                             // 1 pt
            if ( obj.hasOwnProperty(field) ) {       // 1 pt
                    res[field] = obj[field];         // 1 pt
            }
    }

    return res;                                     // 1 pt
}
```

4. JavaScript Functions

Write a function *cached* that takes a function *foo* and returns a function that takes a variable number of arguments – the other function then invokes *foo* with the arguments, and caches the result returned by *foo*. Any future invocations of the function returned by *cached* with the same arguments should not invoke *foo,* but rather return the result from the cache (this does not include recursive invocations). You may not add any global state to the program or you'll get a 0. You may also assume that *foo* does not have any side effects, i.e., does not modify the global state in any way. Finally, for this question, you can assume that the *arguments* behaves like an array in JavaScript and treat is as such **(5 points).**

NOTE: You may not make any other assumption about foo, or no points will be given. In particular, you may not restrict foo's arguments or return value type.

Example of usage:
var foo = function (x, y, z) { return (x + y + z); }
var f = cached(foo);
var res1 = f(1, 2, 3); // should invoke foo(1, 2, 3) and cache the result
var res3 = f(1, 2, 3); // should return the value from the cache

var cached = function(foo) {

```
        var cache = {};                              // 1 point
        return function() {
                //  This part is not correct technically
                args = arguments;
                if ( cache[args] != null)            // 1.5 pts
                        return cache[args];

                var res = foo.apply(null, args);     // 1.5 pts
                cache[ args ] = res;                 // 1 point
                return res;
        }
}
```

};

5. **JavaScript Events and the DOM**

Write a function to traverse the DOM tree starting from a node ‘n’ with a given ‘ID’, and add event handlers to each of the elements (not nodeType ==1) in the subtree rooted at ‘n’. The event handlers should all be for the “click” event, and each event handler should log the level of the element to which it is attached to the console (when invoked), where the node ‘n’ is at level 0, the children of node ‘n’ are at level 1 and so on. You should also add the event handlers for the bubble phase of the event propagation only (you should set the third parameter to false of *addEventListener* for the bubble phase). You should also make sure other existing event handlers are not disturbed in any way by your modifications.

Note that you may not make any assumptions about the depth of the tree or its contents. You should not add any elements to the global state of the program or else you’ll get a 0. **(5 points).**

NOTE: You can assume that there’s always a node with the given ID in the DOM.

```
var addEventHandlers = function( ID ) {

     var root = document.getElementById(ID);       // 1 pt
    // we assume ID is always present, so node !=null

    var handler = function(i) {        // 1 pts
            return function() {
                console.log("Invoking handler at level ", i);
            }
    }

    var recurseHandler = function(node, level) {   // 2.5 pts
          // If it's not an element node, add a click handler
         if (node.nodeType!=1) return;
         node.addEventListener("click", handler(level));
         var children = node.childNodes;
         for (var i=0; i< children.length; i++) {
                  recurseHandler( children[i], level + 1);
         }
   }

   recurseHandler(root, 0);       // 0.5 pts

}

}
```

6. **Node.js**

Consider the following node.js program that opens a file in streaming mode and counts the number of characters in the file. Modify the program to instead count the number of star-words in the file – you only need to show the modified code.
A star-word is defined as a contiguous, non-empty stream of non-space characters that is preceded by *one* or more '*' characters, and succeeded by one or more spaces. Note that there may be multiple stars before a word, and multiple spaces after a word – these should not be identified as separate star words. For example, "***example " is a single star word. On the other hand, "example " is not a star word as it does not have a sequence of "*" preceding it. *Note that you cannot make any assumptions about the size of the blob that is read*, and your program should operate in streaming mode, i.e., it should count the words as and when they are read and not wait till the end (no points will be given if either condition is violated).

HINT: Construct a state machine to keep track of whether you're inside a star word.

```
var fs = require('fs');
var length = 0;
var fileName = "sample.txt";
var readStream = fs.createReadStream(fileName);

readStream.on("data", function(blob) {
     length += blob.length;
} );

readStream.on("end", function() {
     console.log("Total number of chars read = " + length);
} );

readStream.on("error", funcEon() {
     console.log("Error occurred when reading from file " +
fileName);
} );
```

The high level idea is to have a three-state state machine with the following transition matrix, and then go over each blob character by character, and increment length accordingly (they can optionally rename length as numStarWords).

State	'*'	**' ' (space)**	**Anything else**
0	1	0	0
1	1	0	2
2	2	0 (increment length)	2

They need to add a global variable *state* and initialize it to 0 along with *length. (0.5 pts). T*hey'd need to modify the readStream.on("data", function(blob)) as follows:

```
function(blob) {                                          (4 points)
     for (var i=0; i < blob.length; i++) {
          var c = blob[i];
          if  (state==0) {
                if (c=='*') state = 1;
          } else if (state == 1) {
               if (c == '*') state = 1;
               else if (c==' ') state = 0;
               else state = 2;
          } else  if (c==' ') {  // state is 2
               numStarWords += 1;
               state = 0;
          }
}
```

They should also rewrite the readStream.on("end", function(blob) {   (0.5 pt)

```
     console.log("Total number of starwords read = ",
numStarWords); } );
```

7. Promises

Write a function *waitAll* to take an array of *Promise* objects as a parameter, and then waits for all the promises to be either resolved or rejected before proceeding with the execution. For example, if an array [p1, p2, p3] has three promises p1, p2, and p3, then *waitAll* will need to wait for all three promises to be resolved or all three promises to be rejected before returning. Note that *promise.all* will reject if the first promise is rejected, so you can't use *promise.all* directly in your implementation. Note that the return value of *waitAll* should be a promise.

HINT: Use *promise.race* in an iterative fashion. To remove an element from a list/array, you can use list.splice(index, 1), where index is the element's index.

I've written a recursvie solution below, but they could do it iteratively as well.

```
function waitAll(promiseList){
  if (promiseList.length === 0) return
Promise.resolve(promiseList);
  var done;
  promiseList.forEach((item, index, array)=>{
    item.then(()=>{
      done = index;
    }).catch(()=>{
      done = index;
    });
  });
  return Promise.race(promiseList)
    .then((result)=>{
      promiseList.splice(done, 1);
      return waitAll (promiseList);
    }).catch((error)=>{
      promiseList.splice(done, 1);
      return waitAll (promiseList);
    });
}
```

8. Databases

A. Consider a data item that is replicated in two different geographic sites A and B, and assume that there is a single network link between the two sites. Assume that the network link goes down, causing a network partition, and that there are read and write requests that are queued up at each of the sites when that happens. Which of the following requirements are violated (of consistency, availability and partition tolerance) in each of the following scenarios (if any)? (**3 points**)

a. All reads are allowed at both sites, but no writes (1 point)

Partition Tolerance, as writes are not allowed during a partition

b. Both reads and writes are allowed at both the sites (1 point)

Consistency, as the replicas can get out of sync due to writes

c. Reads and writes are allowed only at Site ‘A’, and not site ‘B’ (1 point)

Availability, as the site B becomes unavailable during a partition

B. Consider a MongoDB database consisting of two collections: *movies* and *users*. *users* consists of the user no, user name, followed by the list of movies watched by each user as follows (in JSON format):

```
{
        User no,
        UserName,
         [ movieNo1, movieNo2, movieNo3 ….. ]
};
```

While *movies* consists of the movie numbers followed by the movie names

{ movieNo, movieName }

Complete the following query to find the list of users who watched a movie. You can assume that the database has already been loaded into the Mongo shell. (**2 points**)

```
db.movies.find().forEach(
        function(Object) {
                var users = db.users.find( {“movies”: Object.movieNo } ); // 1 point
                if (! users) return;
                printJson( movieNo );          // 1 point for the 3 print statements
                printJson( moviewName );
                printJson( users );
        }

} );
```

9. **RESTful design**

Consider a guestbook application that allows users to post comments on a public web forum. Each comment is linked to a user id, and multiple users should be able to leave comments. Likewise, a single user should be able to add a comment. However, it is not allowed for users to modify or remove previous comments – even their own. Finally, there should be an option for the system administrator to add or remove users. Design a RESTFul API for the above application – you should adhere to the principles of REST or else no points will be given

A. What are the resources you will have in the application ? (2 points)

The resources exposed should be (1) users, and (2) comments. The former is for the list of users, while the latter is for the list of comments.

(Each of the above gets a point)

B. What're the operations you would allow on the resources identified in part A, and what would they do for each of the resources (3 points) ?

(Each of the operations below gets 0.5 marks for a total of 6 operations.)

There are 3 operations to be supported on the collection "users":

i. GET guestbook/users– returns the list of users, with pagination options (?offset=""&&length=""), or a specific user with a user-id (?id="XX")
ii. POST guestbook/users– allows a new user to be removed from the collection
iii. DELETE guestbook/users– removes a given user from the list of users

There are 3 operations to be supported on the collection "comments":

(1) GET guestbook/comment_id– retrieve a specific comment
(2) GET guestbook/comments?user=user_id – retrieve all comments by a user
(3) PUT, POST and DELETE are not supported as one

Question 10: Design Patterns and AJAX

Consider the function below, which sends an AJAX message to the server every-time it is called. It takes 2 callback functions as arguments as well as a *timeout* parameter, and returns a function that returns the request sent to the caller function after setting up the callbacks.

The 2 questions below require you to modify this web application in specific ways. You don't need to rewrite the entire code for the questions – just say which lines you'll modify (if any) and how – you can use the line numbers below in your answer.

```
1: function ajaxRequest(url, timeout, onSuccess, onFailure) {
2:
3:         return function() {
4:                var req = new XMLHttpRequest();
5:                req.open("GET", url);
6:                req.timeout = timeout;
7:                req.onerror = function() {
8:                        onFailure();
9:                }
10:                req.ontimeout = req.onerror;
11:                req.onabort = req.onerror; // optional
12:                req.onload = function() {
13:                        if (req.status==200) {
14:                                onSuccess();
15:                        }
16:                        else {
17:                                onFailure();
18:                        }
19:                }
20:              // Send the request asynchronously
21:              req.send();
22:              return request;
23:         }
```

A. It is required that there be only one outstanding AJAX message to the server at any time. In other words, if the inner function is called when a message has already been sent (to the same url), then a new message should not be sent, but the message that is in transit should be returned – there is no need to modify the call-back for the message in that case. (2 points)

HINT: Use the singleton pattern

To use the singleton pattern, they need to add a local variable "current" to the outer scope and use it as a cache – they need to remember the request after it's sent, and in case it completes, set it back to null. They also need to check the value of current.

The code would be as follows (they don't need to write the entire code, btw):

```
function ajaxRequest(url, timeout, onSuccess, onFailure) {
        var current = null;
        return function() {
                if (current!=null) req = current;
                var req = new XMLHttpRequest();
                current = req;
                req.open("GET", url);
                req.timeout = timeout;
                req.onerror = function() {
                        onFailure();
                        current = null;
                }
                req.ontimeout = req.onerror;
                req.onabort = req.onerror; // optional
                req.onload = function() {
                        if (req.status==200) {
                                onSuccess();
                                current = null;
                        }
                        else {
                                onFailure();
                                current = null;
                      }
                }
                req.send();
                return req;
        }
}
```

Each of the above lines added (in bold) gets 0.5 points for a total of 2.5 points.

B. Now assume that you want to add multiple handlers for the message whenever a new call is made to the *inner function*, but without sending new requests when a request is outstanding (to the same URL). In other words, if we call *ajaxRequest* when another request to the same URL is outstanding, then a new request is not sent but the outstanding request is returned like in part 'A'. *But the new onSuccess and onFailure handlers are added to the outstanding request's handlers*, and all the handlers get called when the request completes (i.e., succeeds or fails). You don't need to write the code in detail, but you should explain the modifications you'll make to the code in part 'A' in your answer. (2.5 points)

HINT: Use the observer pattern

There are many ways to do this, but the most straightforward one is to maintain a list of success handlers and a list of failure handlers. When the inner function is invoked and a request is in progress (value of *current* is not null), then the success and failure handlers get added to this list and the request is not sent. When the request completes successfully (or fails), the handlers in the list of success (failure) handlers are each invoked in turn – this can be done via a iteration thro' the array.