1 Homework 2

Due Date: Oct 15; To be turned in on paper in class.

No late submissions.

Important Reminder: As per the course *Academic Honesty Statement*, cheating of any kind will minimally result in receiving an F letter grade for the entire course.

Please remember to justify all answers.

Note that some of the questions require you to show code or the output resulting from some code. You may use a JavaScript implementation to verify your answers but you should realize that you will not have access to an implementation during exams.

You are encouraged to use the web or the library but are required to cite any external sources used in your answers.

1. The specification for the DocFinder class for *Project 2* splits the construction of a DocFinder instance into a synchronous constructor call followed by an asynchronous call to the `init()` function. OTOH, the user-store example discussed in class uses a single static asynchronous factory function **`newUserStore()`**. Discuss the tradeoffs between these two approaches. Would one be preferred over the other? *10-points*

2. The documentation for *nodejs modules* mentions that if you want to `export` individual JavaScript entities from a module you can do so "by specifying additional properties on the special `exports` object". The documentation gives an example of a `circle` module:

   ```
   const { PI } = Math;
   exports.area = (r) => PI * r ** 2;
   exports.circumference = (r) => 2 * PI * r;
   ```

   The documentation subsequently makes clear that if you want to package up all your exports, then assigning directly to `exports` as in:

   ```
   const { PI } = Math;
   exports = {
     area: (r) => PI * r ** 2;
     circumference: (r) => 2 * PI * r;
   }
   ```

   will not work. Instead it is necessary to assign to `module.exports` as in:

   ```
   const { PI } = Math;
   ```

```
module.exports = {
  area: (r) => PI * r ** 2;
  circumference: (r) => 2 * PI * r;
}
```

Why is it necessary to assign to `module.exports` and not simply to `exports`? *5-points*

3. Given an `Object` literal:

```
const NAME_VALUES = {
  [key1]: [ val1, docString1 ],
  [key2]: [ val2, docString2 ],
  ...
}
```

for some `key`*i*, `val`*i* and `docString`*i*, how would you write a **single expression** `expr` involving `NAME_VALUES`, such that `const nameValues = expr` results in `nameValues` having the value:

```
{
  [key1]: val1,
  [key2]: val2,
  ...
}
```

That is, you need to *project out* only the `key`*i*, `val`*i* pairs.

The only functions which your answer may use are those provided by the standard JavaScript libraries.

Hint: Consider using `Object.entries()`, `Object.assign()`, destructuring, spread operator. *10-points*

4. Rewrite the `Shapes` example covered in class **without** using ES6 classes. Specifically, you need to provide alternate code for `Shape`, `Rect` and `Circle` without using the `class` keyword. Your modified implementation should still allow the example code to run. *10-points*

5. JavaScript uses the `prototype` property of a function which is used as a constructor to allow accessing the shared prototype of any objects it constructs. Though this design appears strange it, or something like it, seems to be the simplest design given that objects constructed using the same constructor function should inherit the same behavior. Why does this design appear to be the simplest design? *10-points*

6. Many articles on the web over-complicate the rules for *hoisting*. For example, this article gives these rules for function hoisting:

- *function declarations hoist the function definitions.*
- *Function expressions in JavaScript are not hoisted.*

Can you give very simple rules which describe the behavior of `let`, `var`, function declarations and function expressions based on the concepts of **scope**, **declarations and definitions**. *10-points*

7. A set B of small non-negative integers can be represented on computers using a bitset which is a single integer `b`, where integer $i \in B$ iff bit i in `b` is 1 (we can assume that bits are indexed in *little-endian* order with bit 0 corresponding to the LSB).

   With this representation, many set operations can be performed using bitwise operations. For example, assuming that `b1` and `b2` are the representation of sets B_1 and B_2, the set union $B_1 \cup B_2$ is simply `b1 | b2` and the set intersections $B_1 \cap B_2$ is simply `b1 & b2`.

   (a) In JavaScript, what is the value of the largest integer which can be stored in a bitset represented using a single integer.

   (b) Provide a definition for a function **toBitSet**`(list)` which when given an array `list` of small non-negative integers, returns an integer giving a bitset representation of all the integers in `list`.

   For example, **toBitSet**`([1, 5, 3])` should return `42`.

   (c) Provide a definition for a function **fromBitSet**`(bitset)` which when provided with a `bitset` representation of a set, returns an array listing all the integers in `bitset`.

   For example, **fromBitSet**`(42)` should return list `[1, 3, 5]` (any ordering is acceptable in the returned list).

   The above functions are subject to the same restrictions as the functions you wrote in the previous homework; i.e. the body can consist of only a single return statement which returns an expression which computes the desired value.

   Hint: An integer `b` can be converted to its binary representation using `b.toString(2)`. *15-points*

8. Given the following code:

```
const x = 1;

function f(a) {
  let y = x + 1;
  if (a) {
    var x = y + 1;
    y = x + 2;
  }
  return x + y;
```

```
}

console.log(f(1), x);
```

(a) What will be the output of the above program. Explain why it is so.

(b) What will be the output of the above program if the `var x = ..¬ .` statement inside the **if** is changed to a `let x = ...` statement. Explain why it is so.

Hint: arithmetic on undefined values results in a `NaN`. *10-points*

9. Assuming no earlier variable declarations, what will be the output of the following JavaScript code when run in non-strict mode?

```
x = 1;
obj1 = { x: 2, f: function() { return this.x; } }
obj2 = { x: 3, f: function() { return this.x; } }
f = obj1.f.bind(obj2);
console.log(obj1.f() - obj2.f() + f());
```

Explain why it is so. *10-points*

10. Discuss the validity of the following statements. What is more important than whether you ultimately classify the statement as **true** or **false** is your justification for arriving at your conclusion. *10-points*

(a) Constructor functions in JavaScript must be declared using the `constructor` keyword.

(b) Given objects `a` and `b`, `a === b` is true iff the values of all the properties of `a` are recursively equal (using `===`) to the values of the properties of `b`.

(c) The prototype of an object can be changed after it has been created.

(d) `this` for a fat-arrow function can be changed using `bind()`.

(e) It is possible to set things up so that assigning to a single object property changes multiple properties.