1 Homework 2 Solution

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*

   There is a major problem with requiring a separate constructor call followed by a call to `init()`: namely having to document the fact and then trust that the client programmer and generations of subsequent maintenance programmers will comply. It is much better to have an API without dependencies between its separate parts.

   The static factory method requires only a single call to construct an instance. If a module system wraps the class, then it is possible to export the static factory method without exporting the constructor, thus making it impossible to construct a instance without using the factory method. So the static factory method wins hands down.

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*

The "variable" `exports` is really a property of the `module` object and nodejs will access the exports using the property `module.exports`. Since assigning directly to `exports` will not affect `module.exports`, it is necessary to assign to `module.exports`.

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*

```
const nameValues =
  Object.assign({}, ... Object.entries(NAME_VALUES).
                         map(([k, v]) => ({k: v[0]})));
```

`Object.entries(NAME_VALUES)` will return a list of pairs: `[ key`*i*, `[ val`*i*, `docString`*i* `] ]` where the 2nd element is itself a pair. The `map()` operation returns a list of single-property objects `{ key`*i*: `[ val`*i* `}`. These are then spread into `Object.assign()` to accumulate in a newly created object `{}`.

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*

```
function Shape(x, y) { //constructor
  this.x = x; this.y = y;
}

Shape.distance = function(s1, s2) {
  const xDiff = s1.x - s2.x;
  const yDiff = s1.y - s2.y;
  return Math.sqrt(xDiff*xDiff + yDiff*yDiff);
};

function Rect(x, y, w, h) { //Rect constructor
  Shape.call(this, x, y); //call as non-constructor
  this.width = w; this.height = h;
}

Rect.prototype = new Shape();
Rect.prototype.area = function() {
  return this.width*this.height;
}

function Circle(x, y, r) { //Rect constructor
  Shape.call(this, x, y); //call as non-constructor
  this.radius = r;
}

Circle.prototype = new Shape();
Circle.prototype.area = function() {
  return Math.PI*this.radius*this.radius;
```

```
}

//code for exercising shapes remains unchanged
const shapes = [
  new Rect(3, 4, 5, 6),
  new Circle(0, 0, 1),
];

shapes.forEach((s) => console.log(s.x, s.y, s.area()));

console.log(Shape.distance(shapes[0], shapes[1]));
```

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*

   In a class-based language, there are two static things associated with a class (where **static** means independent of a `class` instance):

   (a) The class.

   (b) A constructor for instance objects. This is basically a static method of the class.

   The shared behavior for instance objects is defined by methods in the class.

   JavaScript does not have any classes. So it makes sense to make the constructor the single known thing about the class and be able to access the shared behavior via the constructor; specifically. via its `prototype` property. So by hanging the object prototype on the constructor function, the constructor function provides a single point-of-access for all static class functionality.

   In JavaScript, every function has this `prototype` property which will be initialized with a `constructor` property referencing the function. When a function is called using the `new` operator, `this` is set to reference a newly created object whose prototype points to the same object as the constructor's `prototype` property. Hence shared behavior can be defined in this shared prototype object referenced through the constructor.

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*

[Unfortunately, this question did not quite work out the way it was intended; there is a single simple rule which explains `var`, `let` and `const` declarations; this rule also explains function definitions the way they are normally used, but it does not have a simple explanation for function definitions which are made in dynamic execution contexts (something which should not usually be done).]

First we need to make sure that the distinction between a declaration and definition is clear. A declaration merely states the existence of an entity; a definition gives it a value. (Note that languages like C allow an entity like a function to be declared multiple times as long as all the declarations are consistent, but there can only be a single definition.)

So in a program fragment like:

```
var x = 42;
let y = 22;
```

`var x` and `let y` are the declarations, whereas the part following the `=` is an initializer which gives a definition for the variable.

[JavaScript distinguishes between reading an **undeclared** variable versus reading an **unassigned** variable; the former causes an error, whereas the latter simple results in the `undefined` value.]

A function definition like `function f() { ... }` constitutes both a declaration and definition of `f()`. OTOH, `var f = function() { ...}` is just a `var` declaration followed by an initializer which is a `function` expression.

The behavior of (almost all) JavaScript declarations and definitions can be explained by one simple rule:

The scope of any JavaScript declaration begins at the start of of the syntactic construct associated with that declaration.

Then we simply need to specify which syntactic construct is associated with each type of declaration:

- `var` declarations and function definitions have function scope; i.e. they come into scope at the start of the enclosing function.
- `let` and `const` have block scope; i.e. their scope starts at the beginning of the associated syntactic block (there is a separate scope associated with each loop).

The initializer associated with a `var`, `const` or `let` declaration remains at the point of the declaration. That explains the fact that the use of a variable declared `var` before the point of declaration results in `undefined` and the use of a `const` or `let` results in a *temporal dead-zone* error.

OTOH, function definitions **usually** act as though the actual function body is moved to the start of the enclosing function scope.

(The rest of this answer explores the situations where the simple rule breaks down for function definitions).

The reason for the *usually* qualifier is that it is possible for the function to be defined within some dynamic context: if the function is defined conditionally or in the presence of exceptions, there does not seem to be any simple way of describing how a function definition works.

Note that it is usually a bad idea to define a function inside a syntactic construct like a if-then-else, or a loop and then use it outside that construct, but we will look at some examples to explore these complexities:

```
> function f(a) {
    if (a) {
      function g() { return 42; }
    }
    return g;
  }
undefined
> x = f(1)
[Function: g]
> x()
42
> x = f(0)
undefined
> x()
TypeError: x is not a function
```

The above example shows that even though `g()` is defined within a `if`-condition, it has function scope. Note that when the condition is true `g()` is both declared and defined; when the condition is false, it is declared but not defined (its value is `undefined`).

Note the different behavior if we return a `g1` rather than the function `g()` defined within a false condition:

```
> function f(a) {
    if (a) {
      function g() { return 42; }
```

```
    }
    return g1;
  }
undefined
> x = f(0)
ReferenceError: g1 is not defined
```

In this example since g1 is undeclared, a reference to it causes an error. The very similar earlier example for `f(0)` showed that `g` was declared but not defined.

Note that we can even define a function within a loop; because the definition has function scope, the definition outside the loop will capture the definition on the last execution of the loop.

```
> function f(n) {
    for (let i = 0; i < n; i++) {
      function g() { return 2*i; }
    }
    return g;
  }
undefined
> x = f(3)
[Function: g]
> x()
4
>
```

Up till these examples, the function declaration has been acting exactly like a `var` declaration. However, the initializer for a `var` declaration remains at the point of declaration, but the entire body of the function definition is moved up to the start of the containing function as illustrated by the following example:

```
> function f() {
    console.log(22, g());
    function g() { return 42; }
    return g;
  }
undefined
> f()
22 42
[Function: g]
>
```

Note that we refer to `g` before it's point of definition and can still access its definition; we cannot do so for a `var`:

```
> function f() {
    console.log(22, g);
    var g = 42;
    return g;
  }
undefined
> f()
22 undefined
42
```

Things get weird in the presence of exceptions:

```
> function f() {
    try {
      console.log(22, g());
      throw 'err';
      function g() { return 42; }
    }
    catch (e) {
    };
    return g;
  }
undefined
> x = f()
22 42
undefined
> x()
TypeError: x is not a function
```

So `g` is defined before the `throw` but is not defined at the time the function `f()` returns.

So there does not appear to be any simple rule for these function definitions within dynamic contexts.

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*

(a) JavaScript performs bit-operations on 32-bit integers; hence the largest natural number representable using JavaScript integers will be 31.

(b) All we need to do is set the bit corresponding to each array element. We can do so by reducing the array:

```
function toBitSet(list) {
  return list.reduce((acc, e) => acc | (1 << e),
0);
}
```

We start out with an accumulator `acc` of 0, for each element of the array we update the accumulator with the bit corresponding to that element.

(c) We get the bits of an integer using `toString(2)` and split those into an array. We then reduce the reverse of this array using the array index to concatenate the appropriate natural number into an accumulator (initialized to `[]`) when the bit is a 1.

```
function fromBitSet(bitset) { //comments show when
bitset === 42
  return bitset.   //42
    toString(2).   //'101010'
```

```
            split('').        //[ '1', '0', '1', '0', '1', '0' ]
            reverse().        //[ '0', '1', '0', '1', '0', '1' ]
            reduce((acc, e, i) =>
                      e === '0' ? acc : acc.concat([i]),
                   []);    // [ 1, 3, 5 ]
        }
```

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*

   (a) The `var` declaration (not the initializer) is hoisted to the start of the function; so the code is equivalent to the following:

```
const x = 1;

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

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

   So the `x` in `let y = x + 1` will be `undefined`. Since arithmetic on an `undefined` value results in a `NaN`, `y` will be a `NaN` and the return value of the call to `f(1)` will be a `NaN`. The global `x` will be unaffected by the call to `f()`. Hence the output will be `NaN 1`.

(b) If the `var` is changed to a `let`, the code will be:

```
const x = 1;

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

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

The scope of the `let x` will simply be the `if`-block, so the `x` in the declaration for `let y = x + 1` will refer to the global `x` giving `y` the value 2; when the if-block executes (since `a === 1` is truthy), `x` will get the value 3 and `y` will be assigned 5. The `x` in the return value will refer to the global `x` which will remain unchanged at 1. Hence the function call `f(1)` will return 6 and the output of the `console.log()` will be `6 1`.

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*

The call to `obj1.f()` has `this` set to `obj1`; hence it will return `obj1.x` which is 2. The call to obj2.f() has `this` set to `obj2`; hence it will return `obj2.x` which is 3. Finally, the `f()` call has `this` bound to `obj2`, so it will return `obj2.x` which is 3. Hence the value printed will be `2 - 3 + 3` which is 2.

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 `con-`

structor 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.

The answers follow:

(a) The answer is that it depends on whether the constructor is declared within an ES6 `class` or not. If declared within an ES6 `class`, then the `constructor` keyword must be used (and the constructor is not given any name). OTOH, if the constructor is not for an ES6 `class`, then the `constructor` keyword should not be used and it should be declared as a regular `function` with having a name.

(b) Objects are compared for equality using reference equality; i.e. two objects compare equal iff they are the same object. Hence no recursive equality checking is done and the statement is **false**.

(c) It is definitely possible to change the prototype of an object after it has been created using `Object.setPrototypeOf()`. For example, consider the following:

```
> function F() {}
undefined
> function G() {}
undefined
> G.prototype.f = function() { return 13; }
[Function]
> x = new F()
F {}
> x.f()
TypeError: x.f is not a function
> Object.setPrototypeOf(x, G.prototype)
G {}
> x.f()
13
```

So the prototype of `x` was changed from `F.prototype` to `G.prototype` after it was created. Hence the statement is **true**.

(d) For a fat-arrow function, `this` is always the value which was captured lexically within the scope in which the fat-arrow function was defined and it is not possible to change it using `bind()`.

```
> x = 11
11
//in this.x below, this refers to global context
> obj1 = { x: 22, f: () => () => this.x }
{ x: 22, f: [Function: f] }
> g = obj1.f() //g is returned fat-arrow function
[Function]
> g() //calling it returns global x
11
> obj2 = { x: 5 } //create another object
{ x: 5 }
> h = g.bind(obj2) //bind g to obj2
[Function: bound ]
> h() //still refers to global x
11
>
```

So the statement is **false**.

(e) It is possible to change multiple object properties by assigning to a single property by associating a setter function with the single property. For example,

```
> obj = {
    a: 1,
    b: 5,
    set c(v) { this.a = this.b = v; } }
{ a: 1, b: 5, c: [Setter] }
> [obj.a, obj.b]
[ 1, 5 ]
> obj.c = 42
42
> [obj.a, obj.b]
[ 42, 42 ]
>
```

Hence the statement is **true**.