COMP 333
Spring 2026
Final Practice Exam (Solutions)

The final exam is cumulative. This practice exam, **in addition to** the prior practice exams, assignment 1, in-class handouts, and exams, is intended to be a comprehensive guide for studying. This practice exam only focuses on material since the last exam. You are permitted to bring three 8.5 x 11 sheets of paper into the exam with you, as long with anything printed or handwritten on them. Both sides of both sheets can be used.

Array Routines in JavaScript

1.) Consider the following array definition in JavaScript:

```
let arr = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

1.a.) Use `filter` to get an array of all even elements in `arr`.

```
// filter takes a function that takes an element and returns
// true if the element should be in the returned array, else
// false
arr.filter(e => e % 2 === 0)

// alternative answer
arr.filter(function (element) {
  return element % 2 === 0;
})
```

1.b.) Use `map` to get an array of strings, where each string represents a number in `arr`. As a hint, you can call the `toString()` method on a number (e.g., `5.toString()`) in JavaScript to get its string representation.

```
// map takes a function that takes an element and returns the
// corresponding value which should be in the output array
arr.map(e => e.toString())

// alternative answer
arr.map(function (element) {
  return element.toString()
});
```

1.c.) Use a combination of `filter` and `reduce` to get the sum of all elements in `arr` which are greater than `5`.

```
// this use of reduce uses an explicit starting accumulator of 0
arr.filter(e => e > 5)
   .reduce((accum, element) => accum + element, 0)

// alternative answer
arr.filter(function (e) { return e > 5 })
   .reduce(function (a, e) { return a + e }, 0)
```

1.d.) Use a combination of `filter` and `reduce` to get the product of all positive elements in `arr`.

```
// this use of reduce uses an explicit starting accumulator of 1
arr.filter(e => e > 0)
   .reduce((accum, element) => accum * element, 1)

// alternative answer
arr.filter(function (e) { return e > 0; })
   .reduce(function (a, e) { return a * e }, 1)
```

1.e.) Use `reduce` to get the last element in `arr`.

```
// reduce takes a function that takes an accumulator and an
// element, and returns the value of the new accumulator.  In
// this case, reduce is only given the function, so it will use
// the first array element as the initial accumulator, and start
// iterating from the second array element
arr.reduce((accum, element) => element)

// alternative answer
arr.reduce(function (accum, element) {
  return element;
})
```

Objects in JavaScript

2.) Create an object holding a field named `"foo"`, holding the value `12`.

```
{ 'foo': 12 }
```

3.) Consider the following JavaScript code:

```
let obj1 = { 'foo': 1 };
let obj2 = { 'foo': true, 'bar': 3.14 };
console.log(obj1.foo);
console.log(obj1.bar);
console.log(obj2.foo);
console.log(obj2.bar);

let obj3 = { 'baz': 'hello', '__proto__': obj2 };
console.log(obj3.foo);
console.log(obj3.bar);
console.log(obj3.baz);

let obj4 = { 'foo': 3, '__proto__': obj1 };
console.log(obj4.foo);
console.log(obj4.bar);
console.log(obj4.baz);
```

What is the output of this code?

```
1
undefined
true
3.14
true
3.14
hello
3
undefined
undefined
```

4.a.) Define a constructor for `Dog` objects, where each `Dog` object has a name. An example code snippet is below, illustrating usage:

```
let d = new Dog("Rover"); // line 1
console.log(d.name);      // line 2; prints Rover

// From line 1, we need a Dog constructor that takes one
// parameter. From line 2, the constructor must be setting the
// name field of Dog objects to the parameter.
function Dog(param) {
  this.name = param;
}
```

4.b.) Define a different constructor for `Dog`, which puts a `bark` method **directly** on the `Dog` objects. The `bark` method should print `"Woof!"` when called. Example usage is below:

```
let d = new Dog("Sparky");
d.bark(); // prints Woof!

function Dog(name) {
  this.name = name; // not explicitly required based on the
                    // question
  // bark is directly on created Dog objects, as opposed to
  // being on the prototype chain for Dog objects
  this.bark = function() {
    console.log("Woof!");
  };
}
```

4.c.) Define a different constructor for `Dog`, which puts a `growl` method **directly** on the `Dog` objects. The `growl` method should print `"[dog name] growls"` when called. Example usage is below:

```
let d = new Dog("Rocky");
d.growl(); // prints Rocky growls

function Dog(name) {
  this.name = name;
  // growl is directly on created Dog objects, as opposed to
  // being on the prototype chain for Dog objects
  this.growl = function() {
    console.log(this.name + " growls");
  };
}
```

5.) Consider the following JavaScript code, where multiple methods are put onto `Pair` objects in the constructor:

```
function Pair(a, b) {
    this.first = a;
    this.second = b;
    this.getFirst = function() {
        return this.first;
    };
    this.getSecond = function() {
        return this.second;
    };
}
```

Rewrite this code so that all `Pair` objects share a single prototype object, and this prototype object contains any methods defined on `Pair`.

```
// Using the approach shown in class
let pairPrototype = {
    'getFirst': function() {
        return this.first;
    },
    'getSecond': function() {
        return this.second;
    }
};

function Pair(a, b) {
    this.first = a;
    this.second = b;
    this.__proto__ = pairPrototype;
}


// ALTERNATIVE SOLUTION: using best practices
Pair.prototype.getFirst = function() {
        return this.first;
};
Pair.prototype.getSecond = function() {
        return this.second;
};

function Pair(a, b) {
    this.first = a;
    this.second = b;
}
```

6.) Consider the JavaScript code below and corresponding output:

```
let three = new MyNumber(3);
let five = new MyNumber(5);

let eight = three.add(five);
let fifteen = three.multiply(five);

console.log(three.getValue());
console.log(five.getValue());
console.log(eight.getValue());
console.log(fifteen.getValue());


---OUTPUT---
3
5
8
15
```

Implement any missing code necessary to produce the above output. Multiple solutions are possible. The next page is blank in case you need it.

Looking at the above code:

- There must be a `MyNumber` constructor which takes a parameter
- There must be an `add` method defined on `MyNumber` objects, either directly on the object, or on the prototype.
    - `add` takes another `MyNumber` object and returns something
- There must be a `multiply` method defined on `MyNumber` objects, either directly on the object, or on the prototype.
    - `multiply` takes another `MyNumber` object and returns something
- There must be a `getValue` method defined on `MyNumber` objects, which appears to return the number passed in the constructor
    - It looks like `add` and `multiply` are specifically returning `MyNumber` objects which wrap around the results of the operations

From there, we can take these notes and derive the code on the next page.

```
function MyNumber(value) {  // MyNumber constructor takes param
    this.value = value;

    // There must be an add method defined on MyNumber objects,
    // either directly on the object, or on the prototype.
    // add takes another MyNumber object and returns something
    // It looks like add and multiply are specifically returning
    // MyNumber objects which wrap around the results of the
    // operations.
    this.add = function (other) {
        return new MyNumber(this.value + other.value);
    };

    // There must be a multiply method defined on MyNumber
    // objects, either directly on the object, or on the
    // prototype.  multiply takes another MyNumber object and
    // returns something.
    // It looks like add and multiply are specifically returning
    // MyNumber objects which wrap around the results of the
    // operations.
    this.multiply = function (other) {
        return new MyNumber(this.value * other.value);
    };

    // There must be a getValue method defined on MyNumber
    // objects, which appears to return the number passed in the
    // constructor.
    this.getValue = function () { return this.value; };
}

// ALTERNATIVE SOLUTION using best practices and the prototype
function MyNumber(value) {
    this.value = value;
}

MyNumber.prototype.add = function (other) {
    return new MyNumber(this.value + other.value);
};

MyNumber.prototype.multiply = function (other) {
    return new MyNumber(this.value * other.value);
};

MyNumber.prototype.getValue = function () {
    return this.value;
}
```

7.) Consider the JavaScript code and corresponding output below:

```
let obj1 = new Obj("foo");
console.log(obj1.field); // output: foo

let obj2 = new Obj("bar");
console.log(obj2.field);          // output: bar
console.log(obj2.doubleField()); // output: barbar

let obj3 = new Obj("baz");

console.log(obj3.field); // prints baz
```

Complete any missing elements needed to allow this code to run and produce this output.

```
// Object is a built-in in JavaScript, but not Obj. This
// requires a custom constructor. From obj1, we know that Obj
// must be a constructor, and that Obj objects need a field
// named "field".  The value of this field must be equal to
// whatever its parameter is.
function Obj(param) {
  this.field = param;

  // From obj2, we know that we need a doubleField method on Obj
  // objects.
  this.doubleField = function() {
    // + in this context performs string concatenation; this
    // concatenates this.field onto itself
    return this.field + this.field;
  };
}


// ALTERNATIVE SOLUTION using best practices and the prototype
function Obj(param) {
  this.field = param;
}

Obj.prototype.doubleField = function() {
  return this.field + this.field;
}
```

8.) Consider the following JavaScript code:

```
function Foo() {
    this.toInt = function() {
        return 0;
    };
}

function Bar(m) {
    this.n = m;
    this.toInt = function() {
        return this.n.toInt() + 1;
    };
}

let t1 = new Foo();
console.log(t1.toInt());

let t2 = new Bar(t1);
console.log(t2.toInt());

let t3 = new Bar(new Bar(t2));
console.log(t3.toInt());
```

What is the output of this code?

```
0
1
3
```

9.) Consider the JavaScript code and corresponding output below:

```
let w1 = new Wrapper(1);
console.log(w1.getValue()); // prints 1

let w2 = w1.transform(e => e + 1);
console.log(w1.getValue()); // prints 1
console.log(w2.getValue()); // prints 2

let w3 = w1.transform(e => e + 7);
console.log(w1.getValue()); // prints 1
console.log(w2.getValue()); // prints 2
console.log(w3.getValue()); // prints 8
```

Complete any missing elements needed to allow this code to run and produce this output. Multiple solutions are possible. The next page is blank in case you need it.

Looking at the above code:

- There must be a `Wrapper` constructor which takes a parameter
- There must be a `getValue` method defined on `Wrapper` objects, which appears to return the value passed in the constructor
- There must be a `transform` method defined on `Wrapper` objects, which takes a function
    - This function takes one parameter and returns something
- The `transform` method appears to return another `Wrapper` object, or at least some object with a `getValue` method which works the same as that of `Wrapper`'s `getValue`
- Based on the output, the function passed to `transform` appears to be getting applied to the value passed in the constructor.

From there, we can take these notes and derive the code on the next page.

```
// There must be a Wrapper constructor which takes a parameter
function Wrapper(a) {
    this.around = a;

    // There must be a getValue method defined on Wrapper
    // objects, which appears to return the value passed in the
    // constructor

    this.getValue = function () {
        return this.around;
    };


    // - There must be a transform method defined on Wrapper
    //   objects, which takes a function
    //     - This function takes one parameter and returns
    //       something
    // - The transform method appears to return another Wrapper
    //   object, or at least some object with a getValue method
    //   which works the same as that of Wrapper's getValue
    // - Based on the output, the function passed to transform
    //   appears to be getting applied to the value passed in
    //   the constructor.

    this.transform = function (f) {
        return new Wrapper(f(this.around));
    };
}

// ALTERNATIVE SOLUTION using best practices and the prototype
Wrapper.prototype.getValue = function () {
    return this.around;
};
Wrapper.prototype.transform = function (f) {
    return new Wrapper(f(this.around));
};
function Wrapper(a) {
    this.around = a;
}
```