COMP 333 Midterm #2 Practice Exam Solutions

This is representative of the topics and kind of questions you may be asked on the exam.

Prototype-Based Inheritance in JavaScript

1.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;
}
```

1.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!"); }
}
```

1.c.) Define a method named growl for Dog objects, which prints "[dog name] growls" when called. Use Dog's **prototype**, instead of putting the method directly on Dog objects themselves. Example usage is below:

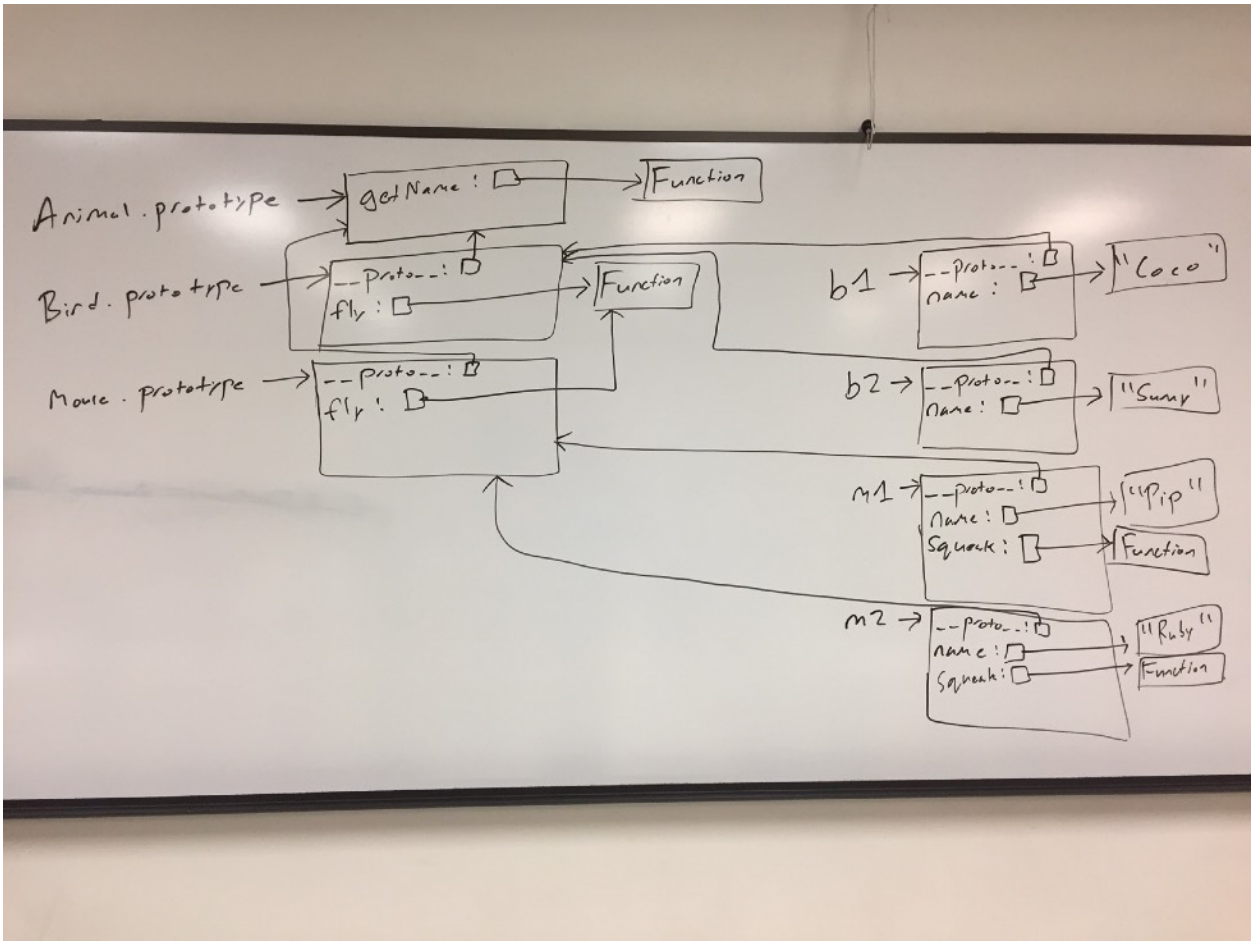
```
let d = new Dog("Rocky");
d.growl(); // prints Rocky growls
```

```
Dog.prototype.growl = function() {
  // assumes constructor initializes this.name, as with 1.a
  console.log(this.name + " growls");
}
```

2.) Consider the JavaScript code below:

```
function Animal(name) { this.name = name; }
Animal.prototype.getName = function() { return this.name; }
function Bird(name) { this.name = name; }
Bird.prototype = { '__proto__': Animal.prototype };
Bird.prototype.fly = function() {
  console.log(this.getName() + " flies");
}
function Mouse(name) {
  this.name = name;
  this.squeak = function() {
    console.log(this.name + " squeaks");
  }
}
Mouse.prototype = { '__proto__': Animal.prototype };
Mouse.prototype.fly = Bird.prototype.fly;
let b1 = new Bird("Coco"); let b2 = new Bird("Sunny");
let m1 = new Mouse("Pip"); let m2 = new Mouse("Ruby");
```

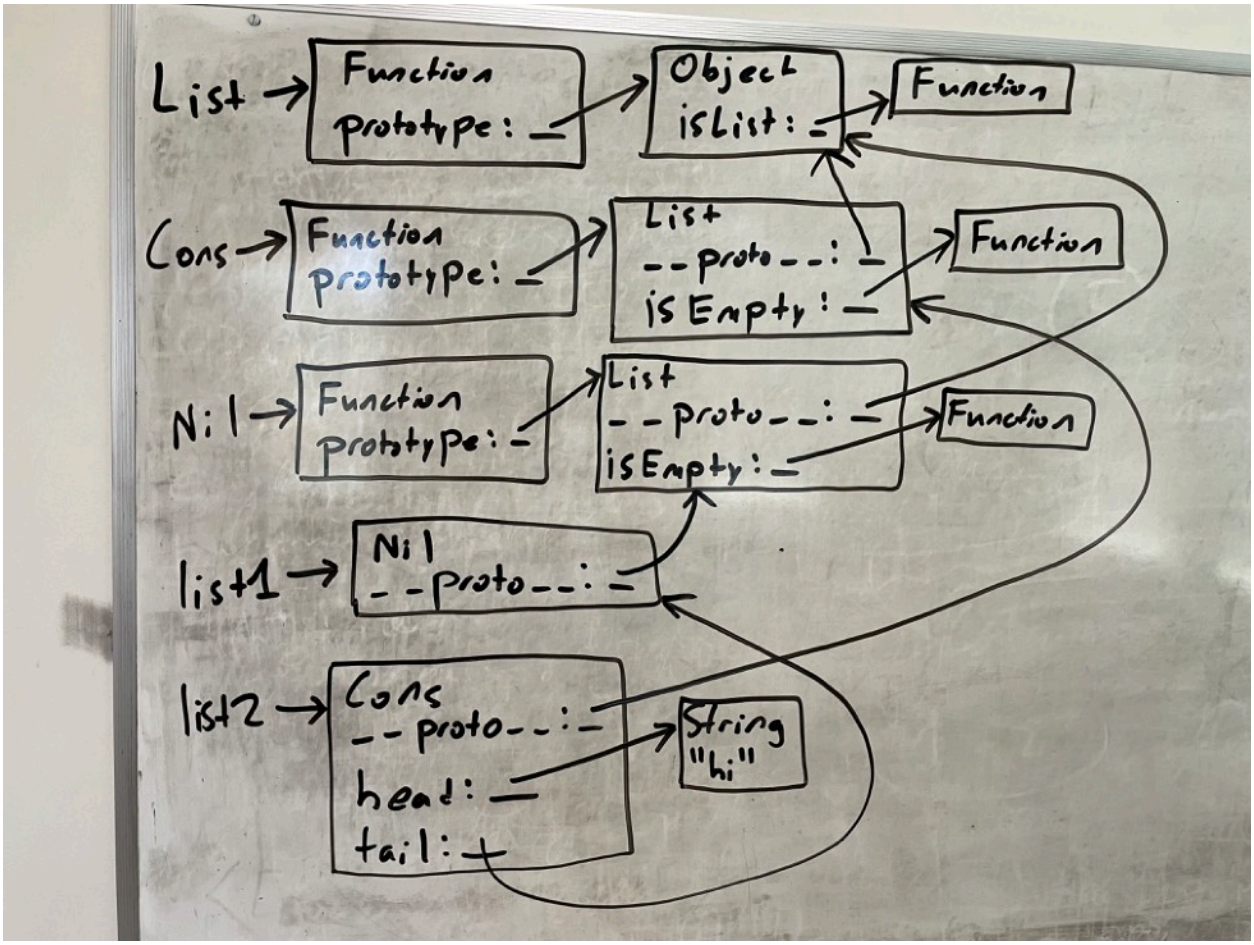
Write a memory diagram which shows how memory looks after this program executes. Your diagram should include the objects and fields associated with `b1`, `b2`, `m1`, `m2`, `Mouse.prototype`, and `Bird.prototype`, `Animal.prototype`. You do not need to show what `Animal`, `Mouse`, and `Bird` refer to.



3.) Consider the JavaScript code below, adapted from the second assignment:

```
function List() {}
List.prototype.isList = function() { return true; }
function Cons(head, tail) {
  this.head = head;
  this.tail = tail;
}
Cons.prototype = new List();
Cons.prototype.isEmpty = function() { return false; }
function Nil() {}
Nil.prototype = new List();
Nil.prototype.isEmpty = function() { return true; }
let list1 = new Nil();
let list2 = new Cons("hi", list1);
```

Write a memory diagram which shows how memory looks after this program executes. Your diagram should include the objects and fields associated with `List`, `Cons`, `Nil`, `list1`, and `list2`.



4.) Consider the test suite below, using `assertEquals` from the second assignment:

```
function test1() {
  let t1 = new Obj("foo");
  assertEquals("foo", t1.field);
}

function test2() {
  let t2 = new Obj("bar");
  assertEquals("barbar", t2.doubleField());
}

function test3() {
  let t3 = new Obj("baz");
  // hasOwnProperty returns true if the object itself has the field,
  // otherwise it returns false.  If the field is on the object's
  // prototype instead (__proto__), it returns false.
  assertEquals(false, t3.hasOwnProperty("doubleField"));
}
```

Write JavaScript code which will make the above tests pass.

```
// Object is a built-in in JavaScript, but not Obj. This requires a
// custom constructor. From test1, 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 test2, we know that we need a doubleField method on Obj
// objects.  From test3, we know that doubleField cannot be directly
// on the Obj objects, so we must put it on Obj's prototype.
Obj.prototype.doubleField = function() {
  // + in this context performs string concatenation; this
  // concatenates this.field onto itself
  return this.field + this.field;
}
```

5.) 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.

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

```
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;
}
```

6.) Consider the JavaScript code below and corresponding output, adapted from the second assignment:

```
function Cons(head, tail) {
  this.head = head;
  this.tail = tail;
}
function Nil() {}

let list = new Cons(1, new Cons(2, new Cons(3, new Nil())));
list.forEach((x) => console.log(x));

---OUTPUT---
1
2
3
```

Implement any missing code necessary to produce the above output.

Looking at the above code:

- There must be a `forEach` method defined on lists
  - This takes a parameter: a higher-order function
    - This function itself takes a parameter
    - Appears to be applying the function to each element of the list
  - Does not appear to be returning anything, at least nothing useful
  - Implementation idea: do something different for `Cons` and `Nil`

```
Cons.prototype.forEach = function (f) {
    f(this.head);
    this.tail.forEach(f);
};
Nil.prototype.forEach =  function (f) {}
```

Language Concepts

7.) In regards to memory management, Swift and Python (specifically `cpython`) both use reference counting, whereas Java and JavaScript both use garbage collection.

7.a.) In 1-3 sentences, in your own words, explain how garbage collection reclaims memory. Your description doesn't have to be detailed enough to implement a garbage collector, only detailed enough to get the gist of when memory would be reclaimed.

Starting from all variables declared on the stack (or globally), we trace through memory and determine which parts of memory are reachable or not. The parts of memory which are not reachable are reclaimed.

7.b.) In 1-3 sentences, in your own words, explain how reference counting reclaims memory. Your description doesn't have to be detailed enough to implement reference counting, only detailed enough to get the gist of when memory would be reclaimed.

Each allocated chuck of memory is given a reference count, which is incremented whenever a new reference is made to this memory, or decremented whenever an existing reference disappears. If the reference count hits zero, the memory is reclaimed.

7.c.) Name one advantage of reference counting over garbage collection.

Possible answers, among others:
- Memory is generally reclaimed as soon as it is no longer needed
- Happens at much more predictable times than garbage collection
- Application performance tends to be much more consistent than with garbage collection

7.d.) Name one advantage of garbage collection over reference counting.

Possible answers, among others:
- Can reclaim cyclic data structures without programmer intervention
- Generally better performance in terms of lower overall execution times

8.) In 1-3 sentences, explain the difference between compilation and interpretation. Your answer does not need to be detailed enough to implement a compiler or interpreter.

Compilers translate programs written in one programming language to another, whereas interpreters directly execute programs written in a given programming language.

9.) The Java Virtual Machine (JVM) is implemented as an interpreter over Java bytecode. Similarly, most JavaScript implementations are implemented as interpreters. However, most Java and JavaScript implementations support just-in-time (JIT) compilation.

9.a.) In 1-3 sentences, explain what JIT compilation does, in the context of an interpreter. Your answer doesn't need to be detailed enough to implement a JIT compiler.

The interpreter dynamically monitors code and can choose to compile chunks of frequently-executed code directly to machine code. When these chunks are executed in the future, it can jump to the compiled chunks of machine code as opposed to interpreting the corresponding original code.

9.b.) JIT compilers can sometimes generate faster code than traditional compilers. Why?

Traditional compilers can only reason about how code *might* execute, whereas JIT compilers can observe exactly how code *is* executed. As such, the JIT compiler has more information to work with, and can use this to generate faster, more optimal code.

10.) Swift, Scala, and Haskell all support type inference. In 1-3 sentences, explain what type inference is, and how it relates to statically-typed and dynamically-typed languages. You don't have to provide enough detail to implement a type inferencer.

Type inference allows a compiler for a statically-typed language to infer what the types of variables must be, without the programmer explicitly saying what the types are. The resulting code may look like dynamically-typed code (because it lacks type information), but it's still statically-typed.

Functions in Swift

11.) Consider the following code snippet in Swift:

```
let temp = addThree(first: 2, 4, third: 8)
print(temp) // prints 14
```

Implement the `addThree` function below.

```
func addThree(first: Int, _ second: Int, third: Int) -> Int {
  return first + second + third
}
```