COMP 333 Practice Exam #2 (Solutions)

This is representative of the kinds of topics and kind of questions you may be asked on the midterm. This practice exam and the handout on prototype-based inheritance in JavaScript are intended to be comprehensive of everything on the exam. That is, I will not ask anything that's not somehow covered by those sources. (I will announce the exact cutoff for the handouts and other material on Wednesday.)

You are permitted to bring two 8.5 x 11 sheets of paper into the exam with you, as long as they have handwritten notes on them. Both sides of both sheets can be used. To be clear, these must be entirely handwritten.

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:

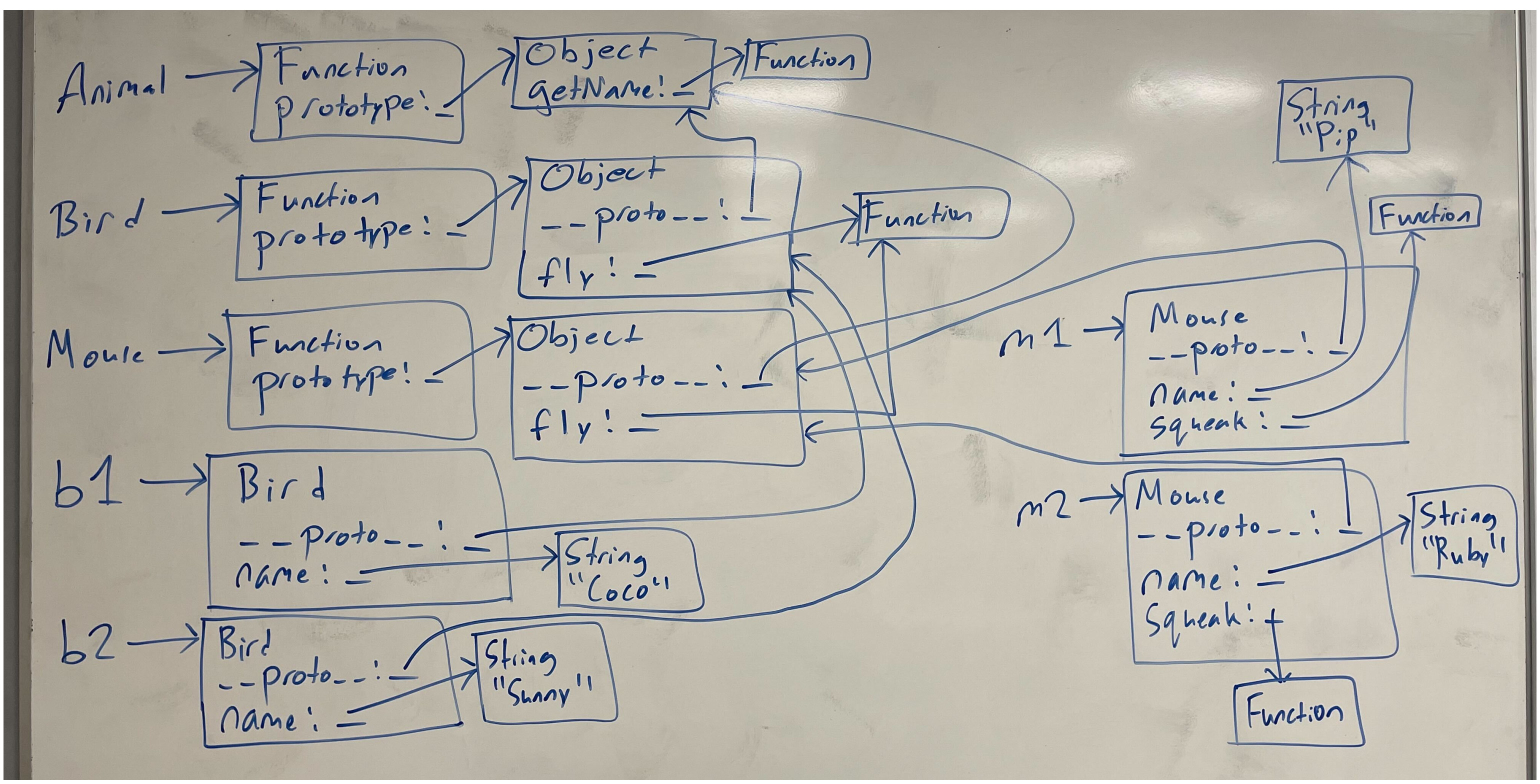
```
let d = new Dog("Rocky");
d.growl(); // prints Rocky growls
```

```
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");
```

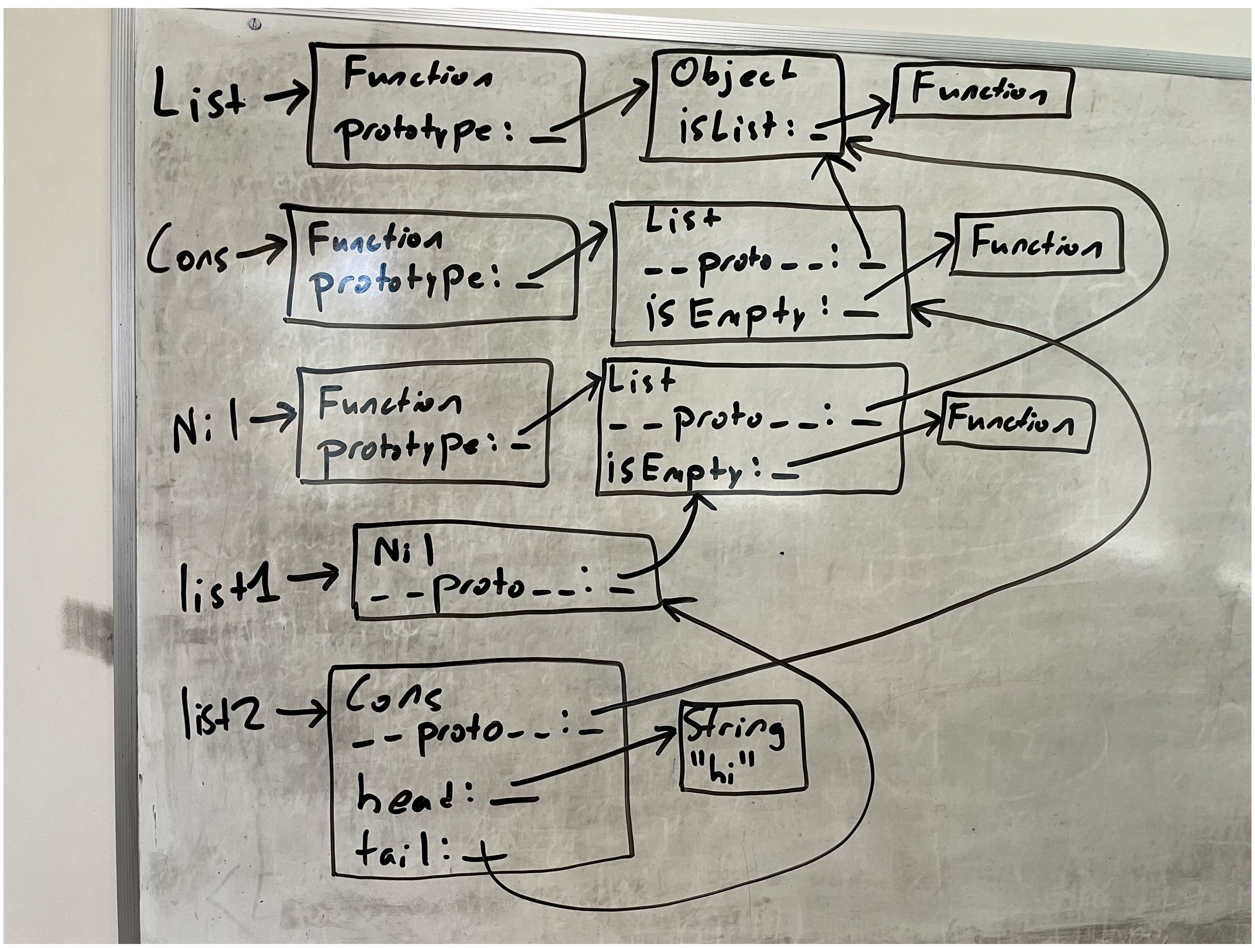
Write a memory diagram which shows how memory looks after this program executes. Your diagram should include the objects and fields associated with `b1`, `b2`, `m1`, `m2`, `Mouse`, `Bird`, and `Animal`.



3.) Consider the JavaScript code below, which implements immutable linked lists:

```
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 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);                                // output: baz
// hasOwnProperty is a built-in method which returns true if the
// object has the field directly, or false if it merely inherits
// the field.
console.log(obj3.hasOwnProperty("doubleField")); // output: false
```

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.  From obj3, 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;
}
```

Language Concepts

6.) 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.

7.) 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.

7.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.

7.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.

8.) 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.