COMP 333 Practice Exam

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

Virtual Dispatch in Java

1.) Consider the following Java code:

```
public interface I1 {
  public void doThing();
}
public class C1 implements I1 {
  public void doThing() { System.out.println("c1"); }
}
public class C2 implements I1 {
  public void doThing() { System.out.println("c2"); }
}
public class Main {
  public void makeCall(I1 value) {
    value.doThing();
  }
  public static void main(String[] args) {
    I1 t1 = new C1();
    I1 t2 = new C2();
    makeCall(t1);
    makeCall(t2);
  }
}
```

What is the output of the `main` method above?

```
c1
c2
```

2.) Consider the following code snippet:

```
public class Main {
  public static void main(String[] args) {
    Operation op1 = new AddOperation();       // line 3
    Operation op2 = new SubtractOperation(); // line 4
    int res1 = op1.doOp(5, 3);               // line 5
    int res2 = op2.doOp(5, 3);               // line 6
    System.out.println(res1); // line 7; should print 8
    System.out.pritnln(res2); // line 8; should print 5
  }
}
```

Define any interfaces and/or classes necessary to make this snippet print 8, followed by 2.

```
// From lines 3-4, we know that Operation must be a superclass of
// AddOperation and SubtractOperation, based on the types of op1
// and op2.  From line 5, we know that Operation must have a doOp
// method, that it must return an int, and that it must take two ints.
// From line 3, 5, and 7, we can infer that AddOperation's doOp must
// be adding its arguments, and similarly from lines 4, 6, and 8, we
// can infer SubtractOperation's doOp must be subtracting its
// arguments.
public interface Operation {
  public int doOp(int first, int second);
}

public class AddOperation implements Operation {
  public int doOp(int first, int second) {
    return first + second;
  }
}

public class SubtractOperation implements Operation {
  public int doOp(int first, int second) {
    return first - second;
  }
}
```

Prototype-Based Inheritance in JavaScript

3.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;
}
```

3.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!"); }
}
```

3.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:

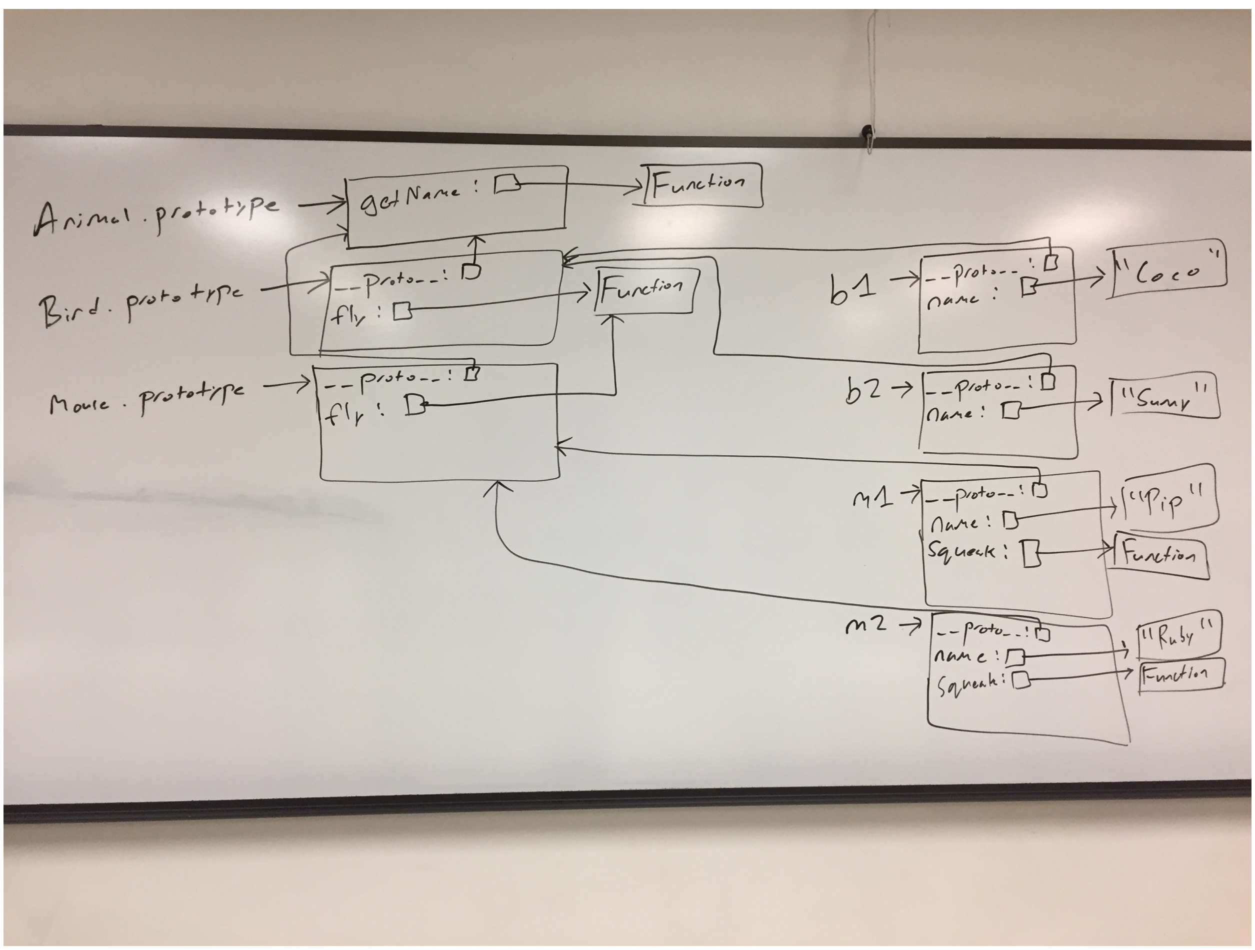
```
let d = new Dog("Rocky");
d.growl(); // prints Rocky growls
```

```
Dog.prototype.growl = function() {
  // assumes constructor initializes this.name, as with 3.a
  console.log(this.name + " growls");
}
```

4.) Consider the JavaScript code below:

```
function Animal(name) { this.name = name; }
Animal.prototype.getName = function() { return this.name; }
function Bird(name) { Animal.call(this, name); }
Bird.prototype = Object.create(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 = Object.create(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`. As a hint, the `__proto__` field on objects refers to the corresponding object's prototype.



5.) Consider the test suite below, using `assertEquals` from the first 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");
  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;
}
```

Higher-Order Functions in JavaScript

6.) Write the output of the following JavaScript code:

```
function foo(fooParam) {
  return function (innerParam) {
    return fooParam - innerParam;
  }
}

let f1 = foo(7);      // fooParam = 7 for f1
let f2 = foo(10);     // fooParam = 10 for f2
console.log(f1(2));   // innerParam = 2 for f1; 7 - 2 = 5
console.log(f2(3));   // innerParam = 3 for f2; 10 - 3 = 7
console.log(f1(4));   // innerParam = 4 for f1; 7 - 4 = 3
console.log(f2(5));   // innerParam = 5 for f2; 10 - 5 = 5
```

```
5
7
3
5
```

7.) Write the output of the following JavaScript code:

```
function guard(thing) {
  try {
    // Call the provided function and return its result.  If the
    // function throws an exception, go to the catch instead.
    return thing();
  } catch (error) {
    // If thing() threw any exceptions, then we get here and return
    // ERROR instead of whatever thing() returned
    return "ERROR";
  }
}

// Always throws an exception when called
function f() {
  throw "hello";
}

// f throws exception when called inside guard, so guard returns
// "ERROR"
console.log(guard(f));
// function passed to guard, when called, returns 42 without throwing
// an exception.  guard returns this result (42) without hitting the
// catch.
console.log(guard(function() { return 42; }));
```

```
ERROR
42
```

8.) Consider the following array definition in JavaScript:

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

8.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;
})
```

8.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()
});
```

8.c) 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 anser
arr.reduce(function (accum, element) {
  return element;
})
```

8.d) 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 (accum, element) { return accum + element }, 0)
```

Algebraic Data Structures in Swift

9.) Consider the following information:

- A `TrafficDevice` is either a `stopSign` or a `trafficLight`. A `trafficLight` is associated with a specific `LightColor`.
- A `LightColor` can be one of `red`, `yellow`, or `green`.

9.a.) Write two `enum` definitions in Swift which represent this information.

```
// LightColor is a separate type, and it can take on the values of
// red, yellow, or green
enum LightColor {
  case red
  case yellow
  case green
}

// TrafficLight is a separate type, and it can be either a stop sign
// or a traffic light.  The traffic light internally holds what color
// the light is.
enum TrafficDevice {
  case stopSign
  case trafficLight(LightColor)
}
```

9.b.) Define a **mutable** variable named `lc` which holds the `red` color. The type of the variable should be `LightColor`.

```
// Mutable variables are introduced with var.
// When making enum values, the type of the enum must be listed
// followed by a .; e.g., it's LightColor.red, as opposed to just red
var lc = LightColor.red

// alternative answer which has an explicit type annotation on lc
// instead of using type inference
var lc: LightColor = LightColor.red
```

9.c.) Define an **immutable** variable named `td` which holds a traffic light with the `green` color. The type of the variable should be `TrafficDevice`.

```
// immutable variables are introduced with let
let td = TrafficDevice.trafficLight(LightColor.green)

// alternative answer
let td: TrafficeDevice = TrafficDevice.trafficLight(LightColor.green)
```