

COMP 333 Practice Exam #1

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

Virtual Dispatch - Conceptual Understanding

1.) Name one reason why someone might want to use virtual dispatch.

2.) Name one reason why someone might **not** want to use virtual dispatch.

Virtual Dispatch in Java

3.) 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?

4.) 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 2  
    }  
}
```

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

5.) Consider the following Java code, which simulates a lock which can be either locked or unlocked. The lock is an immutable data structure, so locking or unlocking returns a new lock in an appropriate state.

```
public class Lock {
    private final boolean locked;

    public Lock(final boolean locked) {
        this.locked = locked;
    }

    public Lock unlock() {
        if (locked) {
            System.out.println("lock unlocked");
            return new Lock(false);
        } else {
            System.out.println("lock already unlocked");
            return this;
        }
    }

    public Lock lock() {
        if (!locked) {
            System.out.println("lock locked");
            return new Lock(true);
        } else {
            System.out.println("lock already locked");
            return this;
        }
    }

    public boolean isLocked() {
        return locked;
    }
}
```

Refactor this code to use virtual dispatch, instead of using if/else. As a hint, you should have a base class/interface for Lock, and subclasses for locked and unlocked locks. (Continued on to next page)

Types

6.) The code below does not compile. Why?

```
public interface MyInterface {  
    public void foo();  
}  
  
public class MyClass extends MyInterface {  
    public void foo() {}  
    public void bar() {}  
  
    public static void main(String[] args) {  
        MyInterface a = new MyClass();  
        a.bar();  
    }  
}
```

7.) Java supports subtyping. Write a Java code snippet that compiles and uses subtyping.

8.) Name one reason why someone might prefer static typing over dynamic typing.

9.) Name one reason why someone might prefer dynamic typing over static typing.

9.) Name one reason why someone might prefer strong typing over weak typing.

10.) Name one reason why someone might prefer weak typing over strong typing.

Higher-Order Functions in JavaScript

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

```
function foo(fooParam) {  
  return function (innerParam) {  
    return fooParam - innerParam;  
  }  
}  
  
let f1 = foo(7);  
let f2 = foo(10);  
console.log(f1(2));  
console.log(f2(3));  
console.log(f1(4));  
console.log(f2(5));
```

12.) Consider the following JavaScript code:

```
function base() {  
    return function (f) {};  
}  
  
function rec(n) {  
    return function (f) {  
        f();  
        n(f);  
    }  
}  
  
function empty() {}  
  
let f1 = rec(rec(base));  
let f2 = rec(rec(rec(base)));  
f1(empty);  
f2(empty);
```

How many times is `empty` called in total in the above code?

13.) Consider the following JavaScript code with corresponding output, which calls an unseen function called `mystery`:

```
function output() {  
    console.log("foo");  
}  
  
let f1 = mystery(output);  
f1();  
console.log();  
  
let f2 = mystery(f1);  
f2();  
console.log();  
  
let f3 = mystery(f2);  
f3();  
console.log();
```

Output:

foo
foo

foo
foo
foo
foo

foo
foo
foo
foo
foo
foo
foo
foo

Define the `mystery` function below.

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

```
function cap(min, max, wrapped) {  
  return function (param) {  
    let temp = wrapped(param);  
    if (temp < min) {  
      return min;  
    } else if (temp > max) {  
      return max;  
    } else {  
      return temp;  
    }  
  };  
}  
  
function addTen(param) {  
  return param + 10;  
}  
  
function subTen(param) {  
  return param - 10;  
}  
  
let f1 = cap(0, 10, addTen);  
let f2 = cap(0, 100, addTen);  
let f3 = cap(0, 10, subTen);  
let f4 = cap(0, 100, subTen);  
  
console.log(f1(0));  
console.log(f1(5));  
console.log();  
  
console.log(f2(0));  
console.log(f2(5));  
console.log();  
  
console.log(f3(0));  
console.log(f3(5));  
console.log();  
  
console.log(f4(0));  
console.log(f4(5));  
console.log();
```

15.) Consider the following JavaScript code and output:

```
console.log(
  ifNotNull(1 + 1,
    a => ifNotNull(2 + 2,
      b => a + b)))
console.log(
  ifNotNull(7,
    function (e) {
      console.log(e);
      return ifNotNull(null,
        function (f) {
          console.log(f);
          return 8;
        })
    })
  ));
```

Output:

```
6
7
null
```

`ifNotNull` takes two parameters:

1. Some arbitrary value, which might be `null`
2. A function. This function is called with the arbitrary value if the value is not `null`, and the result of the function is returned. If the value is `null`, this function isn't called, and `null` is returned instead.

Define the `ifNotNull` function below, so that the output above is produced.

16.) Consider the following array definition in JavaScript:

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

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

16.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.

16.c) Use `reduce` to get the last element in `arr`.

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

Prototype-Based Inheritance in JavaScript

NOTE: This depends on how far we get by Tuesday, and might not be covered (see Canvas at that point)

17.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
```

17.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!
```

17.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
```

18.) 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.

19.) 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.