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.

Non-exhaustive list of possibilities:

- Allows for the same method call to have different behaviors at runtime, which is more flexible
- Allows for abstraction over code behavior. Calling code only needs to know the signature of a method, and the runtime behavior can be dynamically changed by changing the underlying object the method is called on.
- Improves code modularity. Each distinct behavior can be isolated from each other behavior, and behaviors generally don't need to know about each other.

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

Non-exhaustive list of possibilities:

- Code can become more bloated. Usually, multiple classes have to be introduced, and these all have their own boilerplate associated with them.
- Code's behavior can become less explicit, and potentially more difficult to reason about. For example, an if/else clearly indicates a condition to test and code to execute depending on the condition. However, a method call may *implicitly* do something similar.
- It generally has worse performance than an explicit if/else. (Beyond what I'd expect: behind the scenes, indirect jumps are often needed, and these can easily jump out of cached instructions. Moreover, it's difficult to predict where an indirect jump will go.)
-

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?

```
c1
c2
```

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.

```
// 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;
  }
}
```

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. Lock itself doesn't need a constructor, and you do not need to worry about maintaining compatibility with existing code that uses Lock. (Continued on to next page)

```
public interface Lock {
    public Lock unlock();
    public Lock lock();
    public boolean isLocked();
}

public class UnlockedLock implements Lock {
    public Lock unlock() {
        System.out.println("lock already unlocked");
        return this;
    }

    public Lock lock() {
        System.out.println("lock locked");
        return new LockedLock();
    }

    public boolean isLocked() {
        return false;
    }
}

public class LockedLock implements Lock {
    public Lock unlock() {
        System.out.println("lock unlocked");
        return new UnlockedLock();
    }

    public Lock lock() {
        System.out.println("lock already locked");
        return this;
    }

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

Types

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

```
public interface MyInterface {
  public void foo();
}

public class MyClass implements MyInterface {
  public void foo() {}
  public void bar() {}

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

The bar method is only available on MyClass, not MyInterface. a is of type MyInterface, so the bar method is not available, even though the runtime type of a will be MyClass (which does have bar).

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

Object obj = "foo";

("foo" is of type String, and String is a subtype of Object, so values of type String can be assigned to variables of type Object.)

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

Possible answers:
- Can help eliminate programming errors at compile time
- Can help in structuring code (types can be a guide for program development)
- Compilers and runtimes can usually optimize code better with type information

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

Possible answers:
- No need to annotate programs with types, which are a significant amount of code
- More programs are possible, and can fundamentally represent things which cannot be represented in statically-typed code.

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

Possible answers:
- Strongly-typed languages are more predictable, especially in the presence of bugs and errors
- Strongly-typed languages tend to emphasize code correctness

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

Possible answers:
- There is usually a performance benefit. For example, there is no need for runtime array bounds checking.
- Weakly-typed languages tend to be more expressive. (Beyond what I'd expect: C lets you do more than Java in terms of low-level manipulation, but it's easy to hit undefined behavior, meaning programs lose all meaning. This can happen even when the code appears to be working correctly.)

Higher-Order Functions in JavaScript

12.) 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
```

13.) 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); // calls empty twice
f2(empty); // calls empty three times
```

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

5

14.) 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.

```
function mystery(f) {
    return function() {
        f();
        f();
    };
}
```

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

```
// returns a function that will bound the output of the wrapped
// function, so the output is never less than min or greater than max
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();
```

```
10
10

10
15

0
0
```

```
0
0
```

16.) 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.

```
function ifNotNull(value, f) {
    if (value !== null) {
        return f(value);
    } else {
        return value;
    }
}
```

17.) Consider the following array definition in JavaScript:

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

17.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;
})
```

17.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()
});
```

17.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;
})
```

17.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)
```