Introduction

This document is meant to provide you supplementary practice questions for the upcoming midterm. It reflects all material that you will have already seen in labs and lecture. Do not use this as a “be all end all” guide! It is still highly recommended that you review previous and external course material.

For example, I suggest that you do many practice midterms from previous semesters. Use the midterms a heuristic for your knowledge of the material, then use the lab guides and textbook to relearn the material.

Finally, make sure you don’t stress! This class is hard and will only be made harder if you stress yourself out. But you smart, you loyal #blessup. Ask lots of questions and let us the TA’s help you (believe me when I say we want you guys to succeed).

If you find yourself beginning to panic, simply pause, breathe, and believe. In the off chance you don’t believe in yourself, then believe in me who believes in you.

Introduction

Know Your Java

  This Code Sucks!
  What Type is This?
  Functions as Objects
  DBCCCCCCCCCCC
  Pox and Bointers

Arrays

  Easy Mode
  Medium Mode
  Hard Mode

Linked Lists

  Easy Mode
  Medium Mode

Abstract Interfaces

  Easy Mode
  Medium Mode
Know Your Java

This Code Sucks!

This code sucks... no seriously some of the below code examples are broken. For this question, we will be looking at the following classes.

```java
public class A {
    private int valA = 2;
    public void f() {
        this.g();
    }
    public void g() {
        System.out.println("A:" + valA);
    }
    public int h() {
        return valA;
    }
    public static A createA() {
        return new A();
    }
}

public class B extends A {
    private int valB = 15;
    protected int value = 1337;
    public void g() {
        System.out.println("h:" + h() + " z:" + valB);
    }
    public void banana() {
        System.out.println("no");
    }
}

public class C extends B {
    private int valC = 42;
    public void f() {
        this.g();
    }
}
```

Beneath are a number of coding examples. Next to each, determine if the code can run. If it can, write down the output, else state what type of error (compile-time vs runtime) is thrown and for what reason.
// to be placed in class A
public static void main(String[] args) {
    A this = new A();
    this.g();
}

// to be placed in class A
public static void main(String[] args) {
    A thing = this.createA();
    thing.f();
}

// to be placed in class A
// what is this?
public A(int valA) {
    this.valA = valA;
}

// to be placed in class A
public static void main(String[] args) {
    A thingC = new C();
    System.out.println(thingC.valC);
}

// to be placed in class A
public static void main(String[] args) {
    A thingB = new B();
    thingB.banana();
}

// to be placed in class A
public static void main(String[] args) {
    A thingB = new B();
    thingB.g();
}
// to be placed in class B
public static void main(String[] args) {
    A thingB = new B();
    System.out.println(thingB.valA);
}

// to be placed in class B
public static void main(String[] args) {
    B thingB = new B();
    System.out.println(thingB.h());
}

// to be placed in class B
public static void main(String[] args) {
    System.out.println(this.valB);
}

// to be placed in class B
public static void main(String[] args) {
    System.out.println(B.valB);
}

// to be placed in class C
public static void main(String[] args) {
    C thingC = new C();
    System.out.println(thingC.value);
}

// to be placed in class C
public static void main(String[] args) {
    C thingC = new A();
    System.out.println(thingC.valC);
}
What Type is This?

Look at the code blocks below. In the box besides it, write down the highlighted variables static and dynamic type and then write down what the statement would execute to.

```
public class Main {
    public static void method(B arg1) {
        System.out.println(arg1.g());
    }
    public static void main(String[] args) {
        C thingC = new C();
        method(thingC);
    }
}
```

```
public class Main {
    public static void method(A arg1) {
        C arg2 = (C) arg1;
        arg2.f();
    }
    public static void main(String[] args) {
        C thingC = new C();
        method(thingC);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        A thingC = new C();
        thingC.f();
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        A thing = new B();
        B thingy = (B) thing;
        thingy.banana();
    }
}
```
Functions as Objects

Unlike Python, Java doesn’t treat functions as first class objects. This means that we can’t simply pass methods into each other in a functional manner. However, we can employ object orientation to get around this. Consider the following class

```java
/** IntFunction is an object representing functions that return ints. */
public class IntFunction {
    // an instance variable that represents an incoming argument.
    int arg;
    public void setArg(int arg) {
        this.arg = arg;
    }
    public int apply() {
        return -1;
    }
}
```

We can consider IntFunction as a base class for all functions that accepts an int and returns an int. The actual functionality of the method we wish to represent would go into the instance method apply. By default, apply returns -1. To produce more complicated behavior (such as methods that accept arguments) we can simply extend IntFunction. Consider SquareFunction, a method that squares an input argument.

```java
/** SquareFunction represents a functions that squares ints. */
public class SquareFunction extends IntFunction {
    public int apply() {
        return arg * arg;
    }
}
```

And finally, we can now apply it in a functional manner!

```java
public static void main(String[] args) {
    int[] inputs = {1, 2, 3, 4, 5};
    SquareFunction sf = new SquareFunction();
    for (int i = 0; i < inputs.length; i += 1) {
        sf.setArg(inputs[i]);
        inputs[i] = sf.apply();
    }
    // now inputs = {1, 4, 9, 16, 25};
}
```
1) Define an object representing the isMultipleOf function. That is your object should be able to take in an integers and N, returning the number if it is a multiple of N else returning -1.

```java
public class ________________ {
    // what do I need as instance variables?

    // wait there's no return type...
    public _____________(________) {
        __________;
    }

    public void __________(________) {
        __________;
    }

    // what else do I need?
}
```
2) Define the map function in this main class. The map function should take in any functional object and apply it to each element of an int list. **NOTE** for easy-mode write map iteratively, for **hard-mode** write map recursively

```java
public class Main {

    public static void easyMap(__________, __________) {
        for (_________________________ ) {
            ______________________;
            ______________________;
        }
    }

    public static void hardMap(__________, __________) {
        ______________________;
    }

    // helper functions may go here.

}
```
Pox and Bointers

The before picture shows the state of an IntList object and one local variable. In the lines provided, write the Java code statements that transforms the before picture to the after, subject to the following restrictions. (1) do not modify the first instance variable and (2) do not introduce any new variables.

```

```
Arrays

Easy Mode

These questions test basic understanding of manipulating arrays and such.

**Hella Deep Copy** Given a 3D array, make a deep copy of the array. That is return a new 3D array that contains the same elements as the previous. Remember the array may be ragged!

```java
public static int[][][] hellaDeepCopy(int[][][] array) {
    // Implementation...
}
```

**Print All** Given a 2D array, print all the elements. Remember it's possible that the array is ragged! Make sure to line break each int[] iterated through.

```java
public static void printAll(int[][] array) {
    // Implementation...
}
```
Medium Mode

This section provides questions that should match your understanding of arrays after labs.

Iterative Fibonacci Array Given a number \( N \) return an array of arrays such that the \( i \)th array contains all fibonacci numbers up to the \( i \)th one. For example if the \( N=4 \), the return result would be \([[1], [1, 1], [1, 1, 2], [1, 1, 2, 3]]\).

```java
public static int[][] fibonacciArray(int n) {
    int[][] result = ______________;
    for (int i = ____; _________; ________) {
        result[i] = ______________;
        for (int j = ____; _________; ________) {
            if (_________ | _________) {
                result[i][j] = 1;
            } else {
                result[...][...] = ______________;
            }
        }
    }
    return result;
}
```

IntList to Array convert an IntList to an array.

```java
// assume that IntList has a size() method returning the length of the list
public static int[] toArray(IntList list) {
    int[] __________ = __________;
    toArrayHelper(________, ________, ________);
    return __________;
}

public static void toArrayHelper(int index, int[] nList, IntList list) {
    if (list == _____) {
        ____________________________;
    } else {
        nList[_______] = ______________;
        ____________________________;
    }
}
```
Hard Mode

This last section contains non-trivial, purposefully obfuscated applications of arrays. Try your best!

**Recursive Fibonacci Array** Do the fibonacci array question again, but recursively!

```java
public static int[][] fibonacciArray(int n) {
    int[][] results = __________________;
    ___________________________
    return results;
}

public static void fibArrayHelper1(int[][] something, int k) {
    if (____________________) {
        something____ = __________________________;
        __________________________;
        __________________________;
    }
}

public static void fibArrayHelper2(int[] array, int k) {
    if (________________) {
        array[index] = 1;
    } else if (k == 1) {
        fibArrayHelper2(array, k - 1);
        array[index] = 1;
    } else {
        __________________________;
        __________________________;
        array[index] = __________________;
    }
}
```
Linked Lists

Easy Mode

The next couple of questions will have some linked list questions. The first few will focus on basic understanding of linked lists.

Deep Copy given an IntList, return an exact copy of that linked list.

```java
public static IntList deepCopy(IntList list) {

}

// in case you want helper methods
```

Find Max Given a IntList return the max element of that list. If the list is empty, return -1.

```java
public static int maxElement(IntList list) {

}
```
Medium Mode

A few questions that should match on par understanding after lab and practice

**Copy Odd** Given an IntList, non-destructively remove all even elements.

```java
public static IntList copyOdd(IntList list) {

// in case you want helper methods
```
Abstract Interfaces

Easy Mode

Warm-up Questions

1) What are the differences between abstract classes and interfaces? Why might we wish to use one over the other?

2) Why is it okay to write `List<T> list = new ArrayList<>()` but not `ArrayList<T> list = new List<T>()`?
Medium Mode

Computation Network

A simple computational network is composed of value queues (represented by squares in the example below) and computation nodes (circles in the example below). Both value queues and computation nodes have inputs and outputs. Integer values come into each value queue, which accumulates them until the computation node on its output indicates there are enough values for its computation. At that point, the value queue sends its accumulated values to its computation node, which computes a value that is sent to another value queue. In the example below, ‘+’ nodes wait for two inputs and compute their sum; ‘×’ nodes wait for two inputs and compute their product; and ‘P’ nodes wait for one input and print it (producing nothing):

We’ll represent value queues with the class ValueQueue and describe computations with an interface called Computation. Here is ValueQueue:

```java
import java.util.ArrayList;

public class ValueQueue {

    private Computation sink;
    private ArrayList<Integer> queue = new ArrayList<>();

    public void attach(Computation sink) {
        this.sink = sink;
    }

    public void accept(Integer value) {
        queue.add(value);
        if (sink.enabled(queue.size())) {
            sink.consume(queue);
            queue.clear();
        }
    }
}
```
And the code to set up the above network and inputs.

```java
ValueQueue Q1 = new ValueQueue(), Q2 = new ValueQueue()
   Q3 = new ValueQueue(), Q4 = new ValueQueue();
Computation A1 = new Adder(Q3), A2 = new Adder(Q3),
   A3 = new Multiplier(Q4), A4 = new Printer();
Q1.attach(A1); Q2.attach(A2);
Q3.attach(A3); Q4.attach(A4);
Q1.accept(1); Q2.accept(3);
Q2.accept(6); Q1.accept(5);
```

Working backwards from the code, fill in a suitable definition for the Computation interface and the Adder class

```java
public interface Computation

public class Adder
```
There is no need to be upset.