Names: _____

In this lab, we explore the *semantics* of objects in the context of Java. That is, we try to integrate the object—this programming entity with state and behavior—into our mental model of computation. By doing so, we can answer some critical questions about how programs with objects execute in certain unintuitive settings as well as explain some bits of Java syntax that we have glossed over until this point.

Please answer the following questions below in the space provided as we discuss them during class. These are all questions that have "obvious" book answers, but they require a bit of effort on your part to *internalize* what they mean with respect to your programs. Try to (a) answer the question in your own words and (b) give an example illustrating your answer, *e.g.*, example code snippets demonstrating the differences between two different situations.

Problem 1: Value versus Reference Semantics What is the difference in calling the following three `change` methods and why is this the case?

```
public class Cell {
    public int x;
    public Cell(int x) { this.x = x; }
}
public void change1(int x) { x = 5; }
public void change2(Cell c) { c.x = 5; }
public void change3(Cell c) {
    c.x = 5;
    c = new Cell(0);
}
```

What's the rule here? Does java pass parameters by value (*i.e.*, copy) or reference? Is passing an object (with an arbitrary number of fields) to a function more costly than passing a primitive?

Problem 2: The `this` Variable What is the `this` variable in a method and where does it come from? How do these two code classes differ with respect to their `increment` methods?

```
public class Counter1 {
    public int value;
    public void increment() {
        value += 1;
    }
}

public class Counter2 {
    public int value;
    public void increment(int value) {
        value += value;
    }
}
```

What's the rule for variable look-up in Java? How does this differ from regular, old function calls in C?

Without specified by "this", the "value" variable points to the closest definition of "value". Thus, in Counter1, "value" points to the field "value" of Counter1, but in Counter2, "value" points to the parameter "value" of "increment" and the addition does not change the state of the object.

Problem 3: `static` Versus Non-`static` Members What is the distinction between a `static` and non-`static` member (*i.e.*, field or method)? In particular, what does this code do and why?

```
public class TestObject {
    public static int value;
    public TestObject() { value += 1; }
}
```

And why does this code not work? How do you fix it? In general, what is the rule for mixing `static` and non-`static` things?

```
public class Test {
    public void printGreeting() { System.out.println("Hello World!"); }
    public static void main(String[] args) { printGreeting(); }
}
```

Problem 4: Reference Versus Structural Equality Does the following code snippet behave as you expect? Why? How do you fix its behavior?

```
public class Counter {
    public int value;
    public void Counter() { this.value = 0; }
    public void increment() { this.value += 1; }
    public static void main(String[] args) {
        Counter c1 = new Counter();
        Counter c2 = new Counter();
        System.out.println("Are c1 and c2 equal? " + c1 == c2);
    }
}
```

False

With this in mind, does this code behave as you expect?

```
String s1 = "hello";
String s2 = "hello";
System.out.println(s1 == s2);
```

True

How about this snippet? What's the difference between these two snippets and why?

```
Scanner in = new Scanner(System.in);
String s1 = in.readLine();
String s2 = in.readLine();
System.out.println(s1 == s2);
```

False

"The method String.intern() can be used to ensure that equal strings have equal references. String constants are interned, so s1 and s2 will reference the same string. String.toString() simply returns itself, that is, a.toString() returns a, when a is a String. So, s2 also == s3.
In general, strings should not be compared by reference equality, but by value equality, using equals(). The reason is that it's easy to get two strings that are equivalent but different references. For example, when creating substrings. An exception to this rule is that if you know both strings have been interned beforehand (or you intern them as part of the comparison.)
To answer your implied question about heap or stack, Strings are allocated on the heap. Even if they were allocated on the stack, such as with the upcoming escape analysis and stack allocation, the semantics of the program will not change, and you will get the same result for both heap and stack allocation."
(from stackoverflow)