CSM 61B Fall 2017

# Objects & Arrays

Mentoring 3: September 11, 2017

## 1 Switcheroo

The **Golden Rule of Equals** says:

> Given variables `b` and `a`, the assignment statement `b = a` copies all the bits from `a` into `b`.

Passing parameters obeys the same rule: copy the bits to the new scope.

1.1 What is wrong with this definition of `swap`? How can we fix it?

```
class SimpleSwap {
    public static void swap(int a, int b) {
        int temp = b;
        b = a;
        a = temp;
    }
    public static void main(String[] args) {
        int x = 2, y = 5;
        System.out.println("x: " + x + ", y: " + y);
        swap(x, y);
        System.out.println("x: " + x + ", y: " + y);
    }
}
```

```
x: 2, y: 5
x: 2, y: 5
```

In the main method, `x` and `y` won't actually be swapped. Within `swap`, we can change what `a` and `b` point to, but we can't change the variables that were declared in `main`. We can fix this by either in-lining the `swap` functionality in the `main` method or returning and reassigning the swapped values using an object.

**Meta:** Emphasize "Golden Rule of Equals" (pass-by-value). The bits are copied over. It is helpful to talk about the 8 types of primitive variable types (**`byte`**, **`short`**, **`int`**, **`long`**, **`float`**, **`double`**, **`boolean`**, **`char`**), and that Java stores the actual value in the variable. So when a primitive variable is passed to a function, its value itself is copied over.

1.2 How is this implementation of `swap` different?

```
class Coordinate {
    int x, y;
    Coordinate(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class SwapObject {
    public static void swap(Coordinate p) {
        int temp = p.x;
        p.x = p.y;
        p.y = temp;
    }
    public static void main(String[] args) {
        Coordinate p = new Coordinate(2, 5);
        System.out.println("p.x: " + p.x + ", p.y: " + p.y);
        swap(p);
        System.out.println("p.x: " + p.x + ", p.y: " + p.y);
    }
}
```

When calling `swap` with a `Coordinate` object, we're passing a reference to the original `Coordinate` object. The object's instance variables can be changed from within `swap` and will remain changed after we exit from the function.

**Meta:** It can be useful to mention that Java stores only the address of an object in a variable of non-primitive type, and so when passed to a function, the address of the original object gets passed around.

# 2 Flatter Me

**Arrays** are ordered sequences of fixed length. Unlike Python lists, the `length` must be known when creating an array.

```
int[] a = new int[3];
```

It is possible to initialize and fill an array in a single expression.

```
int[] b = new int[]{1, 2, 3};
```

Java can infer the type of the array from its context, yielding this shorthand.

```
int[] c = {1, 2, 3};
```

Uninitialized values have a default value like `0`, **`false`**, or **`null`**.

```
String[] c = new String[1];
c[0] == null;
```

2.1 Implement `middle`, which takes in **`int[]`** and returns the middle element. If no element is in the exact middle, return the element to the left middle.

```
public static int middle(int[] data) {
    return data[(data.length - 1) / 2];
}
```

**Meta:** It is helpful here to review the fact that length is a field and not a method.

2.2 Write a method `flatten` that takes in a two-dimensional array `data` and returns a one-dimensional array that contains all of the arrays in `data` concatenated together.

```
public static int[] flatten(int[][] data) {
    int size = 0;
    for (int[] row : data) {
        size += row.length;
    }
    int[] result = new int[size];
    int i = 0;
    for (int[] row : data) {
        for (int value : row) {
            result[i] = value;
            i += 1;
        }
    }
    return result;
}
```

**Meta:** This may be one of the first times students see this kind of a for loop syntax, so it will be helpful to review it. Additionally, if students are stuck, one starting point is to figure out what the size of the new flattened array should be. Also, a discussion on why the array size needs to be determined at the beginning is in order since in Python, one can use the .append method to dynamically change the size of the array.

# 3 Pony DeTails

Loosely speaking, Java obeys the following rules for **variable lookup**:

1. Look in the local scope. In general, curly braces `{}` define a scope.
2. Look in the instance and class.
3. Look in the superclass.

3.1 What would Java display?

```
public class Pony {
    String name;
    int age;
    public Pony(String s, int a) {
        name = s;
        age = a;
    }
    public void getDeTails() {
        String name = "getInfo";
        System.out.println("My name is " + name);
        System.out.println("My age is " + age);
    }
    public static void main(String[] args) {
        Pony pony = new Pony("Jerry", 300);
        pony.getDeTails();
    }
}
```

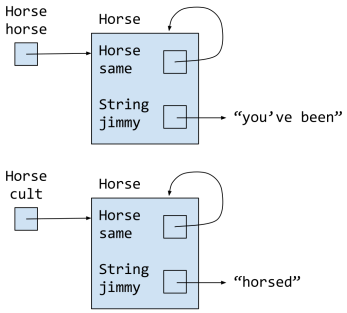
```
My name is getInfo
My age is 300
```

# 4 Samehorse *Extra Practice Midterm Question*

4.1 What would Java display? Draw a box-and-pointer diagram to find out!

```
public class Horse {
    Horse same;
    String jimmy;
    public Horse(String lee) {
        jimmy = lee;
    }
    public Horse same(Horse horse) {
        if (same != null) {
            Horse same = horse;
            same.same = horse;
            same = horse.same;
        }
        return same.same;
    }
    public static void main(String[] args) {
        Horse horse = new Horse("you've been");
        Horse cult = new Horse("horsed");
        cult.same = cult;
        cult = cult.same(horse);
        System.out.println(cult.jimmy);
        System.out.println(horse.jimmy);
    }
}
```

horsed
you've been




**Meta:** One good way to teach this question is by drawing out an environment diagram, and very carefully figuring out what needs to be assigned to what. The purpose of the question is to realize that Java will first look inside the local scope to find a variable and then in the instance/class scope.