CSM Week 1 Review

Pass by Value: works with any primitive type (int, float, char, boolean, double, long, short, byte); copies the actual value
Pass by Reference: works for objects such as a String or int[] (really anything that is not a primitive type); copies the reference to the object instead of the object itself

Method Signature: chosen by THE COMPILER (which looks at the static type of an object only); not affected during runtime when the dynamic type is being considered!
The method signature is chosen based on the **method name**, **parameter types,** and **number of parameters.**

A common misconception is that the return type will change the method signature. However, the scenario below will result in a ***compile time error*** because the method signature of the two methods are exactly the same!

```
class Dog {
    void bark(Dog dog) {
        System.out.println("bark");
    }

    String bark(Dog dog) {
        return "bark";
    }
}
```

Overriding: A class and its subclass contain the same method signature.
If the dynamic type of an object is Poodle and the bark(Dog dog) method is called, then the method in Poodle will *override* the one in Dog.

```
class Dog {
    void bark(Dog dog) {
        System.out.println("bark");
    }
}

class Poodle extends Dog {
    @Override
```

```
    void bark(Dog dog) {
        System.out.println("woof");
    }
}
```

Note: The *@Override* symbol is nice to have, but not necessary to pass the compiler.

What happens if we have an object with static type Dog and dynamic type Poodle calling bark, but Poodle does not have the method bark? The bark method in the Dog class will be called.

Overloading: Two methods in the same class have different parameter types; which method is chosen is based on the method signature.

```
class Dog {
    void bark(Dog dog) {
        System.out.println("bark");
    }

    void bark() {
        System.out.println("woof")
    }

}
```

Arrays

Java arrays are different from Python arrays. You cannot change their size after initializing them. Whenever you approach a coding question involving an array, think about how you would determine the size first.

Different ways to intialize an array:

```
int[] arr1 = new int[3];
int[] arr2 = new int[]{3, 3, 3};
int[] arr3 = {3, 3, 3};
```

Ways to for loop over an array:

```
//if you need to keep track of the index
for (int i = 0; i < arr1.length; i++) {
    print(arr1[i] + " is at index " + i);
}

//for directly accessing the element
//Note: setting number equal to another int value will not modify the actual
element in the array
for (int number: arr1) {
    print(number + " is an element of arr1.")
}
```

Values of unintialized arrays based on their types:
int/short/byte/long: 0
float/double: 0.0
char: '\u0000' (represents the null character)
boolean: false
Object: null

Common mistakes:
The length of the array is a field instead of a method. Do not call *arr1.length()*. Instead, you can access the length by *arr1.length*.

Casting Clear-Ups

The purpose of casting is to trick the compiler into thinking an object's static type is different. Sometimes it succeeds....

```
//upcasting: casting to a superclass
Dog d = (Dog) new Beagle();

//downcasting: casting to a subclass
Dog d = new Beagle();
Beagle b = (Beagle) new Dog();
```

Sometimes, it does not. Casting can be responsible for compile-time errors and runtime errors as well!

Compile-Time Error: occurs when an object is casted to a class that is not in the same class hierarchy (the classes are not at all related)

```
Dog d = (Dog) new Cat();
```

In this context, the Cat is not in the same class hierarchy as Dog. The compiler notices this and complains immediately!

Runtime Error: occurs when an object passes the compiler, but is incorrectly casted to its subclass/superclass; specifically called a ClassCastException

```
Dog d = new Dog();
Beagle b = (Beagle) d;
```

In this context, the class Beagle is a subclass of Dog.
d is downcasted to Beagle and the compiler is okay with it! During runtime, Java discovers the compiler has been lied to since d is actually a Dog, not a Beagle, and throws a ClassCastException.

How would we fix this to have no error? Change the dynamic type of d to Beagle!

```
Dog d = new Beagle();
Beagle b = (Beagle) d;
```

The compiler will pass as it did for the first runtime example. However, during runtime, Java will understand that the dynamic type of d is Beagle and that we weren't lying when we casted d as a Beagle.

A Common Misconception:

```
Dog d = new Beagle();


//casts the result of calling the method
(Dog) d.bark();


//casts the class calling the method
((Dog) d).bark();
```

More Resources

CSM 61B Review Slides (For those who have trouble with casting, Slides 1-96 and 113-122 are really useful)
HKN 61B Review Slides (Slide 52 is good practice for understanding polymorphism and casting)

Midterm 1 Review Document (Supplemental Practice)
SameHorse (Great for understanding Java Scoping)
Java Gotchas

More Casting Problems:
Michael Ju's slides
Matthew Sit's slides

If you don't mind Hilfinger exams:
Fall 2016 Midterm 1 Solutions
Fall 2017 Midterm 1 Solutions