Section 1: Golden Rule of Equals, Arrays, Static and Dynamic Typing

Raymond Guo

Note: I'm a little rusty in 61B and actually relearning with y'all :). These are just my personal notes, so there may be errors in this document; if so, let me know!

Golden Rule of Equals (Q1)

In Java, there are two types of data types (of variables): - primitive: **ints, doubles, booleans, longs, chars**, bytes, shorts, floats - non-primitive (Objects): **Strings, arrays, other Objects**

> The Golden Rule of Equals: Given variables `b` and `a`, the assignment statement `b = a` copies all the bits from `a` into `b`.

What does this mean? There are intricacies: -**For *all* primitives:** Whenever you pass a primitive variable into a function as an argument, you pass a *copy* of its value into the function. This means that when you pass a primitive variable `v` into the function, if the function modifies the variable without returning it, when you exit back to main, the value of `v` still remains the same. -**For *most* Objects (see Strings!):** Whenever you pass an Object variable into a function, you pass a copy of its *reference* to the Object into the function. This is the same as copying over a pointer in an environment diagram to point to the same "stuff". This means that when you pass an array variable `v[]` into the function, if the function modifies `v` by changing `v[0]`, when you exit back to main, the value of `v[0]` has also changed. -**For Strings:** Strings are a special type of Object: immutable. This means that they cannot ever be modified. So, whenever you are modifying a String (e.g. `String a = ""; a = a + "b;`), Java is actually creating a completely new String object. This means that when you pass an String variable `s` into the function, even though you pass its reference, if the function modifies `s`, it actually generates a completely new string, so when you exit back to main, the value of `s` still remains the same. This is true for all immutable Objects!

Arrays (Q2)

One of the most common data structures in Java is an *array*. Arrays are very similar to Python lists, but the biggest difference is that their length is unchangeable. This means that when you instantiate a Java array, you *must* know it's length. (This means that you can't use `.append()`!) You must also know the type of variable that goes into that array. Here are ways to instantiate an array:

```
 int[] a1 = new int[2]; //array of length 2
a1[0] = 0;
a1[1] = 1;
```

```
 int[] a2;
a2 = new int[]{0, 1}; //array with filled in values
```

```
int[] a3 = {0, 1}; //array with filled in values
```

You can also have 2D arrays:

```
int[][] a4 = new int[2][3]; //a4 has 2 "rows" and 3 "columns"
```

Static and Dynamic Typing (Q3)

Compile-Time and Runtime

Whenever you run a Java program, there are 2 "stages", meaning that there can be two types of errors: - Compile-Time error: An issue that the Java compiler finds, usually it has to do with types and syntax issues, but there are others. - Runtime error: An issue that Java runs into after it has fully compiled your code, and is in the middle of running it. For example, array index out of bounds or divide by zero.

Static and Dynamic Types

Object variables have 2 types: static and dynamic:

```
Dog fido = new Beagle();
```

In this case, `fido` has static type `Dog` and dynamic type `Beagle`. One general rule is that**the dynamic type must be equal to or be a "sub-category" of the static type**. In this example, a `Beagle` is a `Dog`, so this is valid. The dynamic type can also be a `Dog` (`Dog fido = new Dog();`), since a `Dog` is a `Dog`. However, the dynamic type cannot be an `Animal` (`Dog fido = new Animal();`), since an `Animal` isn't necessarily a `Dog`. This will lead to a compile-time error.

Method Signature

We often use method signatures to compare if two functions are similar. We say that the*method signature* of a function is its name, the number of its arguments, the order of its arguments, and the types of its arguments. Note that return value type and the name of the arguments do *not* matter.

Combining Concepts

Let's walk through what exactly happens when you run a Java program: - Compiling: The Java compiler starts compiling the code, checking for any syntax errors. One check that it does is **static type checking**. This can be split into 2 main types of checks: - It checks that the dynamic types of all of the Objects are equal to or a "sub-category" of the static types. See above. - It checks that the static types of all of the Objects contain the method that is being called. For example, if we call `int a = 0; fido.bark(a);` somewhere in our program, the compiler checks that the `Dog` class does indeed have a method with the same method signature (the function must be called `bark` and it must take in a single `int`).

```
In addition to static type checking, the compiler also chooses the *method signature* to
```

- Runtime: At runtime, Java chooses which exact method to run. Because the method signature was already chosen at compile-time, at runtime, Java is mainly just deciding which class to run the method from. This is called **dynamic method selection**. Java first starts at the dynamic type class and sees if the previously chosen method signature is in that class (in the example above, it checks if `Beagle` has a function with the same method signature as `bark(int a);`). If so, then it runs that method, otherwise, it continues going up to the next superclass and checks, doing so until it reaches the static type class. Because we know that the static type class *must* have a method with the same method signature (this was our second check at compile-time), this ensures that Java shouldn't crash!

Note: When checking method signatures, specifically when checking the argument types, superclasses are allowed as well (sorry for being confusing lol). For example, when we're checking `Beagle` for a method signature of `bark(Beagle b);`, if `Beagle` doesn't have a method `bark(Beagle b);` but has a method `bark(Dog d);`, it will end up running `bark(Dog d);` (since a `Beagle` is a `Dog`!).

> TLDR; Method signatures are chosen at compile-time. Which class to run the method is chosen at runtime.

General workflow when doing these types of problems

1. Static type checking: Ensure that the dynamic type of the object matches its static type, meaning that it's either equal to or a sub-category of the static type. If not, compile-time error.
2. Static type checking: Check that the static type class has a function with the same method signature as the function being called. If not, compile-time error.
3. Note down the method signature.
4. Now, we enter runtime. Go to the dynamic type class and check if it has a function with the same method signature you previously noted down. If so, run it. If not, repeat this with its superclass until you reach the static type.