CS 61B Data Structures — Java

Spring 2020 Anthony Ding

1 Introduction

Welcome to CS61B! At its heart, this is a software engineering course which poses questions like: "How do we optimize this?" "How can we achieve both correctness and efficiency?" "How do we build more complex programs and have them work with each other?" Usually these questions don't have a single valid answer – some can be rather open-ended. Therefore, I recommend approaching the class with a mindset of considering different approaches and trade-offs, instead of honing in on one correct/"best" answer.

61B was one of my favorite courses at Berkeley, and I hope to make the experience more enjoyable for you through these series of notes. They're not meant to be a comprehensive textbook or anything, but a detail of some important conceptual points and quirks. I'll also throw in some exam tips and tricks, as well as common pitfalls. **In no way is this a replacement for going to lecture or discussion: I highly encourage you to do both!**

Important points that may been helpful on exams will be in red.

Solution: These will be in blue.

2 Java vs. Python

I'm assuming most of you are coming from 61A and/or have some familiarity with an interpreted language (e.g. Python). **Interpreted** means that instructions in Python are executed directly by interpreting them line by line. On the other hand, Java is a **compiled** language, which means source code is aggregated and reduced to an executable file (compilation), which is then run. I like to think of the instructions as written on sheets of paper scattered across a messy office. Python would individually find each sheet as necessary, whereas Java would first compile all the sheets together into a central location and then follow it one by one. We can infer that Java programs usually execute faster.

Java's compiling leads us to another feature: **static typing**. In Python, variables can represent any type, and so variable declarations are of the form [x = 0]. Every variable in Java has a "static type" which tells the compiler to treat them as such. For instance:

```
//Instantiates a dog
Dog doggo = new Dog();
//Assume Puppy and Samoyed are subclasses of Dog
```

```
Dog pupper = new Puppy();
//This will error during compile time!
Samoyed cloud = new Dog();
```

During compilation, Java does type checks to ensure that declarations and instantiations are logical. The second line of code is ok because a Puppy is a valid type of Dog (a Puppy IS a Dog). The third line is not because we cannot force a Dog to be a Samoyed. In this fashion, the left-side variable types ("static types") are like **labels that tell the compiler how to think of the entities on the right.** After assignment, the object pupper (that is actually a Puppy) will be treated, for all intents and purposes, as a Dog by the compiler when it checks if future method calls obey typing rules. However, during runtime, the Java Virtual Machine (JVM) will treat it as a Puppy (I'll go into further detail in a future note about Dynamic Method Selection).

The compiler also does type checks in future method calls to ensure that we are working with Dogs in the right manner. If I have a method from a Person class called feedChocolate(Person p), it will error **during compile time** if I try to call something like

```
Person alice = new Person();
doggo.feedChocolate(alice);
```

because a Dog should not be feeding a Person in our program, and it also errors for

```
Person bob = new Person();
bob.feedChocolate(doggo);
```

because a Dog is not a Person, and we would rather not feed our dog something toxic to them. (Python, meanwhile, would just force-feed the dog chocolate until something errors during interpretation.) So for complex systems that include many classes and types, static typing is a blessing: it makes code more readable and prevents logic errors. **A benefit of erroring at compile time over execution time is that you can catch stuff before running**, which is nice because you only need to compile a given program once. *Keep in mind that if you modify the source code you have to recompile.*

3 Casting

Normally, the compiler rejects any assignments where the dynamic type cannot be assigned to the static one (like assigning Dog objects to Samoyed variables). We can circumvent this with casting, as discussed in lecture. **Casting is really powerful but risky, as you're essentially telling the compiler "I guarantee you that this Dog is actually a Samoyed"** so only use it when absolutely necessary.

Usually, casting is useful when you're assigning a superclass instance to a subclass variable. Say we started off a program by writing

```
Dog topDog = new Samoyed();
```

because we would like to add it to a list of Dogs for a dog show. We do not need to cast here because a Samoyed already IS a Dog, but it is not inherently wrong (it will not error) to cast. But now, suppose we also would like to designate that topDog won best behaved samoyed in the dog show. Can we write something like this?

```
Samoyed bestBehaved = topDog;
```

It turns out that this will error **at compile time** because the compiler only knows that topDog is a dog. Yes, during the assignment it checked that new Samoyed() would actually make a type of Dog. But after that line, it "wraps" the instance in wrapping paper and sticks a Dog label on top. So now when we get to this line, it tells us "Wait a minute, a Dog might not be a Samoyed" (even though we as the programmer know it is). To remedy this, we can cast topDog to a Samoyed, but only because we can *guarantee* that it is, in fact, a Samoyed.

```
//Will not error at compile time
Samoyed bestBehaved = (Samoyed) topDog;
```

Casting does not affect the JVM/runtime behavior of the program. All it does it trick the compiler! On an exam, if a line of code is modified to include a cast, the same runtime errors will still be runtime errors – casting does nothing to solve this. Previous compile time errors could go away, but only if the cast resolves the type error of improper assignment.

4 Typing Quirks

In 61B, there will likely be questions which do a lot of instantiations and then ask you when there are errors. Usually compile time type errors have one of two main causes:

1. Assigning a superclass object to a subclass variable, or assigning one subclass object to another different subclass variable For example, if Samoyed and Retriever both extend Dog, the lines

   ```
   Samoyed badDog = new Dog();
   Samoyed wrongDog = new Retriever();
   ```

   both error, because Dogs and Retrievers are not necessarily Samoyeds. To keep track, it may be helpful to draw out a tree to represent how classes depend on each other, with subclasses child nodes of their superclasses (each line represents an "extend"). Then, when you check if an instantiation errors, you start from the dynamic type (right hand)'s node in the tree and keep going up. If you see the static type (left hand) sometime, it doesn't error.

2. Failing to cast. As in a previous example, writing

   ```
   Dog trustMeImPupper = new Puppy();
   Puppy definitelyPupper = trustMeImPupper;
   ```

   will crash the compilation. However, if you do cast but the cast is incorrect, like

```
Dog imPupper = new Retriever();
Puppy stillPupper = (Puppy) imPupper;
```

it will not go noticed by the compiler, and cause an error at runtime.

5 Primitive and Reference Types

Variable's static types in Java can be one of two main categories: primitive and reference types. **Primitive types** include ints, booleans, chars, shorts, floats, longs, doubles, and bytes. **Reference types** include mostly everything else, including any classes you declare throughout the program.

When a variable is declared, Java should allocate some amount of space. But allocating some space for the entire object can get rather tricky for Java. What if I make a list of lists of lists of lists and so on, and it can't figure out how much space to allocate? For primitive types, this is okay, because they have fixed size. But for non-primitive types, **Java allocates enough space for a "pointer"** to the object which is constructed somewhere else. The pointer consists of some number that allows Java to locate the object. This works similarly to allocating an address book, which tells you where to find someone's house. I'll get into more detail on pointers in a later note.

A key difference, due to this pointer concept, is that primitive type variables always contain **something**. Each of them have a default value (for ints it's 0, for boolean it's false). Reference type variables, however, can be NULL, a special value which indicates nothing is there (we've declared the variable but not assigned an object to it). When we try to use this variable, Java throws a NullPointerException.

6 Golden Rule of Equals

For both primitive and reference types, there is an important rule of assignment, which 61B sometimes refers to as the **Golden Rule of Equals**. When you write a line like

```
y = x;
```

The behavior is the same for primitive and reference types. In all cases, Java will **copy over the bits** of whatever the variable is assigned to. For primitive types, it will directly copy the bits of the value. For reference types, it will copy the bits of the address.

This means that when you pass reference type arguments in to be used by methods, Java simply passes the bits that represent the pointer (known as "pass by value"). *Keep in mind that when you actually modify the object, it's still modifying the real object in memory, not the pointer address. Java automatically looks up the object using the address and works with it.*

Say we execute the following program:

```
public static void doSomething(Task t) {
    t = new Task("Shopping", true);
}

public static void main(String[] args) {
    Task incomplete = new Task("Shopping", false);
    doSomething(incomplete);
}
```

What will the variable incomplete contain at the end of execution?

Solution: Incomplete will still have the original task of "Shopping" and false. The reason is that when it is passed as an argument, the address bits are copied over into t. Incomplete and t now both point to the original object. But then t is reassigned to the new object that has parameters "Shopping" and true, yet incomplete remains pointing to the old object. So incomplete will still contain the bits of the original pointer.

What about this program?

```
public static void doSomething(int b) {
    b *= 10;
}

public static void main(String[] args) {
    int a = 42;
    doSomething(a);
}
```

What will the variable a contain at the end?

Solution: When it calls doSomething(int b), Java will again copy the bits over from the primitive type a. In this case, a simply contains the **value** 42, not the pointer. This will be directly copied into b, and then b is multiplied by 10, not affecting a at all. a ends up still being 42 at the end, while b is 420.