Section 3: Casting, Bit Manipulation, and Access Control

Raymond Guo

Disclaimer: 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! Also, this isn't affilianted with CSM or any Berkeley organization :).

Casting (Q2)

In Java, casting is the process of making a variable behaves as a variable of another type. More specifically, casting forces a change of the static (compile-time) type of an expression or variable temporarily, just for that line. Casting does *not* change the dynamic type of a variable, nor does it change the static type of the variable in future lines:

```
Cat c = new Cat();
Animal a = (Animal) c; // the static type of "(Animal) c" is now Animal
c.meow(); // both the dynamic type and the static type of "c" is still Cat
```

There are 3 main rules to follow when casting: Let's say that we have the line `A a = (C) d;`

1. (Compile-Time) Type of `d` and `C` must have a parent-child or ancestor relationship (child to parent or parent to child or same type). If there is no relationship then we will get a compile error (inconvertible types).
2. (Compile-Time) `C` must be either same type or subclass of `A`; otherwise we will get a compile error (incompatible types).
3. (Runtime) Dynamic type of `d` must be same type or subclass of `C`; otherwise we will get a runtime exception (`ClassCastException Exception`).

To illustrate these 3 rules, here's a few examples:

- `Animal a = (Cat) new Cat();` Valid - you can cast to your own type.
- `Animal a = (Animal) new Cat();` Valid - this is an example of upcasting, always safe
- `Cat c = (Cat) new Animal();` Runtime Error - this is an example of downcasting; in general, downcasting leads to a runtime error if the dynamic type (`Animal`) is a superclass of the static type being casted to (`Cat`)
- `Object o = new NyanCat(); Cat c = (Cat) o;` Valid - this is also downcasting, but OK because the dynamic type (`NyanCat`) is a subclass of the static type being casted to (`Cat`)
- `Dog d = (Dog) new Cat();` Compile-Time Error - no parent-child relationship between `Dog` and `Cat`

Note: Keep in mind that even though casting only changes the "effective" static type and not the dynamic type, it can still affect which methods are being run since method signatures are chosen at compile-time! For example, consider the following code:

```
public class Test {
   public static void method(Object t) {
       system.out.println("Its an object");
   }
   public static void method(Test t) {
       system.out.println("Its a test");
   }
   public static void main(String[] args) {
       Test t = new Test();
       method(t);
```

```
        method((Object) t);
    }
}
```

Running this will lead to the output:

```
Its a test
Its an object
```

Bit Manipulation (Q2)

Bit manipulation is one of the more obscure topics inn CS61B, but it's also one of the most commonly discussed concepts in computer science overall (you'll be dealing with it a lot more in CS61C). Here's a really good guide on bit manipulation in Java: https://www.vojtechruzicka.com/bit-manipulation-java-bitwise-bit-shift-operations/

For this section, I mainly want to focus on specific "tricks" that I commonly use for midterm questions:

Finding the kth bit from the right

You can get the kth bit of an integer by masking it. This is a technique where you AND the number with a multiple of 2 (represented as a 1 followed by a certain number of 0's) in order to extract only the value of that one single bit. Then we can apply a bit-shift remove all of the unrelevant bits.

For example, if you want to get the 3rd digit from the right of a number `x`, you can do `(x & 8) >> 2`. Why does this work? Let's say that `x = 5`, which is 101 in binary. 8 is 100 in binary. When we AND 101 and 100 together, we get 100. Then we right-shift this number by 2, giving us 1. This tells us that the 3rd digit from the right of 5 is 1!

Toggling kth bit from the right

You can conveniently toggle a bit (0 -> 1, 1 -> 0) by XOR-ing it with 1. This is because O ^ 1 = 1, and 1 ^ 1 = 0. Similarly, you can preserve a bit by XOR-ing it with 0. That means that if we want to toggle the kth bit (while preserving all the other bits), we just have to XOR the number with a multiple of 2 (represented as a 1 followed by a certain number of 0's), very similar to masking (but instead of AND, we do XOR).

For example, if you want to toggle the 3rd digit from the right of a number `x`, you can do `x ^ 8`. Why does this work? Let's say that `x = 5`, which is 101 in binary. 8 is 100 in binary. When we XOR 101 and 100 together, we get 001!

Access Control (Q4)

There are four privilege levels in Java, ranging from public (the least restrictive) to private (the most restrictive). I would copy this table down on your cheatsheet!

privilege	accessible by class	accessible by package	accessible by subclass	accessible by everybody
public	yes	yes	yes	yes
protected	yes	yes	yes	no
default (no modifier)	yes	yes	no	no
private	yes	no	no	no