CS61B Lecture 10

Wednesday, February 12, 2020

Review

Examine the following code:

```java
class A {
    void f() {
        System.out.println("A.f");
    }
    void g() {
        f(); /* or this.f() */ }
    }

class B extends A {
    void f() {
        System.out.println("B.f");
    }
}

class C {
    static void main(String[] args) {
        B aB = new B();
        h(aB);
    }

    static void h(A x) {
        x.g();
    }
}
```

What gets printed?

Answer: `B.f`. `C.main` calls `h` and passes it `aB`, whose dynamic type is `B`. `h` calls `x.g()`. Since `g` is inherited by `B`, we execute the code for `g` in class `A`. `g` calls `this.f()`. Now this contains the value of `h`'s argument, whose dynamic type is `B`. Therefore, we execute the definition of `f` that is in `B`. In calls to `f`, in other words, static type is ignored in figuring out what method to call.

What if we made `g` static?

Answer: My interpreter says it's an error, but Prof. Hilfinger says no? Selection of `f` still depends on dynamic type of this. Same for overriding g in B.

What if we made `f` static?

Answer: `A.f`. Since `f` is a static method in any case, the compiler will make its decision on which method to use at the time of compilation, not when the method is called.

If we overrode `g` in `B`?

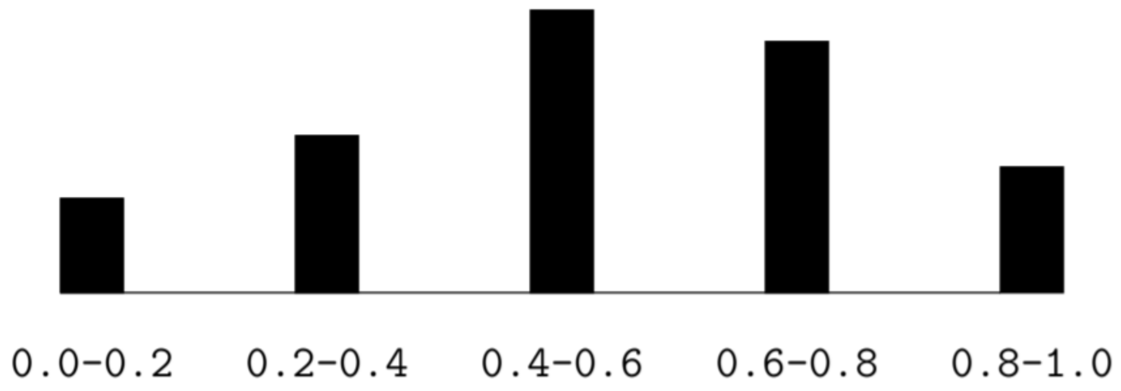
Answer: `B.f`.

What if `f` was not defined in `A`?

Answer: An error.

Designing a Class

Let's say we want a class that represents histograms like this one:



We should carefully analyze what we need from this class:

- Specify buckets and limits.
- Accumulate counts of values.
- Retrieve counts of values.
- Retrieve numbers of buckets and other initial parameters

Specification Seen by Clients

The **clients** of a module (its classes and programs, among other things) are the programs or methods that use that module's exported definitions.

In Java, the intention is that exported definitions are designated `public`. Clients are intended to rely on specifications (called APIs), and not code.

The **syntactic specifications** include method and constructor headers -- the syntax needed to use. The **semantic specifications** define what they do. There is formal notation, so use comments to do this. It is a **contract**. They usually include:

- Conditions the client must satisfy, called preconditions, marked "Pre:" in the examples below.
- Promised results, called postconditions
- Design these to be all the client needs.
- Exceptions communicate errors, specifically the failure to meet preconditions.

Histogram Specification and Use

```
/** A histogram of floating-point values */
public interface Histogram {
    /** The number of buckets in THIS. */
    int size();
    /** Lower bound of bucket #K. Pre: 0<=K<size(). */
    double low(int k);
    /** # of values in bucket #K. Pre: 0<=K<size(). */
    int count(int k);
    /** Add VAL to the histogram. */
    void add(double val);
}
```

```
void fillHistogram(Histogram H, Scanner in) {
    while (in.hasNextDouble())
    H.add(in.nextDouble());
}

void printHistogram(Histogram H) {
    for (int i = 0; i < H.size(); i += 1) {
        System.out.printf(">=%5.2f | %4d%n",
                          H.low(i), H.count(i));
    }
}
```

Sample output:

```
>=  0.00 | 10
>= 10.25 | 80
>= 20.50 | 120
>= 30.75 | 50
```

An Implementation

```
public class FixedHistogram implements Histogram {
    private double low, high; /* From constructor*/
    private int[] count; /* Value counts */
    /** A new histogram with SIZE buckets of values >= LOW and < HIGH. */
    public FixedHistogram(int size, double low, double high) {
        if (low >= high || size <= 0) {
            throw new IllegalArgumentException();
            this.low = low; this.high = high;
            this.count = new int[size];
        }
    }
    public int size() { return count.length; }
    public double low(int k) {
        return low + k * (high-low)/count.length;    }
    public int count(int k) { return count[k]; }
    public void add(double val) {
        if (val >= low && val < high) {
            count[(int) ((val-low)/(high-low) *
                        count.length)] += 1;
        }
    }
}
```

Improvements

Here's an idea: don't require a priori bounds. We will not set the lower bound and higher bound, but only give it the size.

This is a profoundly different method that requires another implementation. However, clients, like `printHistogram` and `fillHistogram` will still work with no changes, which illustrates the power of **separation of concerns**.

How to Implement

It is pointless to pre-allocate the `count` array, and since we don't know the bounds, we must save arguments to `add`. Thus, we are re-computing the `count` array "lazily" when `count(...)` is called. We then invalidate the `count` array whenever the histogram changes.

```
class FlexHistogram implements Histogram {
    private ArrayList<Double> values = new ArrayList<();
    int size;
    private int[] count;

    public FlexHistogram(int size) { this.size = size; this.count = null;
}

public void add(double x) {
    count = null;
    values.add(x);
}

public int count(int k) {
    if (count == null) {
        // compute count from values here.
    }
    return count[k];
}
```

Why do this?

By using the public method for `count` instead of making the array `count` visible, the "tiny change" is transparent to clients.

If the client had to write something like `myHist.count[k]`, it would mean: "the number of items currently in the k-th bucket of histogram `myHist` (which, by the way, is stored in an array called count in `myHist` that always holds the up-to-date count)."

The parenthetical comment is *worse than useless* to the client.

If the `count` array had been visible, after this tiny change, then every use of `count` in client program would have to change.

Using a getter method for the `count` decreases what the client **must** know and therefore has to change.