CS61B Lecture #10: OOP mechanism and Class Design
Review: A Puzzle

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(); }
}

1. What is printed?
2. If we made g static?
3. If we made f static?
4. If we overrode g in B?
5. If f not defined in A?

Choices
a. A.f
b. B.f
c. Some kind of error
Review: A Puzzle

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(); }
}

1. What is printed?
2. If we made g static?
3. If we made f static?
4. If we overrode g in B?
5. If f not defined in A?

Choices
a. A.f
b. B.f
c. Some kind of error
Review: A Puzzle

class A {
    void f() {
        System.out.println("A.f");
    }
    static void g(A y) { y.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) { A.g(x); } // x.g(x) also legal here
}

1. What is printed?
2. If we made g static?
3. If we made f static?
4. If we overrode g in B?
5. If f not defined in A?

Choices
a. A.f
b. B.f
c. Some kind of error
class A {
    void f() {
        System.out.println("A.f");
    }
    static void g(A y) { y.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) { A.g(x); } // x.g(x) also legal here
}

1. What is printed?
2. If we made g static?
3. If we made f static?
4. If we overrode g in B?
5. If f not defined in A?

Choices
a. A.f
b. B.f
c. Some kind of error
Review: A Puzzle

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

class B extends A {
    static 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(); }
}

1. What is printed?  
   Choices:  
   a. A.f  
   b. B.f  
   c. Some kind of error

2. If we made g static?  
3. If we made f static?  
4. If we overrode g in B?  
5. If f not defined in A?
Review: A Puzzle

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

class B extends A {
    static 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(); }
}

1. What is printed?
   Choices
   a. A.f
   b. B.f
   c. Some kind of error

2. If we made g static?
3. If we made f static?
4. If we overrode g in B?
5. If f not defined in A?
Review: A Puzzle

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");
    }
    void g() { f(); }
}

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

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

1. What is printed?
2. If we made g static?
3. If we made f static?
4. If we overrode g in B?
5. If f not defined in A?

 Choices
a. A.f
b. B.f
c. Some kind of error
Review: A Puzzle

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");
    }
    void g() { f(); }
}

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

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

1. What is printed?
2. If we made g static?
3. If we made f static?
4. If we overrode g in B?
5. If f not defined in A?

Choices
a. A.f
b. B.f
c. Some kind of error
class A {
    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(); }
}

1. What is printed?
2. If we made g static?
3. If we made f static?
4. If we overrode g in B?
5. If f not defined in A?

Choices
a. A.f
b. B.f
c. Some kind of error
class A {
    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(); }
}

1. What is printed?
2. If we made g static?
3. If we made f static?
4. If we overrode g in B?
5. If f not defined in A?

Choices
a. A.f
b. B.f
c. Some kind of error
Answer to Puzzle

1. Executing `java C` prints _____, because
   A. `C.main` calls `h` and passes it `aB`, whose dynamic type is `B`.
   B. `h` calls `x.g()`. Since `g` is inherited by `B`, we execute the code for `g` in class `A`.
   C. `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`.
   D. In calls to `f`, in other words, static type is ignored in figuring out what method to call.

2. If `g` were static, we see _____; selection of `f` still depends on dynamic type of `this`. Same for overriding `g` in `B`.

3. If `f` were static, would print _____ because then selection of `f` would depend on static type of `this`, which is `A`.

4. If `f` were not defined in `A`, we'd see _____
Answer to Puzzle

1. Executing `java C` prints `B.f`, because
   
   A. `C.main` calls `h` and passes it `aB`, whose dynamic type is `B`.
   
   B. `h` calls `x.g()`. Since `g` is inherited by `B`, we execute the code for `g` in class `A`.
   
   C. `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`.
   
   D. In calls to `f`, in other words, static type is ignored in figuring out what method to call.

2. If `g` were static, we see `B.f`; selection of `f` still depends on dynamic type of `this`. Same for overriding `g` in `B`.

3. If `f` were static, would print `A.f` because then selection of `f` would depend on static type of `this`, which is `A`.

4. If `f` were not defined in `A`, we’d see a compile-time error
Example: Designing a Class

Problem: Want a class that represents histograms, like this one:

```
0.0-0.2  0.2-0.4  0.4-0.6  0.6-0.8  0.8-1.0
```

Analysis: What do we need from it? At least:

- 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 (class, program, etc.) are the programs or methods that use that module’s exported definitions.

- In Java, intention is that exported definitions are designated **public**.

- Clients are intended to rely on **specifications**, (aka APIs) not code.

- **Syntactic specification:** method and constructor headers—syntax needed to use.

- **Semantic specification:** what they do. No formal notation, so use comments.
  - Semantic specification is a **contract**.
  - Conditions client must satisfy (**preconditions**, marked “Pre:” in examples below).
  - Promised results (**postconditions**).
  - Design these to be **all the client needs**!
  - Exceptions communicate errors, specifically failure to meet pre-conditions.
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);
}

Sample output:
>= 0.00 | 10
>= 10.25 | 80
>= 20.50 | 120
>= 30.75 | 50

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));
}
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;
    }
}
Let's Make a Tiny Change

Don't require *a priori* bounds:

class FlexHistogram implements Histogram {
    /** A new histogram with SIZE buckets. */
    public FlexHistogram(int size) {
        ?
    }
    // What needs to change?
}

• How would you do this? Profoundly changes implementation.
• But clients (like `printHistogram` and `fillHistogram`) still work with no changes.
• Illustrates the power of *separation of concerns*. 
Implementing the Tiny Change

- Pointless to pre-allocate the `count` array.
- Don’t know bounds, so must save arguments to `add`.
- Then recompute `count` array “lazily” when `count(···)` called.
- Invalidate `count` array whenever histogram changes.

```java
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];
    }
}
```
Advantages of Procedural Interface over Visible Fields

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

- If client had to write `myHist.count[k]`, it would mean
  
  “The number of items currently in the $k^{\text{th}}$ 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).”

- Parenthetical comment **worse than useless** to the client.

- If `count` array had been visible, after “tiny change,” every use of `count` in client program would have to change.

- So using a method for the public `count` method decreases what client *has to* know, and (therefore) has to change.