

Review: A Puzzle

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

/* or this.f() */

C {
    void main(String[] args) {
        aB = new B();
        aB.f();
    }

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

Red? **Choices**
g static? a. A.f
f static? b. B.f
de g in B? c. Some kind of error
ned in A?

Review: A Puzzle

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

A y) { y.f(); }

C {
    void main(String[] args) {
        aB = new B();
        aB.f();
    }

    void h(A x) { A.g(x); } // x.g(x) also legal here
}
```

Red? **Choices**
g static? a. A.f
f static? b. B.f
de g in B? c. Some kind of error
ned in A?

Review: A Puzzle

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

/* or this.f() */

C {
    void main(String[] args) {
        aB = new B();
        aB.f();
    }

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

Red? **Choices**
g static? a. A.f
f static? b. B.f
de g in B? c. Some kind of error
ned in A?

Lecture #10: OOP mechanism and Class Design

Review: A Puzzle

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

/* or this.f() */

C {
    void main(String[] args) {
        aB = new B();
        aB.f();
    }

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

Red? **Choices**
g static? a. A.f
f static? b. B.f
de g in B? c. Some kind of error
ned in A?

Review: A Puzzle

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

A y) { y.f(); }

C {
    void main(String[] args) {
        aB = new B();
        aB.f();
    }

    void h(A x) { A.g(x); } // x.g(x) also legal here
}
```

Red? **Choices**
g static? a. A.f
f static? b. B.f
de g in B? c. Some kind of error
ned in A?

Review: A Puzzle

```
System.out.println("A.f");  
/* or this.f() */  
class B extends A {  
    void f() {  
        System.out.println("B.f");  
    }  
    void g() { f(); }  
}  
C {  
    void main(String[] args) {  
        aB = new B();  
        aB;  
    }  
    void h(A x) { x.g(); }  
}
```

What is printed?
Is `static`?
Is `f` static?
Is `g` in `B`?
Is `h` in `A`?

27:59 2016

CS61B: Lecture #10 8

Choices

- a. A.f
- b. B.f
- c. Some kind of error

Review: A Puzzle

```
/* or this.f() */  
class B extends A {  
    void f() {  
        System.out.println("B.f");  
    }  
    void g() { f(); }  
}  
C {  
    void main(String[] args) {  
        aB = new B();  
        aB;  
    }  
    void h(A x) { x.g(); }  
}
```

What is printed?
Is `static`?
Is `f` static?
Is `g` in `B`?
Is `h` in `A`?

27:59 2016

CS61B: Lecture #10 10

Choices

- a. A.f
- b. B.f
- c. Some kind of error

Answer to Puzzle

What does `h` print, because

`h` prints `B.f` and passes it `aB`, whose dynamic type is `B`.

`g()`. Since `g` is inherited by `B`, we execute the code for `A`.

`h` calls `g`. Now `this` contains the value of `h`'s argument, whose dynamic type is `B`. Therefore, we execute the definition of `g` in `B`.

`g` calls `f`, in other words, static type is ignored in figuring out how to call.

What is printed? `static`, we see `static`; selection of `f` still depends on dynamic type of `this`. Same for overriding `g` in `B`.

What is printed, would print `B.f` because then selection of `f` is based on static type of `this`, which is `A`.

What is defined in `A`, we'd see `A.f`.

27:59 2016

CS61B: Lecture #10 12

Review: A Puzzle

```
System.out.println("A.f");  
/* or this.f() */  
class B extends A {  
    static void f() {  
        System.out.println("B.f");  
    }  
    void g() { f(); }  
}  
C {  
    void main(String[] args) {  
        aB = new B();  
        aB;  
    }  
    void h(A x) { x.g(); }  
}
```

What is printed?
Is `static`?
Is `f` static?
Is `g` in `B`?
Is `h` in `A`?

27:59 2016

CS61B: Lecture #10 7

Choices

- a. A.f
- b. B.f
- c. Some kind of error

Review: A Puzzle

```
System.out.println("A.f");  
/* or this.f() */  
class B extends A {  
    void f() {  
        System.out.println("B.f");  
    }  
    void g() { f(); }  
}  
C {  
    void main(String[] args) {  
        aB = new B();  
        aB;  
    }  
    void h(A x) { x.g(); }  
}
```

What is printed?
Is `static`?
Is `f` static?
Is `g` in `B`?
Is `h` in `A`?

27:59 2016

CS61B: Lecture #10 9

Choices

- a. A.f
- b. B.f
- c. Some kind of error

Review: A Puzzle

```
/* or this.f() */  
class B extends A {  
    void f() {  
        System.out.println("B.f");  
    }  
    void g() { f(); }  
}  
C {  
    void main(String[] args) {  
        aB = new B();  
        aB;  
    }  
    void h(A x) { x.g(); }  
}
```

What is printed?
Is `static`?
Is `f` static?
Is `g` in `B`?
Is `h` in `A`?

27:59 2016

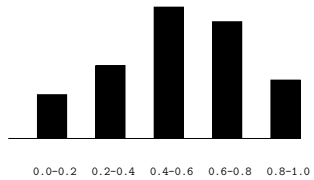
CS61B: Lecture #10 11

Choices

- a. A.f
- b. B.f
- c. Some kind of error

Example: Designing a Class

Write a class that represents histograms, like this one:



What do we need from it? At least:

Bucket limits.

Counts of values.

Low and high limits of values.

Number of buckets and other initial parameters.

27:59 2016

CS61B: Lecture #10 14

Histogram Specification and Use

```
of floating-point values */
Histogram {
  of buckets in THIS. */

  low of bucket #K. Pre: 0<=K<size(). */
  low(k);

  count of values in bucket #K. Pre: 0<=K<size(). */
  count(k);

  print the histogram. */
  printHistogram(Scanner in);
}

Histogram H, Scanner in
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
```

27:59 2016

CS61B: Lecture #10 16

Let's Make a Tiny Change

Change the *priori* bounds:

```
LowHistogram implements Histogram {
  LowHistogram with SIZE buckets. */
  LowHistogram(int size) {
```

What is to change?

How do you do this? Profoundly changes implementation.

Can you make `printHistogram` and `fillHistogram` still work with

the power of *separation of concerns*.

27:59 2016

CS61B: Lecture #10 18

Answer to Puzzle

When `va C` prints `_B.f_`, because

`va` calls `h` and passes it `aB`, whose dynamic type is `B`.

`va.g()`. Since `g` is inherited by `B`, we execute the code for `A.g()`.

`va.is.f()`. Now `this` contains the value of `h`'s argument, whose dynamic type is `B`. Therefore, we execute the definition of `B.is.f()`.

`va.f()`, in other words, static type is ignored in figuring out what method to call.

When `va.f()`, we see `_B.f_`; selection of `f` still depends on dynamic type of `this`. Same for overriding `g` in `B`.

When `va.f()`, would print `_A.f_` because then selection of `f` is based on static type of `this`, which is `A`.

When `va.f()` is defined in `A`, we'd see a compile-time error

27:59 2016

CS61B: Lecture #10 13

Specification Seen by Clients

When a module (class, program, etc.) uses a module, it uses that module's exported definitions.

The module's intention is that exported definitions are designated **public**. Clients are intended to rely on *specifications*, (aka APIs) not code.

Specification: method and constructor headers—syntax and semantics.

Specification: what they do. No formal notation, so use natural language.

A *specification* is a *contract*.

A client must satisfy (*preconditions*, marked "Pre:" in the code below).

Results (*postconditions*).

It is the client's responsibility to ensure that all the client needs!

When a module communicates errors, specifically failure to meet preconditions, it should communicate errors, specifically failure to meet preconditions.

27:59 2016

CS61B: Lecture #10 15

An Implementation

```
LowHistogram implements Histogram {
  low, high; /* From constructor */
  count; /* Value counts */

  LowHistogram with SIZE buckets of values >= LOW and < HIGH. */
  LowHistogram(int size, double low, double high) {
    if (size <= 0) throw new IllegalArgumentException();
    low = this.low = high;
    count = new int[size];
  }

  count() { return count.length; }
  low(int k) { return low + k * (high-low)/count.length; }
  high(int k) { return high - k * (high-low)/count.length; }

  fill(double val) {
    for (int i = 0; i < count.length; i++)
      count[i] += 1;
  }
}
```

27:59 2016

CS61B: Lecture #10 17

of Procedural Interface over Visible Fields

method for `count` instead of making the array `count` "change" is transparent to clients:

to write `myHist.count[k]`, would mean

number of items currently in the k^{th} bucket of histogram
and by the way, there is an array called `count` in
that always holds the up-to-date count."

! comment *useless* to the client.

array had been visible, after "tiny change," every use
client program would have to change.

method for the public `count` decreases what client has to
change (therefore) has to change.

Implementing the Tiny Change

pre-allocate the `count` array.

bounds, so must save arguments to `add`.

compute `count` array "lazily" when `count(...)` called.

`count` array whenever histogram changes.

```
Histogram implements Histogram {
    ArrayList<Double> values = new ArrayList<>();

    int[] count;

    Histogram(int size) { this.size = size; this.count = null;

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

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