ure #10: OOP mechanism and Class Design

Review: A Puzzle

```
t.println("A.f");

; /* or this.f()  */ }
```

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

}
```

```
C {
ic void main(String[] args) {
aB = new B();
aB);


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

ted?
g static?
f static?
de g in B?
ined in A?

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

Review: A Puzzle

```
t.println("A.f");

A y) { y.f(); }
```

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

}
```

```
C {
ic void main(String[] args) {
aB = new B();
aB);


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

ted?
g static?
f static?
de g in B?
ined in A?

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

Review: A Puzzle

```
t.println("A.f");

; /* or this.f()  */ }
```

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

}
```

```
C {
ic void main(String[] args) {
aB = new B();
aB);


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

ted?
g static?
f static?
de g in B?
ined in A?

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

Review: A Puzzle

```
t.println("A.f");

; /* or this.f()  */ }
```

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

}
```

```
C {
ic void main(String[] args) {
aB = new B();
aB);


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

ted?
g static?
f static?
de g in B?
ined in A?

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

Review: A Puzzle

```
t.println("A.f");

A y) { y.f(); }
```

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

}
```

```
C {
ic void main(String[] args) {
aB = new B();
aB);


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

ted?
g static?
f static?
de g in B?
ined in A?

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

Review: A Puzzle

```
t.println("A.f");

; /* or this.f()  */ }
```

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

```
C {
ic void main(String[] args) {
aB = new B();
aB);


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

ted?
g static?
f static?
de g in B?
ined in A?

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");
  }

}
```

```
C {
ic void main(String[] args) {
aB = new B();
aB);


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

ted?
g static?
f static?
de g in B?
ined in A?

Choices

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

Answer to Puzzle

va C prints _____, because

lls h and passes it aB, whose dynamic type is B.

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

is.f(). Now this contains the value of h's argument, amic type is B. Therefore, we execute the definition of n B.

f, in other words, static type is ignored in figuring out hod to call.

atic, we see _____; selection of f still depends on dy- f this. Same for overriding g in B.

tatic, would print _____ because then selection of f d on static type of this, which is A.

t defined in A, we'd see _____

Review: A Puzzle

```
) {
t.println("A.f");

; /* or this.f()  */ }
```

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

}
```

```
C {
ic void main(String[] args) {
aB = new B();
aB);


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

ted?
g static?
f static?
de g in B?
ined in A?

Choices

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

Review: A Puzzle

```
t.println("A.f");

; /* or this.f()  */ }
```

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

```
C {
ic void main(String[] args) {
aB = new B();
aB);


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

ted?
g static?
f static?
de g in B?
ined in A?

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");
  }

}
```

```
C {
ic void main(String[] args) {
aB = new B();
aB);


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

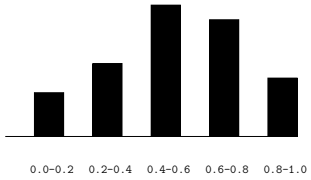
ted?
g static?
f static?
de g in B?
ined in A?

Choices

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

Example: Designing a Class

t a class that represents histograms, like this one:



t do we need from it? At least:

kets and limits.

counts of values.

nts of values.

bers of buckets and other initial parameters.

Answer to Puzzle

va C prints B.f, because

lls h and passes it aB, whose dynamic type is B.

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

is.f(). Now this contains the value of h's argument, namic type is B. Therefore, we execute the definition of n B.

f, in other words, static type is ignored in figuring out hod to call.

atic, we see B.f; selection of f still depends on dy- f this. Same for overriding g in B.

tatic, would print A.f because then selection of f d on static type of this, which is A.

t defined in A, we'd see a compile-time error

Histogram Specification and Use

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


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

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

 the histogram. */
e val);
```

Sample output:

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

```
am(Histogram H,
    Scanner in)

sNextDouble())
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));
}
```


Specification Seen by Clients

of a module (class, program, etc.) are the programs or t *use* that module's exported definitions.

ntion is that exported definitions are designated **public**.

ntended to rely on *specifications*, (*aka* APIs) not code.

ecification: method and constructor headers—syntax e.

ecification: what they do. No formal notation, so use

specification is a *contract*.

s client must satisfy (*preconditions*, marked "Pre:" in below).

results (*postconditions*).

ese to be *all the client needs!*

s communicate errors, specifically failure to meet pre- s.

An Implementation

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

ogram with SIZE buckets of values >= LOW and < HIGH. */
stogram(int size, double low, double high)

igh || size <= 0) throw new IllegalArgumentException();
ow; this.high = high;
 new int[size];


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

nt(int k) { return count[k]; }

d(double val) {
low && val < high)
int) ((val-low)/(high-low) * count.length)] += 1;
```


Let's Make a Tiny Change

priori **bounds:**

```
ogram implements Histogram {
stogram with SIZE buckets. */
istogram(int size) {


ls to change?
```

ou do this? Profoundly changes implementation.

like printHistogram and fillHistogram) still work with

he power of *separation of concerns*.

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

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

l comment *useless* to the client.

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

ethod for the public `count` decreases what client *has* to
herefore) has to change.

Implementing the Tiny Change

pre-allocate the `count` array.

ounds, so must save arguments to `add`.

ute `count` array "lazily" when `count(···)` called.

unt array whenever histogram changes.

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

t[] count;

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


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

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