

# CS61B Lecture #6

## Administrative:

- Make sure to get this week's lab checked off before end of lab *next* Tuesday.
- Are you turning in homework??
- Discussion Section 111 at 10AM Wednesday in 321 Haviland moved: now in 106 Moffitt.

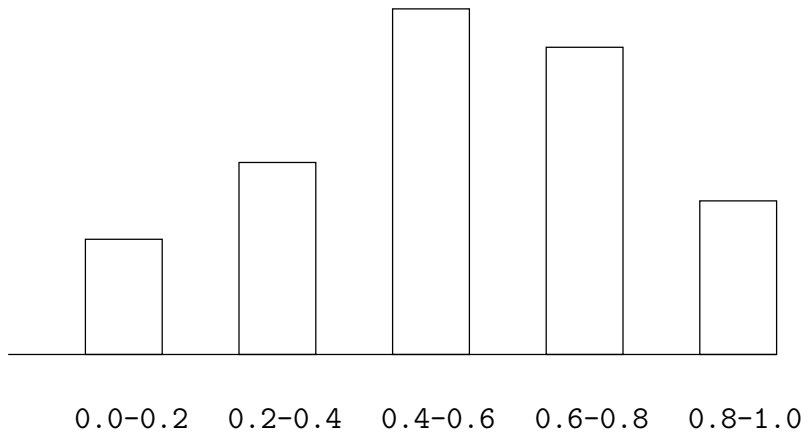
## Today:

- Designing a class
- Syntactic and semantic specification
- Preconditions
- Exceptions (*On to Java*, chapter 32; *Programming into Java*, chapter 7).

**Next Lecture:** Inheritance, method overriding (*On To Java*, Chapters 14-16, 19, 20).

## Example: Designing a Class

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



**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*, 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 preconditions.

# Specification

```
/** A histogram of floating-point values */
public class Histogram {
    /** A new, empty histogram with SIZE buckets,
     * for values in the range LOW -- HIGH
     * (inclusive). Pre: SIZE>0, LOW<HIGH. */
    public Histogram (int size,
                     double low, double high) ...

    /** The number of buckets in THIS. */
    public int size () ...

    /** Lower bound of bucket #K. Pre: 0<=K<size(). */
    public double low (int k) ...

    /** Upper bound of bucket #K. Pre: 0<=K<size(). */
    public double high (int k) ...

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

    /** Add VAL to the histogram. */
    public void add (double val) ...
}
```

## An Implementation

```
public class Histogram {
    private double low, high; /* From constructor*/
    private int[] count; /* Value counts */

    public Histogram (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];
    }

    /** Exception if K an illegal bucket number */
    private void checkIndex (int k) {
        if (k < 0 || k >= count.length)
            throw new IllegalArgumentException ();
    }
}
```

*Continues...*

## Implementation continues

```
public int size () { return count.length; }
```

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

```
public double high (int k) {  
    checkIndex (k);  
    return low + (k+1) * (high-low)/count.length;  
}
```

```
public int count (int k) {  
    checkIndex (k); return count[k];  
}
```

```
public void add (double val) {  
    int k;  
    k = (val == high) ? count.length-1  
        : (int) ( (val-low)/(high-low) * count.length  
    if (k >= 0 && k < count.length)  
        count[k] += 1;  
}
```

## Let's Make a *Tiny Change*

**Add following methods and constructor:**

```
/** Set size () to N; Pre: N>0. */  
public setSize (int N) ...
```

```
/** Set lower bound of bucket #0 to LOW. */  
public setLow (double low) ...
```

```
/** Set upper bound of last bucket to HIGH. */  
public setHigh (double high) ...
```

- How would you do this?
- Profoundly changes implementation.
- But *client* sees very little change: still calls `.add`, `.count`, etc.
- 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.

# Sketch of Changes for New Implementation

```
/** Some data structure that can hold the set of
 * all arguments given to .add(...) */
private ListOfDoubles values = ...;

public Histogram (...) {
    ...
    this.count = null; // Hold off computing count
}

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

public setLow (double low) {
    this.low = low;
    counts = null; // No longer valid.
}

...
public int count (int k) {
    checkIndex (k);
    if (count == null) recompute count;
    return count[k];
}
```

# Advantages of Procedural Interface

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]`, would mean  
"The number of items currently in the  $k^{\text{th}}$  bucket of histogram `myHist` (and by the way, there is an array called `count` in `myHist` that always holds the up-to-date count)."
- Parenthetical comment *useless* to the client.
- But 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` decreases what client *has* to know, and (therefore) has to change.

# What to do About Errors?

- Large amount of any production program devoted to detecting and responding to errors.
- Some errors are external (bad input, network failures); others are internal errors in programs.
- When method has stated precondition, it's client's job to comply.
- Still, its nice to detect and report client's errors.
- In Java, we *throw exception objects*, typically:  

```
throw new SomeException (optional description) ;
```
- Exceptions are objects. By convention, they are given two constructors: one with no arguments, and one with a descriptive string argument (which the exception stores).
- Java system throws some exceptions implicitly, as when you dereference a null pointer, or exceed an array bound.

# Catching Exceptions

- A **throw** causes each active method call to *terminate abruptly*, until (and unless) we come to a **try** block.
- Catch exceptions and do something corrective with **try**:

```
try {  
    Stuff that might throw exception;  
} catch (SomeException e) {  
    Do something reasonable;  
} catch (SomeOtherException e) {  
    Do something else reasonable;  
}  
Go on with life;
```

- When *SomeException* exception occurs in "Stuff...", we immediately "do something reasonable" and then "go on with life."
- Descriptive string (if any) available as `e.getMessage()` for error messages and the like.