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

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

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

static?

3. If we made `f` static?

4. If we overrode `g` in `B`?

5. If `f` not defined in A?

b. `B.f`

c. Some kind of error

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

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

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

static?

3. If we made `f` static?

4. If we overrode `g` in `B`?

5. If `f` not defined in A?

b. `B.f`

c. Some kind of error

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

static?

3. If we made `f` static?

4. If we overrode `g` in `B`?

5. If `f` not defined in A?

b. `B.f`

c. Some kind of error

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

static?

3. If we made `f` static?

4. If we overrode `g` in `B`?

5. If `f` not defined in A?

b. `B.f`

c. Some kind of error

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

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

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

static?

3. If we made `f` static?

4. If we overrode `g` in `B`?

5. If `f` not defined in A?

b. `B.f`

c. Some kind of error

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

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

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

static?

3. If we made f static?

4. If we overrode g in B?

5. If f not defined in A?

b. B.f

c. Some kind of error

```
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(); }
     }
```

static?

3. If we made f static?

4. If we overrode g in B?

5. If f not defined in A?

b. B.f

c. Some kind of error

```
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(); }
     }
```

static?

3. If we made f static?

4. If we overrode g in B?

5. If f not defined in A?

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(); }
     }
```

static?

3. If we made f static?

4. If we overrode g in B?

5. If f not defined in A?

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(); }
}
```

static?

3. If we made f static?

4. If we overrode g in B?

5. If f not defined in A?

b. B.f

c. Some kind of error

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.

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.

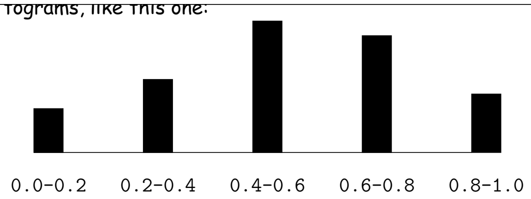
C. g calls this.f(). Now this contains the value of h's argument, whose dynamic type is B. Therefore, we execute the definition of f that is in B.

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.

tograms, 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.

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 preconditions.

```
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);
}
```

output:

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

```
                        Scanner
in)
{
    while
(in.hasNextDouble())
       H.add(in.nextDouble());
}
```

```
H.size(); i += 1)
       System.out.printf
           (">=%5.2f |
%4d%n",
            H.low(i),
H.count(i));
}
```

```
  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)]
```

```
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.*

- Don't know bounds, so must save arguments to `add`.
- Then recompute `count` array "lazily" when `count(···)` called.
- Invalidate `count` array whenever 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) {
```

```
    }
```

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^{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