CS61B Lecture #9: Interfaces and Abstract Classes

Announcements

- Hackers@Berkeley is hosting a HackJam this Saturday (28 Sept) in the Woz—12 hours of (low-pressure) hacking/coding with food and prizes. Starts at 11AM.

- Programming Contest: Sat., 5 October at 10:00 AM.

Abstract Methods and Classes

- Instance method can be *abstract:* No body given; must be supplied in subtypes.

- One good use is in specifying a pure interface to a family of types:

```
/** A drawable object. */
public abstract class Drawable {  // "abstract" = "can't say new Drawable"
  /** Expand THIS by a factor of SIZE */
  public abstract void scale (double size);
  /** Draw THIS on the standard output. */
  public abstract void draw ();
}
```

  Now a `Drawable` is something that has *at least* the operations `scale` and `draw` on it. Can't create a `Drawable` because it's abstract—in particular, it has two methods without any implementation.

- *BUT,* we can write methods that operate on `Drawables`:

```
void drawAll (Drawable[] thingsToDraw) {
   for (Drawable thing : thingsToDraw)
      thing.draw ();
}
```

- But `draw` has no implementation! How can this work?

Concrete Subclasses

- Can define kinds of `Drawable`s that are non-abstract. To do so, must supply implementations for all methods:

```
public class Rectangle extends Drawable {
  public Rectangle (double w, double h) { this.w = w; this.h = h; }
  public void scale (double size) { w *= size; h *= size; }
  public void draw () { draw a w x h rectangle }
  private double w,h;
}
```

  Any `Circle` or `Rectangle` is a Drawable.

```
public class Circle extends Drawable {
  public Circle (double rad) { this.rad = rad; }
  public void scale (double size) { rad *= size; }
  public void draw () { draw a circle with radius rad }
  double rad;
}
```

- So, writing

```
   Drawable[] things = { new Rectangle (3, 4), new Circle (2) };
   drawAll (things);
```

  draws a 3×4 rectangle and a circle with radius 2.

Interfaces

- In generic use, an *interface* is a "point where interaction occurs between two systems, processes, subjects, etc." (*Concise Oxford Dictionary*).

- In programming, often use the term to mean a *description* of this generic interaction, specifically, a description of the functions or variables by which two things interact.

- Java uses the term to refer to a slight variant of an abstract class that contains only abstract methods (and static constants).

- Idea is to treat Java interfaces as the public specifications of data types, and classes as their implementations:

```
public interface Drawable {
  void scale (double size);   // Automatically public abstract.
  void draw ();
}

public class Rectangle implements Drawable { ... }
```

- Interfaces are automatically abstract: can't say new Drawable(); can say new Rectangle(...).

Multiple Inheritance

- Can *extend* one class, but *implement* any number of interfaces.
- Contrived Example:

```
interface Readable {              |   void copy (Readable r,
  Object get ();                  |                 Writable w)
}                                 |   {
                                  |      w.put (r.get ());
interface Writable {              |   }
  void put (Object x);            |
}                                 |
                                  |
class Source implements Readable {|  class Sink implements Writable {
  public Object get () { ... }    |    public void put (Object x) { ... }
}                                 |  }

          class Variable implements Readable, Writable {
            public Object get () { ... }
            public void put (Object x) { ... }
          }
```

- The first argument of `copy` can be a `Source` or a `Variable`. The second can be a `Sink` or a `Variable`.

Review: Higher-Order Functions

- In Scheme, you had *higher-order functions* like this (adapted from *SICP*)

```
(define (map     proc        items)
;                function     list
  (if (null? items)
       nil
       (cons (proc (car items)) (map proc (cdr items)))))
```

and could write

```
(map abs (list -10 2 -11 17))
  ====> (10 2 11 17)
(map (lambda (x) (* x x)) (list 1 2 3 4))
  ====> (1 4 9 16)
```

- Java does not have these directly, but can use abstract classes or interfaces and subtyping to get the same effect (with more writing)

Map in Java

```
/** Function with one integer argument */ | IntList map (IntUnaryFunction proc,
                                           |                  IntList items) {
public interface IntUnaryFunction {        |   if (items == null)
  int apply (int x);                       |     return null;
}                                          |   else return new IntList (
                                           |        proc.apply (items.head),
                                           |        map (proc, items.tail)
                                           |      );
                                           | }
```

- It's the use of this function that's clumsy. First, define class for absolute value function; then create an instance:

```
class Abs implements IntUnaryFunction {
  public int apply (int x) { return Math.abs (x); }
}
-----------------------------------------------
map (new Abs (), some list);
```

- Or, we can write a lambda expression (sort of):

```
map (new IntUnaryFunction () {
        public int apply (int x) { return x*x; }
     }, some list);
```

Review: A Puzzle

```
class A {                                  | class B extends A {
  void f ()        { System.out.println ("A.f"); } |   void f ()        {
  void g () { f (); /* or this.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) { x.g(); }
//static void h (A x) { A.g(x); } // x.g(x) also legal here
          }
```

1. What is printed?   Choices:

2. What if we made `g` static?   a. `A.f`

3. What if we made `f` static?   b. `B.f`

4. What if `f` were not defined in A?   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`.

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 get _____.

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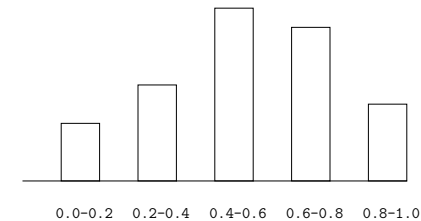
Analysis: What do we need from it? At least:

- 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,* 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

```java
/** 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
```

```java
void fillHistogram (Histogram H,
                    Scanner in)
{
    while (in.hasNextDouble ())
        H.add (in.nextDouble ());
}
```

```java
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));
}
```

An Implementation

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

  /** A new histogram with SIZE buckets recording 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) {
      int k = (int) ((val-low)/(high-low) * count.length);
      if (k >= 0 && k < count.length) count[k] += 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($\cdots$) called.
- Invalidate count array whenever histogram changes.

```
  class FlexHistogram implements Histogram {
      private List<Double> values = ...; // Java library type (later)
      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], would mean

   "The number of items currently in the kth 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.