

CS61B Lecture #11: Some Loose Ends and Tricks

Midterm. Evening of Tues, 15 October, 6:30-8:30PM. Alternate times possible; contact me the week before the exam.

Upcoming topic:

- Numbers. See *PIJ* §6.3, §6.4.
- Scope and Extent. See *PIJ* §5.1

Trick #1: Touching the Unmentionable

Possible to call methods in objects of types you can't name:

```
package utils;                               | package mystuff;
/** A Set of things. */                       |
public interface Collector {                  |
    void add (Object x);                       |
}                                               |
-----|
package utils;                               |
public class Utils {                          |
    public static Collector concat () {        |
        return new Concatenator ();           |
    }                                           |
}                                               |

/** NON-PUBLIC class that collects strings. */ |
class Concatenater implements Collector {      |
    StringBuffer stuff = new StringBuffer ();  |
    int n = 0;                                 |
    public void add (Object x) { stuff.append (x); n += 1; } |
    public Object value () { return stuff.toString (); } |
}                                               |
-----|
class User {                                  |
    Collector c =                               |
        utils.Utils.concat ();                 |
    c.add ("foo"); // OK                       |
    ... c.value (); // ERROR                   |
    ((utils.Collector) c).value ()             |
        // ERROR                               |
}                                               |
-----|
```

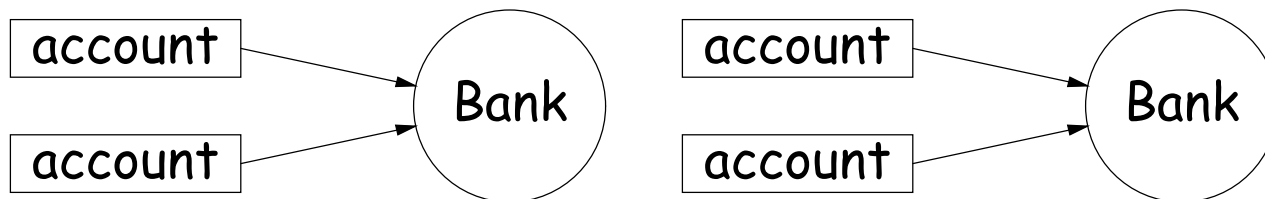
Loose End #1: Nesting Classes

- Sometimes, it makes sense to *nest* one class in another. The nested class might
 - be used only in the implementation of the other, or
 - be conceptually “subservient” to the other
- Nesting such classes can help avoid name clashes or “pollution of the name space” with names that will never be used anywhere else.
- Example: Polynomials from project #1 are sequences of terms. Terms aren’t meaningful outside of Polynomials, so you might define a class to represent a term *inside* the Polynomial class:

```
class Polynomial {  
  
    methods on polynomials  
  
    private Term[] terms;  
    private static class Term {  
        ...  
    }  
}
```

Inner Classes

- Last slide showed a static nested class. Static nested classes are just like any other, except that they can be private or protected, and they can see private variables of the enclosing class.
- Non-static nested classes are called *inner classes*.
- Somewhat rare; used when each instance of the nested class is created by and naturally associated with an instance of the containing class, like Banks and Accounts:



```
class Bank {
    private void connectTo (...) {...}
    public class Account {
        public void call (int number) {
            Bank.this.connectTo (...); ...
        } // Bank.this means "the bank that
    } // created me"
}

| Bank e = new Bank(...);
| Bank.Account p0 =
|     e.new Account (...);
| Bank.Account p1 =
|     e.new Account (...);
|
|
```

Loose End #2: Using an Overridden Method

- Suppose that you wish to *add* to the action defined by a superclass's method, rather than to completely override it.
- The overriding method can refer to overridden methods by using the special prefix `super`.
- For example, you have a class with expensive functions, and you'd like a memoizing version of the class.

```
class ComputeHard {
    int cogitate (String x, int y) { ... }
    ...
}
```

```
class ComputeLazily extends ComputeHard {
    int cogitate (String x, int y) {
        if (already have answer for this x and y) return memoized result;
        else
            int result = super.cogitate (x, y);
            memoize (save) result;
            return result;
    }
}
```

Loose End #3: Constructors of Superclasses

- In Lecture #9, saw instance of a constructor calling the superclass's constructor.
- In fact, constructors *always* call a constructor of the superclass first (well, except for `java.lang.Object`).
- Rationale: superclass *P* is in charge of making sure that any object that *is a P* has certain properties from the start. To *be a C* (where *C* is a subtype of *P*), an object must first *be a P*.
- So either a constructor of *C* starts with a call to the *P*'s constructor, or calls some other constructor in *C* (and so on until a *P* constructor is called), or Java inserts `super ()` by default.

```
class P {
    // id always gets initialized first
    private int id;
    P (int id) { this.id = id; }
    P () { this (-1); }
}

class C extends P {
    int head; C next;
    C (int h, C n) {
        /* super(); */
        head = h; ...
    }
    C (int h) { this(h, null); }
}
```

Trick #2: Delegation and Wrappers

- Not always appropriate to use inheritance to extend something.
- Homework gives example of a `TrReader`, which *contains* another `Reader`, to which it *delegates* the task of actually going out and reading characters.
- Another example: an "interface monitor:"

```
interface Storage {      | class Monitor implements Storage {
    void put (Object x); |     int gets, puts;
    Object get ();       |     private Storage store;
}                          |     Monitor (Storage x) { store = x; gets = puts = 0; }
                          |     public void put (Object x) { puts += 1; store.put (x); }
                          |     public Object get () { gets += 1; return store.get (); }
                          | }

```

- So now, you can *instrument* a program:

```
// ORIGINAL
Storage S = something;
f (S);
```

```
// INSTRUMENTED
Monitor S = new Monitor (something);
f(S);
System.out.println (S.gets + " gets");
```

- Monitor is called a *wrapper class*.

Loose End #4: instanceof

- It is possible to ask about the dynamic type of something:

```
void typeChecker (Reader r) {  
    if (r instanceof TrReader)  
        System.out.print ("Translated characters: ");  
    else  
        System.out.print ("Characters: ");  
    ...  
}
```

- However, this is *seldom* what you want to do. Why do this:

```
if (x instanceof StringReader)  
    read from (StringReader) x;  
else if (x instanceof FileReader)  
    read from (FileReader) x;  
...
```

when you can just call `x.read()`?!

- In general, use instance methods rather than **instanceof**.