CS61B Lecture #11: Some Loose Ends and Tricks

Readings: Chapters 4 and 5 of the Blue Reader for next week.

Today: Wrapping up some loose ends and leaving explicit material on Java for now.

Question: Have you started Project #1 yet?

Loose End #1: Importing

- Writing `java.util.List` every time you mean `List` or `java.lang.regex.Pattern` every time you mean `Pattern` is annoying.
- The purpose of the **import** clause at the beginning of a source file is to define abbreviations:
  - `import java.util.List;` means "within this file, you can use `List` as an abbreviation for `java.util.List`.
  - `import java.util.*;` means "within this file, you can use *any* class name in the package `java.util` without mentioning the package."
- Importing does *not* grant any special access; it *only* allows abbreviation.
- In effect, your program always contains `import java.lang.*;`

Loose End #2: Static importing

- One can easily get tired of writing `System.out` and `Math.sqrt`. Do you really need to be reminded with each use that `out` is in the `java.lang.System` package and that `sqrt` is in the Math package (duh)?
- Both examples are of *static* members. New feature of Java allows you to abbreviate such references:
  - `import static java.lang.System.out;` means "within this file, you can use `out` as an abbreviation for `System.out`.
  - `import static java.lang.System.*;` means "within this file, you can use *any* static member name in `System` without mentioning the package.
- Again, this is *only* an abbreviation. No special access.
- Alas, you can't do this for classes in the anonymous package.

Loose End #3: 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 can be thought of as sequences of terms. Terms aren't meaningful outside of Polynomials, so you might define a class to represent a term *inside* the Polynomial class:

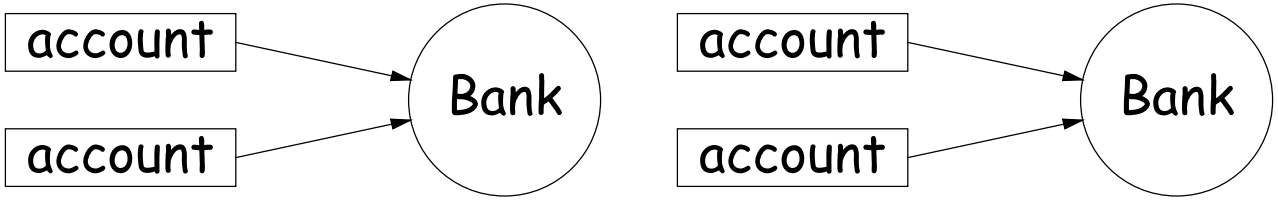
```
class Polynomial {

  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 (and syntax is odd); 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 {                                  | Bank e = new Bank(...);
  private void connectTo (...) {...}          | Bank.Account p0 =
  public class Account {                      |     e.new Account (...);
    public void call (int number) {           | Bank.Account p1 =
       Bank.this.connectTo (...); ...         |     e.new Account (...);
    } // Bank.this means "the bank that       |
  }   // created me"                          |
 }                                            |
```

Loose End #4: 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;
 }
}
```

Trick: 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 #5: 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**.