Lecture #13: Packages, Access, Etc.

on facilities in Java.

:es.

dden method.

ructors.

.

Package Mechanics

espond to things being modeled (represented) in one's

: collections of "related" classes and other packages.

andard libraries and packages in package `java` and `javax`.

class resides in the *anonymous package*.

ewhere, use a `package` declaration at start of file, as in

`latabase;` *or* `package ucb.util;`

uses convention that class `C` in package `P1.P2` goes in
`P1/P2` of any other directory in the *class path*.

e:

```
rt CLASSPATH=.:$HOME/java-utils:$MASTERDIR/lib/classes/junit.jar
 junit.textui.TestRunner MyTests
```

TestRunner.class in ./junit/textui, ~/java-utils/junit/textui
ooks for junit/textui/TestRunner.class in the junit.jar
a single file that is a special compressed archive of an
ory of files).

Access Modifiers

fiers (**private, public, protected**) do not add anything
of Java.

w a programmer to declare what classes are supposed
ccess ("know about") what declarations.

also part of security—prevent programmers from ac-
gs that would "break" the runtime system.

always determined by static types.

nine correctness of writing `x.f()`, look at the definition
static type of `x`.

cause the rules are supposed to be enforced by the
which only knows static types of things (static types
end on what happens at execution time).

The Access Rules

have two packages (not necessarily distinct) and two
es:

```
C1 ... {          package P2;
 named M,         class C2 extends C3 {
                    void f(P1.C1 x) {... x.M ...} // OK?
                    // C4 a subtype of C2 (possibly C2 itself)
)                   void g(C4 y) {... y.M ...  } // OK?
 ... } // OK.     }
```

.M is

is **public**;

is **protected** and $P1$ is $P2$;

is *package private* (default—no keyword) and $P1$ is $P2$;

A is **private**.

, if $C3$ is $C1$, then `y.M` is also legal under the conditions
A is **protected** (i.e., even if $P1$ is not the same as $P2$).

What May be Controlled

nterfaces that are not nested may be public or package
aven't talked explicitly about nested types yet).

ields, methods, constructors, and (later) nested types—
of the four access levels.

a method only with one that has *at least* as permissive
el. Reason: avoid inconsistency:

```
;                              package P2;
ss C1 {                        class C3 {
nt f() { ... }                   void g(C2 y2) {
                                   C1 y1 = y2
                                   y2.f(); // Bad???
ss C2 extends C1 {                 y1.f(); // OK??!!?
lly a compiler error; pretend    }
not and see what happens       }
{ ... }
```

e's no point in restricting C2.f, because access control
tatic types, and C1.f is public.

Intentions of this Design

rations represent *specifications*—what clients of a pack-
osed to rely on.

`vate` declarations are part of the *implementation* of a
ust be known to other classes that assist in the imple-

eclarations are part of the implementation that sub-
ed, but that clients of the subtypes generally won't.

larations are part of the implementation of a class that
ss needs.

Quick Quiz

```
;
{
 OK

y1;
;
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // OK?
    x.y1 = 3; // OK?
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // OK?
    x.y1 = 3; // OK?
    f1();     // OK?
    y1 = 3;   // OK?
    x1 = 3;   // OK?
  }
}
```

hree lines of h have implicit **this.**'s in front. Static type

Quick Quiz

```
;
{
 OK

y1;
;
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // ERROR
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // OK?
    x.y1 = 3; // OK?
    f1();     // OK?
    y1 = 3;   // OK?
    x1 = 3;   // OK?
  }
}
```

hree lines of h have implicit **this.**'s in front. Static type

Quick Quiz

```
;
{
 OK

y1;
;
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // ERROR
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // OK?
    f1();     // ERROR
    y1 = 3;   // OK?
    x1 = 3;   // OK?
  }
}
```

hree lines of h have implicit **this.**'s in front. Static type

Quick Quiz

```
;
{
 OK?

y1;
;
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // OK?
    x.y1 = 3; // OK?
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // OK?
    x.y1 = 3; // OK?
    f1();     // OK?
    y1 = 3;   // OK?
    x1 = 3;   // OK?
  }
}
```

hree lines of h have implicit **this.**'s in front. Static type

Quick Quiz

```
;
{
 OK

y1;
;
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // OK?
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // OK?
    x.y1 = 3; // OK?
    f1();     // OK?
    y1 = 3;   // OK?
    x1 = 3;   // OK?
  }
}
```

hree lines of h have implicit **this.**'s in front. Static type

Quick Quiz

```
;
{
 OK

y1;
;
```

```
// Anonymous package

class A2 {
  void g(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // ERROR
  }
}

class B2 extends SomePack.A1 {
  void h(SomePack.A1 x) {
    x.f1();  // ERROR
    x.y1 = 3; // OK?
    f1();     // OK?
    y1 = 3;   // OK?
    x1 = 3;   // OK?
  }
}
```

hree lines of h have implicit **this.**'s in front. Static type

Quick Quiz

```
;                 // Anonymous package
{
                  class A2 {
                    void g(SomePack.A1 x) {
 OK                   x.f1();  // ERROR
                      x.y1 = 3; // ERROR
y1;                 }
;                 }

                  class B2 extends SomePack.A1 {
                    void h(SomePack.A1 x) {
                      x.f1();  // ERROR
                      x.y1 = 3; // OK?
                      f1();     // ERROR
                      y1 = 3;   // OK
                      x1 = 3;   // ERROR
                    }
                  }
```

hree lines of h have implicit **this.**'s in front. Static type

Access Control Static Only

vate" don't apply to dynamic types; it is possible to call
cts of types you can't name:

```
                                   | package mystuff;
hings. */                          |
ace Collector {                    | class User {
ect x);                            |    utils.Collector c =
                                   |      utils.Utils.concat();
--------------                     |
                                   |    c.add("foo");  // OK
tils {                             |    ... c.value(); // ERROR
c Collector concat() {             |    ((utils.Concatenator) c).value()
 Concatenator();                   |                    // ERROR
                                   |
                                   ----------------------------------

 class that collects strings. */
ater implements Collector {
 stuff = new StringBuffer();

add(Object x) { stuff.append(x); n += 1; }
t value() { return stuff.toString(); }
```


Loose End #2: Static importing

ly get tired of writing `System.out` and `Math.sqrt`. Do
eed to be reminded with each use that `out` is in the
ystem package and that `sqrt` is in the Math package

es are of *static* members. New feature of Java allows
viate such references:

tatic java.lang.System.out; means "within this file,
e `out` as an abbreviation for `System.out`.

tatic java.lang.System.*; means "within this file, you
y static member name in `System` without mentioning the

s *only* an abbreviation. No special access.

't do this for classes in the anonymous package.

Quick Quiz

```
;                 // Anonymous package
{
                  class A2 {
                    void g(SomePack.A1 x) {
 OK                   x.f1();  // ERROR
                      x.y1 = 3; // ERROR
y1;                 }
;                 }

                  class B2 extends SomePack.A1 {
                    void h(SomePack.A1 x) {
                      x.f1();  // ERROR
                      x.y1 = 3; // OK?
                      f1();     // ERROR
                      y1 = 3;   // OK
                      x1 = 3;   // OK?
                    }
                  }
```

hree lines of h have implicit **this.**'s in front. Static type

Quick Quiz

```
;                 // Anonymous package
{
                  class A2 {
                    void g(SomePack.A1 x) {
 OK                   x.f1();  // ERROR
                      x.y1 = 3; // ERROR
y1;                 }
;                 }

                  class B2 extends SomePack.A1 {
                    void h(SomePack.A1 x) {
                      x.f1();  // ERROR
                      x.y1 = 3; // ERROR
                      f1();     // ERROR
                      y1 = 3;   // OK
                      x1 = 3;   // ERROR
                    }
                  }
```

hree lines of h have implicit **this.**'s in front. Static type

Loose End #1: Importing

.util.List every time you mean `List` or
egex.Pattern every time you mean `Pattern` is annoying.

of the **import** clause at the beginning of a source file is
oreviations:

ava.util.List; means "within this file, you can use `List`
reviation for `java.util.List`.

ava.util.*; means "within this file, you can use *any*
e in the package `java.util` without mentioning the pack-

oes *not* grant any special access; it *only* allows abbrevi-

our program always contains `import java.lang.*;`

ose End #3: Parent constructors

- otes #5, talked about how Java allows implementer of a rol all manipulation of objects of that class.
- , this means that Java gives the constructor of a class t at each new object.
- ass extends another, there are two constructors—one nt type and one for the new (child) type.
- Java guarantees that one of the parent's constructors t. In effect, there is a call to a parent constructor at g of every one of the child's constructors.
- he parent's constructor yourself. By default, Java calls ' (parameterless) constructor.

```
re {                          class Rectangle extends Figure {
igure(int sides) {               public Rectangle() {
                                      super(4);
                                  }...
                               }
```


End #4: Using an Overridden Method

- t you wish to *add* to the action defined by a superclass's her than to completely override it.
- ng method can refer to overridden methods by using refix super.
- , you have a class with expensive functions, and you'd zing version of the class.

```
uteHard {
tate(String x, int y) { ... }


uteLazily extends ComputeHard {
tate(String x, int y) {
n't already have answer for this x and y) {
 result = super.cogitate(x, y);  // <<< Calls overridden function
noize (save) result;
urn result;

n memoized result;
```


Loose End #5: Nesting Classes

- t makes sense to *nest* one class in another. The nested
- nly in the implementation of the other, or
- tually "subservient" to the other
- classes can help avoid name clashes or "pollution of the with names that will never be used anywhere else.
- lynomials can be thought of as sequences of terms. meaningful outside of Polynomials, so you might define present a term *inside* the Polynomial class:

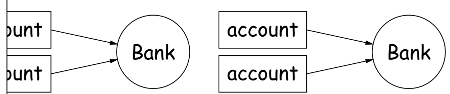
```
nomial {

on polynomials

Term[] terms;
static class Term {
```


Inner Classes

- owed a static nested class. Static nested classes are other, except that they can be private or protected, see private variables of the enclosing class.
- ested classes are called *inner classes.*
- are (and syntax is odd); used when each instance of the is created by and naturally associated with an instance ining class, like Banks and Accounts:




```
d connectTo(...) {...}     | Bank e = new Bank(...);
s Account {                | Bank.Account p0 =
id call(int number) {      |     e.new Account(...);
his.connectTo(...); ...    | Bank.Account p1 =
.this means "the bank that |     e.new Account(...);
ted me"                    |
                           |
```


Trick: Delegation and Wrappers

- ppropriate to use inheritance to extend something.
- ives example of a TrReader, which *contains* another hich it *delegates* the task of actually going out and acters.
- mple: a class that *instruments* objects:

```
e {           |  class Monitor implements Storage {
t x);         |    int gets, puts;
              |    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(); }
              |  }

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

- led a *wrapper class.*

Loose End #6: instanceof

- to ask about the dynamic type of something:

```
ker(Reader r) {
nceof TrReader)
t.print("Translated characters: ");

t.print("Characters: ");
```

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

```
eof StringReader)
StringReader) x;
stanceof FileReader)
FileReader) x;
```

- just call x.read()?!
- se instance methods rather than **instanceof**.