ture #13: Packages, Access, Loose Ends

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;`

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

e:

```
LASSPATH=.:$HOME/java-utils:$MASTERDIR/lib/classes/junit.jar
it.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 which classes are supposed ccess ("know about") what declarations.

also part of security—prevent programmers from ac- s 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`.

static type? Because the rules are supposed to be en- the compiler, which only knows static types of things es don't depend on what happens at execution time).

The Access Rules: Public

of a member depends on (1) how the member's decla- ified and (2) where it is being accessed.

d C4 are distinct classes.

either class C2 itself or a subtype of C2.

```
C1 ... {
ethod, field,...
M ...
)
 ... } // OK.
-----------------
C4 ... {
)
 ... } // OK.
```

```
package P2;
class C2 extends C3 {
  void f(P1.C1 x) {... x.M ...} // OK
  void g(C2a y) {... y.M ...  } // OK
}
```

Public members are available evrywhere.

The Access Rules: Private

4 are distinct classes.

either class C2 itself or a subtype of C2.

```
C1 ... {
ethod, field,...
 M ...
)
 ... } // OK.
-----------------
C4 ... {
)
 ... } // ERROR.
```

```
package P2;
class C2 extends C1 {
  void f(P1.C1 x) {... x.M ...} // ERROR
  void g(C2a y) {... y.M ...  } // ERROR
}
```

Private members are available only within the text of the same class, even for subtypes.

e Access Rules: Package Private

4 are distinct classes.

either class C2 itself or a subtype of C2.

```
C1 ... {
ethod, field,...
)
 ... } // OK.
-----------------
C4 ... {
)
 ... } // OK.
```

```
package P2;
class C2 extends C1 {
  void f(P1.C1 x) {... x.M ...} // ERROR
  void g(C2a y) {... y.M ...  } // ERROR
}
```

Package Private members are available only within the same package (even for subtypes).

The Access Rules: Protected

4 are distinct classes.

either class C2 itself or a subtype of C2.

```
                        package P2;
C1 ... {                class C2 extends C1 {
ethod, field,...          void f(P1.C1 x) {... x.M ...} // ERROR
nt M ...                       // (x's type is not subtype of C2.)
)                         void g(C2a y) {... y.M ...  } // OK
 ... } // OK.             void g2()     {... M ... } // OK (this.M)
                        }
-----------------

C4 ... {
)
 ... } // OK.
```

Protected members of C1 are available within P1, as for package private. Outside P1, they are available within subtypes of C1 such as C2, but only if accessed from expressions whose static types are subtypes of C2.

What May be Controlled

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

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

e a method only with one that has *at least* as permissive vel. 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       }
hot 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 package 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 subed, but that clients of the subtypes generally won't.

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

Quick Quiz

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

                           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

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

                           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

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

                           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

```
;
{
 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();  // ERROR
    x.y1 = 3; // ERROR
  }
}

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

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();  // 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;   // 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. */                       |
ce 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 #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
breviations:

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*
 in the package java.util without mentioning the pack-

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

ur program always contains import java.lang.*;

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

 only an abbreviation. No special access.

't do this for classes in the anonymous package.

Loose End #3: 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:

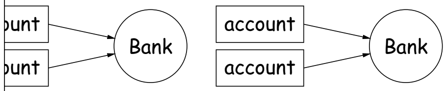
```
nomial {

n polynomials

erm[] terms;
tatic 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.*

re (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:



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

Loose End #4: instanceof

 to ask about the dynamic type of something:

```
ecker(Reader r) {
stanceof TrReader)
.out.print("Translated characters: ");

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

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

```
anceof StringReader)
  (StringReader) x;
 instanceof FileReader)
  (FileReader) x;
```

 just call x.read()?!

se instance methods rather than **instanceof**.