CS61B Lecture #13: Packages, Access, Loose Ends

- Modularization facilities in Java.
- Importing
- Nested classes.
- Using overridden method.
- Parent constructors.
- Type testing.

Package Mechanics

- Classes correspond to things being modeled (represented) in one's program.
- Packages are collections of "related" classes and other packages.
- Java puts standard libraries and packages in package `java` and `javax`.
- By default, a class resides in the *anonymous package.*
- To put it elsewhere, use a `package` declaration at start of file, as in

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

- Oracle's `javac` uses convention that class `C` in package `P1.P2` goes in subdirectory `P1/P2` of any other directory in the *class path*.
- Unix example:

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

  Searches for TestRunner.class in ./junit/textui, ~/java-utils/junit/textui and finally looks for junit/textui/TestRunner.class in the junit.jar file (which is a single file that is a special compressed archive of an entire directory of files).

Access Modifiers

- Access modifiers (**private**, **public**, **protected**) do not add anything to the power of Java.
- Basically allow a programmer to declare which classes are supposed to need to access ("know about") what declarations.
- In Java, are also part of security—prevent programmers from accessing things that would "break" the runtime system.
- Accessibility always determined by static types.
  - To determine correctness of writing `x.f()`, look at the definition of `f` in the *static type* of `x`.
  - Why the static type? Because the rules are supposed to be enforced by the compiler, which only knows static types of things (static types don't depend on what happens at execution time).

The Access Rules: Public

- Accessibility of a member depends on (1) how the member's declaration is qualified and (2) where it is being accessed.
- C1, C2, C3, and C4 are distinct classes.
- Class C2a is either class C2 itself or a subtype of C2.

```
package P1;
public class C1 ... {
  // M is a method, field,...
  public int M ...
  void h(C1 x)
    { ... x.M ... } // OK.
}
------------------------------
package P1;
public class C4 ... {
  void p(C1 x)
    { ... x.M ... } // 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

- C1, C2, and C4 are distinct classes.
- Class C2a is either class C2 itself or a subtype of C2.

```
package P1;
public class C1 ... {
  // M is a method, field,...
  private int M ...
  void h(C1 x)
    { ... x.M ... } // OK.
}
------------------------------
package P1;
public class C4 ... {
  void p(C1 x)
    { ... x.M ... } // 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.

The Access Rules: Package Private

- C1, C2, and C4 are distinct classes.
- Class C2a is either class C2 itself or a subtype of C2.

```
package P1;
public class C1 ... {
  // M is a method, field,...
  int M ...
  void h(C1 x)
    { ... x.M ... } // OK.
}
------------------------------
package P1;
public class C4 ... {
  void p(C1 x)
    { ... x.M ... } // 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

- C1, C2, and C4 are distinct classes.
- Class C2a is either class C2 itself or a subtype of C2.

```
package P1;
public class C1 ... {
  // M is a method, field,...
  protected int M ...
  void h(C1 x)
    { ... x.M ... } // OK.
}
------------------------------
package P1;
public class C4 ... {
  void p(C1 x)
    { ... x.M ... } // OK.
}
```

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

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

- Classes and interfaces that are not nested may be public or package private (we haven't talked explicitly about nested types yet).
- Members—fields, methods, constructors, and (later) nested types—may have any of the four access levels.
- May *override* a method only with one that has *at least* as permissive an access level. Reason: avoid inconsistency:

```
package P1;
public class C1 {
  public int f() { ... }
}

public class C2 extends C1 {
  // Actually a compiler error; pretend
  // it's not and see what happens
  int f() { ... }
}
```

```
package P2;
class C3 {
  void g(C2 y2) {
    C1 y1 = y2
    y2.f(); // Bad???
    y1.f(); // OK??!!?
  }
}
```

That is, there's no point in restricting C2.f, because access control depends on static types, and C1.f is public.

Intentions of this Design

- `public` declarations represent *specifications*—what clients of a package are supposed to rely on.
- `package private` declarations are part of the *implementation* of a class that must be known to other classes that assist in the implementation.
- `protected` declarations are part of the implementation that subtypes may need, but that clients of the subtypes generally won't.
- `private` declarations are part of the implementation of a class that only that class needs.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK?
  }
  protected int y1;
  private int x1;
}
```

```
// 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?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// 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?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// 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?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// 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?
  }
}
```

- **Note:** Last three lines of h have implicit **this**.'s in front. Static type of **this** is B2.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// 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?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// 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?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this.**'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// 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?
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// 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
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Quick Quiz

```
package SomePack;
public class A1 {
  int f1() {
    A1 a = ...
    a.x1 = 3; // OK
  }
  protected int y1;
  private int x1;
}
```

```
// 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
  }
}
```

- **Note:** Last three lines of `h` have implicit **this**.'s in front. Static type of **this** is `B2`.

Access Control Static Only

"Public" and "private" don't apply to dynamic types; it is possible to call methods in objects of types you can't name:

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

/** 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(); }
}
```

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:

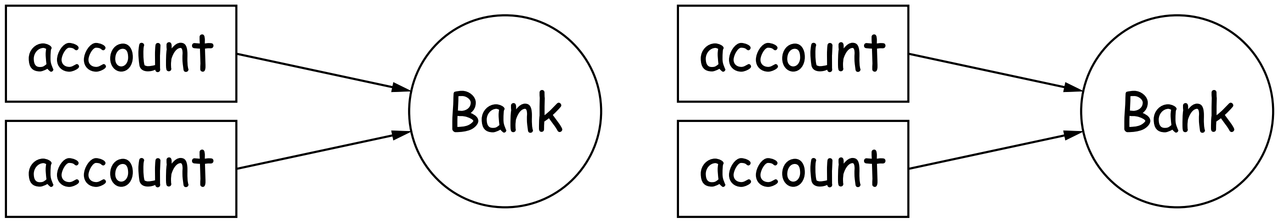
```
class Polynomial {

  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: 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**.