CS61B Lecture 13

Friday, February 21, 2020

Packages

Classes correspond to things being modeled and represented in a program. A **package** is a collection of related classes and other packages. For example, 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 the start of the file, as in:

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

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

Here's an example in Unix:

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

This searches for a file called `TestRunner.class` in `./junit/textui`, then `~/java-utils/junit/textui` and finally in `junit/textui/TestRunner.class` in the `junit.jar` file, 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: they don't give you the ability to do anything you couldn't before. They allow the programmer to declare which classes are supposed to need to access which declarations.

Instance variables are usually `private`, while methods are usually `public`.

In Java, they are also a part of security, they prevent programmers from accessing things that would break the runtime system.

Accessibility is 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 the static types of things.

Access Rules

The accessibility of a member depends on:

1. How the member's declaration is qualified, and

2. Where it is being accessed

`public`

Suppose `C1`, `C2`, `C3` and `C4` are distinct classes, and class `C2a` is either the 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 P2;
class C2 extends C3 {
    void f(P1.C1 x) { ... x.M ... } // OK
    void g(C2a y) { ... y.M ... } // OK
}

package P1;
public class C4 ... {
    void p(C1 x)
        { ... x.M ... } // OK.
}
```

`public` members are available anywhere.

`private`

Suppose `C1`, `C2` and `C4` are distinct classes, and class `C2a` is either the 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 P2;
class C2 extends C1 {
    void f(P1.C1 x) { ... x.M ... } // ERROR
    void g(C2a y) { ... y.M ... } // ERROR
}

package P1;
public class C4 ... {
    void p(C1 x)
        { ... x.M ... } // ERROR.
}
```

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

`package private`

This is the hidden keyword: if you don't declare any access modifier, a function or variable automatically becomes `package private`.

Suppose `C1`, `C2` and `C4` are distinct classes, and class `C2a` is either the 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 P2;
class C2 extends C1 {
    void f(P1.C1 x) { ... x.M ... } // ERROR
    void g(C2a y) { ... y.M ... } // ERROR
}

package P1;
public class C4 ... {
    void p(C1 x)
        { ... x.M ... } // OK.
}
```

`package private` members are available only within the same package (even for subtypes)

`protected`

Suppose `C1`, `C2` and `C4` are distinct classes, and class `C2a` is either the 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 P2;
class C2 extends C1 {
    void f(P1.C1 x) { ... x.M ... }
    // ERROR: (x's type is not a subtype of C2)
    void g(C2a y) { ... y.M ... } // OK
    void g2() {... return M ...} // OK (this.M)
}

package P1;
public class C4 ... {
void p(C1 x)
    { ... x.M ... } // 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`.

Why 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.

Review

Observe the following code. For every comment `OK?`, determine whether the code will error or not:

```
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?
    }
}
```

Solution

```
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
    }
}
```

Access Control is Static Only

`public` and `private` don't apply to dynamic types; it's possible to call methods in objects of types you can't name:

```
package utils;
/** A Set of things. */
public interface Collector {
    void
    (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();
    }
}

package mystuff;
class User {
    utils.Collector c = utils.Utils.concat();

    c.add("foo"); // OK
    ... c.value(); // ERROR
    ((utils.Concatenator) c).value(); // ERROR
}
```

Some Loose Ends

Below are a collection of small Java features we haven't covered in class before we move to the next part of the course, which are important for you to know but didn't quite fit in anywhere else.

Importing

Unlike Python, you don't have to import a package to use its methods in your file. You don't have to import `java.util.List` to be able to use `List`, but importing it means you can use `List` as an abbreviation for `java.util.List`.

You can import all the methods of a package by using a `*` instead of `methodName`, so you can, for example, import `java.util.*` and use all of its methods without needing to mention the package every time.

This does not grant any special access to your program: you gain no additional abilities by importing vs. just using the long form.

You can also import static members of a class: an example is `System.out.println` or `Math.sqrt`, without having to mention the package every time. To do so:

```
import static java.lang.System.out;
// Tells Java out is an abbreviation for System.out

import static java lang.System.*;
// Tells Java we can use any static member name in
// System without mentionin the package
```

Much like before, this does not give you any benefits other than simplicity, and you cannot do this for classes in the anonymous package.

Nesting

Sometimes, like as we saw in Signpost, it makes sense to nest one class in another. We do this when: the class is only to be used in the implementation of the other, or is conceptually only useful to the outer class.

Nesting classes can help avoid name clashes, as we avoid "polluting the name space" with names that will not be used elsewhere.

As an example, polynomials can be thought of as sequences of terms, and terms are not useful outside of polynomials, so we might define a `Term` class inside of `Polynomial`.

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

The above is an example of a static nested class. They are similar to other classes, except that they can be private or protected, an they can see private variables of the otuer class.

Non-static nested classes are called **inner classes**. Somewhat rare is when each instance of the nested class is created by and naturally associated with an instance of the containing class, line `Bank`s and `Account`s.

```
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(...);
```

`instanceOf`

It is possible for us to ask about the dynamic type of an object using `instanceOf`:

```
void stringChecker(Object a) {
    if (a.instanceOf(String)) {
        System.out.println("yes");
    }
    System.out.println("no");
}
```

However, this is rarely what we want to do. Why do this:

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

when we can just call `x.read()`? In fact, C++ did not use to have this feature, because the original designer firmly believed you should not be using something if you didn't know what type it was.

In general, you want to use instance methods rather than `instanceOf`.