

CS 61B: Lecture 13
Friday, February 20, 2009

Today's reading: Sierra & Bates, pp. 154-160, 587-591, 667-668.

JAVA PACKAGES =====

In Java, a "package" is a collection of classes and Java interfaces, and possibly subpackages, that trust each other. Packages have three benefits.

- (1) Packages can contain hidden classes that are used by the package but are not visible or accessible outside the package.
- (2) Classes in packages can have fields and methods that are visible by all classes inside the package, but not outside.
- (3) Different packages can have classes with the same name. For example, `java.awt.Frame` and `photo.Frame`.

Here are two examples of packages.

- (1) `java.io` is a package of I/O-related classes in the standard Java libraries.
- (2) Homework 4 uses "list", a package containing the classes `DList` and `DListNode`. You will be adding two additional classes to the list package.

Package names are hierarchical. `java.awt.image.Model` refers to the class `Model` inside the package `image` inside the package `awt` inside the package `java`.

Using Packages -----

You can address any class, field, or method with a fully-qualified name. Here's an example of all three in one.

```
java.lang.System.out.println("My fingers are tired.");
```

Java's "import" command saves us from the tedium of using fully-qualified names all the time.

```
import java.io.File; // Can now refer to File class, not just java.io.File.
import java.io.*;   // Can now refer to everything in java.io.
```

Every Java program implicitly imports `java.lang.*`, so you don't have to import it explicitly to use `System.out.println()`. However, if you import packages that contain multiple classes with the same name, you'll need to qualify their names explicitly throughout your code.

```
java.awt.Frame.add(photo.Frame.canvas);
```

Any package you create must appear in a directory of the same name. For example, the `photo.Frame` class bytecode appears in `photo/Frame.class`, and `x.y.z.Class` appears in `x/y/z/Class.class`. Where are the `photo` and `x` directories? They can appear in any of the directories on your `CLASSPATH` environment variable (in Unix). For example, if you type "printenv `CLASSPATH`" on the lab machines:

```
% printenv CLASSPATH
./home/ff/cs61b/lib/ucb.jar      [plus some other stuff, omitted]
```

This means that Java first looks in ".", the current directory, and then looks in `/home/ff/cs61b/lib/ucb.jar` (which is a file of packages in a compressed format), when it's looking for the `photo` and `x` directories. The `CLASSPATH` does not include the location of the Java standard library packages (those beginning with `java`). The Java compiler knows where to find them.

Building Packages -----

The files that form a package are annotated with a "package" command, which specifies the name of the package, which must match the name of the directory in which the files appear.

```
/* list/SList.java */           | /* list/SListNode.java */
package list;                   | package list;
                                 |
public class SList {            | class SListNode {
    SListNode head;             |     Object item;
    int size;                   |     SListNode next;
}                                 | }

```

Here, the `SListNode` class and its fields are marked neither `public`, `private`, nor `protected`. Instead, they have "package" protection, which falls somewhere between "private" and "protected". Package protection is specified not by using the word "package", but by using no modifier at all. Variables are package by default unless declared `public`, `private`, or `protected`.

A class or variable with package protection is visible to any class in the same package, but not to classes outside the package (i.e., files outside the directory). The files in a package are presumed to trust each other, and are usually implemented by the same person. Files outside the package can only see the public classes, methods, and fields. (Subclasses outside the package can see the protected methods and fields as well.)

Before we knew about packages, we had to make the fields of `SListNode` `public` so that `SList` could manipulate them. Our list package above solves this problem by giving `SListNode` and its fields package protection, so that the `SList` class may use `SListNodes` freely, but outside applications cannot access them.

In Homework 4, you'll see a different approach. There, the `SListNode` class is `public`, so that `SListNodes` can be directly held by application programs, but the "item" and "next" fields have package protection, so an application cannot access these fields or corrupt the `SList` ADT. But an application can hop quickly from node to node because it can store `SListNode` references and use them as parameters in `SList` method calls.

Each public class must be declared in a file named after the class, but a class with package protection can be declared in a file with a different name (usually found together with a class that uses it). So a `public SList` class and a package `SListNode` class can both be declared in the file `list/SList.java`, if you feel like it.

Compiling and running files in a package is a bit tricky, because it must be done from outside the package, using the following syntax:

```
javac -g list/SList.java
java list.SList
```

Here's the correspondence between declarations and their visibility.

Declaration	Visible: in the same package	in a subclass	everywhere
"public"	X	X	X
"protected"	X	X	
default (package)	X		
"private"			

ITERATORS
=====

In java.util there is a standard Java interface for iterating over sequences of objects.

```
public interface Iterator {
    public boolean hasNext();
    public Object next();
    public void remove();           // The remove() method is optional.
}
```

next() is akin to "nextRun()" in Project 1. Each time you call next(), it returns the next Object in the sequence after the one previously returned. There are three differences.

- (1) Iterator is not implemented by the class it iterates through; rather, it is implemented by a separate class in the same package. For example, we can define an SListIterator in the list package to iterate through an SList. The advantage of this is that you can have many different SListIterators for a single SList, with each of them positioned at a different node.
- (2) There's no method to reset an Iterator to the beginning. (You can add one to your class, but it's not expected.) If you want to start again from the beginning of a list, create a new Iterator.
- (3) next() doesn't return null when there are no more elements. Instead, it throws an exception and halts with an error message. You should call hasNext() to find out if there are more elements before you call next().

Personally, I dislike properties (2) and (3), so I don't declare Iterators for my classes, but it's a de facto Java standard and you should know about it.

```
/* SListIterator.java */
package list;
import java.util.*;

public class SListIterator implements Iterator {
    protected SListNode n;

    public SListIterator(SList l) {
        n = l.head;
    }

    public boolean hasNext() {
        return n != null;
    }

    public Object next() {
        Object i = n.item;
        n = n.next;
        return i;
    }
}
```

There is also an "Iterable" interface for classes that can produce iterators with a method iterator(). In this example, SList would implement Iterable.

An iterator doesn't have to iterate over a data structure. For example, you can implement an Iterator class called Primes that returns each successive prime number.