CS 61B: Lecture 13
Wednesday, February 19, 2014

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". You can specify a classpath on the command line, as when you type

```
javac -cp ".:~jrs/classes:libraries.jar" *.java
```

This means that Java first looks in ".", the current directory, then looks in ~jrs/classes/, then finally in the _Java_archive_ libraries.jar 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 or javax). 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 DListNode class is public, so that DListNodes can be directly held by application programs, but the "prev" and "next" fields have package protection, so an application cannot access these fields or corrupt the DList ADT. But an application can hop quickly from node to node because it can store DListNode references and use them as parameters in DList 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 any .java file (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.

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

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

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

Part of Project 1 is to write a class RunIterator that implements an Iterator for your RunLengthEncoding class. Its purpose is to provide an interface by which other classes can read the runs in your run-length encoding, one by one.

An Iterator is like a bookmark. Just as you can have many bookmarks in a book, you can have many Iterators iterating over the same data structure, each one independent of the others. One Iterator can advance without disturbing other Iterators that are iterating over the same data structure.

The first time next() is called on a newly constructed Iterator, it returns the first item in the sequence. Each subsequent time next() is called, it returns the next item in the sequence. After the Iterator has returned every item in the sequence, every subsequent call to next() throws an exception and halts with an error message. (I find this annoying; I would prefer an interface in which next() returns null. The Java library designers disagree.)

To help you avoid triggering an exception, hasNext() returns true if the Iterator has more items to return, or false if it has already returned every item in the sequence. It is usually considered good practice to check hasNext() before calling next(). (In the next lecture we'll learn how to catch exceptions; that will give us an alternative way to prevent our program from crashing when next() throws an exception.)

There is usually no way to reset an Iterator back to the beginning of the sequence. Instead, you construct a new Iterator.

Most data structures that support Iterators "implement" another interface in java.util called "Iterable".

```
  public interface Iterable {
    Iterator iterator();
  }
```

It is customary for applications that want to iterate over a data structure DS to call DS.iterate(), which constructs and returns a DSIterator whose fields are initialized so it is ready to return the first item in DS.

A benefit of creating an Iterable class with its own Iterator is that Java has a simple built-in loop syntax, a second kind of "for each" loop, that iterates over the items in a data structure. Suppose we design an SList that implements Iterator. The following loop (which can appear in any class) iterates through the items in an SList l.

```
  for (Object o : l) {
    System.out.println(o);
  }
```

This loop is equivalent to

```
  for (Iterator i = l.iterator(); i.hasNext(); ) {
    Object o = i.next();
    System.out.println(o);
  }
```

To make all this more concrete, here is a complete implementation of an SListIterator class and a partial implementation of SList, both in the "list" package.

```
/* list/SListIterator.java */

package list;
import java.util.*;

public class SListIterator implements Iterator {
  SListNode n;

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

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

  public Object next() {
    if (n == null) {
      /* We'll learn about throwing exceptions in the next lecture. */
      throw new NoSuchElementException();                       // In java.util
    }
    Object i = n.item;
    n = n.next;
    return i;
  }

  public void remove() {
    /* Doing it the lazy way.  Remove this, motherf! */
    throw new UnsupportedOperationException("Nice try, bozo."); // In java.lang
  }
}

/* list/SList.java */

package list;
import java.util.*;

public class SList implements Iterable {
  SListNode head;
  int size;

  public Iterator iterator() {
    return new SListIterator(this);
  }

  [other methods here]
}
```

Observe that an Iterator may mess up or even crash the program if the structure it is iterating over changes. For example, if the node "n" that an SListIterator references is removed from the list, the SListIterator will not be able to find the rest of the nodes.

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