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.

```java
/* list/SLlist.java */
package list;

public class SLlist {
    SLlistNode head;
    int size;
    SLlistNode next;
}
```

Here, the SLListNode 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 SLListNode public so that SLList could manipulate them. Our list package above solves this problem by giving SLListNode its fields package protection, so that the SLList class may use SLListNodes freely, but outside applications cannot access them.

In Homework 4, you'll see a different approach. There, the DLListNode class is public, so that DLListNodes 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 DLList ADT. But an application can hop quickly from node to node because it can store DLListNode references and use them as parameters in DLList 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 SLList class and a package SLListNode class can both be declared in the file list/SLlist.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/SLlist.java
java list.SLlist
```

Here's the correspondence between declarations and their visibility.

<table>
<thead>
<tr>
<th>Visible:</th>
<th>in the same package</th>
<th>in a subclass</th>
<th>everywhere</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration</td>
<td>&quot;public&quot;</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td></td>
<td>&quot;protected&quot;</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>default (package)</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&quot;private&quot;</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
In `java.util` there is a standard Java interface for iterating over sequences of objects.

```java
public interface Iterator {
    boolean hasNext();
    Object next();
    // 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`".

```java
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`.

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

This loop is equivalent to

```java
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.

```java
/* 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 o = n.item;
        n = n.next;
        return o;
    }

    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.