Iterators etc

An iterator is an object that gives access to the contents of a collection. Iterators have two important, related benefits:

- They provide a universal, consistent interface. (Abstraction)
- They do not expose the collection's internal structure. (Encapsulation)

Thus using iterators we can write a loop over the elements in a collection that will be the same whether that collection is a list with an array inside, a linked list, or a set with any underlying implementation. As long as collection `c` has a method `iterator()`, we can write a loop like the following (assuming the base type of the collection is `T`):

```
for (Iterator<T> it = c.iterator();  it.hasNext(); ) {
       // do something with it.next();
}
```

Notice that not only is this code independent of whether `c` has an array or linked list or something else inside of it, it is even independent of what `c`'s other public operations are. It is independent both of `c`'s implementation and of `c`'s ADT or interface. An alternate definition is that an iterator is an object that encapsulates the *state of an iteration.* It is a bookmark of sorts hiding the information about where we are in a collection but presenting the programmer or client code with the needed functionality.

For beginner programmers, the tricky part of *using* iterators is that the `next()` method does two things. It both retrieves the next item from the collection and modifies the state of the iterator to get it ready for the next call to `next()` (and `hasNext()`). Compare the template above for an iterator loop with loops for iterating over arrays and linked lists:

```
for (int i = 0;
     i < a.length;
     i++) {
     // do something with a[i]
}
```

```
for (Node current = head;
     current != null;
     current = current.next) {
     // do something with current.datum
}
```

The `next()` method thus plays the role both of the retrieving of the element in the body of the loop (`a[i]` or `current.datum`) and of the increment (`i++` or `current = current.next`). A pitfall for novices is calling `next()` twice in the body of a loop and expecting it to be the same thing. This is especially easy to do when quickly adding debugging output:

```
int sum = 0;
for (Iterator<Integer> it = c.iterator(); it.hasNext(); ) {
    System.out.println(it.next());
    sum += it.next();      // AAAAHHHHH!!!!!!!!
}
```

If you want to do more than one thing with a item you get from an iterator, you must store it in a variable:

```
int sum = 0;
for (Iterator<Integer> it = c.iterator(); it.hasNext(); ) {
    int x = it.next();
    System.out.println(x);
    sum += x;      // much better.
}
```

I am a big fan of the *writing* of iterators as a test or project problem, because it gets students thinking about how to encapsulate the state of an iteration over a collection—among other things. Consider these two static methods that each return an iterator given an array or the head of a linked list.

```
static <T>
Iterator<T> arrayToIterator(final T[] array) {
    return new Iterator<T>() {
        int i =0;
        public boolean hasNext() {
            return i < array.length;
        }
        public T next() {
            if (! hasNext())
                throw new
                    NoSuchElementException();
            else return array[i++];
        }
    };
}
```

```
static <T>
Iterator<T> llToIterator(final Node<T> head) {
    return new Iterator<T>() {
        Node current = head;
        public boolean hasNext() {
            return current != null;
        }
        public T next() {
            if (! hasNext())
                throw new
                    NoSuchElementException();
            else {
                T toReturn = current.datum;
                current = current.next;
                return toReturn;
            }
        }
    };
}
```

Iterators are so useful and so widely used that modern programming languages like Java and Python provide not only library support for them but also integrate them into the languages themselves. (I just referred to Java and Python, which came out in the '90s, as *modern* programming languages. I guess I'm getting old.) Python's for loops are based on the iterator concept, and Java has for-each loops (sometimes called *enhanced for loops*). A for-each loop is really just shorthand for a for loop using an iterator. That is, the following two loop templates compile to the same thing:

```
for (T x : c) {
   // do something with x
}
```

```
for (Iterator<T> it = c.iterator();
     it.hasNext(); ) {
    T x = it.next();
    // do something with x
}
```

In a for-each loop, the iterator itself becomes invisible. Not only is this very readable, it also eliminates the easy mistake mentioned above about calling `next()` twice in a loop body. Now, here's the point that prompted me to write all this: What needs to be assumed about `c` in order to use it in the above form? The collection `c` needs to support a method `iterator()` with return type `Iterator<T>`. Java captures that constraint with the `Iterable` interface:

```
interface Iterable<T> {
    Iterator<T> iterator();
}
```

Thus an *iterator* is an object that encapsulates the state of an iteration over a collection; an *iterable* is a collection that can be iterated over. Our ADT interfaces `List`, `Set`, `Map`, and `Bag` each extend the `Iterable` interface. And this is why the `adjacents()` method in our `Graph` interface returns an `Iterable` instead of an `Iterator`, so that what is returned from that method can be used in a for-each loop:

```
Graph g;
int u;

// ...

for (int v : g.adjacents(u)) {
    // do something with v
}
```

But someone may ask, why can't Java simply allow `c` in the template above to be *either* an `Iterable` or an `Iterator`? Why couldn't the following code on the left be translated into the code on the right if `c` happens to be an `Iterator`?

```
for (T x : c) {
    // do something with x
}
```

```
while (c.hasNext(); ) {
    T x = c.next();
    // do something with x
}
```

Well, in theory Java could have been designed to do that. I think the argument against it is that it would make it easier to write a loop that uses a "stale" iterator, that is, one that is already partly through its iteration, and that some other part of the code might have a reference to, which would make concurrent use of the iterator object more likely. The for-each loop as it is defined forces the loop to be used with a fresh iterator from a call to an `Iterable`'s `iterator()` method.

Exercise. There's always a work-around. Write a static method `iteratorToIterable()` that takes an `Iterator` and returns an `Iterable` wrapper for the given `Iterator`, adapting it to be used in a for-each loop. (This is not difficult at all if you've understood the principles up to this point. You can find a solution on the last page.)

One more thing to learn about iterators. The way we think of iterators (or, really, the way iterators are done in Java and Python and similar popular languages) isn't the only way. The *Design Patterns* book gives an alternative, which seems mostly to have been forgotten over the years. They call the kind of iterator we know the *external* iterator, external in the sense that it uses an object external to the collection being iterated over. With the external iterator, the client code says, "Hey collection, I want do some operation on every element in you. Please give me a way to retrieve each of those elements."

The alternative, called the *internal* iterator, switches it up, with the client code instead saying, "Here's the operation that I want to do on your elements. Could you please see that it gets applied to all of them?" This is more in line with functional programming, since the operation to be performed on every element in a collection is a function object. Here is an array-list-like class that supports the internal iterator pattern:

```
public class IIAL {

    public static interface Operation {
        void execute(int x);
    }

    private int[] internal;

    public void internalIterator(Operation op) {
        for (int i = 0; i < internal.length; i++)
            op.execute(internal[i]);
    }
}
```

And here is a method that would compute the sum of a given `IIAL`:

```
static int sumIIAL(IIAL c) {
    IIAL.Operation summer = new IIAL.Operation() {
        int sum = 0;
        public void execute(int x) { sum += x; }
    }
    c.internalIterator(summer);
    return summer.sum;
}
```

Solution to the exercise.

```
static <T> Iterable<T> iteratorToIterable(final Iterator<T> it) {
    return new Iterable<T>() {
        public Iterator<T> iterator() {
            return it;
        }
    };
}
```