Computer Science 501
Data Structures & Algorithms
The College of Saint Rose
Fall 2014

Topic Notes: Iterators

Interfaces and Abstract classes

Before we discuss `Iterator`s, we need to think about the design of abstract data types in Java. So far, we have seen interfaces and regular classes. There is a level between these called an *abstract class*.

The abstractions provided by interfaces and abstract classes are important for the development of reusable and modular software.

We want to be able to define *what* an abstract data type does without committing to *how* it does it.

The biggest example we've seen so far is a `Vector`. As the user of a `Vector`, we know we can create them, add, retrieve, remove, and modify elements in them, and query information like their size. All of these are independent of how the `Vector` is implemented.

This separation of the public interface from the implementation allows programmers to make use of `Vector`s without needing to know how things work on the inside. It also allows the implementers of `Vector`s to make internal changes without affecting other code that uses it, so long as the public interface does not change.

Java has language constructs to support the development of abstract data types.

- *Interfaces* describe the public functionality of an abstract data type. This includes:
  - method signatures
  - constants

  An `interface` may `extend` another `interface`.

  We have seen and used `interface`s including things like `Comparator`s and `Comparable`s.
- *Abstract base classes* describe a partial implementation. An `abstract` class can define method bodies for some of the methods required by `interface`s it `implements`.

  This can be useful for:
  - methods that can be implemented in terms of other methods

It is possible for a class that extends an `abstract` class to override methods defined in the `abstract` class, in case there is a more efficient way to do some of these things when an actual implementation is developed.

A frequent use of an `abstract` class is to "factor out" implementation of methods that happen to be the same for multiple implementations of an interface.

- Full implementations (classes that you can instantiate) may `implement interface`s, and/or `extend` exactly one `abstract` or fully implemented class.

We will see many examples using interfaces and abstract classes throughout the rest of the course.

The text has examples using "generators" and the design of playing card classes to motivate `interface`s and `abstract` classes, and I encourage you to read them.

However, we consider them in the context of `Iterator`s.

Iterators

How do we "visit" each item in a collection? With a `Vector`, or an array, it's easy. We can write a `for` loop:

```
public <T> void traverse(Vector<T> v) {
  int i;

  for (i=0; i<v.size(); i++) {
    T visitme = v.get(i);
    // do something with visitme
  }
}
```

But imagine if someone has changed the implementation of `Vector`. It no longer has an array, but a linked structure.

We will study linked lists next, but for now, just notice that to get access to the n^{th} element, we need to visit the first $n-1$ elements. If our `Vector` contained one of these linked structures instead of an array, our traverse method suddenly becomes very inefficient.

This is not good. What is the complexity of `get()`? In order to get the item at position i, we have to start at the beginning and we have to follow links until we find the right element.

What we want to do is to use the previous value returned, and take the one pointed to by the list element we just used to get that previous value. But how? We don't have that information!

We often need a way of cycling through all of the elements of a data structure. Java and the structure package provide exactly what we need: `java.util.Iterator<E>`

A data structure can create an object of type `Iterator`, which can be used to cycle through the elements. For example, built-in Java class `Vector` has method:

```
public Iterator<E> iterator()
```

that we can print out the elements of `Vector<E> v` as follows:

```
for (Iterator<E> iter=v.iterator(); iter.hasNext(); )
    System.out.println(iter.next());
```

Or in Java 5 and up, if our class implements the `Iterable interface` (which simply requires the method `iterator`) we can use a "for each" loop:

```
for (E item: v) {
  System.out.println(item);
}
```

See Example:
`/home/cs501/examples/Iterables`

Important Notes:

- Never change the state of a data structure with an active works, or you may end up in an infinite loop!
- There is also a `remove()` methos in Java's `Iterator` interface, but we will ignore that for now, as not all iterators provide it.
- `Iterator`s guarantee a predictable and consistent order of the elements returned.

The structure package defines an abstract class called `AbstractIterator` that `implements` both `java.util.Enumeration` (an older, iterator-like interface) and `java.util.Iterator`.

Notes about `AbstractIterator`s:

- it adds two methods that are not part of either enumerations or iterators in Java:
  - `reset()` – start the iteration over without constructing a brand new `Iterator`
  - `get()` – retrieve the "current value" without advancing the `Iterator`

  Both of these are declared as `abstract` methods, meaning that they are essentially adding to the `interface` defined by the `AbstractIterator` without defining them.
- The `AbstractIterator` also provides *implementations* of the two methods required by enumeration. The fact that these are implemented is what requires that we declare this as an `abstract` class rather than just another `interface`. Note that these are also declared as `final` meaning that classes that extend this class may not override their definitions. This ensures that the `Enumeration` and `Iterator` methods of any class that `extends AbstractIterator` must be identical.

The data structures in the structure package typically return an `AbstractIterator` rather than a `java.util.Iterator`.

See Structure Source:
`/home/cs501/src/structure5/VectorIterator.java`

See Example:
`/home/cs501/examples/Iterators`

In our next lab, you will develop an unusual kind of `Iterator` – one that is iterating over a collection of values that doesn't actually exist!