The Comparable Interface

There are two main ways, in practice, in which interfaces are used in Java programming:

(1) To capture functionality orthogonal to the natural class hierarchy; and

(2) To define a class for which there is no natural default implementation.

We'll look at examples of both, taken from the Java library itself.

Let's look at the first item above. What do I mean by orthogonal functionality? Suppose we are interested in sorting in our university management application, that is, we want to define a method to sort an array of objects.

We can easily sort integers, using some sorting algorithm, e.g., bubble sort. (I like bubble sort — we go back a long way. Don't ever use it, though.)

```
public static int[] bubbleSort (int[] input) {
  // some bubble sort implementation
  // using tests such as input[i] < input[j], etc
}
```

We could also sort strings, presumably, by changing the type of the method to accept arrrays of strings, and changing the implementation by changing the $<$ operator by one suitable for strings. Notice, however, aside from this small bit, the rest of the code ought not to change. Basically, as long as you know how to compare to objects of a given class to be able to say that one is less than the other (for some ordering that you care about), then you can sort those objects. The sorting algorithms you have learned are *parametric* in what's exactly to be sorted, as long as there is a well-define notion of ordering.

So, what we can do is define a method to compare objects in our class, and tell Java that there is such a method by defining an interface and having classes implement that interface. (This is a little bit like providing objects with an `equals` method — the difference of course is that all classes subclass the `Object` class which defines the method; not so for ordered comparisons.) Here is such an interface:

```
public interface Comparable {

  public int compareTo (Object);
```

```
}
```

(This is actually an interface defined in the Java library, but for clarity I am reproducing it here.) The idea is that `v1.compareTo(v2)` returns a negative number ig `v1` is less than `v2` in the ordering, a positive number if `v1` is greater than `v2` in the ordering, and 0 if `v1` and `v2` are equal in the ordering.[1]

Every class in the system for which a notion of ordering can be defined can implement that interface. We can then define a generic sort method such as:

```
public static Comparable[] bubbleSort (Comparable[] input) {
  // same algorithm as before, but using method
  // compareTo() to compare objects with respect to the ordering
}
```

Now, this means that every class we care about ordering in our system subclasses `Comparable`, on top of all the other superclasses it subclasses based on the natural hierarchy. (For instance, `TA` maybe subclasses `Comparable` if we care about sorting `TA`s, on top of subclassing `Student` and `Salaried`.) The subclassing from `Comparable` is not due to the application domain, but rather to something orthogonal, in this case maybe an efficiency issue. (It is easier and faster to manipulate large data if it is sorted than not.)

Let's look at another example, where we make stacks implement the `Comparable` interface. This requires coming up with some ordering on stacks. Let's pick a simple one, where a stack is less than another if the former has less elements than the latter.

```
public class Stack implements Comparable {

  ... // some implementation, such as before

  private int length () {
    if (this.isEmpty())
      return 0;
    return 1 + this.pop().length();
  }

  public int compareTo (Object obj) {
    if (obj instanceof Stack) {
      Stack that = (Stack) obj;
```

[1]There are also implicit specifications for the `compareTo` method which cannot be enforced by Java, but are required for proper behavior. For instance, `compareTo` should be transitive. Also, `v1.compareTo(v2)` > 0 if and only if `v2.compareTo(v1)` < 0, and `v1.compareTo(v2)` $= 0$ if and only if `v2.compareTo(v1)`. There are a few others; see the Java API for more information.

```
        int l1 = this.length();
        int l2 = that.length();
        return (this < that ? -1 : (this > that ? 1 : 0));
    }
}
```

There. Now, we can construct arrays of stacks, and pass it to our (imaginary) bubble sort method, which will sort the stacks in increasing order of size.

(There is some yuckiness in the `Comparable` interface though, the fact that we have to compare against an object because we do not know in advance the type of the objects we will be comparing against. We'll see one way of cleaning that up next time.)

The Functional Iterator Interface

What about item (2) above, classes for which there is no default implementation?

Again, let's look at an example. First, some terminology. An *aggregate* is a data type that contains objects, such as a list data type, or a tree data type, or a hash table data type, or a stack data type. Arrays are typical aggregates. Arrays have the added advantage that there is nice built-in syntax for them, including easy ways of iterating over all the elements of an array using a `for` loop.

This idea of iteration can be generalized to all aggregates. An *iterator* is an object whose sole purpose is to make it easy to iterate over all the elements of an aggregate.

What's an iterator? An iterator is an object that implements the following interface:

```
public interface IFunIterator {
  public boolean hasNext ();
  public int next ();
  public IFunIterator advsnce ();
}
```

(I like to use interface names starting with `I`—this is purely personally, and goes back to my days programming COM.) This is in fact called a *functional iterator* interface. Functional is often used as a synonym for immutable. A functional iterator provides a method `hasNext()` for asking whether we are done iterating over the elements of the underlying aggregate, a method `next()` for getting the next element of the aggregate we have not looked at yet, and a method `advance()` that advances the iterator past the next element so that we can look at the element after than. The `advance` method returns a new iterator that can be queried for that next element.[2] The point here is that different aggregates will require quite different

[2]Java comes with an iterator interface in its basic API that is mutable; it does not have an `advance()`

iterators—there is no notion of a default implementation of an iterator that works for all aggregates. Thus, we use an interface instead of a class above.

Stacks are aggregate, so let's define an iterator for stacks. The first thing we need to do is add a method to the `Stack` class that gives us a functional iterator to iterator over the implicit stack argument. That iterator creates a new instance of a stack iterator, which is a class that we define nested inside the `Stack` class. (It does not need to be nested inside, but since we only ever create instances of it from within the class, we might as well in order not to pollute the namespace.)

There are many ways of implementing iterators. Let's pick a fairly simple minded one that we will improve next time. Roughly, we create an iterator by passing it an array containing all the objects to iterate over. We create such an array in the `getFunIterator()` method that will create the iterator instance. Let's hack on the `Stack` class above with its private `length()` method.

```
public class Stack implements Comparable {

  ... // original code

  // this is new
  public IFunIterator getFunIterator () {
    int[] content = new int[this.length()];
    Stack current = this;
    for (int i = 0; i < content.legnth; i++) {
      content[i] = current.top();
      current = current.pop();
    }
    return new StackFunIterator (content,0);
  }

  private static class StackFunIterator implements IFunIterator {
    private int[] content;
    private int index;

    // constructor takes the array of content and the initial index
    public StackFunIterator (int[] c, int i) {
      content = c;
      index = i;
    }
```

method, and query for the next element in the interface mutates the iterator under the hood so that querying it for the next element returns yet a new element. This mutability, as usual, makes it somewhat more difficult to reason about iteration (but arguably a bit more efficient—the usual trade-off).

```
    public boolean hasNext () {
      return (this.index < content.length);
    }

    public int next () {
      if (this.hasNext())
        return content[this.index];
      throw new java.util.NoSuchElementException ("Stack iterator empty");
    }

    public IFunIterator advance () {
      if (this.hasNext())
        return new StackFunIterator (content,index+1);
      throw new java.util.NoSuchElementException ("Stack iterator empty");
    }
  }
}
```

The exception thrown when trying to get at the next element when there is no such element, or advancing the iterator passed the end of the stack is the one that the Java API requires its iterator to throw, so I have done so here for consistency.

Something to think about for next time: (1) How would you use such an iterator? (2) How can you make the above more efficient? (Because note that you are essentially iterating twice over the content of the stack, once when building the iterator, once when actually iterating over it.)