Java `Comparable` interface

Recall that to define a binary search tree, the elements that we are considering must be comparable to each other, meaning that there must be a well-defined ordering. For strings and numbers, there is a natural ordering, but for more general classes, you need to define the ordering yourself. How?

Java has an interface called `Comparable<T>` which has one method `compareTo(T)` that allows an object of type `T` to compare itself to another object of type `T`. By definition, any class that implements the interface `Comparable<T>` must have a method `compareTo( T )`.

The `compareTo()` method returns an integer and it is defined as follows. Suppose we have variables

```
T  a1, a2;
```

Then the Java API specifies that `a1.compareTo(a2)` should return:

- a negative integer, if the object referenced by `a1` is "less than" the object referenced by `a2`,
- 0, if `a1.equals(a2)` returns true
- a positive integer, if the object referenced by `a1` is "greater than" the object referenced by `a2`.

By "less than" or "greater than" here, I just mean the desired ordering – what you want if the elements are to be sorted or put in a binary search tree or priority queue, etc.

Example: `Circle`

To get a flavour for this, let's define a Java class `Circle` that implements the `Comparable` interface. Let `Circle` have a fields `radius` and perhaps other fields methods such as mentioned last lecture. How should we implement the `compareTo()` method for the class `Circle`? We could compare circles by the comparing their radii.

```
public class Circle implements Comparable<Circle>{
   private radius;

   public Circle(double radius){    // constructor
        this.radius = radius;
   }

    public boolean  equals(Circle c) {
       return radius == c.getRadius();
    }

    public int  compareTo(Circle c) {
         return radius - c.getRadius();
    }
}
```

These definitions make `compareTo()` consistent with `equals()`.

Example: `Rectangle`

The usage of the `Comparable` interface sometimes can be a bit subtle. To get a flavour for this, let's define a Java class `Rectangle`, which has two fields `height` and `width`, along with getters and setters and a method `getArea()`.

When should two `Rectangle` objects be equal? One intuitive definition of equality is that they should have the same `width` and same `height`. The problem with this definition is that it becomes difficult to say if one rectangle is bigger or smaller than another. If one rectangle has bigger width *and* height, then it is obvious how to order them. But if one rectangle has a bigger width but a smaller height, then it is not obvious how to order them. In this sense, it is not obvious how to define a `compareTo` method for rectangles.

There are options. For example, we could compare rectangles by their area. Or we could compare them by only their height, or only their width. In the code below, we choose to compare by area.

```
public class Rectangle implements Comparable<Rectangle>{  // ignore Shape
   private width;
   private height;

   public Rectangle(double width, double height){    // constructor
      this.width = width;
      this.height = height;
   }

   public double getArea(){
      return width * height;
   }

   public boolean equals(Rectangle other){

     //    return  (this.height ==  other.height) && (this.width == other.width);
     //
     //    Not consistent with the compareTo method below.

      return getArea == other.getArea();
   }

   public int  compareTo(Rectangle r) {
      return getArea() - r.getArea;
   }
}
```

Note that I have used the `equals(Rectangle)` method (overloading) rather than `equals(Object)` method (overriding), and I did the same thing in the `Circle` example. Java recommends not doing this, but rather it recommends overriding rather than overloading. This takes extra work because you need to cast from `Object` to `Rectangle` and there are other subtleties that I won't go into.

`Iterator` interface

We have seen many data structures for representing collections of objects including lists, trees, hashtables, graphs. Often we would like to visit all the objects in a collection. We have seen algorithms for doing this, and implementaions for some data structures such as linked lists.

Because iterating through a collection is so common, Java defines an interface `Iterator<E>` that makes this a bit easier to do.

```
interface Iterator<E>{
   boolean   hasNext();
   E         next();    //  returns the next element and advances
}
```

In fact there are a few different motivations for using iterators. One is that you sometimes are writing methods that do certain things to all of the objects of a collections, but the details of how the collection is organized are just not relevant. For example, you might want to set the value of a particular boolean field to true for each object of the collection. You would like to be to this with one or two lines of code.

A related motivation is that for some classes, like the Java `LinkedList` class, if you iterate through by using the `get(i)` method for all `i`, then this is very inefficient namely it would be $O(n^2)$ instead of $O(n)$. You can use an enhanced for loop for this, although as I'll mention again below, the enhanced for loop in fact is implemented "under the hood" with an `Iterator` object.

Another motivation is that you might want to use several different iterators at the same time. This would be awkward to do with a loop, whether it is an enhanced for loop or some other loop. As an analogy in the real world, consider a collection of quizzes that need to be marked (assuming they were written on paper). Each quiz is an object. Suppose each quiz consisted of four questions and suppose there were four T.A.'s marking the quizzes, namely each T.A. marks one question. Think of each T.A. as an iterator that steps through the quizzes. Different TAs might be grading different quizzes at any time.

Consider how the `Iterator` interface might be implemented for a singly linked list class. There would need to be a private field (say `cur`) which would be initialized to reference the same node as `head`. The constructor of the iterator would do that initialization. The `hasNext()` method would checks if `cur == null`. The `next()` method advances to the next element in the list. Naturally, you would only call `next()` if `iterator.hasNext()` returns `true`.

[ASIDE: *The method names* `next` *and* `hasNext()` *are admittedly confusing in the context of linked lists where we think of "next" as the node that comes next, rather than the currrent node. You'll simply have to accept that in the context of Iterators the word "next" will be used more like cur. In particular, if the list has just one element then at the initial stage,* `hasNext` *would return true, namely the* `cur` *variable (in my implementation) would reference that element's node.*]

```
private class  SLL_Iterator<E>   implements   Iterator<E>{

    private SNode<E>    cur;

    SLL_Iterator( SLinkedList<E>     list){            //   constructor
        cur = list.getHead();
    }

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

    public   E   next() {
        E    element  =  cur.getElement;
        cur = cur.getNext();
        return element;
    }
}
```

`Iterable` interface

The `SLinkedList` class does not implement the `Iterator` interface. Rather, a `LinkedList` construct objects that implement the `Iterator` interface. How is this done? Iterator objects are constructed by a method `iterator()` which is a method in a interface called `Iterable`.

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

This means that the `SLinkedList` has a method called `iterator()` that returns an object whose class type implements the `Iterator` interface. The idea here is that if you have a collection such that it makes sense to step through all the objects in the collection, then you can define an iterator object to do this stepping (if you want) and, because you can do this, you would say that the collection is "iterable". This idea applies not just to linked lists, but to any collection you want to iterate over (trees, graphs, heaps, ...).

The following singly linked list class was given to you way back in the exercises when I first covered singly linked lists. Did you check it out then? If not, then now you have another opportunity! Note in particular the inner private class `SLL_Iterator` which I already described above, and also the `iterator` method which has a very simple implementation.

See slides for illustrations which hopefully will be helpful.

```
class   SLinkedList<E> implements Iterable<E> {

    SNode<E>   head;

    private   class   SNode<E> {
        SNode<E>   next;
        E                   element;
        .
    }

    private   class  SLL_Iterator<E>    implements   Iterator<E>{
        ..   //  see previous page
    }

    SLL_Iterator<E>   iterator()   {
        return  new SLL_Iterator( this );
    } ;
}
```

Think of the `iterator()` method as a "constructor" for `Iterator` objects. Note that `Iterator` is not a class it is an interface. Different `Iterable` classes require quite different iterators, since the underlying data structures are so different.

Java "enhanced for loop"

The "killer app" of the `Iterable` interface is the enhanced for loop. For example, if you have list of `String` objects, you can iterate through them with:

```
for (String   s   :   list)  {
   System.out.println( s );
}
```

The enhanced for loop will work for any class that implements `Iterable` interface. Indeed, this enhanced for loop just gets compiled into equivalent statements that use iterators.