Iterators
Any “iterable” collection type provides a way of obtaining an Iterator (or ListIterator) to look at elements in some order.

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

class TreeSet<E>
    implements Iterable<E>
    ...

interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove(); // Optional operation
}
An iterator provides a way of looking at the elements in order, even if the original collection is not a simple list or array.

TreeSet<Stuff> t;

ListIterator<Stuff> i = t.listIterator();
TreeSet<Stuff> t;

ListIterator<Stuff> i = t.listIterator();

The iterator doesn’t really make a new list with the elements in order. It just knows how to keep a finger in the collection.
 TreeSet<Stuff> t;

ListIterator<Stuff> i = t.listIterator();
Stuff s = t.next();

The “next()” method of an iterator moves the finger, and returns the item it moved over.
The iterator finger is actually between elements.

ListIterator<Stuff> i = t.listIterator();
next() moves the finger forward and returns an element

ListIterator<Stuff> i = t.listIterator();

...
The whole purpose of iterators is to make it easier to write loops, like this:

```java
for (ListIterator<Appt> blocks = this.appts.listIterator();
     blocks.hasNext();)
{
    Appt block = blocks.next();
    ...
}
```

or even better, like this (if we don’t need to modify the collection):

```java
for (Appt block: this.appts)
{
    ...
}
```

(which is really creating an iterator and calling the “next()” and “hasNext()” methods for us)