An iterator that is an inner class

We develop an iterator for class Stack of the last video. Below is class Stack, with its method bodies elided because do not have to look at them. We make the iterator an *inner class*, placing it right after the fields.

```
import java.util.EmptyStackException;
import java.util.Iterator;
import java.util.NoSuchElementException;

/** An instance is a stack */
public class Stack<E> implements Iterable<E> {
   private E[] b;  // stack values are b[0..h-1],
   private int h;  // with b[h-1] at the top


// PUT THE INNER CLASS HERE

   /** Constructor: a stack of at most m values */
   public Stack(int m) {}

   /** Push e onto the stack. If there is no room, throw a RuntimeException("no space") */
   public void push(E e) {}

   /** Pop and return the top stack value. If the stack is empty, throw an EmptyStackException. */
   public E pop() {}

   /** = the size of the stack */
   public int isEmpty() {}
}
```

[We will write the inner class below].

We proceed as we did when writing the iterator over the even values of an array, but using an inner class. We begin by putting in the comment and header of the iterator and stubbing in two two required methods, hasNext() and next(). Note that we make the inner class private, so users can't use it. We'll explain why at the end of this video.

```
/** An instance is an iterator over the stack elements, from top to bottom. */
private class StackIterator implements Iterator {

    /** = there is another element to enumerate */
    public @Override boolean hasNext() {
        return false;
    }

    /** Return the next element to enumerate.
      * Throw a NoSuchElementException if there is no next element. */
    public @Override E next() {
         return null;
    }
}
```

We now figure out what fields are needed. First, since this is an inner class, it can refer directly to fields b and n of the outer class, so there is *no* need for fields in StackIterator to contain the collection being enumerated. But we do need a field n, say, to help us keep track of the enumeration. Since the enumeration is from the stack top to the stack bottom, we can use a field n declared and described as follows:

```
int n; // b[n] is the next element to enumerate (n = -1 means there are no more to enumerate)
```

Next, the constructor of the inner class does not need a parameter, since the class uses fields b and h of the outer class as the collection to be enumerated. The constructor must initialize n to thruthify the class invariant. This is done by setting n to the index of the top stack element or to -1 if the stack is empty. This setting of field n is simple enough that no helper method is needed.

```
/** Constructor: an iterator over the stack elements, from top to bottom. */
public StackIterator() { n= h-1; }
```

An iterator that is an inner class

We fix method hasNext. Based on the class invariant, there is another element to enumerate if n is at least 0:

```
/** = there is another element to enumerate */
public @Override boolean hasNext() {
    return 0 <= n;
}
```

We complete method next(). It throws the necessary exception if there are no more elements to enumerate. Because of the simplicity of determining the next element to iterate, we can make the body of next() much simpler than usual. Subtract 1 from n, so that b[n] is the next element to enumerate, and return the desired element, b[n+1].

```
/** Return the next element to enumerate. Throw a NoSuchElementException if there is no next element. */
public @Override E next() {
   if (!hasNext()) throw new NoSuchElementException();
   n= n - 1;
   return b[n+1];
}
```

One more issue. To make it easy for the user to obtain an instance of object StackIterator, insert a method for it. The method creates a new object and returns it. This is why it is OK to make class StackIterator private.

```
/** Return an Iterator for enumerating the Stack, from top to bottom. */
public @Override Iterator iterator() {
    return new StackIterator();
}
```