EXERCISE 0: An ArrayStack Iterator

- Download **`precept2.zip`** from the precepts page, unzip the project and open it using IntelliJ.
- Open **`ArrayStack.java`** and follow along with the preceptor. The next page of this worksheet shows an annotated version of the code.

EXERCISE 1: A LinkedStack Iterator

Open **`LinkedStack.java`** and examine the code carefully. Following the same steps explained in **EXERCISE 0**, do the following:

(a) Make **`LinkedStack`** *Iterable* by implementing the **`Iterable`** interface and adding the method: **`public Iterator iterator().`**

(b) Create an inner class named **`LinkedIterator`** that implements the **`Iterator`** interface. Implement the **`next()`** and **`hasNext()`** methods such that iterating over the elements in the stack returns them in Last-In-First-Out (LIFO) order.

(c) Test the iterator in **`main()`** by creating a stack and pushing the command-line arguments starting at **`args[0]`**. Use a **`for-each`** loop to print out the elements in the stack in LIFO order.

(d) Consider the following piece of code:

```
Stack<Integer> myStack = new Stack<Integer>();

for (int i = 0; i < 3; i++)
    myStack.push(i);

for (int i : myStack)
    for (int j : myStack)
        System.out.println(i + " " + j);
```

- What is the output of this piece of code?

- How many iterator objects does it generate?

```java
public class ArrayStack<Item> implements Iterable<Item> {
    private Item[] a;
    private int n;

    public ArrayStack() {
        a = (Item[]) new Object[2];
        n = 0;
    }

    public void push(Item item) { … }

    public Item pop() { … }

    public Item peek() { … }

    public Iterator<Item> iterator() {
        return new ReverseArrayIterator();
    }

    private class ReverseArrayIterator implements Iterator<Item> {
        private int i;

        public ReverseArrayIterator() {
            i = n-1;
        }

        public boolean hasNext() {
            return i >= 0;
        }

        public Item next() {
            if (!hasNext()) throw new NoSuchElementException();
            return a[i--];
        }

        public void remove() {
            throw new UnsupportedOperationException();
        }
    }

    public static void main(String[] args) {
        ArrayStack<Integer> stack = new ArrayStack<Integer>();
        for (int i = 0; i < args.length; i++)
            stack.push(Integer.parseInt(args[i]));

        for (int num : stack)
            System.out.print(num + " ");
    }
}
```

1 *Promise to have a method named* `iterator()` *that returns an object of type* `Iterator`.

2 *Fulfill the promise! (required by the* `Iterable` *interface).*

3

4 *Promise to have methods* `next()` *and* `hasNext()`.

5 *Fulfill the promise! (required by the* `Iterator` *interface).*

6 *Works only because an* `ArrayStack` *is* `Iterable`.

EXERCISE 2: Insertion Sort

Consider an *organ-pipe* array that contains two copies of the integers 1 through n, first in ascending order, then in descending order. For example, here is the array when $n = 8$:

1 2 3 4 5 6 7 8 8 7 6 5 4 3 2 1

Note that the length of the array is $2n$, not n.

How many compares does **Insertion sort** make to sort the array as a function of n? Use tilde notation to simplify your answer.

EXERCISE 3: Running Time Order-of-Growth Analysis

For each of the following pieces of code, express the number of times **op()** is called as a *summation*. Try to simplify the sum using **Big-Theta** notation.

(a)

```
void f(int n) {
    if (n < 1) return;

    for (int i = 0; i < n; i++)
        op();

    f(n/2);
}
```

(b)

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= n; j += i)
        op();
```

(c)

```
for (int i = n; i >= 1; i--)
    for (int j = 1; j <= i; j *= 2)
        op();
```

(d)

```
for (int i = 1; i <= n; i++)
    for (int j = 1; j <= i; j++)
        for (int k = 1; k <= i; k++)
           op();
```