Client.java

```
import java.util.*; // Random

/**
 * @author joe
 */
public class Client {
    // test
    public static void main(String[] args) {
        testDoubleStack();
        testLeakyStack();
    }

    public static void testLeakyStack() {
        ArrayLeakyStack<Integer> als = new ArrayLeakyStack<>(6);

        for(int i = 0; i < 10; i++) {
            als.push(i);
            System.out.println("ALS: " + als);
        }

    }

    public static void testDoubleStack() {
        ArrayDoubleStack<Integer> ads = new ArrayDoubleStack<>(500, 1500);

        Random rand = new Random();

        boolean result;

        do {
            result = ads.bluePush(rand.nextInt()); // push random integer onto Blue stack
        } while(result);

        do {
            result = ads.redPush(rand.nextInt());
        } while(result);

        System.out.println("Blue stack size: " + ads.blueSize());
        System.out.println("Red stack size:  " + ads.redSize());

        Integer i = 0;

        do {
            i = ads.redPop();
            System.out.println("Red pop: " + i);
        } while(i != null);
    }
}
```

DoubleStack.java

```
/**
 * @author joe
 */
public interface DoubleStack<E> {

    /**
     * @param element element to add to blue stack
     * @return bool whether the operation was successful
     */
    boolean redPush(E element);

    /**
     * @param element E stored here on success
     * @return whether the operation was successful
     */
    E redPop();

    /**
     * @return current number of elements in red stack
     */
    int redSize();

    /**
     * @param element element to add to blue stack
     * @return bool whether the operation was successful
     */
    boolean bluePush(E element);

    /**
     * @param element E stored here on success
     * @return whether the operation was successful
     */
    E bluePop();

    /**
     * @return current number of elements in blue stack
     */
    int blueSize();
}
```

ArrayDoubleStack.java

```
/**
 * @author joe
 */
public class ArrayDoubleStack<E> implements DoubleStack<E> {
    // top is the number of elements currently in the stack
    private E[] redStack;  int redStackTop = 0;
    private E[] blueStack; int blueStackTop = 0;

    /**
     * default constructor, instantiates both stacks with capacity of 1000 elements
     */
    public ArrayDoubleStack() {
        this(1000, 1000);
    }

    public ArrayDoubleStack(int redCapacity, int blueCapacity) {
        redStack  = (E[])new Object[redCapacity];
        blueStack = (E[])new Object[blueCapacity];
    }

    private boolean arrayPush(E[] arr, int top, E element) {
        if(top == arr.length)
            return false;

        arr[top] = element;
        top++;
        return true;
    }

    private E arrayPop(E[] arr, int top) {
        if(top == 0) {
            return null;
        }

        top--;
        return arr[top];
    }

    @Override
    public boolean redPush(E element) {
        if(arrayPush(redStack, redStackTop, element)) {
            redStackTop++;
            return true;
        }
        return false;
    }

    @Override
```

```
    public E redPop() {
        return arrayPop(redStack, redStackTop);
    }

    @Override
    public int redSize() {
        return redStackTop;
    }

    @Override
    public boolean bluePush(E element) {
        if(arrayPush(blueStack, blueStackTop, element)) {
            blueStackTop++;
            return true;
        }
        return false;
    }

    @Override
    public E bluePop() {
        return arrayPop(blueStack, blueStackTop);
    }

    @Override
    public int blueSize() {
        return blueStackTop;
    }
}
```

LeakyStack.java

```
/**
 *
 * @author joey
 */
public interface LeakyStack<E> {
    /**
     * @param element to push onto the stack
     * @return whether the operation was successful
     */
    public boolean push(E element);

    /**
     * @return the popped element or null if failed
     */
    public E pop();

    /**
     * @return number of elements currently on stack
     */
    public int size();
}
```

ArrayLeakyStack.java

```
/**
 *
 * @author joey
 *
 * implemented as a queue with no end
 */
public class ArrayLeakyStack<E> implements LeakyStack<E> {
    // storage for elements
    E[] element_array;

    // updated with every insertion/deletion
    int num_elements = 0;

    public ArrayLeakyStack() {
        this(256); // default size
    }

    public ArrayLeakyStack(int capacity) {
        element_array = (E[])(new Object[capacity]);
    }

    private void shiftArray() {
        int s = element_array.length -1;
        for(int i = 0; i < s; i++) {
            element_array[i] = element_array[i+1];
        }
    }

    @Override
    public boolean push(E element) {
        if(num_elements == element_array.length) {
            shiftArray();
            num_elements--;
        }

        element_array[num_elements] = element;
        num_elements++;

        return true;
    }

    @Override
    public E pop() {
        if(num_elements == 0)
            return null;

        num_elements--;
        return element_array[num_elements];
```

```
    }

    @Override
    public int size() {
        return num_elements;
    }

    // Object method override
    @Override
    public String toString() {
        if(num_elements == 0)
            return "ArrayLeakyStack[::empty::]";
        String ret = "ArrayLeakyStack[ ";

        for(int i = 0; i < num_elements; i++) {
            ret += element_array[i].toString();
            ret += " ";
        }

        return ret + "]";
    }

}
```

screenshot of LeakyStack