Assignment 1

due 13/2/2015

1. This assignment involves the implementation from scratch of a simple ADT named ADT Deque defined overleaf.

2. *Study the ADT description carefully.* It will be impossible to complete a coherent implementation of the deque concept if you have only a woolly grasp of the behaviour of the ADT. In particular a deque is not a queue or a stack, so lazily cutting and pasting existing code isn't going to suffice.

3. Your implementation must be based on the concept of either

   - a left-justified array such as we saw in lectures in connection with `ArrayBasedStack.java`, or
   - (more difficult, but better) a "circular-array" approach akin to that which we saw in `ArrayBasedQueue.java`.

4. Work out how conceptually you might represent a deque using either of the concepts mentioned above. Decide how the various operations might be implemented in terms of this representation. You must map this out carefully before writing any code: *think first, code later*.

5. Your implementation must be generic *i.e.* capable of handling elements of any Java type. Refer to `ArrayBasedStack.java` *etc.* for some ideas on how to achieve this.

6. Ultimately you will need to produce three things

   - A suitable Java interface named `Deque.java`
   - A complete Java implementation named `ArrayBasedDeque.java`
   - A suitable tester class named `TestArrayBasedDeque.java`

7. Tackle the implementation incrementally, recompiling and testing at each stage. Ensure at each stage that what you have (a) compiles cleanly and (b) works correctly for those aspects that you have completed, even if other features remain incomplete temporarily.

   - Create a minimal

     public class . . .
     { . . . }

     file and add the bare minimum to allow it to compile cleanly (e.g. empty "stubs" for the various ADT methods).
   - Add your instance variable definitions and re-compile.
   - Add you constructor and re-compile.
   - And so on. Continue in this fashion fleshing out the various operations working from the simpler ones, such as isEmpty, towards the more complex ones.

8. It will be difficult to complete `ArrayBasedDeque.java` without testing and even more difficult to have complete confidence that your implementation is correct. *Testing is an integral part of the programming effort not an optional extra.* You should test as you go along not leave it all to the end.

   For debugging purposes it may prove helpful to have a method called `print` that prints out the items currently in the deque in some suitable format.

Place your test code within `TestArrayBasedDeque.java`. There is no need to use JUnit testing, plain Java will suffice.

Make sure you your testing is as rigorous as possible. Each feature of the implementation should be tested. For example, you should ensure you test deque containers capable of holding of various types.

The output of the test code should persuasively demonstrate the correctness of the implementation. Make sure that the test output is readable: don't just output an unintelligible series of numbers say.

9. You must adhere to the **cs2500** standards regarding Java coding style in all submission in this course. While some of *my* Java materials may depart from these rules occasionally, *yours* may not- not fair maybe, but my course, my rules!

10. Submissions will use the Moodle system. Submit your work in a single `.tgz` file named `a1.tgz`. This must contain the three files named in 6. You must include your name and id number in the comment of every file you write. Marks will be deducted for incorrect naming of files or using a format other than `.tgz`.

ADT Deque

ADT *Deque* or double-ended queue is a variation of the familiar Queue ADT that allows items to be added to or removed from either the front or the rear of the structure. It is an ordered container abstraction whose elements can be of any type, denoted by the placeholder "EltType" below. It supports the following operations.

- **first():** Return the first element of the deque. Illegal if the deque is empty. *Input:* None; *Output:* EltType.
- **last():** Return the last element of the deque. Illegal if the deque is empty. *Input:* None; *Output:* EltType.
- **insertFirst(e):** Insert a new element e at the front of the deque. *Input:* EltType; *Output:* None.
- **insertLast(e):** Insert a new element e at the rear of the deque. *Input:* EltType; *Output:* None.
- **removeFirst():** Remove and return the element at the front of the deque. Illegal if the deque is empty. *Input:* None; *Output:* EltType.
- **removeLast():** Remove and return the element at the rear of the deque. Illegal if the deque is empty. *Input:* None; *Output:* EltType.
- The ADT also supports the standard operations size and isEmpty.

The following illustrates the effect of a sequence of operations on an initially empty deque.

Operation	Output	D
addFirst(3)	–	(3)
addFirst(5)	–	(5, 3)
removeFirst()	5	(3)
addLast(7)	–	(3, 7)
removeFirst()	3	(7)
removeLast()	7	()
removeFirst()	ERROR	()
isEmpty()	true	()