Université d'Ottawa
Faculté de génie

École de science informatique
et de génie électrique

uOttawa

University of Ottawa
Faculty of Engineering

School of Electrical Engineering
and Computer Science

Introduction to Computing II (ITI 1121)
Midterm Examination: Part 2 of 2

Guy-Vincent Jourdan and Mehrdad Sabetzadeh

March 2021, duration: 60 minutes

Instructions

1. This is an open book examination and the only authorized resources are:
   - Course lecture notes, laboratories, and assignments;
   - Java Development Kit (JDK) on your local computer.
2. By submitting this examination, you agree to the following terms:
   - You understand the importance of professional integrity in your education and future career in engineering or computer science.
   - You hereby certify that you have done and will do all work on this examination entirely by yourself, without outside assistance or the use of unauthorized information sources.
   - Furthermore, you will not provide assistance to others.
3. Anyone who fails to comply with these terms will be charged with academic fraud.

Marking scheme

Question	Maximum
1	**10**
2	**10**
Total	**20**

Question 1 (10 marks)

The goal of this question is to complete the implementation of classes **A** and **B** such that they both override properly the method **equals** that they inherited from the class **Object**, and that they both implement the interface **DeepCopyable** that we provide.

Instances of the class **A** have three instance variables of primitive type **int**, respectively called **a, b** and **c**. These variable are set in the constructor. Instances of the class **B** have two instance variables of reference type **A**, respectively called **a1** and **a2**. These variable are set in the constructor, and getters for both variables are available.

We have provided you with a shell implementation of both classes **A** and **B** . The declaration of the instance variables, as well as the class constructors and the necessary getters have been provided to you and should not be modified. You need to complete these classes to answer the questions below.

Question 1.1 Method equals

The first task is to override the method **equals** inherited from the class **Object**. You need to provide the implementation for both classes **A** and **B**. Your implementation should be accurate and reliable, ready to accommodate any possible input. As could be expected, two instances of one of the class are equal if their instance variables are equal.

Question 1.2 Interface DeepCopyable

We have provided the following interface:

```
public interface DeepCopyable {
        DeepCopyable deepCopy();
}
```

As you have guessed, the method **deepCopy** should return a **deep copy** of the instance on which the method is called. We have introduced the concept of deep copying in assignment 2, and the explanation is repeated here in Appendix A.

The goal of this question is to modify classes **A** and **B** so that they both properly implement the interface **DeepCopyable**, and therefore both provide a reliable deep-copy feature of themselves.

To help you test your solution, we provide you with the following test file:

```
public class TestDeepCopy {

        public static void main(String[] args){

          B b1 = new B(new A(1,2,3), new A(4,5,6));
          B b2 = new B(new A(7,8,9), new A(10,11,12));

          System.out.println(b1.equals(b2)); // false
          System.out.println(b1.equals(b1.deepCopy())); // true
          System.out.println(b1==b1.deepCopy()); // false
          System.out.println(b1.getA1().equals(((B)(b1.deepCopy())).getA1())); //true
          System.out.println(b1.getA1()==(((B)(b1.deepCopy())).getA1())); //false

        }
}
```

Running that test on a correct implementation will produce the following output:

```
$ javac TestDeepCopy.java
$ java TestDeepCopy
false
true
false
true
false
$
```

Important Restrictions for Question 1

- You **cannot** import anything at all.
- You **cannot** use any other class than the ones we provide.
- Each question requires that you add a single method to each class. So that is one method to add to both classes for Question 1.1, and one method to add to both classes for Question 1.2. You **cannot** add any other methods than these two methods, and you **cannot** add any instance or class variables. For each question, you only add the one method.
- You only submit one version of **A.java** and **B.java**, each containing your answer to both questions.

Files

- `A.java` (you need to update this one).
- `B.java` (you need to update this one).
- `DeepCopyable.java`
- `TestDeepCopy.java`

Question 2 (10 marks)

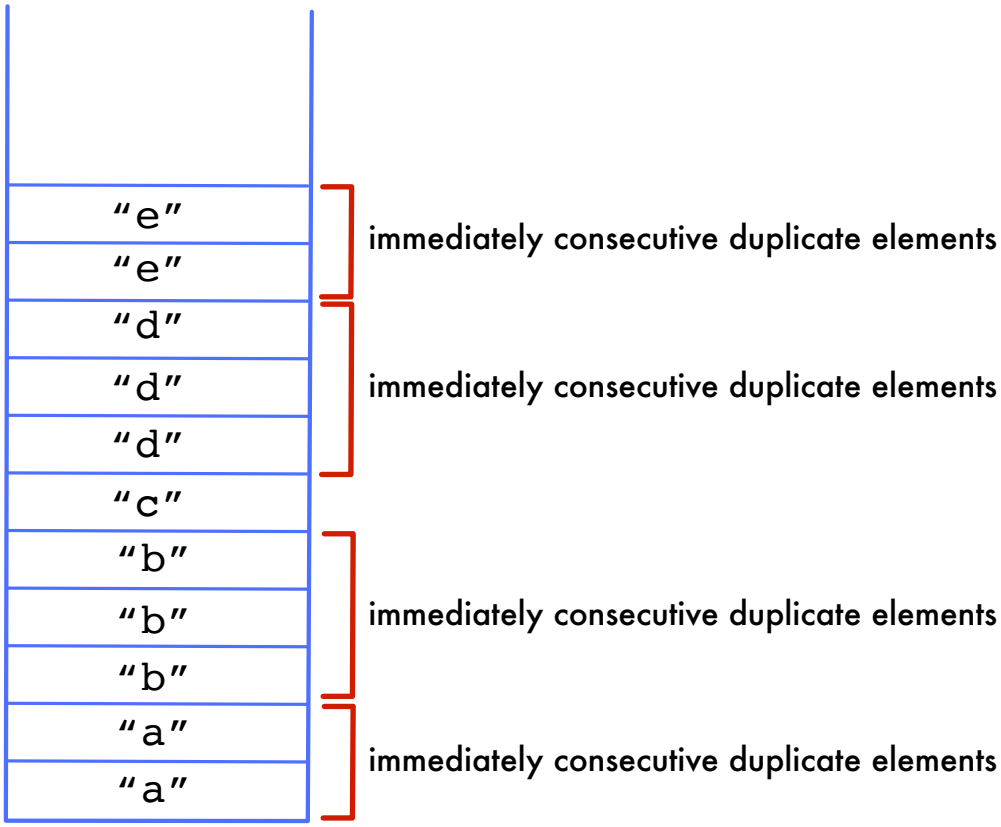
For this question, you are going to develop a **specialization (subclass) of ArrayStack**. ArrayStack has been already covered in the course lectures. The subclass, named **UniquifiableArrayStack**, provides a single additional method, named **uniquify()**.

```
public class UniquifiableArrayStack<E> extends ArrayStack<E> {
        public Stack<E> uniquify() {
          // All the code that you write for Question 2 goes here!
        }
}
```

The method **uniquify()** does the following:

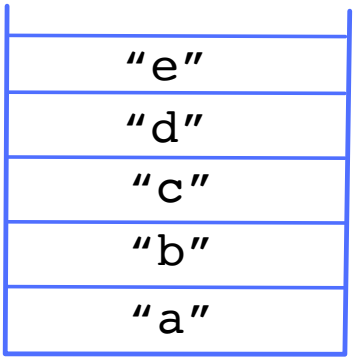
It returns a stack which is the same as the UniquifiableArrayStack instance that the method is called on, except that the returned stack has no immediately consecutive duplicate elements.

To illustrate what "immediately consecutive duplicate elements" are, consider the example stack below:



More precisely, two stack elements are immediately consecutive duplicate elements, when they are adjacent and have identical content. Having identical content for (non-null) stack elements `elem1` and `elem2` means that `elem1.equals(elem2)` is true. **You can assume that all stack elements are non-null.**

In Question 2, you complete the method **uniquify()** in the **UniquifiableArrayStack** class, so that only one instance of immediately consecutive duplicate elements would be retained. For example, if **uniquify()** is called over the example stack above (an instance of `UniquifiableArrayStack<String>`), the result should be as shown in the figure on the top of the next page.



Note that **uniquify() should not have any side effect** on the **UniquifiableArrayStack** instance that the method is called on. Furthermore, **uniquify()** is **not meant to deal with non-consecutive duplicate elements**. In other words, the stack returned by **uniquify()** can have non-consecutive duplicate elements. The absence of side effects on the original stack as well as the carry-over of non-consecutive duplicate elements to the result of **uniquify()** are illustrated in the example output by **Q2Test**, presented next.

Example Output

To test your implementation of **uniquify()**, you can use the **Q2Test.java** in the template code provided to you. The output from running **Q2Test** should be as follows:

```
Original Integer stack: bottom->[0, 1, 1, 1, 2, 2, 3, 3, 3, 4]<-top
Integer stack without immediately consecutive duplicates: bottom->[0, 1, 2, 3, 4]<-top
Original Integer stack (after uniquify): bottom->[0, 1, 1, 1, 2, 2, 3, 3, 3, 4]<-top

Original String stack: bottom->[a, a, b, b, b, c, d, d, d, e, e]<-top
String stack without immediately consecutive duplicates: bottom->[a, b, c, d, e]<-top
Original String stack (after uniquify): bottom->[a, a, b, b, b, c, d, d, d, e, e]<-top

---- Now, testing with some non-consecutive duplicates -----

Original String stack: bottom->[a, b, b, c, a, d, d, e, e, d, d, d, d, b]<-top
String stack without immediately consecutive duplicates: bottom->[a, b, c, a, d, e, d, b]<-top
Original String stack (after uniquify): bottom->[a, b, b, c, a, d, d, e, e, d, d, d, d, b]<-top
```

Important Restrictions for Question 2

- You **cannot** change either `ArrayStack.java` or `Stack.java`. You can change `Q2Test.java` if you would like to perform additional testing, but for your submission, please leave `Q2Test.java` just as you found it in the template code.
- All your variables should be **local variables**. You **cannot** declare any class or instance variables in **UniquifiableArrayStack** (or anywhere else for that matter).
- The local reference variables in **uniquify()** as well as in any helper private methods that you may implement can only be of type **E** (that is, the generic type parameter) or of type **Stack<E>**. Stated otherwise, you **cannot** declare any reference variable that has a different type than either **E** or **Stack<E>**.
- You are allowed to use local primitive variables if you find them necessary, but please note that it is feasible to implement **uniquify()** *without* using any local primitive variables.

Files

- `ArrayStack.java`
- `Q2Test.java`
- `Stack.java`
- `UniquifiableArrayStack.java` (you need to update this one).

Rules and regulation

- Submit your examination through **Brightspace**.
- You must do this examination individually.
- You must use the provided template classes.
- It is your responsibility to make sure that Brightspace has received your examination.
- Late submissions will not be graded.

Files

- Download the archive **m2_3000000.zip**;
- Unzip the file and rename the directory, replacing **3000000** by **your student id**;
- Add your **name** and **student id** in a comment in **A.java**, **B.java** and **UniquifiableArrayStack.java**;

You must submit a **zip** file (no other file format will be accepted). The name of the top directory has to have the following form: **m2_3000000**, where 3000000 is your student number. The name of the folder starts with the letter "m" (lowercase), followed by 2, since this is part 2 of the midterm examination. The segments are separated by the underscore (not the hyphen). There are no spaces in the name of the directory. Your submission must contain the following files, and nothing else. In particular, do not submit the byte-code (.class) files.

- README.txt
  - A text file that contains your name and student id
- `A.java`
- `B.java`
- `DeepCopyable.java`
- `TestDeepCopy.java`
- `ArrayStack.java`
- `Q2Test.java`
- `Stack.java`
- `UniquifiableArrayStack.java`

A Shallow copy versus Deep copy

As you know, objects have variables which are either a primitive type, or a reference type. Primitive variables hold a value from one of the language primitive type, while reference variables hold a reference (the address) of another object (including arrays, which are objects in Java).

If you are copying the current state of an object, in order to obtain a duplicate object, you will create a copy of each of the variables. By doing so, the value of each instance primitive variable will be duplicated (thus, modifying one of these values in one of the copy will not modify the value on the other copy). However, with reference variables, what will be copied is the actual reference, the address of the object that this variable is pointing at. Consequently, the reference variables in both the original object and the duplicated object will point at the same address, and the reference variables will refer to the same objects. This is known as a **shallow** copy: you indeed have two objects, but they share all the objects pointed at by their instance reference variables. The Figure to the left provides an example: the object referenced by variable **b** is a shallow copy of the object referenced by variable **a**: it has its own copies of the instances variables, but the references variables **title** and **time** are referencing the same objects.

Often, a shallow copy is not adequate: what is required is a so-called **deep** copy. A deep copy differs from a shallow copy in that objects referenced by reference variable must also be recursively duplicated, in such a way that when the initial object is (deep) copied, the copy does not share any reference with the initial object. The Figure to the right provides an example: this time, the object referenced by variable **b** is a deep copy of the object referenced by variable **a**: now, the references variables **title** and **time** are referencing different objects. Note that, in turn, the objects referenced by the variable **time** have also been deep-copied. The entire set of objects reachable from **a** have been duplicated.

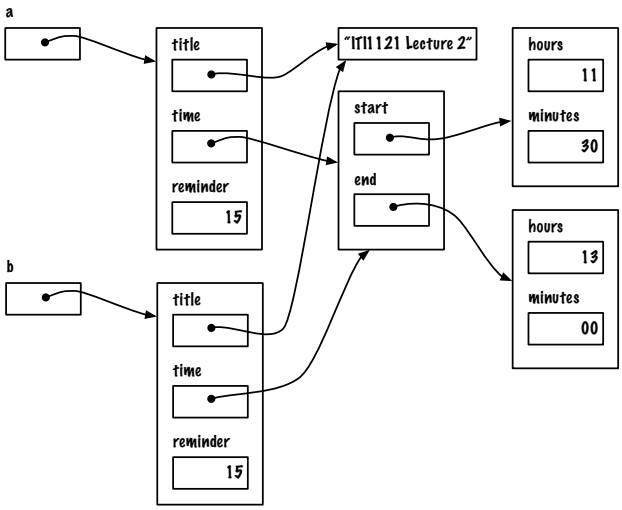


Figure 1: Example of a shallow copy of objects.

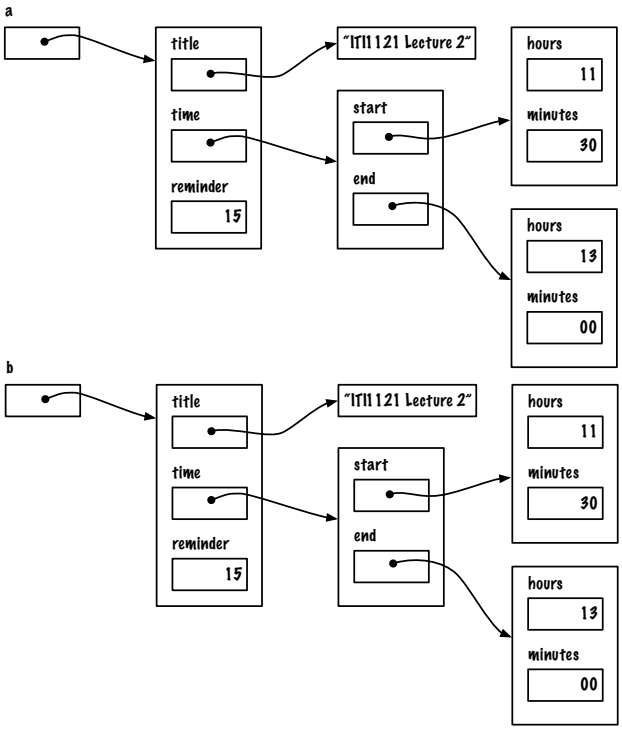


Figure 2: Example of a deep copy of objects.