Exercise 6

1. Given the following double linked list: **`public class DList<T>`**

```
protected class Node<T> {
  Node<T> prev, next;
  T data;
}
private Node<T> first,last;
public DList() {
  first = new Node<T>();
  last = new Node<T>();
  first.next = last;
  last.prev = first;
}
```

```
public DList( T[] d ){
  this();
  for( T x : d ) {add(x);}
}

public T getFirst() {
  return getFirstNode().data;
}
protected Node<T> getFirstNode(){
    return first.next;
}
```

```
public void add( T d ) {
  Node<T> n =new Node<T>();
  n.data = d;
  n.prev = last.prev;
  n.next = last;
  last.prev.next = n;
  last.prev = n;
}
```

```
public static void main( String[] args ) {
  Integer[] ia = new Integer[] {1, 2, 3};
  DList<Integer> dl = new DList<Integer>( ia );
  ia[0] = 5;
  System.out.println("First element: " + dl.getFirst());
}
```

a. Change the implementation, such that an array is used for storing the data. The external observable behavior of the class should not change.
b. The method `getFirstNode` is not needed anymore. Which classes need to be adapted if we remove the method?
c. Is there a different behavior between linked-lists and arrays in the main method?

2. Argue for each of the following classes that they preserve encapsulation. Do all the constructors and methods preserve the invariants? Is it guaranteed that other classes do not violate the consistency of the objects?

```
public class A {
  private int a;
  private int b;

  /*@ invariant a >= b; @*/
  /*@ requires ta >= 0; @*/

  public A(int ta) {
    a = ta;
    b = 0;
  }

  public void increment() {
    ++a;
    ++b;
  }
}
```

```
public class A {
  public int a;
  public int b;

  /*@ invariant a >= b; @*/
  /*@ requires ta >= 0; @*/

  public A(int ta) {
    a = ta;
    b = 0;
  }

  public void increment() {
    ++a;
    ++b;
  }
}
```

```
public class A {
  private int a;
  private int b;
  public int c;


  /*@ invariant a >= b; @*/
  /*@ requires ta >= 0; @*/

  public A(int ta) {
    a = ta;
    b = 0;
  }

  public void increment() {
    ++a;
    ++b;
  }
```

3. **Encapsulation question from a previous exam!**
   This example addresses the relation between encapsulation techniques and security aspects. Given the following scenario: A system environment, represented by the object of type `Environment`, manages what people have access to secure parts of the system. The ID of the persons are stored as an `int` in the class `Authorization` . `Environment` and `Authorization` are implemented in the following way:

```
package System;
public interface Environment {
   public void insertAuthorization ( Authorization b);
   public Authorization getAuthorization();
}
package System;
public class Authorization {
   private int[] ids;
   public Authorization() { ids = new int[5]; }
   protected void setIDs( int[] p ) {
      ids = p; }
   public int[] getIDs() { return ids; }
}
```

   The interaction between `Environment` and `Authorization` looks like the following:
   1. Objects of type `Authorization` can be created by an arbitrary class and can be transfered to the system environment with the method `insertAuthorization`.
   2. `insertAuthorization` saves the transfered reference and stores the ID of the registered person into the field IDs of the `Authorization` object using the method `setIDs`. (Keep in mind, that `Environment` and `Authorization` are declared in the same package!)
   3. An arbitrary user of the system can fetch the IDs of the registered people with the methods `getAuthorization` and `getIDs` accessing them read only. For example, to make comparisons between ids.

The interaction between `Environment` and `Authorization` is called a secure system, if no class outside of the package `System` is allowed to modify the IDs stored into the `Authorization` object.

Exercise:

a. The above implementation is not secure. Describe how an attacker can manipulate the list of `ids` using the method `getIDs`. In this case an attacker is a class declared outside of the package system.
b. Implement your solution for question **a** as method
   `public static void attack(Environment u) {...}`
   in class `Attack` of package `Attacker`.
c. Explain how to modify the implementation of class `Authorization`, to prohibit the attack. The modified `Authorization` class still has to allow the read only interaction described above.
   The interface `Environment` as well as the implementation must not be modified.
d. Describe how an attacker could manipulate the list of IDs **without using** the method `getIDs`.
e. Implement your solution for question **d** as method
   `public static void attack( Environment u) {...}`
   in a class `Attack` in the package `Attacker`.
f. Explain how to modify the implementation of class `Authorization` to prohibit the attack from question **d**. There are the same requirements as in question **c**.