9 Javadocs, Using ArrayList, Implementing Stack and Queue

Goals

The first part of the lab you will learn how to generate *Javadoc* documentation, and practice reading *Javadoc* style documentation for programs.

The second part introduces `ArrayList` class from the **Java Collections Framework** library, lets you practice designing methods that mutate `ArrayList` objects.

In the third part of the lab you will learn how to implement the *queue* and *stack* using the Java `ArrayList`.

9.1 Documentation

For this lab download the following files:

- The file *Balloon.java* — our sample data class
- The file *TopThree.java* will be used to practice working with `ArrayList` in imperative style (using mutation).
- The *Examples.java* file that defines examples of all data and defines all tests.

Create a new **Project** *Lab9* and import into it all files from the zip file. Import the tester.jar and colors.jar.

Generating Documentation

- Once Eclipse shows you that there are no errors in your files select **Generate Javadoc...** from the **Project** pull-down menu. Select to generate docs for all files in your project with the destination *Lab9/doc* directory. Make sure you select all files for which you wish to generate the documentation.

  You should be able to open the *index.html* file in the *Lab9/doc* directory and see the documentation for this project. Compare the documentation for the class `Balloon` with the web pages. You see that all comments from the source file have been converted to the web document.

Observe the format of the comments, especially the `/**` at the beginning of the comment. If you do not understand the rules, ask the TA or one of the tutors, or experiment with new comments. From now on all of your work should have a proper Javadoc style documentation.

- Now use the documentation to see what are the fields in various classes and what methods have been defined already.

- Define a method `isHit` in the class `Balloon` that determines whether a shot aimed at the given `x` and `y` coordinate hits this `Balloon`. Add documentation in the `Javadoc` style. Of course, add tests in the `Examples` class. Run the tests, then rebuild the `Javadoc`s and make sure your documentation shows up correctly.

9.2 Using ArrayList with Mutation

In this part of the lab we will work on lists of balloons, using the Java library class `ArrayList`.

Open the web site that shows the documentation for Java libraries

http://java.sun.com/j2se/1.5.0/docs/api/.

Find the documentation for `ArrayList`.

Here are some of the methods defined in the class `ArrayList`:

```
// how many items are in the collection
int size();


// add the given object of the type E at the end of this collection
// false if no space is available
boolean add(E obj);


// return the object of the type E at the given index
E get(int index);


// replace the object of the type E at the given index
// with the given element
// produce the element that was at the given index before this change
E set(int index, E obj);
```

Other methods of this class are `isEmpty` (checks whether we have added any elements to the `ArrayList`), `contains` (checks if a given element exists in the `ArrayList` — using the `equals` method).

9.3 Using the ArrayList class

Notice that, in order to use an `ArrayList`, we have to add

```
import java.util.ArrayList;
```

at the beginning of our class file.

The first method you design will be within the class `TopThree`. The remaining methods will be defined within the `Examples` class. Of course, the tests for all methods will still be inside the `Examples` class.

1. The class `TopThree` now stores the values of the three elements in an `ArrayList`. Complete the definition of the `reorder` method. Use the previous two parts as a model. Look up the documentation for the Java class `ArrayList` to understand what methods you can use.

   Do not forget to run your tests.

2. Design the method `isSmallerThanAtIndex` that determines whether the radius of the balloon at the given position (index) in the given `ArrayList` of `Balloon`s is smaller than the given limit.

3. Design the method `isSameAsAtIndex` that determines whether the balloon at the given position in the given `ArrayList` of `Balloon`s has the same size and location as the given `Balloon`.

4. Design the method `inflateAtIndex` that increases the radius of a `Balloon` at the given index by 5.

5. Design the method `swapAtIndices` that swaps the elements of the given `ArrayList` at the two given positions (indices).

Note 1: We have used the words *position* in the `ArrayList` and *index* in the `ArrayList` interchangeably in the previous descriptions of tasks. Both are commonly used and we wanted to make sure you get used to both ways of describing an element in an `ArrayList`.

Note 2: Of course, the tests for these methods will also appear in the `Examples` class. Make sure that every test can be run independently of all other tests. To do this, you must initialize the needed data inside of the test method, evaluate the test by invoking the appropriate `checkExpect` method, and reset the data to the original state after the test is completed.

9.4 Implementing Stack and Queue using ArrayList

We can easily implement the `IQueue` and the `Stack` interfaces using the Java `ArrayList`. The behavior of the *stack* and *queue* is provided by the corresponding *interfaces*. The programmer that needs to work with one of these data structures can write the entire program referring only to the methods given by the interface for that data structure. Later, the programmer can decide which *implementation* of the desired data structure will be used when running the program.

The *interface* that describes the behavior of a data structure is called **Abstract Data Type** or **ADT**. The goal of this lab is to see that we can have several different implementations of an *ADT*. We have seen the first variants in the previous lab and in the homework assignment.

1. Recall the definition of the `IQueue` interface:

```
// Interface that constructs a queue
public interface IQueue<T>{

  // Is this an empty queue?
  public boolean isEmpty();

  // Adds an Object to the queue
  public IQueue<T> enQueue(T t);

  // Returns the element at the head of the list
  public T element();

  // Returns the resulting queue after the head of the list
  // has been removed
  public IQueue<T> deQueue();
}
```

   Start with the following partial class definition:

```
// A class that implements a queue
public class ArrQueue<T>{

  // The ArrayList to hold the data
  ArrayList<T> arlist;

  ArrQueue(){}

  // Is this an empty queue?
  public boolean isEmpty(){ ............ }

  // Adds an Object to the queue
  public IQueue<T> enQueue(T t){ ............ }
```

```
  // Returns the element at the head of the list
  public T element(){
    ... throw an exception if there are no elements...
  }

  // Returns the resulting queue after the head of the list
  // has been removed
  public IQueue<T> deQueue(){ ............ }
}
```

Implement the four methods and run the tests for a queue of `String`s.

2. Here is the definition of the `IStack` interface:

```
// Interface that constructs a stack
public interface IStack<T>{

  // Is this an empty stack?
  public boolean isEmpty();

  // Adds an Object to the top of the stack
  public IStack<T> push(T t);

  // Returns the element at the the top of the stack
  public T peek();

  // Returns the resulting stack after the top of the stack
  // has been removed
  public IStack<T> pop();
}
```

Start with the following partial class definition:

```
// A class that implements a stack
public class ArrStack<T>{

  // The ArrayList to hold the data
  ArrayList<T> arlist;

  ArrStack(){}

  // Is this an empty stack?
  boolean isEmpty(){ ............ }

  // Adds an Object to the top of the stack
  public IStack<T> push(T t){ ............ }

  // Returns the element at the the top of the stack
  public T peek(){
     ...... throw an exception if empty .....
  }
```

```
  // Returns the resulting stack after the top of the stack
  // has been removed
  public IStack<T> pop(){
     ...... throw an exception if empty .....
  }
}
```

Implement the three methods and run the tests for a stack of `Strings`.