ArrayWrapper

1 Introduction

This assignment will have you implement a primitive Collection called ArrayWrapper that behaves very similarly to an ArrayList. You must use a standard array to write this assignment, but for all assignments after this one, any Collection class will be fair game.

2 Problem Description

Implement the `SimpleCollection` interface provided in a class called `ArrayWrapper`. Use a backing array to store your elements. This `ArrayWrapper` class should use a generic parameter for the type of objects it is storing.

Additionally, we have provided a testing suite for you to use to test your code. The testing suite is in the form of a JUnit test. To run the JUnit, run the following command(replacing ":" with ";" on Windows):

```
$ javac -cp .:/your/path/to/hamcrest-core-1.3.jar:/your/path/to/junit-4.11.jar *.java

$ java -cp .:/your/path/to/hamcrest-core-1.3.jar:/your/path/to/junit-4.11.jar
    org.junit.runner.JUnitCore ArrayWrapperTest
```

3 Solution Description

We provide the following classes:

- `SimpleCollection`: An interface that simple collections must implement.
- `ArrayWrapperTest`: A JUnit testing suite that tests your simple collection for errors.

You need to create the `ArrayWrapper` class, filling out the methods in the manner that `SimpleCollection` describes so that the unit tests in `ArrayWrapperTest` pass.

4 Tips

- If you're having difficulty getting the generics to work correctly, start by hardcoding the type to be String. It can always be changed later to the correct generic type.
- Instantiating a generic array is a bit strange - you cannot simply make a `new T[5]`. Instead, you must cast an `Object` array to a T array.
- When resizing the array in `add()`, think carefully about how you will copy the elements over.
- Take care to avoid any nulls in the your output. Specifically, `SimpleCollections` cannot contain null.

5 Checkstyle

Review the Style Guide and download the Checkstyle jar and associated XML file. Run Checkstyle on your code like so:

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java
Starting audit...
Audit done.
```

The message above means there were no Checkstyle errors. You can easily count Checkstyle errors by piping the output of Checkstyle through `wc -l` and subtracting 2 for the two non-error lines printed above (which is how we will deduct points). For example:

```
$ java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | wc -l
      2
```

Alternatively, if you are on Windows, you can use the following instead:

```
C:\> java -jar checkstyle-5.6-all.jar -c cs1331-checkstyle.xml *.java | findstr /v "Starting
    audit..." | findstr /v "Audit done" | find /c /v "hashcode()"
0
```

> Food for thought: is there a one-liner like above that shows you only the number of errors? Hint: `man grep`.

The Java source files we provide contain no Checkstyle errors. For this assignment, there will be a maximum of **50** points lost due to Checkstyle errors (1 point per error). In future homeworks we will be increasing this cap, so get into the habit of fixing these style errors early!

6 Javadocs

We are going to have you do Javadocs for this assignment (and for all assignments here on out). Javadocs are a clean and useful way to document your code's functionality. For more information on what Javadocs are and why they are awesome, the online documentation for them is very detailed and helpful. The relevant tags that you need to have are `@author`, `@version`, `@param`, and `@return`. Here is an example of a properly Javadoc'd class:

```
import java.util.Scanner;

/**
 * This class represents a Dog object.
 * @author George P. Burdell
 * @version 13.31
 */
public class Dog {

   /**
    * Creates an awesome dog (NOT a dawg!)
    */
   public Dog(){
      ...
   }

   /**
    * This method takes in two ints and returns their sum
    * @param a first number
    * @param b second number
    * @return sum of a and b
    */
   public int add(int a, int b){
      ...
   }
}
```

7 Turn-in Procedure

Submit all of the Java source files and resources your program requires to run to T-Square. Do not submit any compiled bytecode (`.class` files), the Checkstyle jar file, or the `cs1331-checkstyle.xml` file. When you're ready, double-check that you have submitted and not just saved a draft.

Please remember to run your code with Checkstyle!

Verify the Success of Your Submission to T-Square
Practice safe submission! Verify that your HW files were truly submitted correctly, the upload was successful, and that the files compile and run. It is solely your responsibility to turn in your homework and practice this safe submission safeguard.

1. After uploading the files to T-Square you should receive an email from T-Square listing the names of the files that were uploaded and received. If you do not get the confirmation email almost immediately, something is wrong with your HW submission and/or your email. Even receiving the email does not guarantee that you turned in exactly what you intended.
2. After submitting the files to T-Square, return to the Assignment menu option and this homework. It should show the submitted files.
3. Download copies of your submitted files from the T-Square Assignment page placing them in a new folder.
4. Recompile and test those exact files.
5. This helps guard against a few things.
   (a) It helps insure that you turn in the correct files.
   (b) It helps you realize if you omit a file or files.[1] (If you do discover that you omitted a file, submit all of your files again, not just the missing one.)
   (c) Helps find last minute causes of files not compiling and/or running.

[1] Missing files will not be given any credit, and non-compiling homework solutions will receive few to zero points. Also recall that late homework will not be accepted regardless of excuse. Treat the due date with respect. The real due date is midnight. Do not wait until the last minute!