Project 0: Project Skills Check

DUE: Sunday, August 27th at 11:59pm (NO EMERGENCY TOKENS)

Setup

- Download the `p0.zip` and unzip it. This will create a folder `section-yourGMUUserName-p0`.
- Rename the folder replacing `section` with the `002`, `006`, `008` , etc. based on the lecture section you are in.
- Rename the folder replacing `yourGMUUserName` with the first part of your GMU email address.
- After renaming, your folder should be named something like: `001-yzhong-p0`.
- Complete the `readme.txt` file (an example file is included: `exampleReadmeFile.txt`).

Submission Instructions

- Make a backup copy of your user folder!
- Remove all test files, jar files, class files, etc.
- You should just submit your java files and your readme.txt
- Zip your user folder (***not just the files***) and name the zip `section-username-p0.zip` (***no other type of archive*** – no rars, 7zips, etc.) following the same rules for `section` and `username` as described above.
    - The submitted file should look something like this:

```
001-yzhong-p0.zip --> 001-yzhong-p0 --> BananaCoconut.java
                                        OneItemBag.java
                                        readme.txt
```

- Submit to blackboard. DOWNLOAD AND VERIFY WHAT YOU HAVE UPLOADED THE RIGHT THING. ***Submitting the wrong files will result in a 0 on the assignment!***

Basic Procedures

You must:

- Have code that compiles with the command: **`javac *.java`** in your user directory *without errors or warnings*. You may not use **`@SuppressWarnings`** to remove warnings, you must fix them.
- Have code that runs with the following command for part 1:

```
java BagUsageDemo
```

- Have code that runs with the following command for part 2:

```
java BananaCoconut [number] [number] [...]
```

You may:

- Add additional methods and variables not specifically mentioned in the project description, however these methods and variables **must be private**.

You may NOT:

- Make your program part of a package.
- Add additional public methods or class/instance variables (local variables are ok).
- Use any built in Java Collections Framework classes anywhere in your program (e.g. no **`ArrayList`**, **`LinkedList`**, **`HashSet`**, etc.).
- Use any arrays anywhere in your **`OneItemBag`** program (you may use them in **`BananaCoconut`**).
- Add any additional import statements (or use the "fully qualified name" to get around adding import statements).
- Add any additional libraries/packages which require downloading from the internet.

Grading Rubric

Due to the complexity of this assignment, an accompanying grading rubric pdf has been included with this assignment. Please refer to this document for a complete explanation of the grading.

Overview

This project should be very straight forward and take only an hour or so to implement if you have all the prerequisite knowledge from the prior courses. However, if it's been a while since you took CS211, or if you took CS211 at another university, or even if you just didn't get the best grade you could have in that class, you may have additional "catching up" to do. Below are the major topics required to complete this assignment and links to resources to help you with them:

1. **Basic Java Programming** – this is required, and you need to have *at least one additional semester* of programming under your belt to be successful in CS310. If you're not quite comfortable:
    a. Review chapters 1-4 in our textbook (Weiss).
    b. Review the following Java Tutorials:
        i. Trail: Getting Started: https://docs.oracle.com/javase/tutorial/getStarted/index.html
        ii. Trail: Learning the Java Language: https://docs.oracle.com/javase/tutorial/java/index.html
        iii. Trail: Essential Classes (for exceptions and basic I/O): https://docs.oracle.com/javase/tutorial/essential/index.html
    c. For writing JavaDocs, see:
        i. This tutorial: https://www.baeldung.com/javadoc
        ii. The official JavaDoc documentation: https://www.oracle.com/technetwork/java/javase/documentation/index-137868.html

2. **Using a Terminal / Command Line** – this is a required skill for every *programmer*, not just everyone in CS310. For this class all you really need to be comfortable changing directories, compiling and running java programs, and providing/using command line arguments in Java. Here are some quick references for both OS systems:
    a. Table of simple commands in Windows/Linux: https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/4/html/Step_by_Step_Guide/ap-doslinux.html
    b. Working with command line arguments (for/from Java): https://docs.oracle.com/javase/tutorial/essential/environment/cmdLineArgs.html

If you're interested in specific topics, there is a table of contents (in the front) and an index in your textbook (in the back), but here are some quick references for topics will definitely play a role in this particular assignment:

- Basics of Objects and References (pg. 30)
- Strings (pg. 35)
- Command Line Arguments (pg. 45)
- Exception Handling (pg. 47)
- Input and Output (pg. 51)
- JavaDocs (pg. 73)
- Generics (pg. 150)
- Local Classes (pg. 161)

Requirements

An overview of the requirements are listed below, please see the grading rubric for more details.

- **Implementing the classes** - You will need to implement required classes/methods.
- **Style** – You must follow the coding and conventions specified by the style checker.
- **JavaDocs** - You are required to write JavaDoc comments for all the required classes and methods.

JavaDocs?? Style Checker?? Yes, as you may often be asked to do in a professional setting, you'll need to document your code correctly and conform to a given code style (many companies have their own style requirements, but there are also some standard ones like Sun and Google).

TL;DR You need to know some things to be successful in this class. You (hopefully) learned them in CS211. This project checks that. There are resources and links if you need help. If you can't do something, learn now!

Part 1: Make a bag that can hold anything

This first part checks that you know how to use generics, write JavaDocs, and use the basic command line tools you'll use throughout the semester. You need to write a class called **`OneItemBag`** (in a file called **`OneItemBag.java`** that you create). This class represents a bag that can hold a single type of object (decided at bag-creation time), and only one item of that type at a time. You may have a zero-parameter constructor if you want one, but you must have the following three features:

1. a method that puts an item in the bag (**`addItem()`**) which has a single parameter and returns whether or not it was successfully added
2. a method that removes an item from the bag and returns it (**`removeItem()`**), return **`null`** if there is no item
3. a method to check if an item is in the bag (**`hasItem()`**) which returns true or false

Make sure to comment your code *as you go* in proper JavaDoc style.

To check that you've got the right idea, we've provided a class called **`BagUsageDemo.java`**. You should not alter this class to "make it work", but rather alter your bag to allow the provided code to work. You can use the command "**`java BagUsageDemo`**" to run the testing code defined in **`main()`**. You could also edit this **`main()`** to perform additional testing (we won't be using **`BagUsageDemo`** for testing, it's just a demo for you). Note that JUnit test cases will not be provided for **`OneItemBag`**, but feel free to create JUnit tests for yourself. A part of your grade *will* be based on automatic grading using test cases made from the specifications provided.

TL;DR You're making a bag (**`OneItemBag`**) and using JavaDoc comments. There's test code in **`BagUsageDemo`**.

Checking That You've Got Style

Every professional developer needs to adhere to a coding style (indentation, variable naming, etc.). This is non-optional in 99.99999% of professional settings (aka. "jobs"). Normally, in school, a grader checks your style manually, but this is very inefficient since you only get feedback after your project is completed (not while you're writing), and it's very hard for someone to manually check these things.

We're going to help move you along the path to "coding with style" this semester. We have provided you with a command line tool that checks your style for you: **`checkstyle`** (https://checkstyle.org). This tool has plugins for Eclipse, NetBeans, jGRASP, and many others (see their website), but there is also a command line interface (CLI) which has been provided with this project (**`checkstyle.jar`**). This automatic checker is similar to ones used at large companies (like Google, Oracle, and Facebook). If you're curious, we're using a subset of Google's style requirements for this class.

The provided **`cs310code.xml`** checks for common coding convention mistakes (such as indentation and naming issues). You can use the following to check your style:

```
java -jar checkstyle.jar -c [style.xml file] [userDirectory]/*.java
```

For example, for a user directory **`001-yzhong-p0`** checking for JavaDoc mistakes I would use:

```
java -jar checkstyle.jar -c cs310code.xml 001-yzhong-p0/*.java
```

Note: if you have a lot of messages from **`checkstyle`**, you can add the following to the end of the command to output it to a file called out.txt: **`> out.txt`**

If checkstyle finds nothing wrong you'll see "Starting audit... Audit done." and nothing else.

Checking Your JavaDoc Comments

You should verify that your JavaDoc comments adhere to the proper format (especially if you're new to writing JavaDocs). We've provided a second **`checkstyle`** file (**`cs310comments.xml`**) that looks for JavaDoc issues and in-line comment indentation issues. You can run it the same way as **`cs310code.xml`**, for example:

```
java -jar checkstyle.jar -c cs310comments.xml 001-yzhong-p0/*.java
```

TL;DR There are some tools you need to try out. They are provided. They help you code more professionally.

Part 2: Putt Putt Banana Coconut!

This second part of the project checks that you know how to use command line arguments and write basic Java code (including exception handling), and it gives you more practice writing JavaDocs and using **`checkstyle`**. You'll also be able to see and use JUnit tests (such as the ones we use when grading).

Specification: Write a program (**`BananaCoconut`**, that lives in **`BananaCoconut.java`**). This program should accept multiple numbers as command line arguments and print those numbers out again space separated, but for multiples of 3 print "banana" instead of the number, for multiples of 7 print "coconut" instead of the number, and for the multiples of both 3 and 7 print "banana-coconut" instead of the number. For numbers smaller than 1, print "puttputt". Don't forget to document as you go. Example program run (coloring added for readability):

```
> java BananaCoconut 1 2 3 4 5
1 2 banana 4 5

> java BananaCoconut 0 1 3 5 7 17 21
puttputt 1 banana 5 coconut 17 banana-coconut

> java BananaCoconut 3 7 21 7 3
banana coconut banana-coconut coconut banana
```

Checking for Invalid Input

Your program should print the following (exactly) if the command line argument is missing or if any of the command line arguments are not a valid integer. This message should be sent to **`System.err`** not **`System.out`**. After printing the error message, your program should stop immediately. **NOTE**: do NOT use **`System.exit()`** to terminate since it will interfere with JUnit testing. See below (page 6) for more details. Example commands which generate this error include:

```
> java BananaCoconut
> java BananaCoconut apple
> java BananaCoconut 2 apple
```

Error message:

```
One or more numbers required as a command line argument.
Example Usage: java BananaCoconut [number] [number] [...]
```

Exactly Matching Output

Eight unit tests have been provided to automatically check that you are exactly matching the output required (**`BananaCoconutTest.java`**) since it is hard to eyeball differences in text. To run these tests, navigate to the directory *above* your user directory (from your user directory type **`cd ..`** to go up one directory) and do the following...

Compile and run the tests with the following in Windows:

```
javac -cp .;junit-4.11.jar;[userDirectory] BananaCoconutTest.java
java -cp .;junit-4.11.jar;[userDirectory] BananaCoconutTest
```

Or for Linux/Mac, replace all the semicolons wit colons in the classpath:

```
javac -cp .:junit-4.11.jar:[userDirectory] BananaCoconutTest.java
java -cp .:junit-4.11.jar:[userDirectory] BananaCoconutTest
```

Replace [**`userDirectory`**] with your actual user directory name. For example, on my computer (running Windows) from my "p0 directory" (not from my user directory which would be "`001-yzhong-p0`"), I would type:

```
javac -cp .;junit-4.11.jar;001-yzhong-p0 BananaCoconutTest.java
java -cp .;junit-4.11.jar;001-yzhong-p0 BananaCoconutTest
```

Check Your Style and JavaDocs

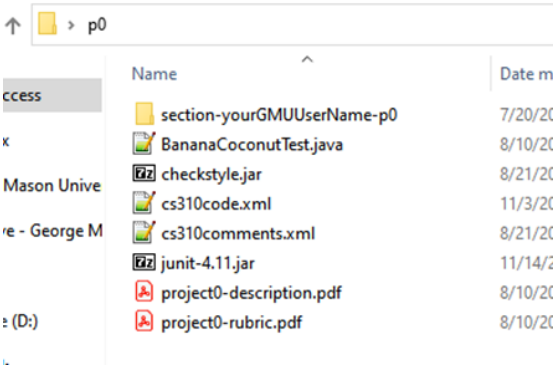
Run both **`checkstyle`** files again on this new program to verify you got those basics down.

TL;DR You need to make a program (**`BananaCoconut`**) with exact output. JUnit tests are provided.

Common Questions

What is the directory structure expected? Why are there folders? Should I move files around?

When you unzip the provided file, it will place several files and a folder. For example:



The folder you rename from "section-yourGMUUserName-p0" (your "user folder/directory") will contain your work and any provided non-unit test java files. The folder where you unzipped everything is "above" your "user directory". This "outer folder/directory" will contain JUnit tests, jars, pdfs, and other files we provide for your project. In the above example this is the folder "p0".

This separation helps ensure you aren't submitting files we don't want (things outside your user directory) and also gives you some practice with commands which span multiple directories, which is something you'll need to understand for working professionally. As a result, all commands provided are meant to be run from certain locations (such as "from your user directory" or "from above your user directory"). You should *not* move files around and expect the commands to work as provided. So you'll need to either use our structure, or learn to modify the commands to fit the structure you'd like *on your own*.

What is a JUnit test? What should the output look like for Junit? The JUnit Tests don't finish?

We expect students coming from CS211 to be familiar with JUnit testing, but if you are coming from another university here's the one sentence version: JUnit tests are individual "units of Java" (methods) that test your code. That's really all they are. They work almost exactly like the "yay" tests we provided: (1) setup some interesting scenario, like create a bag, (2) do something interesting, like put an item in the bag, and (3) check that the code had the right "effect" and give a message about it, like the if-then-yays. JUnit tests are just fancier packaging around that simple idea. Here is an example:

```
@Test(timeout = 2000)
public void test() {
   //Scenario: I have a peanut and a bag
   Peanut p = new Peanut();
   OneItemBag<Peanut> bag = new OneItemBag<>();

   //Do Something Interesting: Put the peanut in the bag
   bag.addItem(p);

   //Check and Give Message: the bag reports that there is an item in it
   assertTrue("Putting an item in the bag should make hasItem() true", bag.hasItem());
}
```

When you run the JUnit tests, they give some nice output telling you if the test worked or error messages saying they didn't. They're just bits of Java code so you can look at them and see what they are testing. This is what the output should be if the tests pass:

```
JUnit version 4.11
........
Time: 0.044

OK (8 tests)
```

If your code isn't passing the tests, you'll see a number of "failures" instead. If you see the JUnit tests "end early" without saying how many tests pass/fail it's usually caused by using **`System.exit()`**. This issue has to do with how JUnit works: it is Java code running Java code. Using **`System.exit()`** will kill the Java process, so using **`System.exit()`** will kill the JUnit tester. Don't use it. It's bad practice in code not to just return from main for this (and other) reasons.

Is it realistic in "real life" to forbid using X or Y when coding?

Yep. Believe it or not, there are many professional settings which will forbid you from programming using certain languages, libraries, and packages (think jobs that require government clearances). So this is good practice for your professional life. Trust me, it'll be just as "frustrating" there as it is here, but when it happens in industry you'll know what to do if you've practiced it.

Command Reference

All commands assume you haven't been moving files around – which means that you left the style items and jar files in an outer folder. See "Common Questions" if the commands don't work before messaging a professor/TA.

From *in* your user directory (e.g. "`001-yzhong-p0`"):

Compile:	**`javac *.java`**
Run Part 1:	**`java BagUsageDemo`**
Run Part 2:	**`java BananaCoconut [number] [number] [...]`**
Compile JavaDocs:	**`javadoc -private -d ../docs  *.java`**

From *above* your user directory (e.g. "`p0`"):

Style Checker:	**`java -jar checkstyle.jar -c cs310code.xml [userDirectory]/*.java`**
Comments Checker:	**`java -jar checkstyle.jar -c cs310comments.xml [userDirectory]/*.java`**
Compile Unit Tests:	**`javac -cp .;junit-4.11.jar;[userDirectory] BananaCoconutTest.java`**
Run Unit Tests:	**`java -cp .;junit-4.11.jar;[userDirectory] BananaCoconutTest`**

Or for Linux/Mac users, make sure that you use **`:`** instead of **`;`** for the classpath (the argument to **`-cp`**).