P01 Piggy Bank

Overview

When you have been a child, you might have a piggy bank, jar, or some other moneybox to hold your spare coins. In this first programming assignment, we are going to implement a piggy bank which holds US currency coins. This piggy bank is a kind of bag that holds your coins but with no organization. It can also contain duplicate coins. This application defines mainly the following operations.

- You can add a coin to the piggy bank.
- You can get the balance of your piggy bank.
- You can remove an arbitrary coin from the piggy bank (in real life scenario, you can shake the bank, so you have no control over what coin falls out).
- You can remove all coins from your piggy bank.

Grading Rubric

5 points	**Pre-Assignment Quiz:** The P01 pre-assignment quiz is accessible through Canvas before having access to this specification by **11:59PM on Wednesday 06/17/2020**. Access to the pre-assignment quiz will be unavailable passing its deadline.
20 points	**Immediate Automated Tests:** Upon submission of your assignment to Gradescope, you will receive feedback from automated grading tests about whether specific parts of your submission conform to this write-up specification. If these tests detect problems in your code, they will attempt to give you some feedback about the kind of defect that they noticed. Note that passing all of these tests does NOT mean your program is otherwise correct. To become more confident of this, you should run additional tests of your own.
15 points	**Additional Automated Tests:** When your manual grading feedback appears on Gradescope, you will also see the feedback from these additional automated grading tests. These tests are similar to the Immediate Automated Tests, but may test different parts of your submission in different ways.
10 points	**Manual Grading Feedback:** After the deadline for an assignment has passed, the course staff will begin manually grading your submission. We will focus on looking at your algorithms, use of programming constructs, and the style and readability of your code. This grading usually takes about a week from the hard deadline, after which you will find feedback on Gradescope.

Learning Objectives

The goals of this assignment include:

- reviewing the use of procedure oriented code (prerequisites for this course),
- practicing the use of control structures, custom static methods, and using arrays in methods.
- practicing how to manage an unordered collection of data that may contain duplicates (multiple occurrences of the same element),
- learning how to develop tests to demonstrate the functionality of code, and familiarizing yourself with CS300 grading tests.

Additional Assignment Requirements and Notes

- The only import statement that you may include in your `PiggyBank` class is `import java.util.Random`.
- You are not allowed to add any constant or variable outside of any method not defined or provided in this write-up.
- You CAN define local variables that you may need to implement the methods defined in this program.
- You CAN define private static helper methods to help implement the different public static methods defined in this write-up.
- In addition to the required test methods, we HIGHLY recommend that you develop additional own unit tests (public static methods that return a boolean) to convince yourself of the correctness of every public static method implemented in your `PiggyBank` class. It is highly recommended to design the test scenarios for every method before starting its implementation.
- All your test methods must be public static. Also, they must take zero arguments, return a boolean, and must be defined and implemented in your `PiggyBankTester.java`.
- All implemented methods MUST have their own javadoc-style method headers, according to the CS300 Course Style Guide.
- Feel free to reuse any of the provided source code in this write-up in your own submission.

1 Review of the CS300 Assignments Requirements

Before getting started, if not yet done, make sure to read carefully the advice through the following links which applies to all the CS300 programming assignments.

- Academic Conduct Expectations and Advice,
- **Course Style Guide** Review well the CS300 Course Style Guide, otherwise you are likely to lose points for submitting code that does not conform to these requirements on this and future programming assignments. At the end of the course style guide page are some helpful tips for configuring Eclipse to help with these requirements. Also, pay close attention to the requirements for commenting and the use of JavaDoc style comments in your code. Additional examples of Javadoc style class and method headers are available in the following zyBook reading section. Make also sure to complete all the entries in the file header appropriately and include it at the beginning of every java source file to be submitted on Gradescope.

2 Getting Started

Start by creating a new Java Project in eclipse which you may call **P01 Piggy Bank**, for instance. You have to ensure that your new project uses Java 11, by setting the "Use an execution environment JRE:" drop down setting to "JavaSE-11" within the new Java Project dialog box. Then, create two Java classes / source files within that project's src folder (both inside the default package) called `PiggyBank` and `PiggyBankTester`. Note that ONLY the `PiggyBankTester` class should include a main method. DO NOT generate or add a main method to the `PiggyBank` class. For a reminder or introduction to creating projects with source files in Eclipse, please review these instructions.

In this assignment, we are going to use procedural programming paradigm to develop our piggy bank application. This means that all the operations that you are going to implement will be *static methods*. This assignment involves also the use of arrays within methods in Java. So, before start working on the next steps, make sure that you have already completed the following reading activities on **zyBooks**. You may also refer to these sections at any level of the implementation of this project.

- Common errors: Methods and arrays,
- Perfect Size Arrays,
- Oversize Arrays,
- Methods With Oversize Arrays,
- Comparing perfect size and oversize arrays.

3 Implement the PiggyBank Class

In this assignment, we define a piggy bank as an oversize array which stores the integer values in cents of its coins. Your `PiggyBank` class must implement all the main operations related to adding a coin to the piggy bank, removing a random coin from a piggy bank, removing all the coins from a piggy bank, getting the balance of a piggy bank, and printing its content. In the following, we are going to first define the constants that we are going to use in this class. Then, we are going to implement and test the operational methods of our piggy bank application.

3.1 PiggyBank class constants

Add the following constants to your `PiggyBank` class. The top of the class body is a good placement where to declare them. These constants must be put outside of any of the methods that you are going to develop later. NO additional constants must be added to this class.

```
public final static int[] COINS_VALUES = {1, 5, 10, 25}; // coins values in cents
// coins names
public final static String[] COINS_NAMES = {"PENNY", "NICKEL", "DIME", "QUARTER"};
public final static String NO_SUCH_COIN = "N/A"; // N/A string
private final static Random RAND_GEN = new Random(100); // generator of random integers
```

3.2 Creating the first test methods

To practice good structured programming, we will be organizing our PiggyBank implementation into several easy-to-digest sized methods. This means that we are going to start with the implementation of the easy to solve methods before getting to the relatively harder ones. In addition, we want to test these methods before writing code that makes use of calling them. When we do find bugs in the future, we will add additional tests to demonstrate whether those defects exist in our code. This will help us see when a bug is fixed, and it will help us notice if similar bugs surface or return in the future. This design approach is also known as **Test Driven Development** process to design this first program.

We provide you in the following with the JavaDoc style comments and signature for the first method that we are going to write a unit test first.

```
/**
 * Returns the name of a specified coin value
 * @param coin represents a coin value in cents.
 * @return the String name of a specified coin value if it is valid and N/A if the
 *         coin value is not valid
 */
public static String getCoinName(int coin) {
  return ""; // return an empty string
}
```

To allow your code to compile, we suggest to add a return statement (return an empty string for instance). We would like to highlight that you can submit an incomplete implementation to Gradescope. But, never submit a code which does not compile. A submission that includes compile errors won't pass any of the automated tests on gradescope.

Now, based on the information provided in the above Javadoc style comments, we are going to write a test method to check the well functioning of any implementation of the *getCoinName()* method (not necessarily yours). This first test method should be added to the `PiggyBankTester` class and must have exactly the following signature.

```
/**
 * Checks whether PiggyBank.getCoinName() method works as expected.
 * @return true when this test verifies a correct functionality, and false otherwise
 */
public static boolean testGetCoinName() {}
```

A common trap when writing tests is to make the test code as complex or even more complex than the code that it is meant to test. This can lead to there being more bugs and more development time required for testing code, than for the code being tested. To avoid this trap, we aim to make our test code as simple as possible. It is important also that your test method takes checks whether the *getCoinName()* method returns the expected output considering different input values. We would like also to note that *getCoinName()* method returns a String. The == operator is used to compare only primitive types in Java which are int, double, float, short, long, char, boolean, and byte. To compare strings, you have to use the `String.equals()` method.

Now, we suggest that you take a piece of paper and write down which test scenarios you may consider in your test method. Then, compare your proposal with our suggestion. Recall that valid coin values and their related names are provided respectively in **COINS_VALUES** and **COINS_NAMES** arrays (cf. 3.1). Notice also that the arrays COINS_VALUES, COINS_NAMES, and the string NO_SUCH_COIN are declared public. So, you can access to them from your `PiggyBankTester` class.

.

.

.

.

.

.

.

The following is our proposal. You can copy it into your `PiggyBankTester` class as an example. If you would like to print out more specific feedback when tests fail (before returning false), that can be helpful.

```java
/**
 * Checks whether PiggyBank.getCoinName() method works as expected.
 * @return true when this test verifies a correct functionality, and false otherwise
 */
public static boolean testGetCoinName() {
  // consider all coin values as input arguments
    for(int i=0; i < PiggyBank.COINS_VALUES.length; i++)
    if(!PiggyBank.getCoinName(PiggyBank.COINS_VALUES[i]).equals(PiggyBank.COINS_NAMES[i]))
      return false;
    // consider input argument which does not correspond to a coin value
    if(!PiggyBank.getCoinName(7).equals(PiggyBank.NO_SUCH_COIN))
      return false;
    if(!PiggyBank.getCoinName(-10).equals(PiggyBank.NO_SUCH_COIN))
      return false;
    return true;
}
```

Recall that you should strive to keep all of your tests as simple as possible. For instance, notice that the *getCoinName()* method should return `PiggyBank.NO_SUCH_COIN` if it is called with a non valid coin value (an int value not defined within the `PiggyBank.COINS_VALUES` array). You do not have to consider "all" possible non valid coin values in your test method. You can choose one, two, or three possible non valid values. Note also that in CS300, we generally will NOT worry about the possibility of overflowing ints in cases like this either, unless the write-up specification explicitly mentions such concerns.

You can also notice how the above test method helps clarify the requirements and expected behavior of the *getCoinName()* method. Note that when you encounter a problem or question about your code, start by creating a test like this to verify your understanding of what is happening, versus what should be happening when your code runs. Sharing tests like this with the course staff who are helping you throughout this term is a great way to help the course staff help you more efficiently.

Now, you can add a call to this test method from the *main* method of your `PiggyBankTester` class. The following is an example.

```java
/**
 * Calls the test methods implemented in this class and displays their output
 * @param args input arguments if any
 */
public static void main(String[] args) {
  System.out.println("testGetCoinName(): " + testGetCoinName());
}
```

If you run now the above *main()*, the test method should not pass. You should implement the *PiggyBank.getCoinName()* method, and make sure that it passes your own test. It is

worth noting that you **HAVE TO** avoid hard-coding if conditions or switch statements on the coin argument in your implementation of the *PiggyBank.getCoinName()* method. Your implementation must be independent of the values that we provided in either the COINS_VALUES or the COINS_NAMES array. We suggest that you update your *testGetCoinName()* as follows and check whether your implementation passes the test. If not, try to locate the source of the error (it may be a logic error) and fix it.

```
public static boolean testGetCoinName() {
  // change some coin values and names
  PiggyBank.COINS_NAMES[1] = "Two cents";
  PiggyBank.COINS_NAMES[3] = "Fifty Cents";
  PiggyBank.COINS_VALUES[1] = 2;
  PiggyBank.COINS_VALUES[3] = 50;
  // consider all coin values as input arguments
  for (int i = 0; i < PiggyBank.COINS_VALUES.length; i++)
    if (!PiggyBank.getCoinName(PiggyBank.COINS_VALUES[i]).equals(PiggyBank.COINS_NAMES[i]))
      return false;
  // consider input argument which does not correspond to a coin value
  if (!PiggyBank.getCoinName(7).equals(PiggyBank.NO_SUCH_COIN))
    return false;
  if (!PiggyBank.getCoinName(-10).equals(PiggyBank.NO_SUCH_COIN))
    return false;
  return true;
}
```

3.3 Additional simple methods to implement and test

Below are Javadoc style method headers and signatures for two additional simple methods that you have to add to your `PiggyBank` class.

```
/**
 * Returns the balance of a piggy bank in cents
 * @param coins an oversize array which contains all the coins held in a piggy bank
 * @param size the total number of coins stored in coins array
 * @return the total value of the coins held in a piggy bank in cents
 */
public static int getBalance(int[] coins, int size) {}
```

```java
/**
 * Returns the total number of coins of a specific coin value held in a piggy bank
 *
 * @param coinValue the value of a specific coin
 * @param coins an oversize array which contains all the coins held in a piggy
 *       bank
 * @param size the total number of coins stored in coins array
 * @return the number of coins of value coinValue stored in the array coins
 */
public static int getSpecificCoinCount(int coinValue, int[] coins, int size) {}
```

Recall that the piggy bank is represented by the oversize array defined by the two input parameters coins and size passed to both *getBalance()* and *getSpecificCoinCount()* methods. All coins held in the piggy array are stored in the elements at the range of indexes from 0 to size-1 within the coins array.

Following the same design thinking, let's first design and implement the following two test methods with exactly the following signatures. Recall that the test methods must be added to the `PiggyBankTester` class.

```java
public static boolean testGetBalance() {}
public static boolean testGetSpecificCoinCount() {}
```

Here are some tips for the implementation of *testGetBalance()*. First, you have to read the method's javadoc style header and determine how this method is supposed to operate with different input parameters. For instance, you can ask yourself the following questions. What should *getBalance()* method return if it is passed an empty piggy bank?
How may the content of coins array look like if the passed piggy bank is not empty and what would be the returned balance?

While answering these two questions, take a paper and write down at least three different scenarios that can be considered in your *testGetBalance()*.

.
.
.
.
.
.
.
.

Our suggestion is at the top of the next page.

```java
/**
 * Checks whether PiggyBank.getBalance() method works as expected.
 * @return true when this test verifies a correct functionality, and false otherwise
 */
public static boolean testGetBalance() {
  // scenario 1 - empty piggy bank
  int[] coins = new int[10]; // array storing the coins held in a piggy bank
  int size = 0; // number of coins held in coins array
  if(PiggyBank.getBalance(coins, size)!= 0) {
    System.out.println("Problem detected. Your PiggyBank.getBalance() did not "
        + "return the expected output when passed an empty piggy bank.");
    return false;
  }
  // scenario 2 - non empty piggy bank
  coins = new int[] {10, 1, 5, 25, 1, 0, 0};
  size = 5;
  if(PiggyBank.getBalance(coins, size)!= 42) {
    System.out.println("Problem detected. Your PiggyBank.getBalance() did not "
        + "return the expected output when passed an piggy bank that holds "
        + "two pennies, a nickel, a dime, and a quarter.");
    return false;
  }
  // scenario 3 - another non empty piggy bank
  coins = new int[] {10, 1, 5, 25, 1, 0};
  size = 3;
  if(PiggyBank.getBalance(coins, size)!= 16) {
    System.out.println("Problem detected. Your PiggyBank.getBalance() did not "
        + "return the expected output when passed an piggy bank that holds "
        + "a penny, a nickel, and a dime, only.");
    return false;
  }
  return true;
}
```

You should be able now to design and implement the *testGetSpecificCoinCount()* test method. Try to define at least three different scenarios. Then, implement your `PiggyBank.`*getBalance()* and `PiggyBank.`*getSpecificCoinCount()* methods and check the correctness of their implementation with respect to your own test methods.

Now, to display the content of a piggy bank, add the following method to your `PiggyBank` class.

```
/**
 * Displays information about the content of a piggy bank
 *
 * @param coins an oversize array storing the coins held in a piggy bank
 * @param size number of coin held array coins
 */
public static void printPiggyBank(int[] coins, int size) {
  System.out.println("QUARTERS: " + getSpecificCoinCount(25, coins, size));
  System.out.println("DIMES: " + getSpecificCoinCount(10, coins, size));
  System.out.println("NICKELS: " + getSpecificCoinCount(5, coins, size));
  System.out.println("PENNIES: "+getSpecificCoinCount(1, coins, size));
  System.out.println("Balance: $" + getBalance(coins, size)*0.01);
}
```

3.4 Implementing adding a coin operation

Next, you are going to add and implement the following method into your `PiggyBank` class.

```
/**
 * Adds a given coin to a piggy bank. This operation can terminate
 * successfully or unsuccessfully. For either cases, this method
 * displays a descriptive message for either cases.
 *
 * @param coin the coin value in cents to be added to the array coins
 * @param coins an oversize array storing the coins held in a piggy bank
 * @param size the total number of coins contained in the array coins
 *       before this addCoin method is called.
 * @return the new size of the coins array after trying to add coin.
 */
public static int addCoin(int coin, int[] coins, int size) {}
```

Here are further specifications related to *addCoin()* method. If the integer coin value to be added to the piggy bank is not valid, this method terminates without changing the content of coins array and displays the following error message.

```
<coin>  + " cents is not a valid US currency coin."
```

If the *addCoin()* method is called with a full coins array as input argument, the method should display the following error message and terminates without adding the specified coin to the piggy bank.

```
"Tried to add a " + <coin_name> + ", but could not because the piggy bank is full."
```

Otherwise, the specified coin must be added to the piggy bank, and the following message must be displayed before the method returns.

```
"Added a " + <coin_name> + "."
```

To check the correctness of a `PiggyBank.` *addCoin()* method, add and implement a test method with exactly the following signature into your `PiggyBankTester` class.

```
public static boolean testAddCoin() {}
```

3.5 Implement remove coins operations

Now, you are going to implement the two following functions to remove coin(s) from a piggy bank 1) remove one arbitrary coin, and 2) remove all coins held in a piggy bank.

3.5.1 Remove an arbitrary coin from a piggy bank

```
/**
 * Removes an arbitrary coin from a piggy bank. This method may terminate
 * successfully or unsuccessfully. In either cases, a descriptive message
 * is displayed.
 *
 * @param coins an oversize array which contains the coins held in a piggy bank
 * @param size the number of coins held in the coins array before this method
 *       is called
 * @return the size of coins array after this method returns successfully
 */
public static int removeCoin(int[] coins, int size) {}
```

Here are further specification related to the *removeCoin()* method. If it is called with an empty piggy bank, the method displays the following message without changing the content of coins array.

```
"Tried to remove a coin, but could not because the piggy bank is empty."
```

If the provided piggy bank is not empty, *removeCoins()* method must remove an arbitrary coin from the oversize array coins and display the following message.

```
"Removed a " + <removed_coin_name> + "."
```

It is worth to note that we do not provide any restriction on which coin should be removed from the piggy bank. It must be an arbitrary (meaning randomly chosen) coin. **[Hint]:** Think of using the class constant `RAND_GEN` provided at section 3.1 to randomly generate the index of

the coin to be removed. Recall that coins values held in the piggy bank must be stored in the range of indices from 0 to size-1 of coins array. Elements of coins array located outside of that range of indices should be unused regardless of their values (0 or not).

We note also that the piggy bank represents a bag of coins. So, you do not have to maintain the order of elements in coins array as they have been originally added.

You can develop a test method to check the correctness of your *PiggyBank.removeCoin()* method. But, this is not required.

3.5.2 Empty the piggy bank

Below are the javadoc style method header comments and the signature of the function related to removing all the coins in a piggy bank. This method must be implemented and added to your `PiggyBank` class.

```
/**
 * Removes all the coins in a piggy bank. It also displays the total value
 * of the removed coins
 *
 * @param coins an oversize array storing the coins held in a piggy bank application
 * @param size number of coins held in coins array before this method is called
 * @return the new size of the piggy bank after removing all its coins.
 */
public static int emptyPiggyBank(int[] coins, int size) {}
```

- if the *emptyPiggyBank()* method is passed an empty piggy bank as input parameter, it displays the following message.

  `"Zero coin removed. The piggy bank is already empty."`

- Otherwise, the following message will be displayed.

  `"All done. " + <sum_of_all_removed_coins> + " cents removed."`

You have now to implement the test method for the `PiggyBank`.*emptyPiggyBank()* method in your `PiggyBankTester` class with exactly the following signature. After your *emptyPiggyBank()* returns, the size of the piggy bank must be zero. You do not have to check for the correctness of the displayed messages in your test method.

```
public static boolean testEmptyPiggyBank() {}
```

4 Piggy Bank Driver

Now, we have all the operations defined for our piggy bank application implemented and tested. It is time to compose the driver application. To do so, download the already implemented PiggyBankDriver.java source file and add it to the src folder of your project. Make sure to read carefully the provided source code. Notice that we organized the driver main method into more specific static private helper ones. Note that you ARE NOT going to submit the `PiggyBankDriver.java` source file on gradescope. Our grading tests will only check for the correctness of the methods implemented in `PiggyBank` and `PiggyBankTester` classes.

We provide you in the following with a few notes about the `PiggyBankDriver` class.

- Running the main method within the `PiggyBankDriver` class should result in an interaction section comparable to the sample shown at the top of this write-up (cf. Section 4).
- The provided implementation does not consider user inputs that contain syntax error. For instance, the cases where the user enters for instance `a` or `a 25 10 blabla` or `a 10`. We did not worry about how our program would operate in such cases.
- This program does not consider any type mismatched user inputs either. We considered that all entered user command lines are properly encoded. For instance, we assume that the the second argument within an *add one coin* command entered by the user is always an integer. If the user enters a string or a double as value of the coin to be added, the program will crash and terminates and that is all right. We do not have to worry about such cases at this level of the course.

The following illustrates a demo (sample of run) of the driver method of this programming assignment.

```
======== WELCOME to the Piggy Bank Application ========

COMMAND MENU:
[A <coin>] Add one coin value in cents to the piggy bank.
[B] Display the Balance of the piggy bank in dollars.
[E] Empty the piggy bank (remove all coins from the piggy bank).
[R] Remove an arbitrary coin from the piggy bank.
[P] Print/display the content of the piggy bank.
[Q] Quit the application.
[H] Help (display this Menu).

ENTER COMMAND: r
Tried to remove a coin, but could not because the piggy bank is empty.

ENTER COMMAND: a 10
Added a DIME.
```

```
ENTER COMMAND: a 5
Added a NICKEL.

ENTER COMMAND: a 25
Added a QUARTER.

ENTER COMMAND: a 10
Added a DIME.

ENTER COMMAND: a 1
Added a PENNY.

ENTER COMMAND: a 15
15 is not a valid US currency coin in cents.

ENTER COMMAND: a 1
Added a PENNY.

ENTER COMMAND: a 25
Added a QUARTER.

ENTER COMMAND: b
Balance: $0.77.

ENTER COMMAND: p
QUARTERS: 2
DIMES: 2
NICKELS: 1
PENNIES: 2
Balance: $0.77

ENTER COMMAND: r
Removed a QUARTER.

ENTER COMMAND: b
Balance: $0.52.

ENTER COMMAND: R
Removed a PENNY.

ENTER COMMAND: r
Removed a PENNY.

ENTER COMMAND: P
QUARTERS: 1
DIMES: 2
```

```
NICKELS: 1
PENNIES: 0
Balance: $0.5

ENTER COMMAND: e
All done. 50 cents removed.

ENTER COMMAND: a 10
Added a DIME.

ENTER COMMAND: b
Balance: $0.1.

ENTER COMMAND: q
======== Thank you for using this Application! ========
```

5 Assignment Submission

Congratulations on finishing this CS300 assignment! After verifying that your work is correct, and written clearly in a style that is consistent with the CS300 Course Style Guide, you should submit your final work through Gradescope. The only 2 files that you must submit include: `PiggyBank.java` and `PiggyBankTester.java`. Your score for this assignment will be based on your **"active"** submission made prior to the hard deadline of Due: **11:59PM on June 22**nd. The second portion of your grade for this assignment will be determined by running that same submission against additional offline automated grading tests after the submission deadline. Finally, the third portion of your grade for your submission will be determined by humans looking for organization, clarity, commenting, and adherence to the CS300 Course Style Guide.