CS201 - Assignment 5, Part 2
Due: Friday March 21, by 5pm

For this assignment you may (and are encouraged to) work with a partner if you'd like. If you do, you must do **all** of the work together. That means if either of you is working on it, the other person should be there. Only turn in one copy of the assignment, but make sure both of your names are in the comments at the top of the file.

1 Warm-Up

For the main part of this assignment we will be using `ArrayList`s, so to make sure you're comfortable using them, there's a few warm up methods.

Note: To use the `ArrayList` class, you need to import it at the top of your file:

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

To help you make sure you understand creating and using `ArrayList`s, I've include a JUnit test class (and a few supporting files) for the warm up called `WarmUpTester.java` at:

`http://www.cs.middlebury.edu/~dkauchak/classes/cs201/assignments/assign5/`

(If you're rusty on how to create a `JUnit` test, look back at Assignment 3, part 1.)

In a file called `WarmUp.java`, write the following static methods:

- `addFirstAndLast`: takes an `ArrayList` of `Double`s as a parameter and returns the sum of the first and last element in the `ArrayList`.
- `max`: takes an `ArrayList` of `Integer`s as a parameter and returns the largest one in the `ArrayList`.
- `getCards`: takes `int num` as a parameter and returns an `ArrayList` of `Card`s with `num` cards in it. You can use the `CardDealer` class to get the cards. I've included the `CardDealer` class and the `Card` class in the starter code along with the JUnit test so you don't have to search for them, but feel free to copy your own version.

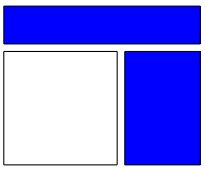
2 Making Dough!

For the main part of the assignment you are to write a dough allocator class. An object in the class starts with a square sheet of dough. In response to requests, the object gives out squares from

the sheet. As the original square is cut into smaller rectangles, the object must keep track of all the pieces. Once cut, the dough stays cut. A rectangle cannot be rejoined with other rectangles to make a larger piece.

The object maintains an inventory of rectangles of dough. When a request is made, the object chooses a suitable rectangle from its inventory. (Later, we will have more to say about how the choice is made.) The object then cuts the square from the selected rectangle and returns any remainders to the inventory. There are three possibilities:

1. The requested square is exactly the same size as the chosen rectangle. In this case, the rectangle is simply removed from the inventory.
2. The side of the requested square exactly matches one side of the rectangle. The rectangle is removed from the inventory, the square is cut out, and the remaining rectangle is returned to the inventory.
3. The side of the requested square is smaller than both dimensions of the rectangle. The rectangle is removed from the inventory, the square is cut out, and *two* smaller rectangles are returned to the inventory (see the illustration to the left). One of your implementation decisions is to decide between the two ways to divide a rectangle. We use rectangles because they are more easily stored and manipulated than more complex shapes, and there is no point in dividing the original rectangle into more than two pieces.



3 Making Dough in Java

You are to write a Java class, named `DoughAllocator`, that contains the following public methods:

`public DoughAllocator(int size)`, a constructor which begins with a single square sheet whose side is the specified size. If the size is not greater than 0 *or* if the size is greater than 16,000 your program should print an error and then set the size to a default size of 100. The upper bound on `size` is chosen so that area of the square of `size` is still an ordinary integer.

`public int area()`, a method that returns the total area of dough remaining.

`public int largest()`, a method that returns the length of the **side** of the largest **square** that could currently be allocated.

`public void get(int size)`, a method that provides a square of dough of the specified size. If this were a real allocator, the method would provide actual dough, and the return type would be something other than `void`. The method will succeed if `size` is positive and not greater

than the largest square available. Otherwise, the method will print out an error and NOT change the dough.

In addition to the class methods above, your code must use an `ArrayList` to maintain the inventory of rectangles.

Please read the specifications carefully; some details are easy to overlook. For example, the call `get(0)` must not succeed, even though the request could logically be granted.

To keep track of the individual rectangles of dough, you should also implement a class called `DoughRectangle`. When the dough allocation begins, you will just have a single rectangle (really a square) of dough. However, as you start to grant requests, you will need to keep track of what rectangular scraps are remaining.

Besides the methods described above, you are free to make your own implementation decisions.

3.1 Dough allocation strategy

After several steps, the dough will be divided into many rectangles, and it is not obvious which rectangle to select when the next request comes in. There is no optimum allocation strategy, because we do not know what the future requests will be. Even if we did, finding the best strategy is a very hard problem. We do, however, place a few requirements on the allocation.

- The class must grant a request for a square of size 8 immediately after constructing a sheet of size 8. This requirement eliminates the possibility of the constructor immediately dividing the dough into squares of size 1.
- Similarly, the class must grant requests for squares of size 3 and 5 immediately after constructing a sheet of size 8.
- From a newly-constructed sheet of size 8, the class must grant requests of sizes 3, 3, 3, 2, and 2. It may or may not be possible to grant a subsequent request of size 3, depending on the choices made for previous squares.
- If `area()` reports a value of 6, then it must be possible to satisfy six successive requests for squares of size 1.

3.2 Path to implementation

Since I'm giving you a fair amount of freedom on this assignment, make sure that you spend some time thinking through how you're going to implement everything (in particular, your `DoughRectangle` class and how that will be used:

- What data must a `DoughRectangle` keep track of? Put another way, what makes one `DoughRectangle` different from another. This should help you decide on the private instance variables.

- What types of questions will you want to ask about a `DoughRectangle`? Any such questions will likely become `public methods`.

As always, I strongly recommend working incrementally, writing a method or two, testing it and then working from there. Start with the `DoughRectangle` class and get the basics working. You may not have all of the methods figured out at the beginning, but get something in place. Then, you can move on to the `DoughAllocator` class. You may have to go back to the `DoughRectangle` class, however, to add another method or two as you work through the rest of the problem.

3.3 A few test cases

Here are a few test cases you can try out once you get your `DoughAllocator` working (or you think it's working). It will also help you understand how we call the different methods:

```
public static void test1(){
    ArrayList<Integer> sizes = new ArrayList<Integer>();

    sizes.add(5);
    sizes.add(3);

    testHelper(sizes);
}

public static void test2(){
    ArrayList<Integer> sizes = new ArrayList<Integer>();

    sizes.add(3);
    sizes.add(3);
    sizes.add(3);
    sizes.add(2);
    sizes.add(2);
    sizes.add(5);

    testHelper(sizes);
}

private static void testHelper(ArrayList<Integer> sizes){
    DoughAllocator allocator = new DoughAllocator(8);
    System.out.println("Area before: " + allocator.area());
    System.out.println("Largest before: " + allocator.largest());

    for( Integer s: sizes ){
        System.out.println("-----------");
        System.out.println("Requesting a square of size: " + s);
```

```
        allocator.get(s);
        System.out.println("Area: " + allocator.area());
        System.out.println("Largest: " + allocator.largest());
    }
}
```

For example, the output from `test1` should be:

```
Area before: 64
Largest before: 8
-----------
Requesting a square of size: 5
Area: 39
Largest: 3
-----------
Requesting a square of size: 3
Area: 30
Largest: 3
```

4 JUnit

To give you extra motivation to use JUnit (and practice with it), you must also include a class called `TestDoughAllocator` that includes at least two tests for your `DoughAllocator` or `DoughRectangle` classes (just two tests total). Of course, I encourage you to write more!

5 Extra Credit

If you turn in this assignment by the beginning of class on Wednesday, you will receive 5% extra credit.

6 When You're Done

Make sure you have comments at the top of any file/class that you wrote with with your name and the assignment number.

JavaDoc

For this assignment, we're going to start using JavaDoc style comments. All of your methods MUST have JavaDoc style headers.

To group these files into a single file, you need to export your project. To do this:

1. Right-click on the project (on Mac, ctrl+click) and select `Export`.
2. You'll see a number of options. Open up the `Java` folder and select `JAR file` and click `Next`.
3. You should just see your project selected. Below, make sure **only** the following two options are checked:
   - "Export Java source files and resources"
   - "Compress the contents of the JAR File"
4. Click on the `Browse...` button and pick a location to save the output file. Give the file a name like "kauchak1.jar", where "kauchak" is your last name and "1" is the assignment number.
5. Click `Finish`.

If all went well this will generate a single `.jar` file wherever you picked to save it.

Submit this JAR file as assignment number "5.2" via the online submission mechanism on the course web page.