Assignment 3: Abstracting over Data Definitions; Custom constructors; World building

Goals: Learn to use the `String` class, practice working with lists. Learn to design methods for complex class hierarchies.

Instructions

This assignment is long. Start early.

Be very, very careful with naming! Again, the solution files expect your submissions to be named a certain way, so that they can define their own `Examples` class with which to test your code. Therefore, whenever the assignment specifies:

- the names of classes,
- the names and types of the fields within classes,
- the names, types and order of the arguments to the constructor,
- the names, types and order of arguments to methods, or
- filenames,

...be sure that your submission uses exactly those names.

Make sure you follow the style guidelines that Bottlenose enforces. For now the most important ones are: using spaces instead of tabs, indenting by 2 characters, following the naming conventions (data type names start with a capital letter, names of fields and methods start with a lower case letter), and having spaces before curly braces.

You will submit this assignment by the deadline using the Bottlenose submission system. You may submit as many times as you wish. Be aware of the fact that close to the deadline the system may have a long queue of submissions which means it takes longer for your code to be submitted - so try to finish early.

The submission will be organized as follows:

- **Homework 3 Problem 1:** The `Media.java` file
- **Homework 3 Problem 2:** The `Bagel.java` file
- **Homework 3 Problem 3:** The `Centipede.java` file

Due Dates:

- Problem 1 – Monday, January 27th, 9:00pm
- Problem 2 – Monday, January 27th, 9:00pm

- Problem 3 – Thursday, January 30th, 9:00pm

Problem 1: Abstracting over Data Definitions

Related files:
Media.java

In this problem, our data will represent different kinds of media. Each piece of media has a title and list of available languages for captions. Different kinds of media can also contain additional information as shown in the supplied `Media.java` file.

Note: none of these methods are properly implemented. As given in the file, they are all *stubs* that currently return a dummy value, so the code will compile but not yet work.

Warmup: Download the file and work out the following problems:

- Make at least two examples of data for each of the three classes.
- Design the `isReallyOld` method for each class. TV and YouTube are fairly modern formats, but movies that came out before 1930 are considered really old.
- Design the method `isCaptionAvailable` for each class, which determines if a given language is available for captions.
- Design the method `format` which produces a `String` showing the proper way to display the piece of media in text. Movies are formatted by the title followed by the year in parentheses, such as `"The Favourite (2018)"`. TV episodes are formatted by the show's title followed by the season and episode number, separated by a period, followed by a dash and then the episode title, such as `"Mad Men 1.1 - Smoke Gets In Your Eyes"`. YouTube videos are formatted by the video's title followed by the word by followed by the channel name, such as `"Threw It On The Ground by thelonelyisland"`.

Once you have finished these methods and are confident that they work properly, save the work you have done to a separate file. Do not submit the code as written so far. The problems below are the main point of this exercise, and it will be helpful for you to preserve the code written so far as a reference against which to compare your revised code below. *Again, submit only the work below.*

- Look at the code and identify all places where the code repeats — the opportunity for abstraction.

  Lift the common fields to an abstract class `AMedia`, which should implement `IMedia`.

  Make sure you include a constructor in the abstract class, and change the constructors in the derived classes accordingly. Run the program and make sure all test cases work as before.

- For each method that is defined in all three classes decide to which category it belongs:
  - The method bodies in the different classes are all different, and so the method has to be declared as `abstract` in the `abstract` class.

- The method bodies are the same in all classes and it can be implemented concretely in the `abstract` class.
- The method bodies are the same for two of the classes, but are different in one class — therefore we can define the common body in the `abstract` class and override it in only one derived class.

Now, lift the methods that can be lifted and run all tests again.

Problem 2: Working with Custom Constructors, A Taste of Equality

A `BagelRecipe` can be represented as the amount of flour, water, yeast, salt, and malt in the recipe. A perfect bagel results when the ratios of the weights are right:

- the weight of the flour should be equal to the weight of the water
- the weight of the yeast should be equal the weight of the malt
- the weight of the salt + yeast should be 1/20th the weight of the flour

Design the `BagelRecipe` class. The fields should be of type `double` and represent the weight of the ingredients as ounces. Provide three constructors for this class:

- Your main constructor should take in all of the fields and *enforce* all above constraints to ensure a perfect bagel recipe.
- Provide another constructor that only requires the weights of flour and yeast, and produces a perfect bagel recipe.
- Provide another constructor that takes in the flour, yeast and salt **as volumes rather than weights**, and tries to produce a perfect recipe. Here, too, a perfect recipe should be enforced.

Flour and water volumes are measured in cups, while yeast, salt, and malt volumes are measured in teaspoons.

Helpful conversion factors:

- 48 teaspoons = 1 cup
- 1 cup of yeast = 5 ounces
- 1 cup of salt = 10 ounces
- 1 cup of malt = 11 ounces
- 1 cup of water = 8 ounces
- 1 cup of flour = 4 and 1⁄4 ounces

Once you've completed the above constructors, you should:

- Remove as much duplicate code as possible from these constructors.

- Implement the method `sameRecipe(BagelRecipe other)` which returns true if the same ingredients have the same weights to within 0.001 ounces.

Problem 3: Centipede, Part 1

You are going to build a version of the game Centipede. If you aren't familiar with this classic arcade game: you are a garden gnome, tasked with keeping your garden free of the dreaded scourge of centipedes. A long centipede starts at the top of the screen and zig-zags its way down to the bottom, attempting to eat the player, before making its way back to top to start all over again if both it and the player survives that long. When a centipede hits an obstacle, it moves down (or up) a level and reverses its direction (except for obstacle avoiding, a centipede only ever moves left or right). The player can, in turn, shoot darts at the centipede to try to defeat it. Just to be clear, you will not be implementing the entire game on this assignment, but will incrementally complete the game over a handful of assignments.

Your game will use the `javalib.funworld` library, which provides an implementation of `World`s and `bigBang` similar to what was used in Fundies 1. Read the documentation carefully for more information.

> You may find constructors other than the one described here helpful, which is fine, so long as this one can be used by the player.

Our game will take place on an m by n grid, where both m and n are integers greater than one. When someone launches the game, they should only have to pass these two numbers to their constructor.

The player should be presented with an entirely green grid of tiles (your garden is initially a well maintained lawn of green grass). A game like this is quite boring though: the centipede will simply traverse the lawn from side to side. We need a way to add obstacles to the board. We'll allow the player to place two kinds of obstacles on the board: dandelions and pebble piles. When the player *left-clicks* on a grass-filled grid cell, you should place a dandelion on that square. When the player *right-clicks* on a grass-filled grid cell, you should place a pile of pebbles on that square. (Both kinds of obstacles will affect the centipede's motion, in ways we'll describe in Part 2 of this project.) If a player left-clicks on a non-grass-filled cell, it should be reset to grass.

Clicking on cells to place obstacles is quite tiresome, so we also want to support adding multiple obstacles at once. If the player presses the 'D' key, you should *randomly fill* 5% of the available squares with dandelions If the player presses the 'P' key, you should *randomly fill* 5% of the available squares with piles of pebbles. If the player presses the 'R' key, you should *reset* all cells back to grass. (If the user presses two keys, one after another, it's possible to "overwrite" one of the obstacles with another obstacle. That's ok; you don't have to ensure that every single selected cell is currently grassy, before adding an obstacle to it.)

No obstacles of any kind are permitted in the bottom row.

The player needs to start in the *bottom* row, in the leftmost square. If the player presses the left or right arrow keys, you should move the player one cell to the left or right, accordingly.

Finally, when the player presses the 'S' key to start the game, the world should end (for now).

This is by far the largest program you have designed for this course or Fundies 1. Keep the Design Recipe in mind **at all times** and follow it **at all times**. After properly designing your data definitions and their interpretations, templates and examples, there are two general programming practices you could follow: wish-list-as-you-go, often called top-down programming, or build-from-the-wish-list, often called bottom-up programming. In the former, you would begin by writing your on-tick function with helpers you don't actually have, commenting it out, and then adding those helpers to your wish list with their proper signature and purpose statement. In the latter, you start by building your wish list, writing those helpers, and then finish with writing the on-tick function. The latter has the disadvantage in that only after you have written helpers might you realize that the helper is not as helpful as you would hope, but the former has the disadvantage in that it is much harder to test code you don't actually have helpers for. All programmers do a mixture of both to some extent, and it is up to you decide what balance will best help you write this program.

Start early, test thoroughly, and come to office hours if you need any help.

3.1 A note about randomness

There are two ways to generate random numbers in Java. The easiest is to use `Math.random()`, which generates a `double` between 0 (inclusive) and 1 (exclusive). You can multiply this number by some integer to make it bigger, then coerce to an `int` to produce a random integer in the range you wish. However, this is not easily testable: you'll get different random values every time.

The better way to generate random numbers is: First, `import java.util.Random` at the top of your file. Next, create a `new Random()` object, and use its `nextInt(int maxVal)` method, which will give you a random integer between zero (inclusive) and `maxVal` (exclusive).

> This is known as a "pseudorandom number generator", since the numbers aren't really random if they can be reliably repeated...

The reason this is better is because there is a second constructor, `new Random(int initialSeed)`, which has the property that every time you create a `Random` with the same initial seed, it will generate the same "random" numbers in the same order every time your program runs. You should therefore design your world classes with two constructors:

- One of them, to be used for testing, should take in a `Random` object whose seed value you specify. This way your game will be utterly predictable every single time you test it.
- The second constructor should *not* take in a `Random` object, but should call the other constructor, and pass along a really random object:

```
import java.util.Random;

class YourWorld {
  Random rand
  // The constructor for use in "real" games
  YourWorld() { this(new Random()); }
  // The constructor for use in testing, with a specified Random object
```

```
    YourWorld(Random rand) { this.rand = rand; ... }
  }
```

Now, your tests can be predictable while your game can still be random, and the rest of your code doesn't need to change at all. While there is only one element of randomness in this iteration of the game, more is likely to follow.