Programming Exercise: Interface and Abstract Class

In this task you will add in a common interface. You will need to keep a copy of your previous program, so that you will have the old version for the quizzes. The new version will organize the program in a different way, using an interface and an abstract class. You will also use a different class in place of the **MarkovRunner** class to run the program in a different way to take advantage of the new structure.

For this assignment we will give you an incomplete program with some new files. You should look at each of these and understand their parts. They are:

- The **IMarkovModel** interface. It has two signatures: the void method **setTraining** that has one String parameter named **text**, and the method **getRandomText** that has one int parameter named **numChars** and returns a String.
- The class **MarkovRunnerWithInterface** for running your program to generate random text. This class has several methods:
    - A void method named **runModel** that has three parameters: an IMarkovModel variable named **markov**, a String named **text** and an int named **size**. This method will work with any **markov** object that implements IMarkovModel.
    - A void method named **runMarkov**. This method creates one of the types of Markov models, and calls **runModel** with it to generate random text.
    - A void method named **printOut** that formats and prints the randomly generated text.
- The abstract class **AbstractMarkovModel** that implements IMarkovModel. This class has several items:
    - Two protected fields **myText**, a String, and **myRandom**, of type Random.
    - A constructor that creates **myRandom**.
    - A **setTraining** method that is public. This method sets the the private String variable **myText** to the parameter **text**.
    - A signature for the abstract method **getRandomText** that has one integer parameter named **numChars** indicating the length of the randomly generated text.

Assignment 1: IMarkovModel Interface

Specifically, for this assignment, you will:

- Copy over four of the .java files from the the work you did with the previous assignment. (You will need to keep a copy of the program you created from the first two assignments for testing.) Specifically you should copy over the four Java files: MarkovZero.java, MarkovOne.java, MarkovFour.java, and MarkovModel.java. You can ignore the following files from the previous lesson—MarkovRunner.java and Tester.java. We will modify the program to organize it in a different way, using an interface and an abstract class.
- Modify your classes **MarkovZero**, **MarkovOne**, **MarkovFour**, and **MarkovModel** to implement the **IMarkovModel** interface. Each of these classes should already have the two required methods **setTraining** and **getRandomText**, so the only change needed is the first line to add

  ```
  implements IMarkovModel
  ```
- Run the method **runMarkov** that is in the **MarkovRunnerWithInterface** class. This method should run several Markov objects and generate random text for a MarkovZero, a MarkovOne, a MarkovModel with number 3, and a MarkovFour. Notice that **runMarkov** is called with each one of these. You can observe that the text gets more like the training text as you move from MarkovZero to MarkovFour.

Assignment 2: AbstractMarkovModel Abstract Class

Now we will integrate the **AbstractMarkovModel** abstract class—you will need to make several changes, specifically:

- The method **getFollows** you wrote is the same everywhere. You should move it into the **AbstractMarkovModel** class and change this method from public to protected.
- Anything that is protected in **AbstractMarkovModel** should be deleted in the other files they are in. In each of MarkovZero, MarkovOne, MarkovFour and MarkovModel, delete the declarations of the two private variables **myText** and **myRandom**, and delete the **getFollows** method if it has one.
- Each of MarkovZero, MarkovOne, MarkovFour, and MarkovModel need to change in their definition “implements IMarkovModel” to “extends AbstractMarkovModel”
- Run the method **runMarkov** that is in the **MarkovRunnerWithInterface** class. It should run as before.
- Notice that when you print any **markov** object, you get its name and an address location. These objects don’t know how to print themselves. Write a **toString** method, so each one prints out “MarkovModel of order n,” where n is the number. For example, a MarkovFour object should print out “MarkovModel of order 4.”
- Modify your program so that you can set a seed in the **runMarkov** method in the **MarkovRunnerWithInterface** class and pass it to the **runModel** method (add a 4th parameter named **seed**) to set the random seed for any **markov** object. In the **runModel** class you should be able to set this seed with the line:

  ```
  markov.setRandom(seed).
  ```

  If you try this you get a compile error. What changes in other files will you need to make to get this to work?

Assignment 3: HashMap for Efficiency

It was noted that the **getRandomText** method is inefficient. Suppose it sees the String “he” 50 times. Each time, it will calculate the follow set again, which could take a long time, each time if the training text is long.

Specifically, you should:

- Write a new class named **EfficientMarkovModel** (make a copy of MarkovModel to start with) that extends AbstractMarkovModel and that builds a HashMap to calculate the **follows** ArrayList for each possible substring only once, and then uses the HashMap to look at the list of characters following when it is needed. This class should include:
    - a **toString** method to print that this is the EfficientMarkovModel class of a specific number
    - a method named **buildMap** to build the HashMap (Be sure to handle the case where there may not be a follow character. If that key is not in the HashMap yet, then it should be put in mapped to an empty ArrayList.) Think carefully about where to call this method, considering that you will want to build a map for each new training text.
    - a **getFollows** method, but this **getFollows** method should be much shorter, as it can look up the ArrayList of Strings, instead of computing it each time.
- To test your HashMap to make sure it is built correctly, write the void method **printHashMapInfo** in the EfficientMarkovModel class. Make sure to call this method immediately after building the map. This method should print out the following information about the HashMap:
    - Print the HashMap (all the keys and their corresponding values). Only do this if the HashMap is small.
    - Print the number of keys in the HashMap
    - Print the size of the largest value in the HashMap—that is, the size of the largest ArrayList of characters
    - Print the keys that have the maximum size value.
- Write a new method named **testHashMap** in the MarkovRunnerWithInterface class. This method should create an order-2 EfficientMarkovModel with
    - seed 42

  - the training text is “yes-this-is-a-thin-pretty-pink-thistle”
  - the size of the text generated is 50
  - Note that “le” occurs only once at the end of the training text
- In the MarkovRunnerWithInterface class, call **testHashMap**. You should see that the HashMap has the following information:
  - It has 25 keys in the HashMap
  - The maximum number of keys following a key is 3
  - Keys that have the largest ArrayList (of size 3 in this case) are: “hi”, “s-”, “-t”, “is”, and “th”
  - After running it, you’ll probably want to comment out the call to **printHashMapInfo** in the EfficientMarkovModel class.
- In the MarkovRunnerWithInterface class, create a void method named **compareMethods** that runs a MarkovModel and an EfficientMarkovModel. In particular,
  - Make both order-2 Markov models
  - Use seed 42 and set the length of text to generate to be 1000
  - Both should call **runModel** that generates random text three times for each.
  - Run the MarkovModel first and then the EfficientMarkovModel with the file “hawthorne.txt” as the training text. One of them should be noticeably faster. You can calculate the time each takes by using **System.nanoTime()** for the current time.