Generating Random Text

Word N-Grams

Review

Which of the following methods would you create when designing a WordGram? (Check all that apply)

- [ ] .compareTo
- [ ] .length()
- [ ] .toString()
- [ ] .equals()
- [ ] .wordAt(i)

Skip Submit

Have a question? Discuss this lecture in the week forums.

Interactive Transcript

Search Transcript English

0:03
As part of creating the design of a class, you'll typically want to think about how the class is used. This is often called generating use cases, and can be done for a class as well as for a program. Well, think about how a WordGram object will be used in our Markov programs. We have some experience to build on here. We'll create a WordGram from an array of strings. This is an analog of creating a string from N characters as when substring is used. We'll need to add a new string to the end of a WordGram. This is an analog of adding a following character to make a new key when generating random text. We're ready to summarize our initial design of WordGram. The state will be an array of strings. We'll store this in an instance variable just as my text was a string in the Character Markov programs. We'll look at Simple Behavior first. We'll need a get method to obtain the length of a WordGram just as String has a .length. This is called a get method because it simply gets a value. We'll need a method that's an analog of .charAt for Strings, but to get the word at a specific index. We'll want programmers to be able to use WordGram like other classes. This means we'll need a .toString() method for printing and other uses and we'll need a .equals() method for finding follows, as we'll see soon.

1:28
We might think about a .compareTo() method and having the WordGram class implement the comparable interface. That might make it general, but in our analysis of how WordGram would be used, we didn't come up with sorting. We won't design a feature that isn't going to be used, we can add features later, as needed.

1:47
Let's design the constructor for WordGram. Constructors initialize the state. You can see the code for the constructor here. In this case, the state is an array of strings we've named myWords. The strings in this array are copied from the source array passed as a parameter to the constructor. The number of strings to copy is another parameter to the constructor. As is the starting index of where to copy the strings from. The method System.arraycopy copies values from a source array, to a destination array. In this case, the destination is instance variable myWords. We'll look at the code for simple behavior in WordGram, these methods are direct analogs of string methods.

2:33
Method .wordAt returns the string at a specified index, just as .charAt returns the char at a specified index in a string. If the index isn't valid, either too low or too high, the method throws an exception. You've likely had such exceptions when you've made a bad index, in accessing array or array lists, or string values. Here, our code is like other classes in throwing an exception. You'll learn more about exceptions if you continue to study Java.

3:04
For now, we want our class to act like other classes.

3:09
The number of words is returned by .length, just as the number of characters is returned by .length in the string class. We've got two more methods so that WordGram behaves well with other classes. Just as our .wordAt method acted like related classes in throwing an exception for a bad index, we need these methods because programmers expect them. We'll create a .toString() method, as you've seen with other examples. This will help with printing WordGrams either in output or as debugging help. We'll also need .toString() to append values to a StringBuilder object in the get random text method. We'll create a .equals() method too. This will be important in determining when one WordGram object is equal to another in writing the get follows helper method. WordGram objects are equal if their lengths are equal, and if the strings at corresponding indices are equal too.

4:04
We'll need to use .equals rather than equals equals to check whether two strings are equal.

4:10
We'll look briefly at how code changes and moving from characters to words, once the WordGram class is completed. You'll do more on your own and in later lessons. You'll need to generate random text in a get random text method. In the character Markov 2 class, random text would be two character sequences like th and er, that appear in the training text. These are obtained using the .substring method. In the word Markov 2 class, random text is sequences like how long and no such. These sequences are formed by creating a new WordGram object, using the same arguments as in the character class myText, index, and myOrder. Happy programming.

Downloads

Lecture Video mp4
Subtitles (English) WebVTT
Transcript (English) txt

Would you like to help us translate the transcript and subtitles into additional languages?