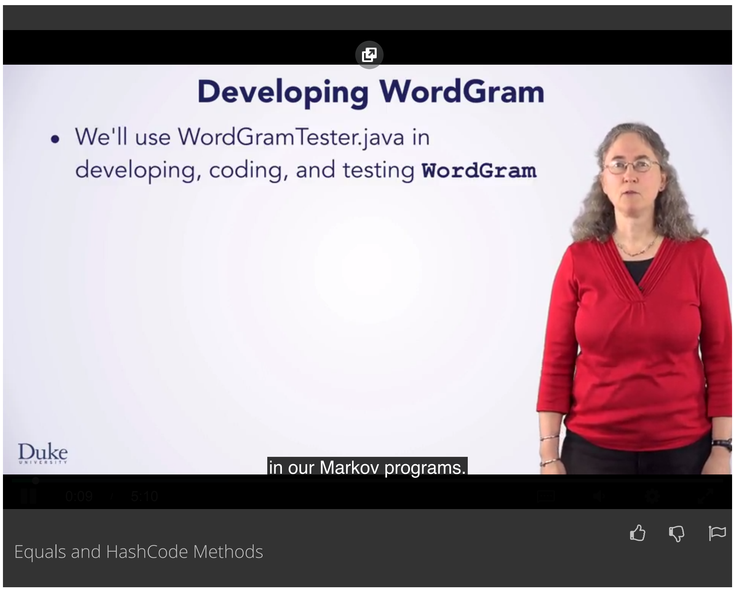
Generating Random Text

Word N-Grams

Review



Equals and HashCode Methods

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

Interactive Transcript

Search Transcript English

0:03
Hi! We're going to develop a few of the methods that make WordGram functional in our Markov programs. We'll use the program WordGramTester.java to demonstrate techniques, ideas, and test methods you might implement in any class, but in particular, for WordGram. We'll show how to test .toString and the WordGram constructor, though these are already written. We'll see why .equals is needed, and although we've discussed .equals in a previous course, this will be a brief discussion on implementing .equals, not just calling it. The .equals method will be needed for MarkovWord.getFollows to work correctly, that will be enough for the mark off runner class work with Markov Word and generating random text, we'll touch briefly on an advanced method.

0:53
We'll see how to implement .hashcode() so that WordGram objects can be added to a hash map. With knowledge we've .tostring, .equals, and .hashcode, you'll be ready to tackle lots of class and program design.

1:09
Let's look at .equals() first. In previous programs you've seen why you need to call .equals() with Strings. Using == doesn't work, because that tests whether two objects are the same object. That is, the same memory location, not whether they contain the same information. In other programs you've called .equals. Now we'll look at what's needed to write it. We must adhere to Java's requirements for writing .equals. The first requirement is that the parameter has type object. That's the base or parent class for every class in Java. The reasons for requiring this type will be explored if you take a more advanced course in object oriented programming.

1:53
We won't be calling .equals with any types other than WordGram. So, the first thing we do is cast the parameter o so that the compiler will treat it as a WordGram object. Casting, by putting the type in parenthesis, makes the compiler treat the object reference by object reference by parameter of as having type WordGram, which we do using the variable we've named other. But any name can be used.

2:19
Then we compare our links with the link of the WordGram object, reference by other and return fault as the links differ. That's the first step in implementing .equals. With .equals and that toString we're ready to code the mark off word classes. But we'll take a look ahead at what you'll study in later courses. And as an enhancement to the mark off classes we've been using. We can store all the keys used in generating text is a HashMap. We'd map the keys to a list of all characters that follow the key. This technique works with letter or word Markov models. We'll show this with this training text, and a two layered Markov model. The training text starts with, the herd then, and continues with more letters not shown. The ideas to avoid scanning the text many, many times for keys like he.

3:13
In our model, we find he and a follow character,

3:19
then the next occurrence of he, then the next occurrence of he. If we ever see the key he again, we need to rescan the text looking for follow characters, repeating work we've all ready done before. This could happen every time we generate he as a key and need to find the follow characters. Instead of scanning we could look up the key in a HashMap and retrieve the following characters stored in a list. This can be efficient as we'll see. For this to work we need to implement the .hashCode method.

3:50
We'll provide a high level overview of HashMaps, enough to get some understanding, but not lots of detail. The idea is to convert an object into an integer hash code. This hash code works as an index into hash map. For strings, the object hello might be converted to have the hash code 3.217. It's some number that tells us where in the HashMap to find hello.

4:17
The simplest idea for a hash code is to make every object have the same index. A number like 17. If we did this our code would work correctly if .equals was correct. But the performance would be really bad. Because every object would be stored in the same bucket, the word used for a place in a HashMap where objects are stored. Ideally, each object has a different number, so finding an object in a bucket is easy, because it's the only object in the bucket. You'll see details of how this works if you take another Java course, like the UCSD specialization that follows this one.

4:55
A simple idea with better performance is possible too, simply add the hash codes of each string in the WordGram.

5:03
Have fun with hashing objects, hopefully into lots of different buckets so your hash is fast.

Downloads

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

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