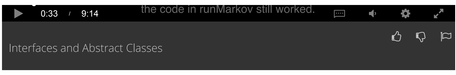
Pause and Reflect

Why would the code still work even though MarkovZero was changed to MarkovOne?

```
public void runMarkov() {
  FileResource fr = new FileResource();
  String st = fr.asString();
  st = st.replace('\n', ' ');
  MarkovZero markov = new MarkovZero();
  markov.setTraining(st);
  for(int k=0; k < 3; k++){
    String text = markov.getRandomText(200);
    printOut(text);
  }
}
```

```
public void runMarkov() {
  FileResource fr = new FileResource();
  String st = fr.asString();
  st = st.replace('\n', ' ');
  MarkovOne markov = new MarkovOne();
  markov.setTraining(st);
  for(int k=0; k < 3; k++){
    String text = markov.getRandomText(200);
    printOut(text);
  }
}
```

Continue



Interfaces and Abstract Classes

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

Interactive Transcript

Search Transcript

English

0:03
Hello. We're going to look at using interfaces and abstract classes to capture common features shared among many classes in the Markov programs.

0:13
We developed code in the method runMarkov of the MarkovRunner class to create random text we first used MarkovZero to generate random text. Then, we changed the variable Markov from the MarkovZero class to the MarkovOne class and the code in runMarkov still worked. This happened because we used the same method names in MarkovZero and MarkovOne. This means Markov.setTraining worked in both classes and Markov.getRandomText worked in both classes. We'd like to capture these commonalities with a Java interface so that we can use the classes in many ways. You may remember the interfaces Comparable and Comparator when sorting and the filter interface we developed to search for earthquake data.

1:03
We'll design and implement a new interface to capture the commonalities here.

1:09
Let's look at developing the interface.

1:12
We capture the common methods in the Markov classes by including their signatures in the interface. Here you see setTraining and getRandomText. The interface name here starts with an I. That's a common practice. So we've created IMarkovModel.

1:33
Each class that implements the interface indicates that with the Java keyword implements. As we see here with MarkovOne and again here with MarkovTwo. We already have the required methods with the required names in each class.

1:51
Using an interface will provide both utility and flexibility. Let's look at these. We can write a method that has a parameter with an IMarkovModel interface as seen here with the method runModel. The first parameter markov has type IMarkovModel. This means we can call runModel and pass a MarkovZero object as the first parameter, or a MarkovTwo object as the first parameter. The object named mz can be passed because it has type, MarkovZero, which implements the IMarkovModel interface. And we can pass the object named m2, because MarkovTwo also implements the IMarkovModel interface.

2:38
This call to markov.setTraining will call the appropriate method in MarkovZero or MarkovTwo, or any other class that implements the IMarkovModel interface.

2:50
This call to markov.getRandomText also calls code that is specific to the object and class passed as the first parameter.

3:01
This example illustrates what's called the open-closed software design principle. The idea is that classes are open to be extended, but closed for modification. Using an interface and other concepts we'll see soon, you can create a new class and use it in place of an already tested and proven class. You shouldn't modify code that works to extend the functionality of that code.

3:25
The IMarkovModel interface provides flexibility. As you've seen, we can develop a new general MarkovModel class that takes the place of Markov1, Markov2, and so on if this class implements the IMarkovModel interface we can use it with existing code.

3:45
That means we don't have to modify runModel for example to use the new class.

3:50
We could develop a more efficient implementation using a HashMap and use this in place of MarkovModel. For example, in the code you wrote the helper function getFollows that might be called hundreds of times to find the characters that follow th. Each time the entire text is re-scanned to find the characters that follow th.

4:13
By storing and reusing the follow characters, your code might be more efficient. And you could use it with runModel and other code because of the interface.

4:23
The IMarkovModel interface provides great flexibility, but there is some sum shared code that can avoid by developing what's called an abstract class.

4:35
Each Markov class we developed shares state and code, sometimes that's duplicated in each class. For example, each class has a random object, and the text used to model random texts, stored in its variables myRandom and myText, respectively.

4:53
Many of the classes share the exact same getFollows helper method that was copy-pasted into each .java file.

5:03
We'd like to avoid this duplication by capturing the common state in code in what's called an Abstract Base Class. This will rely on inheritance, an extremely important object oriented concept we'll touch on briefly here. You can learn more about this and other object-oriented concepts in the UCST specialization in Coursera.

5:26
Abstract base classes are used extensively in the java.util package with classes like AbstractList and AbstractMap, classes like ArrayList and HashMap. Or sub-classes of these abstract classes, sub-classes can inherit state and code or behavior. Let's take a look at the abstract base class. The abstract base class, AbstractMorkovModel is marked as abstract with the abstract keyword. We'll see what this means soon. The state shared across all classes like Markov1, Markov2, and MarkovModel, the subclasses of this class, is marked as protected rather than private. These instance variables would be accessible in each subclass that extends this abstract base class. We'll discuss the word extends next.

6:19
Let's take a closer look at Abstract and Shared Methods. The class is abstract because there is one method in the AbstractMarkovModel class, labelled as abstract. That method is the getRandomText method

6:35
which is implemented differently in each class that extends AbstractMarkovModel. These are called sub-classes. The helper function getFollows is labelled as protected. It can be called in each sub-class just as the protected instant variables can be accessed.

6:55
Let's look at extending the base class.

6:58
You can see the class MarkovModel that inherits state and behavior from the class AbstractMarkovModel we've been looking at.

7:08
The keyword extends means that this class gets instance variables and getFollows code from this super or parent class. Because the super or base class implements the IMarkovModel interface. This MarkovModel class implements the same interface that's inherited from the super class. This class has its own instance variable, myOrder, marked as private, as you've done in previous coding examples.

7:37
Classes that extend an abstract class must implement the abstract methods. The AbstractMarkovModel class has one abstract method, getRandomText.

7:50
This means the class, MarkovModel, must implement this method as you see here.

7:56
The MarkovModel subclass inherits protected state and behavior from the superclass.

8:02
This includes the protected instant variables, myRandom and myText, the training text.

8:09
Subclasses can call inherited methods too like getFollows.

8:15
We'll summarize the interface and the inheritance example.

8:20
The key idea of an abstract based class is to implement the interface and to supply default functionality when that's possible to avoid duplicating state or behavior in each subclass.

8:33
Sub classes like MarkovZero or MarkovOne will extend the base class.

8:39
Extending the class means that the subclasses inherit the interfaces of the parent or super class so that subclasses implement IMarkovModel too. This means client code won't change. The code that relied on the interface will still work. Some methods in the abstract base class are labeled as abstract, subclasses must provide implementations.

9:05
We saw different implementations of getRandomText in MarkovOne and MarkovTwo for example. Happy programming.

Downloads

Lecture Video mp4
Subtitles (English) WebVTT
Transcript txt

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