**Getting started with object oriented programming**

**Objectives**

* Get started with object-oriented programming
* Create a class
* Test the class.
* Learn the terminology

We are going to develop a class for an object-oriented program. From this lab forward, you will rarely use **static** methods besides the **main** method. The **static** methods are useful for what is known as a “procedural” programming paradigm. In your prior class you mainly learned procedural programming. In the remainder of this course, you will learn how to construct programs using the object-oriented programming paradigm.

With object-oriented programming, a class represents *a single thing* or *a single* *concept* and the class name should describe that thing or concept.

You may have previously heard (or read about) the words *encapsulation* and *abstraction*.

*Encapsulation* means that all of the data that represents the thing that you are dealing with *along with* all of the code that uses that data are included *within* the class.

*Abstraction* means that our class will contain code that performs actions based on the data in the class according to the purpose of the class, and that we will provide an interface that lets someone use the class without knowing about the details of how it is implemented.

Even if you have created object-oriented classes in the past, please follow the steps in this document.

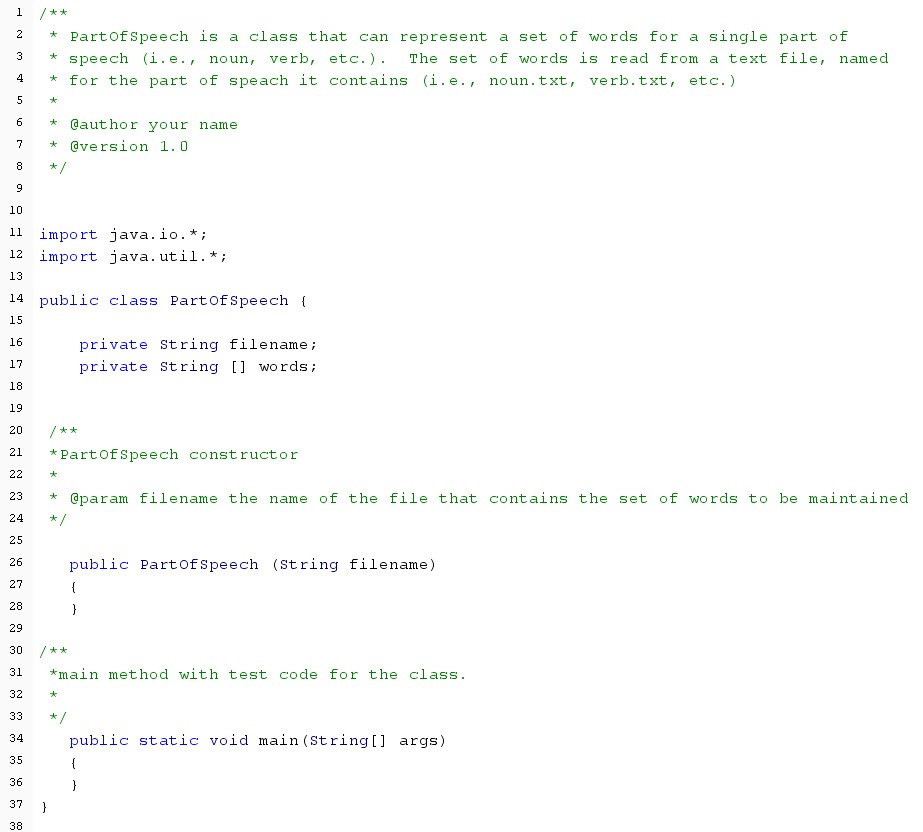
Besides going through the process of creating a class, it also shows a suggested method for creating your code in a structured and stepwise manner.

Start on the next page and type in what is shown exactly, including indentation, comments, blank lines, upper/lowercase, etc. Do not enter the line numbers. They are for reference and should be turned on in your Java IDE.

Compile and run where indicated.

|  |
| --- |
| Please use Camel Case when doing object-oriented programming. Camel Case means that each new word  (Except for usually the first one, but not always) is capitalized. As you construct your class, observe the case for each coding construct.  If you are used to separating words with underscores, you will  have to get out of that habit now. |

The text below is from Dr. Java. If you use eclipse, you will need to start a project for Lab 2 and add a class named PartOfSpeech to the project.



A

*constructor*

is a special method that allows an object

to be created.

Note some things about it - there is no explicit return type,

and the name is exactly the same as the class.

Otherwise, it takes parameters, just as any other method.

The

*access specifier*

public means that objects can be

constructed for the class.

Our class wil

l use an array to store the set of words that are of the part of

speech that it represents (so words will contain all nouns, or all verbs, or all

adverbs, etc.). We'll also store the filename that contains our data.

Variables defined within a class are called

*instance variables.*

The

keyword

*private*

means that these are only visible within the scope of the

class.

Previously you gave your class a name that was a general description of your

program. Now you will give your class a name that represents the thing or

concept that it represents.

Here, we are creating a class that represents a set of words that represents a

part of speech, (nouns, verbs, etc.), so the name is

*PartOfSpeech*

.

*Note*

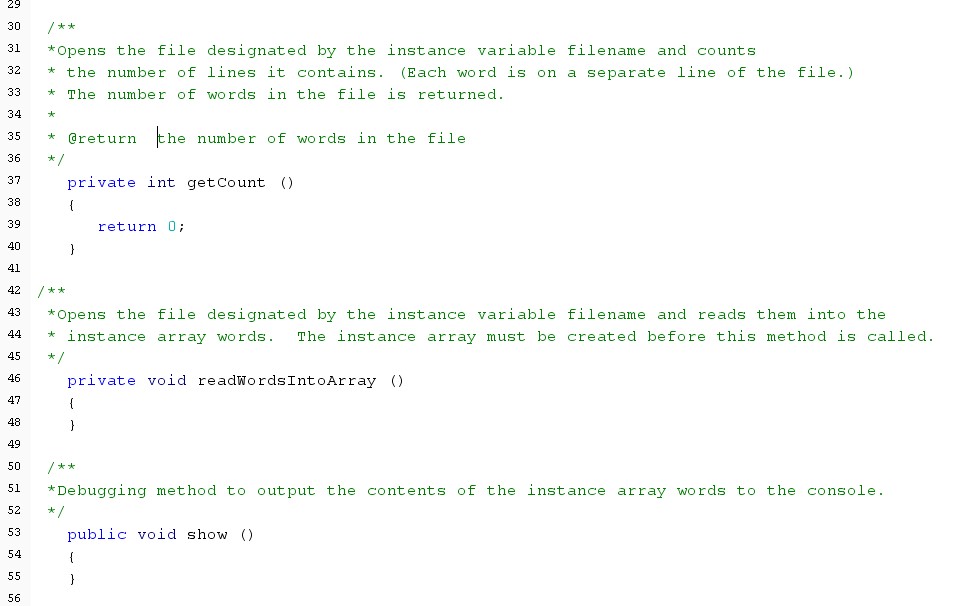
*that for class names, the first letter is also capitalized.*

Notice that no method is defined as static except the main method. Methods not defined as static are called *instance methods* and to use them, you need to call them using an object.

*Make sure that your code compiles before moving on.* This should be your usual approach. Create the basic structure of your class and shells for constructors and then methods, compiling and testing as you go.

Now that we have our class, some instance data, and a constructor outlined, we'll add some additional methods.

Add the method shells shown below. Shell is a term for adding the method signature, braces, and a return statement sufficient to satisfy the compiler. Be sure to add the comments as well.



Notice that two of these methods are declared as

*private*

. This means that these methods can only

be called

*within the scope*

of the class. They cannot

be called using a class object.

Notice also that class methods should begin with a

lower case letter.

Notice that this method is declared as

*public*

. This means that this

method can be used both within the scope of the class and can also

be called using an object. For example, we can have the following

code:

PartOfSpeech nouns = new PartOfSpeech("nouns.txt");

nouns.show();

Compare this to something like using a Scanner object:

Scanner in = new Scanner(System.in);

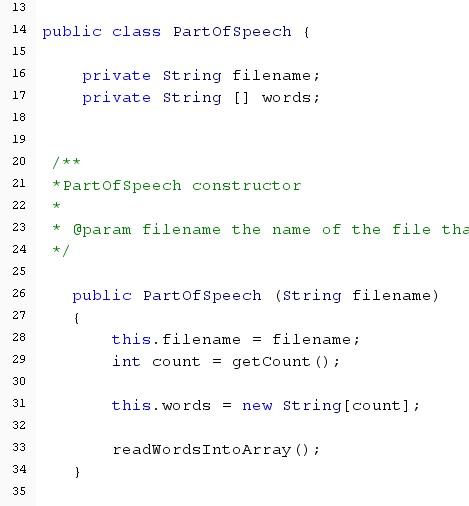
int value = in.nextInt();

This tells you that nextInt() is also defined as a public method of class

Scanner, since it is called using an object.

*Make sure that your code compiles before moving on.*

Next we will start filling in our methods. Let's start with the constructor. Type in the code for the constructor body (lines 28-33).



This line of code is taking the value passed in

as the parameter filename and saving in our

instance variable, also named filename.

The keyword

*this*

is used to designate that

we are using the instance variable (declared

on line 16).

And the same for this.words - the code on

line 31 is setting the value of the variable

declared on line 17.

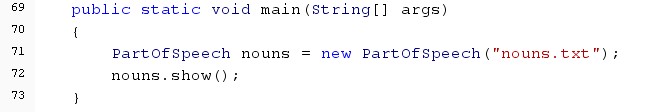
Instance variables are within the scope of the

class and can be used by all of the instance

methods (or non-static methods) declared

within the class.

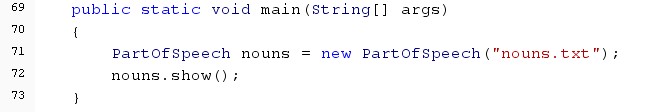
Next add code to the main method to create a PartOfSpeech object and call the show() method:



*Make sure that your code compiles before moving on.* You can also run it, but nothing will happen until we complete our other methods.

Let's pause for a moment and look at what happens when we use the *new operator* to create an object:

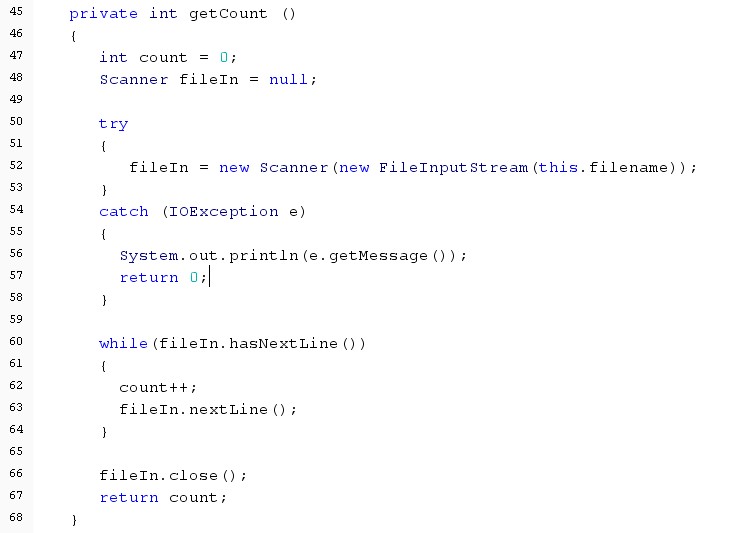
*A class is a data type*. First we are declaring a variable of type PartOfSpeech named nouns.



Then we use the new operator to create the object. When we use the new operator that causes the constructor to be called. Even though we don't put a return value when we write the constructor code, the constructor actually returns the object that is assigned to the variable.

|  |  |
| --- | --- |
|  | In the code above, "nouns.txt" is sent to the constructor as the actual parameter data.    The code in the constructor takes that value and assigns it to the instance variable called filename.    So after the call to constructor, a new object exists and has "nouns.txt" stored for filename.    Every time a new object is created, memory is reserved for the instance variables that are defined for the object. If you create another **PartOfSpeech** object, then different memory is reserved for the instance variables for that object, and so on. |

Next we will continue filling in our methods. Type in the code for the instance method getCount ( here as lines 47-67).



Declare a Scanner variable and set it to null.

Setting it to null is necessary because we are

going to create the object in the try block,

but want the Scanner object to be used

outside of the try block.

We are

using the filename that was

stored for the object in the constructor

code. Remember that the this

operator indicates that we are using an

instance variable.

We will loop as long as there are lines left in the file.

Remember, the Scanner object allows only

sequential access of the file - we start at the front,

move through the file by reading data until we reach

the end of the file

Remember that fileIn.hasNextLine() does not cause

the file pointer to move through the file. We have to

call the nextLine() method to read the data and

advance the file pointer to the next thing.

Since we are done with the file, close it and

return

the count of lines in the file.

*Why are we getting the count of things in the file?*

We want to create an array to hold the items in the file, and it is easiest to create an array when we know at the start how big it needs to be.

The file has words that are one to a line, so counting the lines in the file gets us the number of words in the file.

We can use a File object to find out how many bytes are in the file, but since our data is not defined in the file in terms of bytes, we have to figure out how many words are in the file. Since the words are in the file one to a line, it is pretty easy to count the lines.

*Compile and run your code. What happens?*

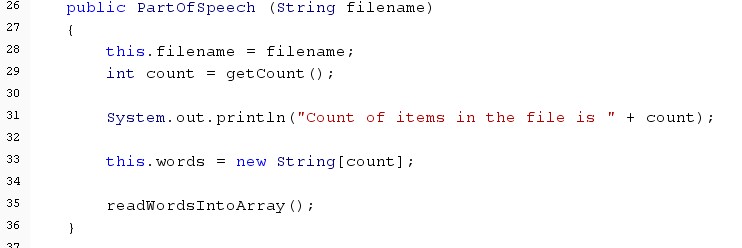


Since we have not yet added the data file to the working folder of the program, we have triggered an exception and moved to the catch part of the try-catch structure. We've output the exception message and returned a 0 for the count of items in the file.

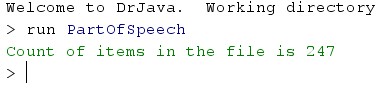
Using the exceptions to your advantage lets us more easily debug our code. If we had just ignored and re-thrown the exception, our program would have ended and we would have no idea why.

Move the file **nouns**.txt into the working folder of your program. If you are using DrJava, that is the same folder as your source code. If you are using Eclipse, it is in the project folder.

Put a debugging message in the constructor code (line 31 here) and rerun the program with the data file in the right place. Note that in the final version of the class, you remove or comment out line 31.

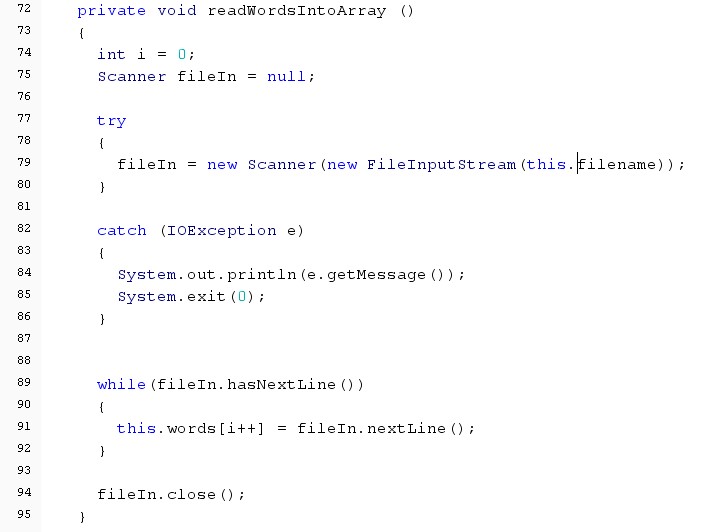


Now we see that we have 247 words in our file.



Look at the constructor above. What is the next method that we have to complete?

If you thought readWordsIntoArray(), you were correct.



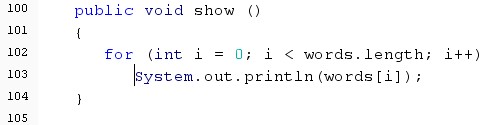
This looks a lot like the getCount() method, except that

instead of just counting the lines, we'll read the word

on each line and add it to our instance array of String

objects.

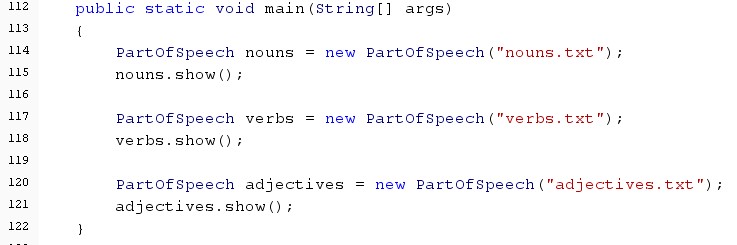
*Compile your code, then complete the show() method in order to see the contents of the array.*



|  |  |
| --- | --- |
| *Compile and run your code.*    You should see the contents of the array in the console window. |  |

Now we have an object that represents a set of words of one part of speech. Let's create objects for different parts.

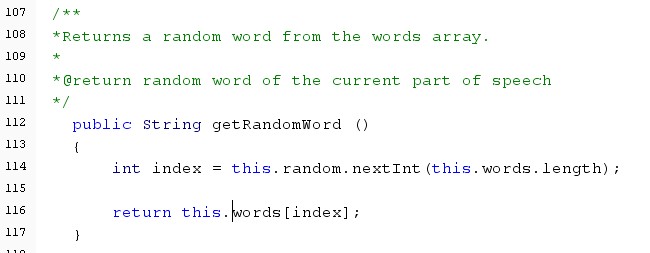
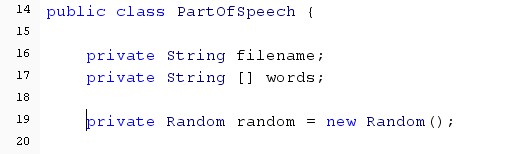
Add code to main to create and show the words for other parts of speech.



Now we want to add a method to our class that lets us get one random word from our set.

First, add an *instance variable* that is an *object of type Random*.

Add the following method to class PartOfSpeech:



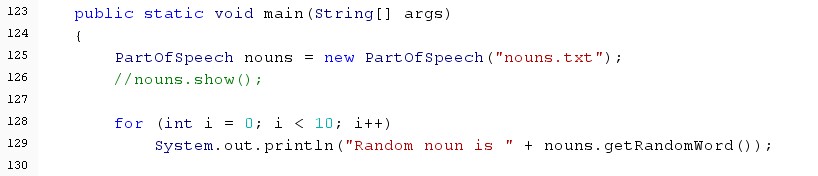
This is using the instance variable random.

Why wouldn't we just put the creation of the Random

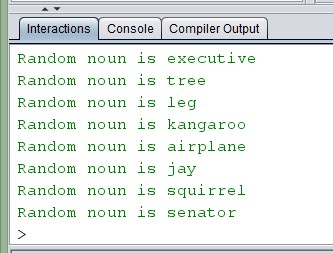
object here in the method?

*Make sure your code compiles before moving on.*

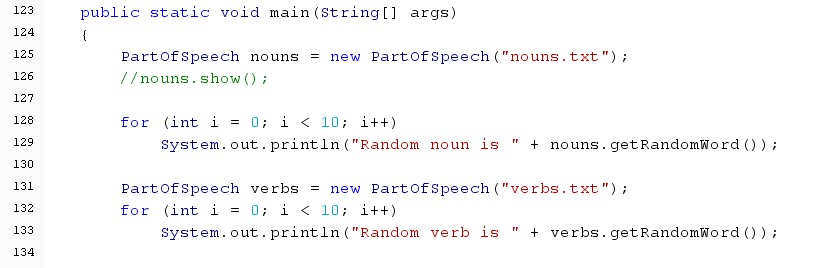
Put some code in the main method to test your new method:



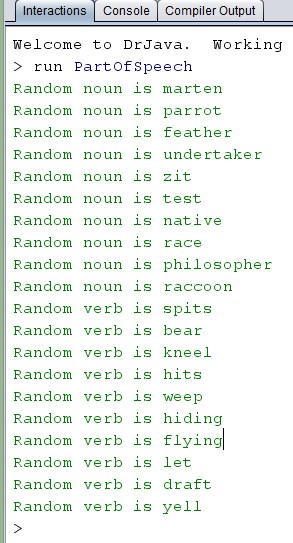
And observe the result:



Test another object:



And observe the result:



Now add code to the main method to create an adjective object and display 10 random adjectives from the **adjectives**.txt text file..

Copy the output of a successful run of **PartOfSpeech** into a comment at the end of your **PartOfSPeech**.**java** file.

Add your ID block to the top of **PartOfSpeech**.**java** .

Submit the java file to canvas.