**Name**

**Advanced Programming in Java**

**Lab Exercise 11/5/2020**

If you go into a bank and ask for a cashier's check, your bank computer system will print the check for the appropriate amount; look closely at that check and you'll see that it has the entire amount spelled out in English. For example, if you got a cashier's check for $1,200.60, the check would read One thousand two hundred dollars and sixty cents. This is not an unusual use for a software program, and the ability to translate numbers to words can be applied in many different types of applications, from education to finance.

The design of the code system that performs this kind of translation is very interesting. The process for this is always done the same way. You break the number — no matter how large — down into hundreds, and then parse the results into English. For example, if you are given the number 123,456, you look first at the 123 and append a thousand to it — resulting in one hundred twenty-three thousand. Further, within a block of hundreds, you will always look at numbers from one to twenty, then multiples of ten, then multiples of a hundred. For example, you will list the numbers from one to nineteen for a given hundred, then it is twenty, twenty-one, thirty, thirty-one, and so forth.

A process that breaks something into smaller pieces — and then assembles the pieces into larger components — should naturally make you think about objects. In this case, you can see that there are objects for the one-to-twenty conversion, the twenty-and-up conversion, and the hundreds conversion. The thousands conversion is really just a variant of the hundreds. All these cases have a common set of things to look at:

* the specific range of the value
* convert that range into a string

Let's look at an example, because this is all rather confusing to explain and much easier to show. If you start with the number 123,456 and want to convert it to English, you would do the following:

1. First, break the number down into the highest unit, in this case thousands. The first part of our given number produces the number 123, with a unit of thousand.
2. Next, split off the hundreds. So, you have one hundred.
3. The next step is to look at the tens unit. If this number were zero, you would skip it. In this case, it is a two, so the number is twenty. An important exception here is the number one. In this case, you have to apply special English rules (i.e. eleven, twelve, thirteen) and skip the ones digit. So, because the second digit is a two, you now have one hundred twenty.
4. Finally, look at the ones digit. In our case, it is a three, so you have one hundred twenty three.
5. Append the units from step 1: one hundred twenty three thousand.
6. Repeat for the next block. If you are under a thousand, skip the units part. Put both blocks together to produce: one hundred twenty three thousand four hundred fifty six.

From an object-oriented design viewpoint, the process shows that its cases have some elements in common elements — as well as some elements that are discrete for different cases. This suggests that you have a common base class, and then derived classes that manage those discrete elements. Furthermore, you can build some of the elements from the base classes to create new extended classes, such as when you create thousands from ones and tens.

This technique shows you how to convert numbers into written English.

Always take a step back from the problem when you are trying to do an object-oriented design. Doing so gives you the opportunity to see the problem from a big-picture perspective, which often allows you to break it down into small components much more easily. When you see all the pieces, you can also usually see the overlap between them — which can be factored into your base classes.

You can save a lot of time in the long-run by getting the design right from the beginning. Understanding how all the pieces fit together is essential to getting that design right.

Your task today is to write a program that will take a numeric value currency (maximum value of 9999.99) and turn it into English written words. For example, 123.45 should give you one hundred twenty three dollars and forty five cents. Your deliverables for this project will include a detailed design specification, a flowchart, and the source code for your working project.

**## Programming Exercise 10-28-2015 - supplement**

**## checkWriter.py**

**## Prints numbers as words. Max value is 9999**

#define conversion dictionary

converter = {0:'zero', 1:'one', 2:'two', 3:'three', 4:'four', 5:'five', 6:'six',

7:'seven', 8:'eight', 9:'nine'}

#get user input

number = input("Enter an integer (1 - 9999): ")

#break number apart

thousands = number / 1000

hundreds = number / 100 % 10

tens = number / 10 % 10

units = number % 10

print converter[thousands], "thousand", converter[hundreds], "hundred",

if tens == 9:

print "ninety", converter[units]

elif tens == 8:

print "eighty", converter[units]

elif tens == 7:

print "seventy", converter[units]

elif tens == 6:

print "sixty", converter[units]

elif tens == 5:

print "fifty", converter[units]

elif tens == 4:

print "forty", converter[units]

elif tens == 3:

print "thirty", converter[units]

elif tens == 2:

print "twenty", converter[units]

elif tens == 0:

print converter[units]

#teens must be handled separately

else:

numTeens = number % 100

if numTeens == 10:

print 'ten'

elif numTeens == 11:

print 'eleven'

elif numTeens == 12:

print 'twelve'

elif numTeens == 13:

print 'thirteen'

elif numTeens == 14:

print 'fourteen'

elif numTeens == 15:

print 'fifteen'

elif numTeens == 16:

print 'sixteen'

elif numTeens == 17:

print 'seventeen'

elif numTeens == 18:

print 'eighteen'

else:

print 'nineteen'