Anglicizing integers

To the right is the specification and header of the function *ang*. We have written a return statement so the function compiles, and it returns *n* as a String, though not anglicized. As examples, we show the results of a call *ang*(1), *ang*(37), *ang*(450), and *ang*(300206):

```
/** Anglicize n.
  * Precondition: 0 < n < 1,000,000 */
public static String ang(int n) {
   return "" + n;
}
```

ang(1) is "one"
ang(37) is "thirty seven"
ang(450) is "four hundred fifty"
ang(300206) is "three hundred thousand two hundred six"

We write a Junit testing procedure before writing the method, putting in all the test cases we think we will need. We have elided some to help make it clear what the testing procedure does.

Processing small n

We are ready to think about writing the body of function *ang*. For small numbers 1, 2, 3, … of *n*, there seems to be no way to "calculate" the answer. If *n* is 1, return "one", if *n* is 2, return "two", and so on. Two questions for you: (1) what code would you write to take care of these small *n*, and (2) for what small *n* would you do this? Stop the video and answer these questions for yourself.

We answer question (2) first. Start by processing the integers in 1..19. Each has to be handled individually. There is no algorithm to calculate one from another.

For question (1), there is an obvious way to do it: Write a series of 19 if-statements. A better way involves using a static array of size 20, each element *k* being the word for integer *k*:

```
if (n == 1) return "one";
if (n == 2) return "two";
…
if (n == 19) return "nineteen";
```

```
private final static String[] ang19= new String[]{"", "one", "two",
"three", "four", "five", "six", "seven", "eight", "nine", "ten", "eleven",
"twelve", "thirteen", "fourteen", "fifteen", "sixteen", "seventeen", "eighteen", "nineteen"};
```

Then, write the body of the method as shown to the right. We insert the comment "20 <= *n*" to help us remember what cases still have to be handled.

```
if (n < 20) return ang19[n];
// 20 <= n
return "" + n;
```

Now, the first 19 test cases in our Junit testing class should work. Note that *all* 19 test cases are *necessary*! Having all 19 is the only way to make sure we spelled the words "one", …, "nineteen" correctly in the declaration of *ang19*.

Processing the second range of integers

We handled the case *n* <= 19. Here is the next question for you: What is the next range of integers to process? Stop the video and figure it out.

The next range of integers to process is 20..99. Anything larger requires the word "hundred".

We have to split any value *n* in 20..99 into two pieces. For example, for *n* = 23, we need to extract the 2 and the 3 and replace them by "twenty" and "three". We can extract them using *n*/10 = 2 and *n*%10 = 3. Thus, not worrying about anglicizing yet, we write the statement to handle these integers like this:

```
if (n < 100) return (n/10) + " " + (n%10);
```

We should test at this point to make sure this extracts the two digits properly, but we don't do that here.

How do we anglicize *n*/10 and *n*%10? Since *n*%10 < 10, we anglicize *n*%10 using array *ang19*:

```
if (n < 100) return (n/10) + " " + ang19[n%10];
```

For *n*/10 = 2, we want the value "twenty", for *n*/10 = 3, "thirty", etc. So, we introduce a new static array:

```
/** For i in 2..9, tenW[i] is the "tens word" for i, e.g. ten[2] is twenty. */
private final static String[] tenW= {"", "", "twenty", "thirty", "forty",  "fifty", "sixty",
                                     "seventy", "eighty", "ninety"};
```

and rewrite the return statement:

if (n < 100) **return** *tenW*[n/10] + " " + *ang19*[*n*%10];

But running the JUnit testing class shows an error. Can you find it? Stop the video and think about it.

Suppose *n* = 20. Then the returned String should be "twenty" but it is "twenty " —the blank at the end does not belong there. So we rewrite the return statement using a conditional expression.

It is most important that we tested before moving on to consider the cases 100 ≤ *n*. The same kind of error would have happened in later code, too. Make alternating programming and testing a habit.

```
if (n < 20) return ang19[n];

// 20 <= n
if (n < 100)
   return tenW[n/10] +
          (n%10 == 0 ? "" : " " + ang19[n%10]);

// 100 <= n
return "" + n;
```

Processing the third range of integers

The next range of numbers to process is 100..999. The string for each of them will have the text "hundred" in it.

For example, for *n* = 345, the result is "three hundred forty five". From 345, we have to extract the 3 and the 45. We can do this using *n*/100 and *n*%100. Taking a cue from the previous development, and not yet anglicizing, we write the statement

if (*n* < 1000) **return** (*n*/100) + " hundred" + (*n*%100 == 0 ? "" : " " + (*n*%100))

Since *n* < 1000, the value of *n*/100 is less than 10, it can be anglicized using array *ang19*. The value of *n*%100 is less than 100 —how are we going to anglicize it? Stop the video and look around for a function that can be used.

Yes, to anglicize *n*%100, use a recursive call on the function being written, *ang*! Recursion is the natural tool to use here.

if (*n* < 1000) **return** (*n*/100) + " hundred" + (n%100 == 0 ? "" : " " + *ang*(n%100))

Since *n* < 1000, the value of n/100 is less than 10, so it can be anglicized using array *ang19*.

if (*n* < 1000) **return** ang[*n*/100] + " hundred" + (n%100 == 0 ? "" : " " + *ang*(n%100))

Processing the last range of integers

The last range to process is 1000..999,999. The resulting string will contain the word "thousand". This case is quite similar to the previous case of *n* in the range 100..999, so we handle it by copy/paste/edit.

We call *ang* recursively to anglicize *n*/1000. Then comes the word "thousand" with a blank before it. The conditional expression gives either the empty string or a blank followed by *n*%1000, anglicized of course.

That completes the development of the function to anglicize integers. Interestingly enough, it contains no assignment statements.

Important points of the development

Important points in the development were:

1. Handling one case at a time, starting with the smallest range.
2. Testing after coding the processing of each range.
3. Breaking the development of each case into two steps:
   a. First, figure out the pieces, e.g. *n*/100, " hundred" and the conditional expression for *n*%100.
   b. Second, figure out how to anglicize each piece; here, use array *ang19* for *n*/100 and function *ang* for *n*%100.

Anglicizing integers

```
public void testAng() {
    assertEquals("one", Ang.ang(1));
    assertEquals("two", Ang.ang(2));
    assertEquals("three", Ang.ang(3));
    assertEquals("four", Ang.ang(4));
    …
    assertEquals("eighteen", Ang.ang(18));
    assertEquals("nineteen", Ang.ang(19));

    assertEquals("twenty", Ang.ang(20));
    assertEquals("thirty", Ang.ang(30));
    …
    assertEquals("eighty", Ang.ang(80));
    assertEquals("ninety", Ang.ang(90));

    assertEquals("thirty five", Ang.ang(35));
    assertEquals("twenty three", Ang.ang(23));

    assertEquals("two hundred", Ang.ang(200));
    assertEquals("five hundred thirty two", Ang.ang(532));

    assertEquals("two thousand", Ang.ang(2000));
    assertEquals("two thousand one", Ang.ang(2001));
    assertEquals("six thousand fifty", Ang.ang(6050));
    assertEquals("nine hundred ninety nine thousand" +
          " nine hundred ninety nine", Ang.ang(999999));
```

Anglicizing integers

1.