Step by step development of project code

Contributors: Mihai Pop

last modified: 03/19/16

Introduction

Now that you've had to struggle with the code in the first projects in the class, it's time for me to show you one strategy for approaching such projects.

Cipher project

The first task required you to implement Caesar's cipher – given a string of text and an integer you had to 'shift' each character in the text by the integer provided to you wrapping around the beginning/end of the alphabet as necessary.

Starting this project, a first question I would have is exactly how the shifting works, and how I can make sure to wrap around the alphabet. This seems to be a key part of the project, and sorting it out first would make life easier.

One useful strategy in programming is to isolate common operations that you are likely to use again as separate methods, so we'll start by doing so here.

First, to make things more formal we can write out in English what this method should do: "*I want a method that takes a character and a number and returns the character shifted in the alphabet by the number of positions provided, wrapping around if necessary*".

This description alone gives us hints about the prototype of the method – it will return a character, and take as parameters a character and a number, yielding the code:

```
public static char shiftChar(char c, int n)
{
}
```

Writing this as a method also allows us to test it more easily through the JUnit framework, making sure that when we write the cipher code the method will work correctly. We can actually implement a few tests already without even writing the method:

```
assertEquals('A', shiftChar('A', 0)); // shifting by 0 we stay the same
assertEquals('E', shiftChar('A', 4)); // E is 4 away from A
```

It's also useful at this point to figure out whether we want to accept negative numbers as well. A negative shift would be useful for decoding a cipher, thus they make sense as well, and we can write a few such tests as well:

```
assertEquals('C', shiftChar('D', -1)); // C is one back from D
assertEquals('L', shiftChar('P', -4)); // L is 4 characters before P
```

And of course, the special cases where we wrap around the alphabet:

```
assertEquals('A', shiftChar('Z', 1));  // A follows Z if we wrap around
assertEquals('Y', shiftChar('B', -3)); // going back from B three spots we
                                       //end up on Y
```

Now when we implement the method we'll be able to make sure we did so correctly.

Coming back to the method, we can implement the most simple version to make sure we know how to add characters to integers:

```
public static char shiftChar(char c, int n)
{
      char result;

      result = (char)(c + n); // note the cast to force narrowing from int
                                    // to char
      return result;
}
```

We could write this in even a more compact way:

```
public static char shiftChar(char c, int n)
{
      return (char)(c + n);
}
```

We can already test this simple method by running the JUnit tests we created and should be able to see that we pass all tests with the exception of the ones wrapping around the ends of the alphabet. Time to sort out that logic.

Let's look at the JUnit tests that we did – how can we work out the code we'll write. Let's think for a second: Z is the 26th character in the alphabet. Adding 1 to it, we want to come to the 1st character in the alphabet, or A. In other words, after we add 1 to Z we have to subtract 26 from it to get the right character. In code:

```
if (result > 'Z')
  result -= 26;
```

Just on paper we can try this out for some other letters. Take X and shift by 8: Y, Z, A, B, C, D, E, F. X + 8 = 'Z' + 5, 'Z' + 5 – 26 = 'F'. Ok, this should work. We could even add a new test at this point:

```
assertEquals('F', shiftChar('X', 8));
```

How about going backwards, - again start with the original JUnit test case: 'B' – 3 = 'Y' – or starting from the 2nd letter in the alphabet and going backwards 3 positions we need to end at the 25th position in the alphabet. Or, 2 – 3 + what? = 25 – the shift must now be +26. Again in code:

```
if (result < 'A')
  result += 26;
```

The final code is:

```
public static char shiftChar(char c, int n)
{
      char result;

      result = (char)(c + n); // note the cast to force narrowing from int
                                    // to char
      if (result > 'Z')
            result -= 26;
```

```
      if (result < 'A')
            result += 26;
      return result;
}
```

With a bit of work we could make this a bit more compact, but this version of the code is quite legible so we can stick with it.

Now we quickly check whether it passes the tests, and it should.

STOP AND THINK.

Will the code we just created pass the following test?

```
assertEquals('F', shiftChar('F', 26));
```

How about?

```
assertEquals('A', shiftChar('X', 29));
```

And how about?

```
assertEquals('C', shiftChar('C', 78));
```

The results of the tests above should indicate that the code doesn't quite behave correctly if we try to shift by more than 26 characters. In other words, if we loop through the alphabet more than once, the code simply doesn't work.

But how do we decide what to do in such situations? First, it's easy to see that if the shift is exactly 26 characters, we don't have to do anything, we simply loop back to the same position, i.e. a shift of 26 is the same as a shift of 0. This holds for any shift that is a multiple of 26 as it doesn't matter how many times we come back to the same spot. But how about a shift of 27? Since we can write it as 26 + 1, the 26 just brings us back where we started and the extra 1 makes us go one character further, i.e., a shift of 27 is the same as a shift of 1. We can more generally claim that a shift by a number that can be written as 26 * a + b is the same as a shift of b. Or in other words, the actual shift we have to do is simply the remainder of the division of the original number by 26, or `n % 26`.

We can now update our code to make sure long shifts do not confuse us:

```
public static char shiftChar(char c, int n)
{
      char result;

      n = n % 26; // make sure large shifts don't confuse the code

      result = (char)(c + n); // note the cast to force narrowing from int
                                    // to char
      if (result > 'Z')
            result -= 26;
      if (result < 'A')
            result += 26;
      return result;
}
```

A quick test should tell us that now our code passes all the JUnit tests we have created, thus all is OK.

As an aside, the example above also shows you why Java has ensured that the modulo operator

returns a negative value when provided a negative value (the property that confused our isOdd method). Our code works correctly even for negative shifts.

Time to return to the actual cipher code. The Caesar cipher can simply be stated in English: *"For each character in the string, shift the character by the number of positions indicated by the key"*. Also remember that you were given the starter code:

```
public static String encryptCesar(String text, int key)
{
      String output = "";

      return output
}
```

We now need to simply translate the English into code, and we can do that with the all too familiar for loop:

```
public static String encryptCesar(String text, int key)
{
      String output = "";

      for (int i = 0; i < text.length(); i++){
            output = output + shiftChar(text.charAt(i), key);
      }

      return output
}
```

Note that we've used the method we have just created. Since we have made this method work for both positive and negative numbers, we can simply apply it with a negative value to implement the `decryptCesar` method.

And there it is – the first assignment in just a few lines of code.

The Vigenère cipher adds one complication – at each position in the key we use a different shift, which is computed from the actual text of the key. If we look at just one position we can easily figure out the key – the shift implied by character `c` is simply the number of characters between `'A'` and `c`, or `c - 'A'`. Once we know the number of characters to shift by, the same `shiftChar` method can be used:

```
output = output + shiftChar(text.charAt(i), key.charAt(j) - 'A');
```

In the code above we simply indicated that we need the key at position `j` when we are inspecting the text at position `i`, but what is the relationship between the two? To figure it out we can write down a little example:

```
           111
i 123456789012
  ATTACKATDAWN

  KEYKEYKEYKEY
j 123123123123
```

The numbers above the text and below the key are the values taken by the variables `i` and `j` as we execute the code. Thinking about their relationship a little bit, you can observe that `j = i % key.length()`.

Using this observation we can write the Vigenère code as:

```
public static String encryptVigenere(String text, String key)
{
      String output = "";

      for (int i = 0; i < text.length(); i++){
            output = output
                     + shiftChar(text.charAt(i),
                                 key.charAt(i % key.length() - 'A');
      }

      return output
}
```

And that's all.

Maritime flag project

The maritime flag project required you to think in two dimensions. For strings, the English description *"for each character in the text do something"* translated into the code

```
for (int i = 0; i < text.length(); i++){
      //do something
}
```

Exploring the grid forming a flag can be expressed in English as: "*for each row and for each column, do something with the pixel at coordinates (row, column)"*, which can be translated into code with a double for loop:

```
for (int row = 0; row < grid.getHt(); row++){
  for (int col = 0; col < grid.getWd(); col++) {
    // do something at pixel (row, col)
  }
}
```

To fill just one part of the grid, as I suggested for the fillRectangle method, you simply adjust the start and end of the for loops:

```
for (int row = rowT; row < rowB; row++){
  for (int col = colL; col < colR; col++) {
    // do something at pixel (row, col)
  }
}
```

visits just the rectangle contained between the top and bottom rows `rowT` and `rowB`, and the left and right columns `colL` and `colR`.

Using the method `fillRectangle(rowT, rowB, colL, colR, color)` we can now easily build several flags by specifying the right boundaries for the rectangle.

Quebec is simply a yellow flag and can be painted with:

```
fillRectangle(0, grid.getHt(), 0, grid.getWd(), Color.yellow);
```

The Echo flag (blue top and red bottom) can be painted with:

```
fillRectangle(0, grid.getHt() / 2, 0, grid.getWd(), Color.blue);
     // from top to middle
fillRectangle(grid.getHt()/2, grid.getHt(), 0, grid.getWd(), Color.red);
     // from middle to bottom
```

And so on.

The Oscar flag, diagonally split with red and yellow, cannot be made out of rectangles, rather opens up the opportunity to think about painting flags in a slightly different way. Returning to the general access pattern for the grid:

```
for (int row = 0; row < grid.getHt(); row++){
  for (int col = 0; col < grid.getWd(); col++) {
    // do something at pixel (row, col)
  }
}
```

we can change what we do at each pixel based on a conditional statement checking the relationship between the row and the column. The diagonal is simply the place in the grid where the row and the column have the same value. Coloring the diagonal red can be easily done with the code:

```
for (int row = 0; row < grid.getHt(); row++){
  for (int col = 0; col < grid.getWd(); col++) {
    if (row == col)
      grid.setColor(row, col, Color.red);
  }
}
```

The pixels above the diagonal simply have the value of the row lower than that of the column and vice versa, yielding to the code for the Oscar flag:

```
for (int row = 0; row < grid.getHt(); row++){
  for (int col = 0; col < grid.getWd(); col++) {
    if (row <= col)
      grid.setColor(row, col, Color.red); // top right is red
    else
      grid.setColor(row, col, Color.yellow); // bottom left is yellow
  }
}
```

We can use the same type of logic to pain the Lima flag that has four square blocks, top left and bottom right are yellow, and bottom left and top right are black. Again, we have to decide how the actual values of the row and col variables relates to color. The top left quadrant simply has both the row and column coordinates less than the half the size of the flag, and it can be filled out yellow:

```
for (int row = 0; row < grid.getHt(); row++){
  for (int col = 0; col < grid.getWd(); col++) {
    if (row < grid.getWHt() / 2
        && col < grid.getWd() / 2)
      grid.setColor(row, col, Color.yellow); // top left quadrant

  }
}
```

Similarly we can add further if () conditions to make all the quadrants.

We could also think about this flag in a slightly different way. If both the row and the column are in the same 'half' of the flag the color is yellow, otherwise black. But how do we write in code a

statement such as 'row is in the first half of the flag'? Let's think a bit about dividing the value of the row by half the height of the flag. The result of the integer division will be 0 if the row position is less than half the height, or 1 otherwise. In other words, the result of the integer division:

```
row / (grid.getHt() / 2) // STOP AND THINK : why are the parentheses necessary?
```

also equivalently written as:

```
2 * row / grid.getHt()
```

tells us which half we are in. We can then write the whole flag as:

```
for (int row = 0; row < grid.getHt(); row++){
  for (int col = 0; col < grid.getWd(); col++) {
    if (2 * row / grid.getHt() ==
        2 * col / grid.getWd() ) // row and column in the same half
      grid.setColor(row, col, Color.yellow);
    else
      grid.setColor(row, col, Color.black);
  }
}
```

Finally, we can use the same idea to quickly knock out the November flag – a 4x4 grid with alternating blue and white squares.

To make the grid 4x4 we simply divide not by `grid.getHt() / 2`, but by `grid.getHt() / 4`. The result of the integer division will now be a number between 0 and 3, indicating which quarter of the flag we are within.

To decide how to use this information to paint the squares, we simply think about the positions of the blue squares: (0, 0), (1, 1), (2, 2), (3, 3), (0, 2), (2, 0), (0, 4), (4, 0), ... and so on. A pattern emerges. Either the coordinates are the same, or they are both even numbers. We can, thus, write the if statement:

```
if (4 * row / grid.getHt() == 4 * col / grid.getWd()  // diagonal
    || (4 * row / grid.getHt()) % 2 == 0        // row position is even
      && (4 * col / grid.getWd()) % 2 == 0)     // col position is even
    grid.setColor(row, col, Color.blue());
else
    grid.setColor(row, col, Color.white());
```

And that's all for the November flag.

STOP AND THINK. How would you write the JUnit tests for the flag project?