CSC1021 Programming I

Project 1

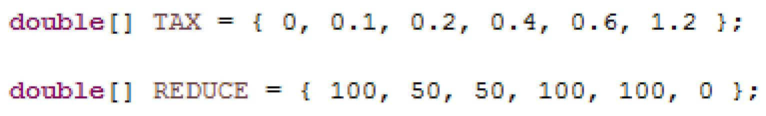
Olivija Guzelyte

November 9th, 2016

Part 1

Writing the code

To begin with, the cleanness or reusability of the code unconsidered, I have done the *TaxCalculator* class's calculations mainly using *If* statements, that eventually (due to complex instructions) became nested *If* statements. Needless to say, the style of the code had to be improved, thus I introduced the *for* statement and decided to create two floating-point arrays of constants. One of them marked the taxation of each income range and the other represented the differences of income ranges: 100, 50, 50, 100, 100 and 0, respectively. This is how much was needed to be reduced each time a loop cycle was completed.



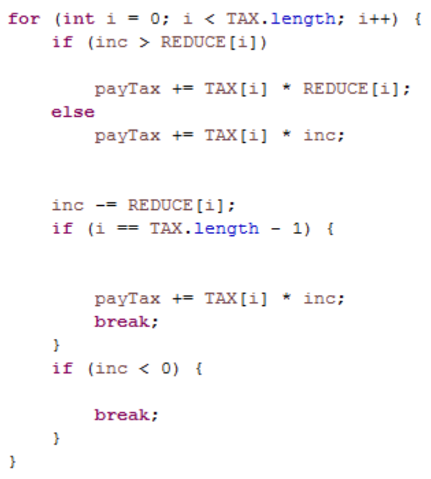
```
double[] TAX = { 0, 0.1, 0.2, 0.4, 0.6, 1.2 };

double[] REDUCE = { 100, 50, 50, 100, 100, 0 };
```

1. Arrays of constants.

The *for* loop was made to execute until the end of the *TAX* array. Inside of it is an *If* statement that is made to check if the passed to the function (and later decremented) child's income is bigger than the value in the *REDUCE* array. This is done to multiply the relevant amount of money by the tax constant corresponding to that particular income range. Otherwise the whole income, which might be a colossal number, could be multiplied by a false tax range. The *If* is followed by an *Else* statement, which instead applies the aforementioned instructions. Since, in this case, the income is smaller than the reduction amount, the remaining income gets used up and multiplied by the relative tax constant. It is assumed that the loop will then be terminated (because the income will become a negative value).

If the loop goes on until the end, in order not to go out of the boundaries of *TAX* and *REDUCE* arrays, an *If* statement is inserted, which uses up the remainder of income with the last taxation element. Therefore, the loop doesn't crash and is prone to work given any income.



```
for (int i = 0; i < TAX.length; i++) {
    if (inc > REDUCE[i])

        payTax += TAX[i] * REDUCE[i];
    else
        payTax += TAX[i] * inc;


    inc -= REDUCE[i];
    if (i == TAX.length - 1) {


        payTax += TAX[i] * inc;
        break;
    }
    if (inc < 0) {

        break;
    }
}
```

2. The main TaxCalculator calculation

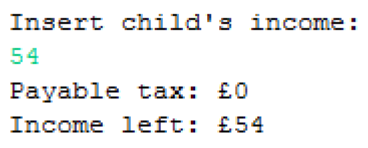
The floating-point value is then converted into an integer and fits the corresponding method. Only one statement is used in the *incomeLeft* method. That is the most efficient use of code, since instead of copy pasting the code from *payableTax*, one call to it is made and an integer value of decremented income is returned.

Testing

For testing purposes in the *BogOff* class I created the main method, in which an ability to input a value by the user is created. Subsequently, the two aforementioned methods (*payableTax* and *incomeLeft*) are both called and assigned to different variables and printed out in the console. The testing needs to be done for each of the income ranges. For the second range two inputs are necessary to clearly demonstrate that the program does not count any tax for the values 101-104 (104 included), because it gets rounded to the nearest pound. Therefore, the testing inputs are: 54, 104, 105, 171, 204, 310 and 435.

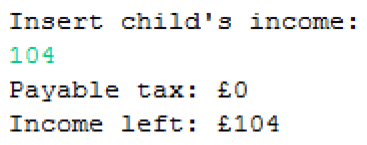
Respective taxes and net incomes for the inputs:

54, 54 * 100 = 0, rounded = 0



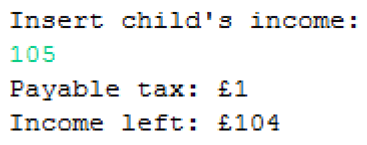
```
Insert child's income:
54
Payable tax: £0
Income left: £54
```

104, 100 * 0 + 4 * 0.1 = 0.4, rounded = 0



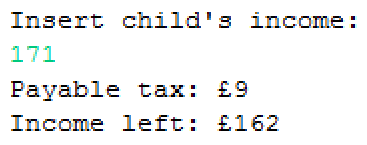
```
Insert child's income:
104
Payable tax: £0
Income left: £104
```

105, 100 * 0 + 5 * 0.1 = 0.5, rounded = 1



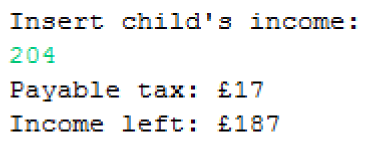
```
Insert child's income:
105
Payable tax: £1
Income left: £104
```

171, 100 * 0 + 50 * 0.1 + 21 * 0.2 = 9.2, rounded = 9



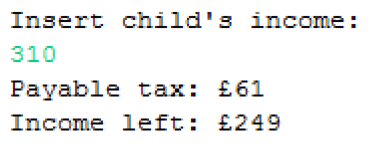
```
Insert child's income:
171
Payable tax: £9
Income left: £162
```

204, 100 * 0 + 50 * 0.1 + 50 * 0.2 + 4 * 0.4 = 16.6, rounded = 17



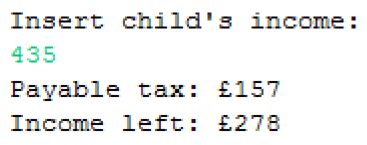
```
Insert child's income:
204
Payable tax: £17
Income left: £187
```

310, 100 * 0 + 50 * 0.1 + 50 * 0.2 + 100 * 0.4 + 10 * 0.6 = 61, rounded = 61



```
Insert child's income:
310
Payable tax: £61
Income left: £249
```

435, 100 * 0 + 50 * 0.1 + 50 * 0.2 + 100 * 0.4 + 100 * 0.6 + 35 * 1.2 = 157, rounded = 157



```
Insert child's income:
435
Payable tax: £157
Income left: £278
```

Part 2

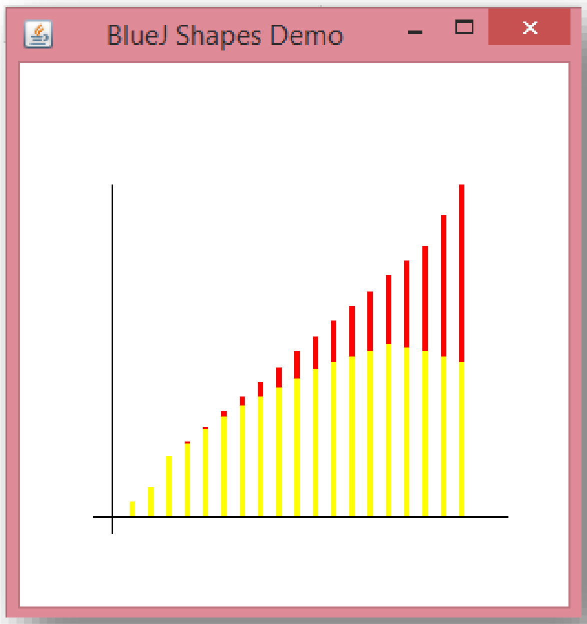
Writing the code and testing

The constructor, as a blueprint, provides the income array to the whole *TaxChart* class. All the public methods in that class see the array, therefore, the *draw* method can access the *income* array by a mere reference. Constants, such as: *WIDTH* and *SHRINK*, are created in the draw method. Their purpose is to change the width of the bars and the size of the overall table depending on preference. The *b, maxY* and *maxX* variables prevent bars from overlapping, determine the height of Y axis and the width of the X axis, respectively.

In order to draw the bars, I created a *for* loop, which would execute until the last element of *income* array. Subsequently, a bar for taxes is created. Using the *changeSize* method its height is a call to the *payableTax* method and shrunk 3 times to fit into the *BlueJ Shapes Demo* screen. The same logic is applicable to the net income bar, except that the call is made to the *incomeLeft method*. Since the classes for bars are written in such way, that the drawn shapes are upside down, inverting them is necessary. Therefore, both bars are flipped using the *moveVertical* method with their height values and a minus sign in front. A few positioning adjustments are made and both bars are situated on top of each other.

In order to position the axes in accordance to the height and width of the bars, using an if statement, the highest bar is found and thus becomes the height of the Y axis. The width of X axis is also calculated counting the gaps in-between the bars and the width of the bars, so that they interact with the input.

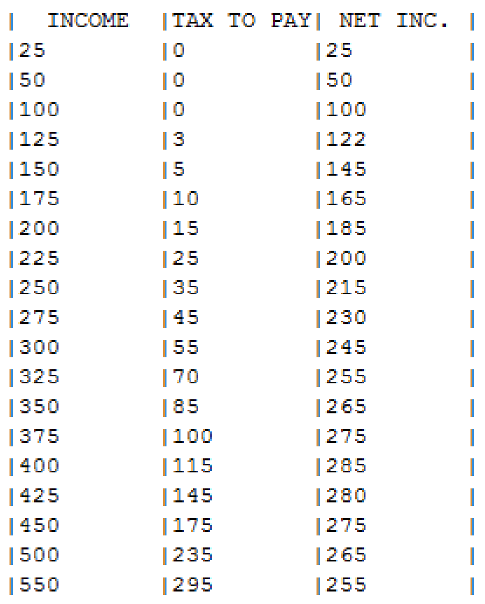
The drawing to the requested input:



Income array: {25, 50, 100, 125, 150, 175, 200, 225, 250, 275, 300, 325, 350, 375, 400, 425, 450, 500, 550}.

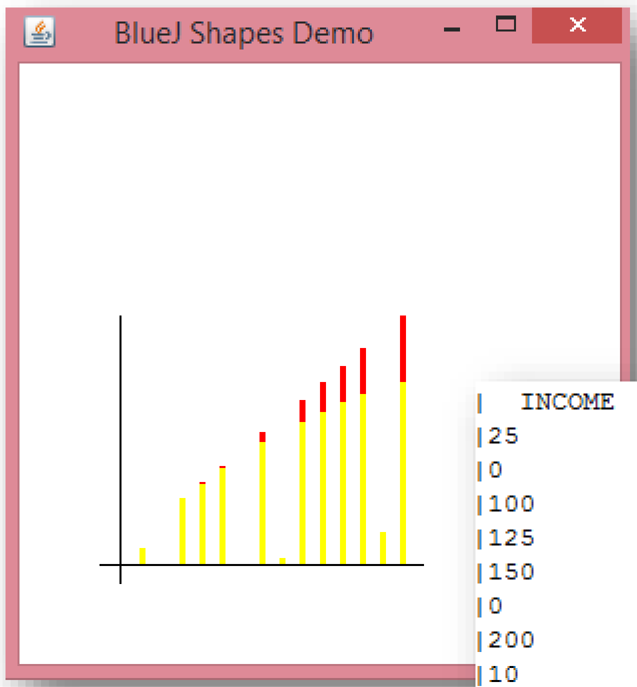
The values of the income array are passed to the constructor using the *BogOff* class, in which a specific method, called *taxTables* is created. For this purpose, a new *TaxChart* object is created and it calls upon the *method* draw.

However, a new method, called *printTable* is created in the *TaxChart* class. Using the main method and the income values in the BogOff class a table of 3 columns, each of them 10 spaces wide, left-aligned, then gets printed off on the Console panel.

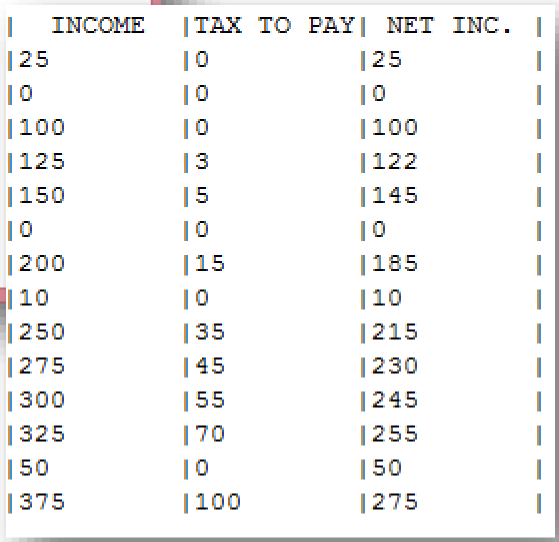


```
|  INCOME   |TAX TO PAY| NET INC. |
|25        |0         |25        |
|50        |0         |50        |
|100       |0         |100       |
|125       |3         |122       |
|150       |5         |145       |
|175       |10        |165       |
|200       |15        |185       |
|225       |25        |200       |
|250       |35        |215       |
|275       |45        |230       |
|300       |55        |245       |
|325       |70        |255       |
|350       |85        |265       |
|375       |100       |275       |
|400       |115       |285       |
|425       |145       |280       |
|450       |175       |275       |
|500       |235       |265       |
|550       |295       |255       |
```

A smaller array with zero incomes can be chosen in order to check the flexibility of the chart and table. This time the income array is: {25, 0, 100, 125, 150, 0, 200, 10, 250, 275, 300, 325, 50, 375}.



The height of the Y axis and the width of the X axis got reduced, therefore, it is visible that they are working in correspondence to the given array. Also there are gaps instead of bars where the zero income is given, meaning that the calculations and calls to *TaxCalculator's* methods are working as well. The table responds to the new values as well.



```
|  INCOME   |TAX TO PAY| NET INC. |
|25        |0         |25        |
|0         |0         |0         |
|100       |0         |100       |
|125       |3         |122       |
|150       |5         |145       |
|0         |0         |0         |
|200       |15        |185       |
|10        |0         |10        |
|250       |35        |215       |
|275       |45        |230       |
|300       |55        |245       |
|325       |70        |255       |
|50        |0         |50        |
|375       |100       |275       |
```