30.11.2015

Nils Gawlik

Lab Report for the 24th November 2015

Exercise 1:

I tested my P1 solution and it gave the right result.

For easier testing of P2 I made a new BlueJ project. In there I created the sumBetween method and typed my code from the pre lab into it.

```
public static int sumBetween(int a, int b)
{
    return (a+b-1)+(b-a)/2;
}
```

To test the sumBetween, I calculated sumBetween(1, 4), sumBetween(9,13) and a few others on a calculator. I then used *assert* to create some test cases so I see if the results given by the functions were the same as the ones I calculated.

Exercise 2:

First off I added the listNotes method which does not have any arguments and also no return value. It iterates through the notes variable using a for-each loop storing the individual notes into the variable note and prints out the content of note in every iteration of the loop using System.out.println().

Exercise 3:

I declare a variable "number" and initialize it as 0 before loop. I print the variable by adding it to the note variable from the left; I also increment the variable just before printing it using the ++ operator after the variable. It should be noted that it is important, that I write number++, not ++number or else the list would start at 1 because it would be increased before being accessed, now it is the other way around, it starts at 0. I also added a " - " string in between – it looks nicer.

Exercise 4:

I first implemented the using a for-loop (I noticed later, that the assignment says while-loop, but I figured that in the context of this task the idea is the same).

In the first part of the for loop the variable i is created and set to 0. The condition for the for loop is if i is smaller than numberOfNotes(). The third part of the for-loop increases i by one. Inside the loop I check if the string given is contained is the note I get by calling notes.get(i) using .contains(...) - in short notes.get(i).contains(searchStr).

If the string is found I remove the element with the index i and also decrease i by one so I don't skip the next element, because the size of the list just changed.

Everything worked splendidly.

30.11.2015

Nils Gawlik

The second implementation uses an iterator. I get the iterator (I will call it "it") by calling notes.iterator(). I can now use a while loop, which uses the iterator's hasNext() method as a running condition, to loop through all the notes. I get a note by calling it.next() and use .contains(...) on it to test if it has to be removed. I use it.remove() to remove the note.

This also worked without a problem.

I did not run into any problems doing these tasks, but I can see that the potential for programming mistakes is higher with the indexed version. If you forget to decrease the index variable, your code will still mostly work, but not remove two consecutive elements, which is very odd behaviour if you do not know what is going on.

Exercise 5:

To test if an element was removed I added a boolean variable removedSomething initialized with false. If the string is found and the if clause is entered this variable is set to true. At the end of the method I check for the variable and if it is still false I print out an error message saying that the note was not found.

Exercise 6:

This task confused me at first, because this problem does, to my knowledge, not exist when using the iterator. So I already fixed it, right?

But if you were still using the while- or for-loop with an index, this exercise makes more sense. I still had the code, I commented it out at some point. So I put it back in place.

To reduce the number of times numberOfNotes() (which is equivalent to notes.size()) is called to one, I save the result of numberOfNotes() in the variable n at the beginning of the method and then use this variable instead of the numberOfNotes() method. When a note gets removed I decrease n by one (makes sense doesn't it?).

It worked. Halleluia.

(In exercise 8 I shortly talk about the advantages of looping backwards, which would have been an alternate solution for this exercise)

30.11.2015

Nils Gawlik

Exercise 7:

This is my isPrime pre lab solution.

```
public static boolean isPrime(int num)
{
    if(num < 2)
        return false;

    int i = 2;
    while(i < num)
    {
        if((num % i++) == 0)
            return false;
    }
    return true;
}
```

I acknowledge that this is a very slow way of finding primes in general, but it uses hardly any memory compared to other methods I know, so it has that going for it, which is nice.

I actually ran into a very weird error while doing this, where no numbers would evaluate to primes. I never really found out what the mistake was, as rewriting the code without really changing anything fixed it. So I am assuming it was some small but deadly error like a missing ++ or some mistake with how I tested.

The actual task was not as difficult. I created a test method and in there I declared a variable primes of type ArrayList<Integer> initialized with a new ArrayList<Integer>().

I made a for-loop which counts from 1 to 1000 and in there added a number if isPrime evaluates to true.

Printing out the size of the list gave 168, so there are 168 primes between 1 and 1000.

Exercise 8:

If you read the code (or this report) you will notice that I did not create any methods like move(int numberOfSteps) or printRow(), scanRow(), etc.. It would have maybe been considered good style, but for such a small exercise with very readable code, I did not want to put in the extra effort. It would have also fragmented the code and possibly made it less readable, actually.

I figured that every call of the act method means copying one line. On top of that the comments describe the rough algorithm to be used:

1) walk left & scan

30.11.2015

Nils Gawlik

2) walk to middle
3) walk right & print
4) walk back to the middle
5) move up one line

to move left I used a while loop checking if there is no tree in front. Inside the loop I move a step and append the result of onLeaf() to the list.

After the while loop (Kara has reached the trees) I let her turn around using turnLeft twice. I then use a for loop to call the move method "leafRow.size()" times, effectively walking as many steps back, as I initially walked until I hit the tree.

I could have also archieved this by using a variable, but I generally avoid using extra variables if I can, as they create more possible program states, which can get confusing, for example while debugging.

Now that I'm in the middle again I use a for-loop to loop through the list backwards. In every iteration of this loop I move and then put down a leaf if the current list item is true. It has to be backwards, so that the image is not inverted (because I am going the right instead of the left now).

I remember that I used to always loop through for loops backwards, basically until they told me in school that you are supposed to do it the other way around. It is weird, because looping backwards gives you the advantage that you only have to access the upper limit once (setting i to it), which is faster in a lot of cases. I now realize that this would have been a funny solution to exercise 6.

Anyway Kara is now at the very right of the screen, so I let her turn around and let her walk backwards using the same methods as before (turnLeft and for loop).

Back in the middle Kara now turns right. If there is a tree in front of her I call Greenfoot.stop(), otherwise she moves up and is ready for the next call of the act method to copy another row.

30.11.2015

Nils Gawlik

Exercise 9:

I was a little bit surprised about the difficulty increase here (provided I understood the task correctly), but in the end it was not as bad as I expected.

I figure my code is going to be harder to explain than to write, so I will actually put it right here to refer to it. I also went through and commented a lot of it. Alternatively I could have also divided it into different methods to make it clearer, but I feel like having it all in one place.

```
private boolean match(String text, String expression)
{
    int pos = 0;//position in the string we are searching through
    int exprPos = 0;//position in the expression (string to search for)

    //loop through the string
    while(pos <  text.length())
    {
        //Skip all stars, because they are not evaluated as characters,
but seen as fall-back-points
        while(expression.charAt(exprPos) == '*')
            exprPos++;

        //check if we've succeeded already
        if(exprPos >= expression.length())
            return true;

        //see if the current char in our string matches the current
char in the expression
        if(text.charAt(pos) == expression.charAt(exprPos) ||
expression.charAt(exprPos) == '?')
        {
            //Nice! it (still) lines up, so keep going
            exprPos++;
            //check if we've succeeded already
            if(exprPos >= expression.length())
                return true;
        }
        else
            //Dang! the expression and the string don't line up. Maybe
there's a star we can fall back to.
            while(expression.charAt(exprPos) != '*' && exprPos > 0)
                exprPos--;

        pos++;
    }

    //we didn't succeed, which means we failed
    return false;
}
```

30.11.2015

Nils Gawlik

If you read through the code you can probably understand a lot of it already, so I am not going through all of the variable definitions, loops, if-clauses, ... like I did in other exercises, but instead focus on the basic idea of the algorithm.

The idea is that you have two counters, both representing a position. “pos” represents the position in the string that you search through, “exprPos” the position in the string that you search for (let us forget about the ?'s and *'s for now).

The program loops through the text until it sees two letters matching (at some position in the text and at the first position of the search string). From then on it goes through both strings to see if the following letters are matching as well. If that is not the case it sets the position in the search string back to zero and the search starts over again (from this position). If the counter ever reaches the end of the search string that means that all characters are matched and the string was found; The program returns true breaking out of the method completely. If that never happens the methods returns false.

What I just described implements the String.contains(…) method. It is still missing the ? and * functionality.

The support for ?'s was added very quickly by not only advancing the exprPos counter when the chars match, but also counting a ? in the expression as a sure match.

Finding a way to implement the * took a little more imagination. I solved it by seeing the star as a sort of fall-back-character. This means that in the case of a mismatch the exprPos counter just goes back to the last *. This last star is then seen as a replacement for all the mismatching stuff that was inbetween then and now. This process can be repeated as often as possible, until at some point the remainder of the expression gets matched.

I also had to add code to skip the *'s in the expression so they are not evaluated in the normal matching process and immediately after that I had to test if I reached the end of the expression (in the case that the expression was ending in a star), in which case I also return true.

I wrote a variety of test notes using a lot of different but similar strings and to my knowledge this algorithm is working correctly.

Exercise 10:

Well I'm probably not quite “bored” enough for this exercise. But I want to point out, that Java.lang.String.matches(String regex) exists in case I ever need it.

Monday evening edit: I actually started implementing this exercise, but I couldn't finish it. So you will find some code for the interpreter in the project; you can look at it if you want, but it's not that well commented.

Summary:

On the matter of time: I actually only completed all the regular exercises (up to 7) in the lab and did the two bonus ones in the bus, train and at home. And I didn't have any

bigger problems, so I would say, that the time was pretty limited.

I liked exercise 9 and I'd like to give 10 a try, but I did not have the time or motivation, really.

I still wouldn't say that I learned much in this lab, considering that I already have a bunch of programming experience and even exercise 9 was not that new to me. I did things pretty similar to it before, it was still a nice challenge, though.