Homework 3

1. Loan calculations

Suppose you take a loan of 100,000 ILS (Shekels) at an annual interest rate of 5%, for 10 years, with equal annual payments. How much should you pay each year, so that at the end of the 10th year the loan will be fully paid? For example, suppose that the annual payment is 10,000 ILS. In this case, your balance (the sum that you still owe) at the end of the first year will be (100,000 – 10,000) * 1.05 = 94,500 ILS. At the end of the second year, the balance will be (94,500 – 10,000) * 1.05 = 88,725 ILS. In general, if you have to make *n* equal payments, we can make *n* such calculations, and check the balance at the end of the *n*-th year. If the ending balance is positive (meaning that you still owe money), it implies that you should have paid more each year. If the ending balance is negative, it implies that you paid too much. This logic is illustrated in the following spreadsheet:

	A	B	C	D	E	F
1	Loan:	100000				
2	Interest rate:	5				
3	Periods:	10				
4	Periodical payment:	10000		Change this value		
5						
6		Ending balance				
7	Period 0	100000				
8	Period 1	94500				
9	Period 2	88725				
10	Period 3	82661				
11	Period 4	76294				
12	Period 5	69609				
13	Period 6	62589				
14	Period 7	55219				
15	Period 8	47480				
16	Period 9	39354				
17	Period 10	30822		And observe the impact on this value		
18						

Start by playing with this spreadsheet, which is supplied with the homework (you can open it using either Excel of Google Sheets). Change the value of the annual payment, and observe the impact on the last period's ending balance. Using trial and error, try to come up with a periodical payment that brings the ending balance close to 0. This is an important exercise, which will add intuition and help you complete the exercise successfully, so do it.

Inspect the spreadsheet formulas that compute the periodical ending balances, and make sure that you understand the model. If you are not familiar with spreadsheet modeling, this is a good opportunity to become familiar with this important tool.

We will now turn to write a Java program that computes the loan's annual payment, using two techniques: *Brute force* search, and *bi-section* search. Inspect the supplied skeletal `LoanCalc.java` class file, and make sure that you understand its overall logic and structure.

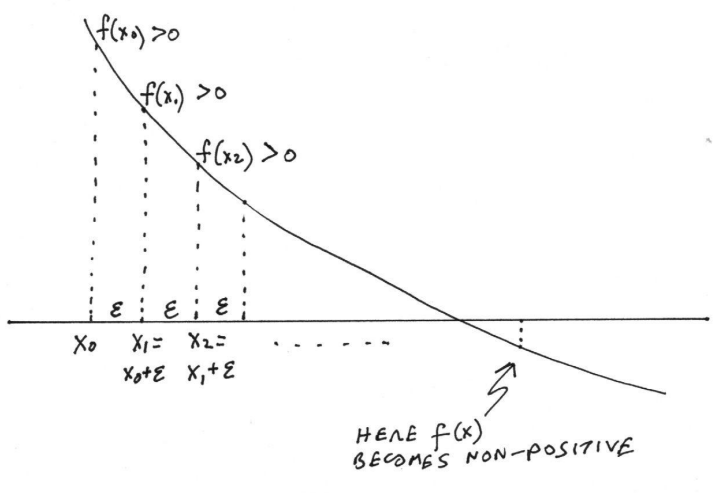
Computing the ending balance: Implement the function `endBalance(loan, rate, n, payment)` in the supplied `LoanCalc.java` class.

Implementation tip: Use a loop to carry out the same computation done by the spreadsheet. Test your implementation by evaluating this function on several different payment values, and compare the returned values to those computed by the spreadsheet.

The key question in this exercise is as follows: How to compute the periodical payment that will bring the loan's ending balance close to zero? Formally, we wish to find *x* such that $f_{loan,rate,n}(x) = 0$, where *f* is the `endBalance` function, `loan` is the initial loan sum, `rate` is the interest rate, `n` is the number of payments, and *x* is the annual payment. We treat the first three values as fixed parameters, so *x* is the only variable of this function. The goal is to find an *x* value for which the function is close to 0.

Note that f is monotonically decreasing in *x*: As *x* increases, f decreases: The more you pay each year, the lower will be your ending balance. As we learned in lecture 3-1, the solution of monotonic functions can be approximated using brute force search, and bisection search.

Brute force search: We start with an initial value g, for which we know that $f(g) > 0$. Then, we set g to $g + \epsilon$, where ϵ is a small value, and check if $f(g) > 0$. We repeat this process until $f(g)$ will become non-positive. At this point we return g, which will be an approximation of the correct solution. How good an approximation? The correct solution will be somewhere in the interval $[g - \epsilon, g + \epsilon]$. So, if ϵ will be reasonably small, we'll be ok. In the following image, the value of g in iteration *i* is represented as x_i:



Implement the `bruteForceSolver` function in the supplied `LoanCalc.java` class.

Implementation tips:

- In this application, f is the `endBalance` function.
- Since the function computes the ending balance of an *n*-period loan, it is reasonable to set the initial guess of the brute force search as follows: $g = loan/n$. Why? Because if in each period we pay *loan* / *n*, it means that we are not paying interest. Therefore, the ending balance will surely be positive, i.e. $f(loan/n) > 0$

- Keep track of the number of iterations by incrementing the static variable `iterationCounter` in each iteration.

Bisection search: As we learned in lecture 3-1, the solution of a monotonic function can be found using an elegant and efficient algorithm – *bisection search*. Here is a version of this algorithm, adapted to this application:

```
// Sets L and H to initial values such that f(L) > 0, f(H) < 0,
// implying that the function evaluates to zero somewhere between L and H.
// So, let's assume that L and H were set to such initial values.
// Set g to (L + H)/2
while (H − L) > ϵ {
    // Sets L and H for the next iteration
    if f(g) · f(L) > 0
        // the solution must be between g and H
        //  so set L or H accordingly
    else
        // the solution must be between L and g
        //  so set L or H accordingly
    // Computes the mid-value (g) for the next iteration
}
return g
```

(In lecture 3-1, we used the notation *M*, for "middle", instead of g)

Implement the `bisectionSolver` function in the supplied `LoanCalc.java` class.

Implementation tips:

- Use the algorithm described above. Note that the algorithm is missing some details. See the bi-section algorithm presented in lecture 3-1, and complete the algorithm accordingly.
- In this application, *f* is the `endBalance` function.
- Choose initial *lo* and *hi* values, using similar considerations to what we did in the brute force search.
- Keep track of the number of iterations by incrementing the static variable `iterationCounter` in each iteration (and make sure to set it to 0 before starting the search).

2. Lower case

(15 points) Write a program (`LowerCase.java`) that gets a command-line string argument, and prints a string which is identical to the given string, except that all the upper-case letters are converted to lower-case letters. Notice that characters that are not letters are left as is. Here are two examples of the program's execution:

```
% java LowerCase B2C
b2c
```

```
% java LowerCase "TLV to LA: 15 Hours."
tlv to la: 15 hours.
```

Implementation tips: Start by defining an empty string that you will gradually evolve into the answer string. Use a loop that iterates through the given string's characters, using the `str.length()` and `str.charAt(int)` functions. If the character is a letter, convert it. Then add the character to the answer string. How to know if a given character is a letter, lower-case, or upper-case? Consult the ASCII table, which will give you hints how to write your code.

3. Unique characters

(15 points) Write a program (`UniqueChars.java`) that gets a command-line string argument, and prints a string which is identical to the original string, except that all duplicate characters are removed, unless they are space characters. Here are two examples of the program's execution:

```
% java UniqueChars committee
comite
```

```
% java UniqueChars "yael played the yokelele"
yael pd th ok
```

Implementation tips: Create a new empty string, and grow it iteratively, using the processed characters. You will need to use some `String` functions, including `str.indexOf(char)`.

4. Calendar

We wish to write a program that gets a year, like 2021 or 1912, and prints the *calendar* of this year. We'll approach this objective gradually. First, we will write and test two key calendar-oriented functions. Second, we'll write a program that prints all the dates from January 1, 1900 to December 31, 1999 (covering the entire 20th century). Finally, we will write the required calendar program.

Calendar0

Write a program (`Calendar0`) that takes a year (like 2021 or 1492) as a command-line argument, and prints two things (1) whether the given year is a common year or a leap year, and (2) The number of days in every month in that year. Here are two separate examples of the program's execution:

```
% java Calendar0 2018
```

```
% java Calendar0 2020
```

```
2018 is a common year      2020 is a leap year
Month 1 has 31 days        Month 1 has 31 days
Month 2 has 28 days        Month 2 has 29 days
Month 3 has 31 days        Month 3 has 31 days
Month 4 has 30 days        Month 4 has 30 days
Month 5 has 31 days        Month 5 has 31 days
Month 6 has 30 days        Month 6 has 30 days
Month 7 has 31 days        Month 7 has 31 days
Month 8 has 31 days        Month 8 has 31 days
Month 9 has 30 days        Month 9 has 30 days
Month 10 has 31 days       Month 10 has 31 days
Month 11 has 30 days       Month 11 has 30 days
Month 12 has 31 days       Month 12 has 31 days
```

In order to perform its operation, the program uses two functions: The `isLeapYear(int)` function checks if the given year is a leap year. The `nDaysInMonth(int,int)` function returns the number of days in the given month of the given year. Both functions are defined in the given `Calendar0.java` file. Each function has a tester function: The `isLeapYearTest(int)` tester is given; the `nDaysInMonthTest(int,int)` tester should be written by you.

Inspect the given `Calendar0.java` file, compile it, run it, and make sure that you understand its logic. Notice that the program produces nonsense, since the tested functions are not yet implemented. Notice also that this class provides a complete, executable, skeleton of all the code that has to be developed, including what is known as "unit testing". This is an example of good software engineering.

Complete and test the `isLeapYear` function. We wrote the basic logic of this function in lecture 1-2. Next, write a basic version of the `nDaysInMonthTest` function. This basic version should call the `nDaysInMonth` function (which was not yet implemented) with a fixed month value. Now implement the `nDaysInMonth` function. We propose using either `switch (better)`, or nested `if-else`. Notice that `nDaysInMonthTest` should call `isLeapYear` – that's a good example of modular design. Test the function using the basic `nDaysInMonthTest` implementation. Next, complete the `nDaysInMonthTest` implementation. Tip: Use a loop that calls `nDaysInMonth` 12 times.

Calendar1

The 20th century is defined as the period between 1.1900 and 31.12.1999, inclusive. Let us start by noting an obscure fact: 1.1.1900 was a Monday. It turns out that this information is all we need in order to print the calendar of every year after this date.

Write a program (`Calendar1`) that prints the calendars of all 100 years of the 20th century. In addition, the program counts and prints how many Sundays fell on the first day of the month during the 20th century. In other words, you have to handle about 36,500 days; if the current day happens to be a Sunday, and also the 1st day of the month, you have to count it. Here is an example of the program's execution:

```
% java Calendar1
```

```
1/1/1900
2/1/1900
3/1/1900
4/1/1900
5/1/1900
6/1/1900
7/1/1900 Sunday
8/1/1900
...
30/8/1901
31/8/1901
1/9/1901 Sunday
2/9/1901
3/9/1901
4/9/1901
5/9/1901
6/9/1901
7/9/1901
8/9/1901 Sunday
9/9/1901
10/9/1901
...
29/12/1999
30/12/1999
31/12/1999

During the 20th century, 172 Sundays fell on the first day of the month
```

Notice that 1.9.1901 is one of these special Sundays that we wish to count.

The program generates about 36500 days and prints that many lines. If you want the program to stop after *n* days, you can do it by setting a debugging variable (explained in the code). To check the output's correctness, consult this website. Focus on the end of the output, and check the dates of a few Sundays. If they fall on the same dates as shown on the website, your program seems to be working properly.

Implementation tips: We know that 1.1.1900 was a Monday. Starting from this anchor, you have to advance several cyclical counters: The day of the week (1, 2, ..., 7, 1, 2, ..., 7, ...), the day of the month (1, 2, ..., *n*1, 1, 2, ..., *n*2, ...) where *n*1, *n*2 , ... are 28, 29, 30, or 31, depending on which month and year we are in, the month of the year (1, 2, …, 12, 1, 2, …, 12, ...), and the year of the century (1900, 1901, …).

The key function in this program is advance, which advances the counters mentioned above. We recommend starting by writing this function, incrementally: write code that advances one counter, and prints the counter's value after each change, for testing purpose. Then add another counter, print its values, and so on.

Write your solution by completing the given `Calendar1.java` class file.

Calendar

We are ready to complete the final version of the program (`Calendar.java`). This program gets a given year, like 2021, and prints the calendar of that year. Here is an example of this program's execution:

```
% java Calendar 2021
1/1/2021
2/1/2021
3/1/2021 Sunday
4/1/2021
5/1/2021
6/1/2021
7/1/2021
8/1/2021
9/1/2021
10/1/2021 Sunday
11/1/2021
12/1/2021
...
25/12/2021
26/12/2021 Sunday
27/12/2021
28/12/2021
29/12/2021
30/12/2021
31/12/2021
```

Implementation tips: This program can be implemented easily by making several simple modifications to the Calendar1 program. Start by inputting the given year, as a command-line argument. Then enter a loop that advances the days (without doing anything else) from 1.1.1900 until the last day of the year before the given year. Then enter a second loop that prints the calendar of the given year. There is no need to count and print the number of Sundays that fall on the first day of the month, so just erase all the statements that deal with this logic.

For this program we provide no skeleton. Make a copy of your working `Calendar1.java` file, change the file name and the class name to `Calendar.java`, and proceed to make the necessary changes.

Submission

Before submitting your work for grading, make sure that your code is written according to our Java Coding Style Guidelines.

Submit the following files only:

- `LoanCalc.java`
- `LowerCase.java`
- `UniqueChars.java`
- `Calendar.java`

Compress the files into a file named `HW3.zip`, and upload the single zip file using Moodle.
Submission deadline: February 9th, 2024, 23:55.