LoanCalc

```
/*
 * Computes the periodical payment necessary to re-pay a given loan.
 */
public class LoanCalc {


   static double epsilon = 0.001; // The computation tolerance (estimation
error)
   static int iterationCounter; // Monitors the efficiency of the calculation


   /**
    * Gets the loan data and computes the periodical payment.
    * Expects to get three command-line arguments: sum of the loan (double),
    * interest rate (double, as a percentage), and number of payments (int).
    */
   public static void main(String[] args) {
      // Gets the loan data
      double loan = Double.parseDouble(args[0]);
      double rate = Double.parseDouble(args[1]);
      int n = Integer.parseInt(args[2]);
      System.out.println("Loan sum = " + loan + ", interest rate = " + rate + "%,
periods = " + n);


      // Computes the periodical payment using brute force search
      System.out.print("Periodical payment, using brute force: ");
      System.out.printf("%.2f", bruteForceSolver(loan, rate, n, epsilon));
      System.out.println();
      System.out.println("number of iterations: " + iterationCounter);


      // Computes the periodical payment using bisection search
      System.out.print("Periodical payment, using bi-section search: ");
      System.out.printf("%.2f", bisectionSolver(loan, rate, n, epsilon));
      System.out.println();
```

```
        System.out.println("number of iterations: " + iterationCounter);
    }

    /**
     * Uses a sequential search method ("brute force") to compute an
approximation
     * of the periodical payment that will bring the ending balance of a loan
close
     * to 0.
     * Given: the sum of the loan, the periodical interest rate (as a percentage),
     * the number of periods (n), and epsilon, a tolerance level.
     */
    // Side effect: modifies the class variable iterationCounter.
    public static double bruteForceSolver(double loan, double rate, int n,
double epsilon) {

        // Determines the initial periodical payment
        double g = loan / n;

        // Reset the iteration counter
        iterationCounter = 0;

        while ((endBalance(loan, rate, n, g) >= epsilon) && (g <= loan)) {

            // increases g by epsilon
            g += epsilon;

            // Increases the interaction by 1
            iterationCounter++;
        }
        return g;
    }
```

```
    /**
     * Uses bisection search to compute an approximation of the periodical payment
     * that will bring the ending balance of a loan close to 0.
     * Given: the sum of theloan, the periodical interest rate (as a percentage),
     * the number of periods (n), and epsilon, a tolerance level.
     */
    // Side effect: modifies the class variable iterationCounter.
    public static double bisectionSolver(double loan, double rate, int n, double epsilon) {

        // Determines low and high such that f(low)>0 and f(high)<0
        double low = loan / n, high = loan;

        // Determines the mid-value (g)
        double g = (low + high) / 2;

        // Reset the iteration counter
        iterationCounter = 0;

        while ((high - low) > epsilon) {

            // Sets L and H for the next iteration
            if (endBalance(loan, rate, n, g) * endBalance(loan, rate, n, low) > 0) {
                low = g;
            } else {
                high = g;
            }

            // Computes the mid-value (g) to the next iteration
            g = (low + high) / 2;

            // Increases the iteration by 1
```

```
            iterationCounter++;
        }
        return g;
    }

    /**
     * Computes the ending balance of a loan, given the sum of the loan, the
     * periodical
     * interest rate (as a percentage), the number of periods (n), and the
     * periodical payment.
     */
    private static double endBalance(double loan, double rate, int n, double
payment) {

        // Determines the final balance of the loan
        double endingBalance = loan;

        // Converts the interest rate from a percentage to a decimal number
        double decimalRate = (rate / 100) + 1;

        // Reduces the periodic payment from the loan ending balance and adds
the
        // periodic interest for each period
        for (int i = 1; i <= n; i++) {
            endingBalance = (endingBalance - payment) * decimalRate;
        }

        return endingBalance;
    }
}
```

LowerCase

```
/** String processing exercise 1. */
public class LowerCase {
    public static void main(String[] args) {
        String str = args[0];
        System.out.println(lowerCase(str));
    }

    /**
     * Returns a string which is identical to the original string,
     * except that all the upper-case letters are converted to lower-case letters.
     * Non-letter characters are left as is.
     */
    public static String lowerCase(String s) {
        // Set new string
        String newS = "";

        for (int i = 0; i < s.length(); i++) {

            // Set x to a certain letter
            int x = s.charAt(i);

            // Check is it a capital letter
            if (x > 64 && x < 91) {
                newS += (char) (x + 32);
            } else {
                newS += (char) (x);
            }
        }
        return newS;
    }
}
```

UniqueChars

```
/** String processing exercise 2. */
public class UniqueChars {
    public static void main(String[] args) {
        String str = args[0];
        System.out.println(uniqueChars(str));
    }

    /**
     * Returns a string which is identical to the original string,
     * except that all the duplicate characters are removed,
     * unless they are space characters.
     */
    public static String uniqueChars(String s) {

        // Sets new string
        String newS = "";

        for (int i = 0; i < s.length(); i++) {

            // Checks if the letter has already appeared

            if (newS.indexOf(s.charAt(i)) == -1) {

                // Adds the letter to the new word
                newS += s.charAt(i);
            } else {
                if (s.charAt(i) == ' ') {
                    newS += s.charAt(i);
                }
            }
        }
```

```
        return newS;
    }
}
```

Calendar

```
/**
 * Prints the calendar of a certain year.
 */
public class Calendar {
    // Starting the calendar on 1/1/1900
    static int dayOfMonth = 1;
    static int month = 1;
    static int year = 1900;
    static int dayOfWeek = 2; // 1.1.1900 was a Monday
    static int nDaysInMonth = 31; // Number of days in January


    /**
     * Prints the calendar of a received year.
     */
    public static void main(String args[]) {
        // Advances the date and the day-of-the-week from 1/1/1900 till 31/12 of
the
        // year received, inclusive.
        // Prints each date dd/mm/yyyy in a separate line of the received year. If
the
        // day is a Sunday, prints "Sunday".
        int thisYear = Integer.parseInt(args[0]);


        // Checks that the received year has not passed yet
        while ((thisYear + 1) > year) {


            // Checks that a year is the received year
            if (thisYear == year) {


                // Checks if the day is Sunday
                if (dayOfWeek == 1) {
```

```
                System.out.println(dayOfMonth + "/" + month + "/" + year + "
Sunday");
                if (dayOfMonth == 1) {

                }
            } else {
                System.out.println(dayOfMonth + "/" + month + "/" + year);
            }
        }
        advance();
    }
}

// Advances the date (day, month, year) and the day-of-the-week.
// If the month changes, sets the number of days in this month.
// Side effects: changes the static variables dayOfMonth, month, year,
// dayOfWeek, nDaysInMonth.
private static void advance() {

    // Checks if the week is over
    if (dayOfWeek == 7) {
        dayOfWeek = 1;
    } else {
        dayOfWeek++;
    }

    // Checks is the month is over
    if (dayOfMonth == nDaysInMonth) {
        month++;

        // Checks if the year is over
        if (month == 13) {
```

```
            month = 1;
            year++;
        }
        nDaysInMonth = nDaysInMonth(month, year);
        dayOfMonth = 1;
    } else {
        dayOfMonth++;
    }
}

// Returns true if the given year is a leap year, false otherwise.
private static boolean isLeapYear(int year) {

    boolean isLeapYear;

    // Check if the year is divisible by 400
    isLeapYear = ((year % 400) == 0);

    // Then checks if the year is divisible by 4 but not by 100
    isLeapYear = isLeapYear || (((year % 4) == 0) && ((year % 100) != 0));

    return isLeapYear;

}

// Returns the number of days in the given month and year.
// April, June, September, and November have 30 days each.
// February has 28 days in a common year, and 29 days in a leap year.
// All the other months have 31 days.
private static int nDaysInMonth(int month, int year) {
    int numberOfDaysInMonth = 0;
```

```
// Checks which month was received and enters the number of days in
// numberOfDaysInMonth
switch (month) {
    case 1:
    case 3:
    case 5:
    case 7:
    case 8:
    case 10:
    case 12:
        numberOfDaysInMonth = 31;
        break;
    case 4:
    case 6:
    case 9:
    case 11:
        numberOfDaysInMonth = 30;
        break;
    case 2:

        // Checks if the year is lean
        if (isLeapYear(year) && month == 2) {
            numberOfDaysInMonth = 29;
        } else {
            numberOfDaysInMonth = 28;
        }
        break;
    default:
        numberOfDaysInMonth = 0;
        break;
}
return numberOfDaysInMonth;
```

```
    }
}
```