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) {
        iterationCounter = 0;
        double payment = 0;
        while (Math.abs(endBalance(loan, rate, n, payment)) >= epsilon) {
            payment += epsilon;
            iterationCounter++;
        }
        return payment;
    }

    /**
    * 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) {
        iterationCounter = 0;
        // initialize low and high bounds of the search
        double L = 0, H = loan, payment = (L + H) / 2.0;
        double endBalance = endBalance(loan, rate, n, payment);
        // loops and sets bounds based on endBalance
        while (Math.abs(endBalance) >= epsilon) {
            if (endBalance > 0) {
                L = payment;
            } else if (endBalance < 0) {
                H = payment;
            }
            payment = (L + H) / 2;
            iterationCounter++;
            endBalance = endBalance(loan, rate, n, payment);
        }
        return payment;
    }

    /**
    * 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) {
        // loop function iterating with n where payment is subtracted from loan
and rate is added to new loan
        for (int i = 0; i < n; i++) {
            loan -= payment;
            loan *= (1 + (rate / 100));
        }
        return loan;
    }
}
```

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) {
        //initialize a new string
        String newString = "";
        int length = s.length();
        int i = 0;
        // loop through the string and add lower case characters to the new
string
        while (i < length) {
            if (s.charAt(i) >= 65 && s.charAt(i) <= 90) {
                newString += (char) (s.charAt(i) + 32);
            } else {
                newString += s.charAt(i);
            }
            i++;
        }

        return newString;
```

```
    }
}
```

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) {
        // initialize a new string
        String newString = "";
        int length = s.length();

        // loop through the string and add unique characters to the new string
        for (int i = 0; i < length; i++) {
            if (i == s.indexOf(s.charAt(i)) || s.charAt(i) == ' ') {
                newString += s.charAt(i);
            }
        }

        return newString;
    }
}
```

CALENDAR

```
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 the given year.
     */
    public static void main(String args[]) {
        int calendarYear = Integer.parseInt(args[0]);
        String sunday = "";
        while (year <= calendarYear) {
            if (year == calendarYear) {
                sunday = (dayOfWeek == 1) ? " Sunday" : "";
                System.out.println(dayOfMonth + "/" + month + "/" + year +
sunday);
            }
            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() {
        dayOfMonth++;
        dayOfWeek++;
        if (dayOfMonth > nDaysInMonth(month, year)) {
            dayOfMonth = 1;
            month++;
        }
        if (month > 12) {
            month = 1;
            year++;
        }
        if (dayOfWeek > 7) {
            dayOfWeek = 1;
        }
     }

    // Returns true if the given year is a leap year, false otherwise.
    private static boolean isLeapYear(int year) {
        boolean isLeap = false;
        if (year % 400 == 0) {
```

```
            isLeap = true;
        } else if (year % 100 == 0) {
            isLeap = false;
        } else if (year % 4 == 0) {
            isLeap = true;
        }
        return isLeap;
    }

    // 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) {
        switch (month) {
            case 4:
            case 6:
            case 9:
            case 11:
                return 30;
            case 2:
                if (isLeapYear(year)) {
                    return 29;
                } else {
                    return 28;
                }
            default:
                return 31;
        }
    }
}
```