

```

/**
 * 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]); // סכום הלוואה
        double rate = Double.parseDouble(args[1]); // ריבית
        int n = Integer.parseInt(args[2]); // מס תשלומים
        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) {
        Double g = loan/n;
        Double balance = endBalance(loan, rate, n, g);
        iterationCounter = 0;

        while (balance >= epsilon){
            g += 0.0001;
            balance = endBalance(loan, rate, n, g);
            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 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 bisectionSolver(double loan, double rate, int n,
double epsilon) {
        Double H = (loan/(n/2));
        Double L = loan/n;
        Double g = (L + H) / 2.0;

        iterationCounter = 0;

        while ((H - L) >= epsilon ){

            Double L_balance = endBalance(loan, rate, n, L);
            Double g_balance = endBalance(loan, rate, n, g);

            if (((g_balance)*(L_balance)) > 0){
                L = g;
            }
        }
    }

```

```

        else{
            H = g;
        }
        g = (L + H) / 2;
        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) {
    Double balance = loan;
    double percent = (1+(rate/100));
    for ( int i = 0; i < n; i++){
        balance = ((balance - payment)*percent);
    }
    return balance;
}
}

```

```

/** 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) {
        String allLow = "";
        for(int i = 0; i < s.length(); i++){
            char chr = s.charAt(i);
            if ( chr >= 'A' && chr <= 'Z' ){
                chr = ((char)(chr + 32));
                allLow += chr;
            }
            else{
                allLow += chr;
            }
        }
        return allLow;
    }
}

```

```
/** 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) {
        String allUnique = "";
        for( int i = 0; i < s.length(); i++){
            char chr = s.charAt(i);
            if (s.indexOf(chr) == i || chr == ' ' ){
                allUnique += s.charAt(i);
            }
        }
        return allUnique;
    }
}
```

```

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

    public static void main(String args[]) {
        int wantedYear = Integer.parseInt(args[0]);
        int debugDaysCounter = 0;
        int totalSundays = 0;

        while (year <= wantedYear) {

            advance();
            debugDaysCounter++;

            if ( year == wantedYear ){

                if (dayOfWeek == 1) {
                    System.out.println(dayOfMonth + "/" + month + "/" + year +
" Sunday");
                    if ( dayOfMonth == 1 ){
                        totalSundays++;
                    }
                } else {
                    System.out.println(dayOfMonth + "/" + month + "/" + year);
                }
            }
        }
    }

    private static void advance() {
        // Increase day of the week
        dayOfWeek = (dayOfWeek % 7) + 1;

        // Increase day of the month
        dayOfMonth = (dayOfMonth % nDaysInMonth) + 1;

        // Increase month of the year
        if (dayOfMonth == 1) {
            month = (month % 12) + 1;
            nDaysInMonth = nDaysInMonth(month, year);
        }
    }
}

```

```

        // Increase year of the century
        if (month == 1 && dayOfMonth == 1) {
            year++;
        }
    }

    public static boolean isLeapYear(int year) {
        return ((year % 400) == 0) || (((year % 4) == 0) && ((year % 100)
!= 0));
    }

    public static int nDaysInMonth(int month, int year) {
        if (month == 4 || month == 6 || month == 9 || month == 11) {
            return 30;
        } else if (month == 2) {
            return isLeapYear(year) ? 29 : 28;
        } else {
            return 31;
        }
    }
}

```