```
/**
 * 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 = 0;    // 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 from the user
        double loan = Double.parseDouble(args[0]);

        // get the rate data from the user
        double rate = Double.parseDouble(args[1]);

        // get the data of how many payments from the user
        int n = Integer.parseInt(args[2]);

        // print the given data
        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();

        // print the number of iteration
        System.out.println("number of iterations: " + iterationCounter);
```

```
    // set new couting
    iterationCounter = 0;

    // 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();

    // print number of iteration in the bisection algorithm
    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) {

    // init the first guess of payment
    double payment  = loan / (double)n;

    // calculate the balance at the end of the loan
    double end_balance  = endBalance(loan, rate , n , payment);

    // iterate the while if the balance greater than 0
    while (end_balance > 0) {

        // increase the guess
        payment =   payment + epsilon;
```

```
        // calclate the new balance
        end_balance = endBalance(loan, rate , n , payment);

        // increase number of iterations
        iterationCounter ++;
    }

    // return the final answer
    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) {



    // init the lowest possible guess
    double low  = loan / (double)n;

    // init the highest possible guess
    double high = loan;

    // init the mid guess according to the boundaries
    double g = (low + high) / 2;

    // calculate the balance of the avrage option
    double current_end_balance  = endBalance(loan, rate , n , g);
```

```
    // go to the while of the search has't ended yet
    while ( (high - low) >= epsilon ) {


        // calculate the new middle
        g = (high + low) / 2;

        // calculate the balance according to the new g
        current_end_balance = endBalance(loan, rate , n , g);



        // check if the you still have to give money at the end of month
        if (current_end_balance >= epsilon){

            // get new low bound
            low = g;
        } else {

            // set new high bound
            high = g;
        }

        // increase number of guess
        iterationCounter ++;

    }

    // return the final answer
    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) {

        // convert the percents to rational number
        rate = (rate + 100) / 100.0;

        // init the remaining balance
        double balance = loan;

        // calculate the balance at each payment (n times)
        for (int i = 0; i < n; i++) {
            balance = ( balance - payment ) * rate;
        }

        // return the final balance
        return balance;
    }
}
```

```
/** String processing exercise 1. */
public class LowerCase {
    public static void main(String[] args) {

        // init the stirng that given from the user
        String str = args[0];

        // call the function and return the given answer
        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) {

        // init variable that holds the new string
        String lower_string  = "";

        // for each char at the string convert every capital letter to small letter
        for (int i = 0; i < s.length(); i++) {

            // get the current i char
            char new_lower = (char) (s.charAt(i));

            // convert to small letter if the letter is capital
            if (new_lower >= 'A' && new_lower <= 'Z'){
               new_lower = (char) (s.charAt(i) + 32);
            }

            // add to the new string the current char
            lower_string += new_lower;
```

```
        }

        // return the final string
        return lower_string;
    }
}
```

```
/** String processing exercise 2. */
public class UniqueChars {
    public static void main(String[] args) {

        // init the string that given from the user
        String str = args[0];

        // call the function and return the given answer
        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) {

        // init the variable that holds the new string
        String new_string = "";

        // for each char of the string do the loop
        for (int i = 0; i < s.length(); i++) {


            // check if the current letter is space
            if (s.charAt(i) == ' '){
                new_string += s.charAt(i);
            } else {

                // add the current char if it is not already in the new string
                if (new_string.indexOf(s.charAt(i)) == -1){
                    new_string += s.charAt(i);
                }
```

```
            }

        }

        // return the final string
        return new_string;
    }
}
```

```
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

    static int end_stop = -1;

    /**
     * Prints the calendars of all the years in the 20th century. Also prints the
     * number of Sundays that occured on the first day of the month during this period.
     */
    public static void main(String args[]) {
        // Advances the date and the day-of-the-week from 1/1/1900 till 31/12/1999, inclusive.
        // Prints each date dd/mm/yyyy in a separate line. If the day is a Sunday, prints "Sunday".
        // The following variable, used for debugging purposes, counts how many days were
advanced so far.

        // init the counter of number of days (for self use)
        int debugDaysCounter = 0;

        // init variable that counts sunday on first day
        int count_first_sunday = 0;
        // init the variable of the requested year
        int requested_year = Integer.parseInt(args[0]);

        while (true) {
            // print the dates if the current year is the requested year
            if (year == requested_year){
                String msg =
String.valueOf(dayOfMonth)+'/'+String.valueOf(month)+'/'+String.valueOf(year);
                // add to the message if the current day is sunday
```

```
            if (dayOfWeek == 1){
                msg += " Sunday";
            }
            // print the current day
            System.out.println(msg);
        }

        // add 1 to the counter if the first day is sunday
        if (dayOfMonth == 1 && dayOfWeek == 1){
            count_first_sunday ++;
        }

        // get the number of days in the current month
        nDaysInMonth = nDaysInMonth(month, year);
        // advance to the next day
        advance();



        debugDaysCounter++;



        // if the requested year has ended
        if ((year > requested_year && month == 12 && dayOfMonth == 31) || debugDaysCounter
== end_stop) {
            break;
        }
      }
   }

   // 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() {
```

```
    // check if end of month
    if (dayOfMonth < nDaysInMonth){
        dayOfMonth ++;
    } else {
        // check if this is the last day of the year
        if (month == 12){
            // set the date of new year
            month = 1;
            year ++;
        } else {
            // go to new month
            month ++;
        }
        // set the first day of the month
        dayOfMonth = 1;
    }

    // check if the week has ended
    if (dayOfWeek == 7){
        dayOfWeek = 1;
    } else {
        dayOfWeek ++;
    }

}

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

    if ( (year % 4 == 0 && year % 100 != 0) || year % 400 == 0 ){
        return true;
    }
    return false;
}
```

```
    // 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) {
        // check if month with 30 days
        if (month == 4 || month == 6 || month == 9 || month == 11){
            return 30;
        }

        // check if febuary
        if (month == 2){

            // if leap year
            if (isLeapYear(year)){
                return 29;
            } else {
                return 28;
            }
        }

        // month with 31 days
        return 31;

    }
}
```