```java
/**
* 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) {
```

```
        double payment = loan/ n; // Set first value to check Payments
        iterationCounter = 0 ;
        double endBalance = endBalance(loan, rate, n, payment);
        while (endBalance - epsilon > 0) { //checks if the endbalance is at
good acurate if not its increasing paymnet by epsilon.
            payment += epsilon;
            endBalance = endBalance(loan, rate, n, payment);
            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) {

        double l = loan/n;
        double h = loan;      //define the 3 values needed for the bi section
search high, low and middle.
        double g = (h + l)/2;
        double endLow = 0;
        double endHigh = 0;
        iterationCounter = 0 ;

        while(h -l  > epsilon ){ // Continue bisection until high and low
values are within epsilon range, adjusting bounds based on end balances.
            endLow = endBalance(loan, rate, n, l);
            endHigh = endBalance(loan, rate, n, g);
            if(endHigh * endLow > 0){
                l = g;
            }
            else{
                h = g;
            }
            g = (h + l) / 2.0;
            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 current = loan;
        double interest = 1 + (rate/100);

        for(int i = 0; i< n; i++)
        {
            current = ((current-payment)*interest);
        }
        return current;

    }
}
```

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

```java
import javax.print.DocFlavor.STRING;

/** 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 newString = "";
        for(int i = 0; i < s.length(); i++){
           if(newString.length() > 0){
                if(s.charAt(i) != ' ')
                {
                    if(newString.indexOf(s.charAt(i))== -1){
                        newString += s.charAt(i);
                    }
                }
                else{
                    newString += s.charAt(i);
                }
            }
           else{
                newString += s.charAt(i);
           }
        }
        return newString;
    }
}
```

```java
/**
 * Prints the calendars of all the years in the 20th century.
 */
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 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[]) {
        int Wanted_year = Integer.parseInt(args[0]);
        // 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.
        int debugDaysCounter = 0;
        //// Write the necessary initialization code, and replace the
condition
        //// of the while loop with the necessary condition

        while (dayOfMonth != 31 || month != 12 || year != Wanted_year - 1) {
// Fast forward to the end of the year before the wanted year
            advance();
            debugDaysCounter++;
        }
        dayOfMonth =1;// Reset to the start of the wanted year and print each
day until the end of that year
        month =1;
        while(year != Wanted_year+1){
            if(dayOfWeek == 1)
            {
                System.out.println(dayOfMonth + "/" + month + "/" +
Wanted_year + " Sunday");
            }
            else{
                System.out.println(dayOfMonth + "/" + month + "/" +
Wanted_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() {

        if(dayOfWeek == 7){
            dayOfWeek = 1;
        }
        else{
            dayOfWeek += 1;
        }

        if(dayOfMonth == nDaysInMonth(month,year)){
            dayOfMonth = 1;
            if(month == 12){
                year += 1;
                month =1;
            }
            else{
                month += 1;
            }
            nDaysInMonth = nDaysInMonth(month,year);
        }
        else{
            dayOfMonth += 1;
        }

    }

   // Returns true if the given year is a leap year, false otherwise.
   private static boolean isLeapYear(int year) {
        boolean isLeapYear = false;
        if((year % 400) == 0 || (year % 4 == 0 && year % 100 != 0)){
            isLeapYear = true;
        }
        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 DaysInMonth = 0;
        if(month == 2){
            if(isLeapYear(year) == true){
                DaysInMonth = 29;
            }
            else{
                DaysInMonth = 28;
            }
        }
        else if(month == 4 || month == 6 || month == 9 || month == 11){
            DaysInMonth = 30;
        }
        else{
            DaysInMonth = 31;
        }
        return DaysInMonth;
    }
}
```