```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) {
        iterationCounter = 0;
        double payment = loan / n;
        while (endBalance(loan, rate, n, payment) > epsilon) {
            iterationCounter++;
            payment += 0.001;
        }
        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;
        double L = loan / n;
        double H = loan;
        double g = (L + H) / 2.0;
        while ((H - L) > epsilon) {
            iterationCounter++;
            // Sets L and H for the next iteration
            if ((endBalance(loan, rate, n, g) * endBalance(loan, rate, n, L)) > 0) {
                // the solution must be between g and H
                // so sets L accordingly
                L = g;
            } else {
                // the solution must be between L and g
                // so sets H accordingly
                H = g;
            }
            // Computes the mid-value (g) for the next iteration
            g = (L + H) / 2.0;
        }
        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;
        for (int i = 0; i < n; i++) {
            balance = (balance - payment) * (1 + 0.01 * rate);
        }
        return balance;
    }
}
```

```java
/** String processing exercise 1. */
public class LowerCase {
    public static void main(String[] args) {
        String str = args[0]; // Gets the string to process
        System.out.println(lowerCase(str)); // Prints the processed string
    }

   /**
    * 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 newString = ""; // Defies an empty string that will be gradually evolved into the answer string
        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i); // Gets the char in the i-th place at the original string
            // Checks if the char is an upper-case letter
            if ((ch >= 65) && (ch <= 90)) {
                ch += 32; // Converts an upper-case letter to a low-case letter by changing it's ASCII value.
            }
            newString += ch; // Adds the processed char to the processed string
        }
        return newString; // Returns the processed string
    }
}
```

```java
/** String processing exercise 2. */
public class UniqueChars {
    public static void main(String[] args) {
        String str = args[0]; // Gets the string to process
        System.out.println(uniqueChars(str)); // Prints the processed string
    }

    /**
     * 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 = ""; // Defies an empty string that will be gradually evolved into the answer string
        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i); // Gets the char in the i-th place at the original string
            // Adds all the space characters from the original string to the processed string
            if (ch == 32) {
                newString += ch; // Adds the char to the processed string
            } else {
                // Removes all the duplicate characters of the original string that are not space characters
                if (s.indexOf(ch) == i) {
                    newString += ch; // Adds the char to the processed string
                }
            }
        }
        return newString; // Returns the processed string
    }
}
```

```java
/*
 * Checks if a given year is a leap year or a common year,
 * and computes the number of days in a given month and a given year.
 */
public class Calendar0 {

    // Gets a year (command-line argument), and tests the functions isLeapYear and nDaysInMonth.
    public static void main(String args[]) {
        int year = Integer.parseInt(args[0]);
        isLeapYearTest(year);
        nDaysInMonthTest(year);
    }

    // Tests the isLeapYear function.
    private static void isLeapYearTest(int year) {
        String commonOrLeap = "common";
        if (isLeapYear(year)) {
            commonOrLeap = "leap";
        }
        System.out.println(year + " is a " + commonOrLeap + " year");
    }

    // Tests the nDaysInMonth function.
    private static void nDaysInMonthTest(int year) {
        for (int i = 1; i <= 12; i++) {
            System.out.println("Month " + i + " has " + nDaysInMonth(i, year) + " days");
        }
    }

    // Returns true if the given year is a leap year, false otherwise.
    public static boolean isLeapYear(int year) {
        if (year % 4 == 0) {
            if (year % 100 != 0) {
                return true;
            } else {
                if (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.
    public static int nDaysInMonth(int month, int year) {
        switch (month) {
            case 2:
                return (isLeapYear(year) ? 29 : 28);
            case 4:
                return 30;
            case 6:
                return 30;
            case 9:
                return 30;
            case 11:
                return 30;
            default:
                return 31;
        }
    }
}


```

```java
/**
 * Prints the calendars of all the years in the 20th century.
 */
public class Calendar1 {
    // 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[]) {
        // 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;
        int sundaysCounter = 0; // Counts the number of sundays that fell on the 1st day of month in the 20th century
        while (year < 2000) {
            System.out.println(dayOfMonth + "/" + month + "/" + year + ((dayOfWeek == 1) ? " Sunday" : ""));
            if (dayOfWeek == 1 && dayOfMonth == 1) {
                sundaysCounter++;
            }
            advance();
            debugDaysCounter++;
            //// If you want to stop the loop after n days, replace the condition of the
            //// if statement with the condition (debugDaysCounter == n)
            if (debugDaysCounter == 37000) {
                break;
            }
        }
        System.out.println("During the 20th century, " + sundaysCounter + " Sundays fell on the first day of the month");
    }

    // 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() {
        dayOfWeek = ((dayOfWeek == 7) ? 1 : dayOfWeek + 1);
        if (dayOfMonth == nDaysInMonth(month, year)) {
            dayOfMonth = 1;
            if (month == 12) {
                year++;
                month = 1;
            } else {
                month++;
            }
        } else {
            dayOfMonth++;
        }
    }

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