```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.
     */
    public static double bruteForceSolver(double loan, double rate, int n, double epsilon) {
        double payment = loan / n;
        double increment = 0.0009;
        iterationCounter = 0;
        while (endBalance(loan, rate, n, payment) > epsilon) {
            payment += increment;
            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.
     */
    public static double bisectionSolver(double loan, double rate, int n, double epsilon) {
        double l = loan / n;
        double h = loan;
        double payment = (l + h) / 2;
        iterationCounter = 0;
        while ((h - l) > epsilon) {
            if (endBalance(loan, rate, n, payment) > epsilon) {
                l = payment;
            } else {
                h = payment;
            }
            payment = (l + h) / 2;
            iterationCounter++;
        }
        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) {
        for (int i = 0; i < n; i++) {
            loan = (loan - payment) * (1 + rate / 100);
        }
        return loan;
    }
}
```

```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 = "";
        // Running threw the string letters
        for (int i = 0; i < s.length(); i++) {
            int chr = s.charAt(i);
            // If the char is A-Z upper-case transform it to lower-case
            if (chr >= 'A' && chr <= 'Z') {
                chr += 32;
            }
            lowerCase += (char) chr;
        }
        return lowerCase;
    }
}
```

```java
/** 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 uniqueChars = "";
        for (int i = 0; i < s.length(); i++) {
            int chr = s.charAt(i);
            if (chr != 32) {
                if (uniqueChars.indexOf(s.charAt(i)) == -1)
                    uniqueChars += s.charAt(i);
            } else {
                uniqueChars += s.charAt(i);
            }
        }
        return uniqueChars;
    }
}
```

```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) {
                if (year % 400 == 0) {
                    return true;
                } else {
                    return false;
                }
            }
            return true;
        }
        return false;
    }

    // Returns the number of days in the given month and year.
    public static int nDaysInMonth(int month, int year) {
        if (month == 2) {
            if (isLeapYear(year)) {
                return 29;
            } else {
                return 28;
            }
        } else if (month == 4 || month == 6 || month == 9 || month == 11) {
            return 30;
        }
        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 occurred 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".
        int debugDaysCounter = 0; // Use for debugging purposes, counts how many days were advanced so far.
        int specialSundays = 0;
        while (year <= 1999 && month <= 12 && dayOfMonth <= 31) {
            if (dayOfWeek == 1 && dayOfMonth == 1) {
                specialSundays++;
            }
            if (dayOfWeek == 1) {
                System.out.println(dayOfMonth + "/" + month + "/" + year + " Sunday");
            } else {
                System.out.println(dayOfMonth + "/" + month + "/" + year);
            }
            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 < 0) {
                break;
            }
        }
        System.out
                .println("During the 20th century, " + specialSundays + " Sundays fell on the first day of the month");
    }

    // Advances the date (day, month, year) and the day-of-the-week.
    private static void advance() {
        dayOfMonth++;
        if (nDaysInMonth(month, year) < dayOfMonth) {
            dayOfMonth = 1;
            month++;
        }
        if (month > 12) {
            month = 1;
            year++;
        }
        dayOfWeek++;
        if (dayOfWeek > 7) {
            dayOfWeek = 1;
        }
    }

    // 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) {
                if (year % 400 == 0) {
                    return true;
                } else {
                    return false;
                }
            }
            return true;
        }
        return false;
    }

    // Returns the number of days in the given month and year.
    private static int nDaysInMonth(int month, int year) {
        if (month == 2) {
            if (isLeapYear(year)) {
                return 29;
            } else {
                return 28;
            }
        } else if (month == 4 || month == 6 || month == 9 || month == 11) {
            return 30;
        }
        return 31;
    }
}
```