

```

/**
 * 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 period = Integer.parseInt(args[2]);
        System.out.println("Loan sum = " + loan + ", interest rate = " + rate
+ "%, periods = " + period);

        // Computes the periodical payment using brute force search
        System.out.print("Periodical payment, using brute force: ");
        System.out.printf("%.2f", bruteForceSolver(loan, rate, period,
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, period,
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
period, double epsilon) {
        double payment = loan/period;
        iterationCounter = 0;
        while (endBalance(loan, rate, period, payment) > epsilon){
            payment = payment + epsilon;
            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 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 period,
double epsilon) {
        // Sets L and H to initial values such that  $f(L) > 0$ ,  $f(H) < 0$ ,
        // implying that the function evaluates to zero somewhere between L
and H .
        // So, let's assume that L and H were set to such initial values.
        // Set g to  $(L + H)/2$ 
        iterationCounter = 0;
        double lowPayment = loan/period;
        double highPayment = loan;
        double g = lowPayment + (highPayment - lowPayment) / 2;
        while ((highPayment - lowPayment) > epsilon) {
            // Sets L and H for the next iteration
            if ((endBalance(loan, rate, period, g) * (endBalance(loan, rate,
period, lowPayment)) > 0)){
                // the solution must be between g and H
                // so set L or H accordingly
                lowPayment = g;
            } else {
                // the solution must be between L and g
                // so set L or H accordingly
                // Computes the mid-value (g) for the next iteration
                highPayment = g;
            }
            g = lowPayment + (highPayment - lowPayment) / 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 period,
double payment) {
        for (int i =0; i < period; i++) {
            loan = (loan - payment) * (1 + rate / 100);
        }
        return loan;
    }
}

```

```

/** 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 str) {
        int i = 0;
        String ans = "";
        while (i < str.length()){
            char ch = str.charAt(i);
            if (ch <= 90 && ch >= 65){
                ans = ans + (char) (str.charAt(i) + 32);
            } else {
                ans = ans + (char) ch;
            }
            i++;
        }
        return ans;
    }
}

```

```

/** 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 str) {
        String ans = "";
        for (int i = 0; i < str.length(); i++){
            char ch = str.charAt(i);
            if (ch == ' '){
                ans = ans + " ";
            } else if (ans.indexOf(ch) == -1){
                ans = ans + ch;
            }
        }
        return ans;
    }
}

```

```

/**
 * 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 month = 1; month <= 12; month++){
        System.out.println("Month " + month + " has " +
nDaysInMonth(month, year) + " days");
    }
}

// Returns true if the given year is a leap year, false otherwise.
public static boolean isLeapYear(int year) {
    return (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;
}

// 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) {
    int days = 31;
    switch (month) {
        case 1: days = 31; break;
        case 2: if (isLeapYear(year)) {
                    days = 29;
                } else {
                    days = 28;
                }
                break;
        case 3: days = 31; break;
        case 4: days = 30; break;
        case 5: days = 31; break;
        case 6: days = 30; break;
        case 7: days = 31; break;
        case 8: days = 31; break;
        case 9: days = 30; break;
        case 10: days = 31; break;
        case 11: days = 30; break;
        case 12: days = 31; break;
    }
    return days;
}
}

```

```

/**
 * 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".
        // The following variable, used for debugging purposes, counts how
many days were advanced so far.
        int debugDaysCounter = 0;
        int sundays = 0;
        //// Write the necessary initialization code, and replace the
condition
        //// of the while loop with the necessary condition
        while (year < 2000) {
            // checks if the day is sunday
            if (dayOfWeek == 1) {
                System.out.println(dayOfMonth + "/" + month + "/" + year + "
Sunday");
                if (dayOfMonth == 1) {
                    sundays ++;
                }
            } 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 (false) {
                break;
            }
        }
    }
}

```

```

    }
}
    System.out.println("During the 20th century, "+ sundays + " 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() {
        // Updates the day of the week
        if (dayOfWeek == 7){
            dayOfWeek = 1;
        } else {
            dayOfWeek ++ ;
        }
        // Updates the day of the month and then accordingly updates the month
and the year if needed
        if (dayOfMonth == nDaysInMonth){
            dayOfMonth = 1;
            if (month == 12){
                month = 1;
                year ++;
            } else {
                month ++;
            }
            nDaysInMonth = nDaysInMonth(month, year);
        } else {
            dayOfMonth ++ ;
        }
    }
}

    // Returns true if the given year is a leap year, false otherwise.
    private static boolean isLeapYear(int year) {
        return (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;
    }

    // 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 days = 31;
        switch (month) {
            case 1: days = 31; break;

```

```

        case 2: if (isLeapYear(year)) {
                    days = 29;
                } else {
                    days = 28;
                }
                break;
        case 3: days = 31; break;
        case 4: days = 30; break;
        case 5: days = 31; break;
        case 6: days = 30; break;
        case 7: days = 31; break;
        case 8: days = 31; break;
        case 9: days = 30; break;
        case 10: days = 31; break;
        case 11: days = 30; break;
        case 12: days = 31; break;
    }
    return days;
}
}

//don't need to submit calender 1

```

```

/**
 * 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 occurred on the first day of the month during
     this period.
     */
    public static void main(String args[]) {
        int newYear = Integer.parseInt(args[0]);
        while (year < newYear){
            advance();
        }
    }
}

```



```

        // 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 (year == newYear) {
            // checks if the day is sunday
            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 (false) {
                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() {
        // Updates the day of the week
        if (dayOfWeek == 7){
            dayOfWeek = 1;
        } else {
            dayOfWeek ++ ;
        }
        // Updates the day of the month and then accordingly updates the month
        and the year if needed
        if (dayOfMonth == nDaysInMonth){
            dayOfMonth = 1;
            if (month == 12){
                month = 1;
                year ++;
            }
        }
    }
}

```

```

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

// Returns true if the given year is a leap year, false otherwise.
private static boolean isLeapYear(int year) {
    return (year % 4 == 0 && year % 100 != 0) || year % 400 == 0;
}

// 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 days = 31;
    switch (month) {
        case 1: days = 31; break;
        case 2: if (isLeapYear(year)) {
            days = 29;
        } else {
            days = 28;
        }
        break;
        case 3: days = 31; break;
        case 4: days = 30; break;
        case 5: days = 31; break;
        case 6: days = 30; break;
        case 7: days = 31; break;
        case 8: days = 31; break;
        case 9: days = 30; break;
        case 10: days = 31; break;
        case 11: days = 30; break;
        case 12: days = 31; break;
    }
    return days;
}
}

//don't need to submit calender 1

```