LoanCalc

```
/**
* 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) {
        // Replace the following statement with your code
        double g = loan / n;
        while (endBalance(loan, rate, n, g) >= epsilon) {
            g += epsilon;
            iterationCounter++;
        }
        return g;
    }

    /**
    * 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) {
        // Replace the following statement with your code
        iterationCounter = 0;
        double L = epsilon , H = loan;
        double g = (L + H) / 2;
        while ((H - L) > epsilon) {
            if ((endBalance(loan, rate, n, g) * endBalance(loan, rate,
n, L)) > 0) {
                L = g;
            }
            else {
                H = g;
            }
            g = (L + H) / 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 n,
double payment) {
        // Replace the following statement with your code
        double endingBalance = loan;
        while (n > 0)
        {
            endingBalance = ((endingBalance - payment) * (1 + rate /
100));
            n = n-1;
        }
        return endingBalance;
    }
}
```

LowerCase

```java
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) {
        // Replace the following statement with your code
        String newStr = "";
        for(int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);
            if (65 <= (int)ch && (int)ch <= 90) {
                ch += 32;
            }
            newStr += ch;
        }
        return newStr;
    }
}
```

UniqueChars

```java
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) {
        // Replace the following statement with your code
        String newStr = "";
        int length = s.length();
        for (int i = 0; i < length; i++){
            char currentChar = s.charAt(i);
            if (newStr.indexOf(currentChar)  == -1) {
                newStr = newStr + s.charAt(i);
            }
            else if (s.charAt(i) == ' ') {
                newStr = newStr + s.charAt(i);
            }
        }
        return newStr;
    }
}
```

Calendar0

```
/*
 * 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) {
        // Replace this comment with your code
        for(int month = 1; month <= 12; month++){
            if (nDaysInMonth(month, year) == 30) {
                System.out.println("Month " + month + " has 30 days");
            }
            else if (month == 2) {
                if (isLeapYear(year)) {
                    System.out.println("Month " + month + " has 29
days");
                }
                else {
                    System.out.println("Month " + month + " has 28
days");
                }
            }
            else if ((nDaysInMonth(month, year) == 31)){
                System.out.println("Month " + month + " has 31 days");
            }
        }
    }
```

```
    // Returns true if the given year is a leap year, false otherwise.
    public static boolean isLeapYear(int year) {
        // Replace the following statement with your code
        boolean isLeapYear;
        // Checks if the year is divisible by 400
        isLeapYear = ((year % 400) == 0);
        // Then checks if the year is divisible by 4 but not by 100
        isLeapYear = isLeapYear || (((year % 4) == 0) && ((year % 100)
!= 0));
        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.
    public static int nDaysInMonth(int month, int year) {
        // Replace the following statement with your code
        int days = 0;
        if (month == 4 || month == 6 || month == 9 || month == 11) {
                days = 30;
            }
        else if (month == 2) {
                if(isLeapYear(year)) {
                    days = 29;
                }
                else {
                    days = 28;
                }
            }
        else {
            days = 31;
        }
        return days;
    }
}
```

Calendar1

```
/**
 * 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
    static int SundaysOnFirst = 0;

    /**
     * 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".
        //// Write the necessary initialization code, and replace the
condition
        //// of the while loop with the necessary condition
        while (year <= 1999) {
            advance();
            // Print current date
            System.out.println(dayOfMonth + "/" + month + "/" + year +
(dayOfWeek == 1 ? " Sunday" : ""));
            // Check for Sundays on the first day
            if (dayOfMonth == 1 && dayOfWeek == 1) {
                SundaysOnFirst++;
            }
        }
        System.out.println("During the 20th century, " +
SundaysOnFirst + " 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() {
```

```
        dayOfMonth++;

        if (dayOfMonth > nDaysInMonth) {  // Move to next month if
needed
            dayOfMonth = 1;
            month++;
        }

        if (month > 12) {  // Move to next year if needed
            month = 1;
            year++;
        }

        dayOfWeek = (dayOfWeek % 7) + 1;  // Ensure dayOfWeek cycles
from 1 to 7

        nDaysInMonth = nDaysInMonth(month, year);  // Update month
length for new month/year
     }

    // Returns true if the given year is a leap year, false otherwise.
    private static boolean isLeapYear(int year) {
        // Replace the following statement with your code
        boolean isLeapYear;
        // Checks if the year is divisible by 400
        isLeapYear = ((year % 400) == 0);
        // Then checks if the year is divisible by 4 but not by 100
        isLeapYear = isLeapYear || (((year % 4) == 0) && ((year % 100)
!= 0));
        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) {
        // Replace the following statement with your code
        int days = 0;
        if (month == 4 || month == 6 || month == 9 || month == 11) {
                days = 30;
            }
        else if (month == 2) {
                if(isLeapYear(year)) {
                    days = 29;
```

```
                }
                else {
                    days = 28;
                }
            }
        else {
            days = 31;
        }
        return days;
    }
}
```