LoanCalc.java

```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);
    }

    /**
```

```java
    * 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 periodicalPayment = loan / n;
        double balance = LoanCalc.endBalance(loan, rate, n,
periodicalPayment); // This statement calls to another endBalance
function to
        iterationCounter =
0;                                                  // culculate
the the remain balance for this periodical payment

        while((Math.abs(balance)) >= epsilon && (balance >= 0)) { //
The loop stops when the balance stops on a number that very close to
0.
            periodicalPayment += epsilon;  // this statement
increase the annual payment by very tiny steps to get a the
accuarate result
            balance = LoanCalc.endBalance(loan, rate, n,
periodicalPayment);
            iterationCounter++;  // Add 1 to the counter of
iteration
        }

        return periodicalPayment;
    }

    /**
    * 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), H = loan; // L - lower payment, H -
higher payment
        double g = (H + L) / 2; // g - the midlle of H and L
```

```
        double balance = LoanCalc.endBalance(loan, rate, n, g); //
This statement calls to another endBalance function to culculate the
the remain balance for this periodical payment
        iterationCounter = 0;   // Reset the variable to the other
search

        while((Math.abs(H - L)) >= epsilon) { // The loop stops when
the balance stops on a number that very close to 0.
            if(balance > 0) {
                L = g;
            } else {
                H = g;
            }
            g = (L + H) / 2;
            balance = LoanCalc.endBalance(loan, rate, n, g);
            iterationCounter++;  // Add 1 to the counter of
iteration
        }
        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++) { // This loop return the remain
balance for given payment.
            balance = (balance - payment) * ((rate / 100) + 1);
        }
        return balance;
    }
}
```

LowerCase.java

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

UniqueChars.java

```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 newStr = "";
        for(int i = 0; i < s.length(); i++) {
            if(newStr.indexOf(s.charAt(i)) == -1 || s.charAt(i) == '
')  {
                newStr += s.charAt(i);
            }
        }
        return newStr;
    }
}
```

```
/**
 * 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[]) {
        // 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
        int userYear = Integer.parseInt(args[0]);
        while (year < userYear) {
            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 == 365001) {
     //         break;
     //     }
        }
        while (year < (userYear + 1)) {
            System.out.print(dayOfMonth + "/" + month + "/" + year);
            if(dayOfWeek == 1 &&  dayOfMonth == 1) {
                System.out.println(" Sunday");
            } else {
                System.out.println();
            }
```

```
            advance();
            debugDaysCounter++;
        }
     }

     // 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++;
    } else {
        dayOfWeek = 1;
    }

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

            }
        }
     }

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

    // 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 febDays = (isLeapYear(year)) ? 29 : 28;
```

```
        switch(month) {
            case 1: return 31;
            case 2: return febDays; // 28 common year, 29 years leap
year.
            case 3: return 31;
            case 4: return 30;
            case 5: return 31;
            case 6: return 30;
            case 7: return 31;
            case 8: return 31;
            case 9: return 30;
            case 10: return 30;
            case 11: return 31;
            case 12: return 31;
        }
        return 0;
    }
}
```