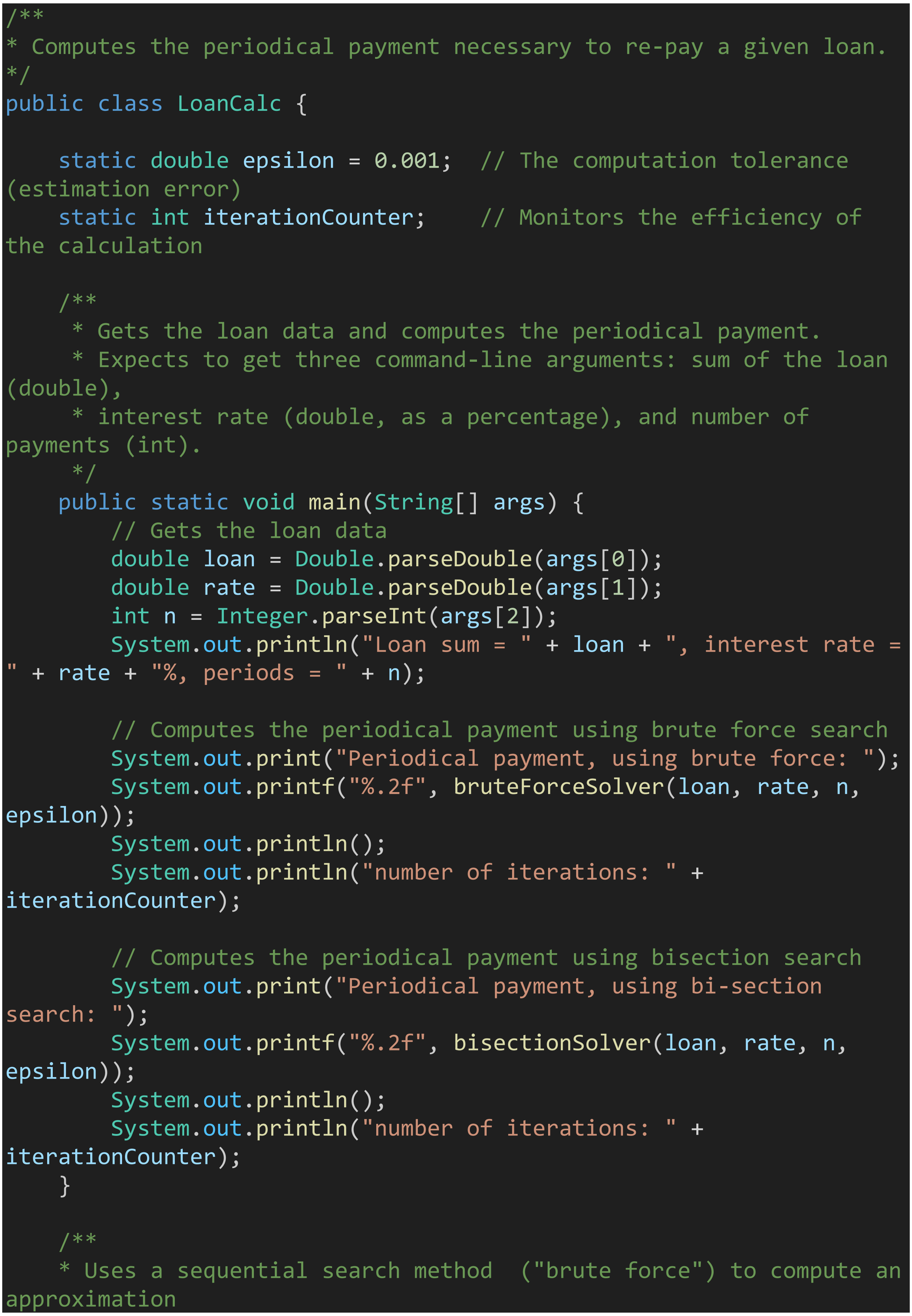
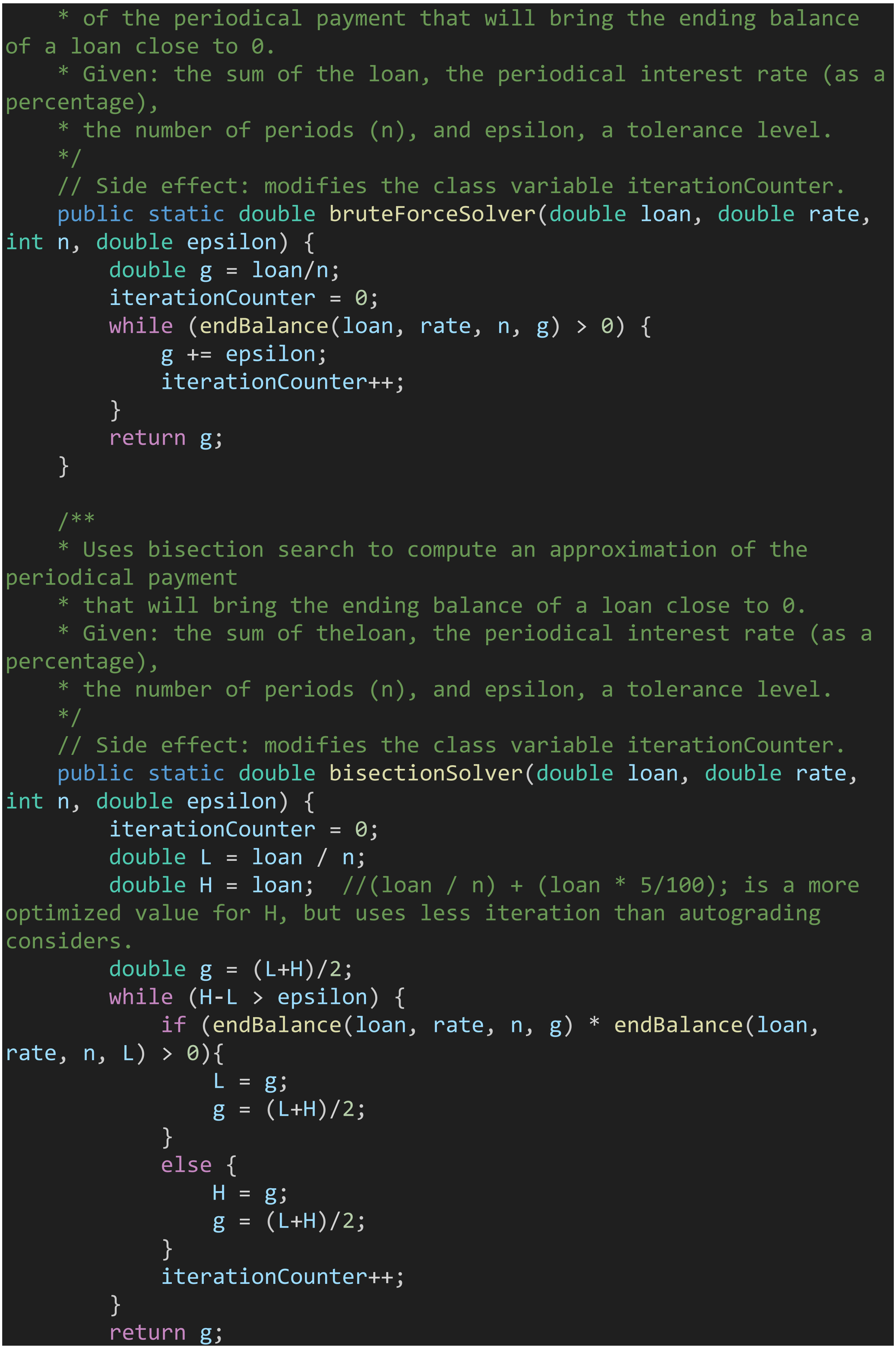
LoanCalc.java

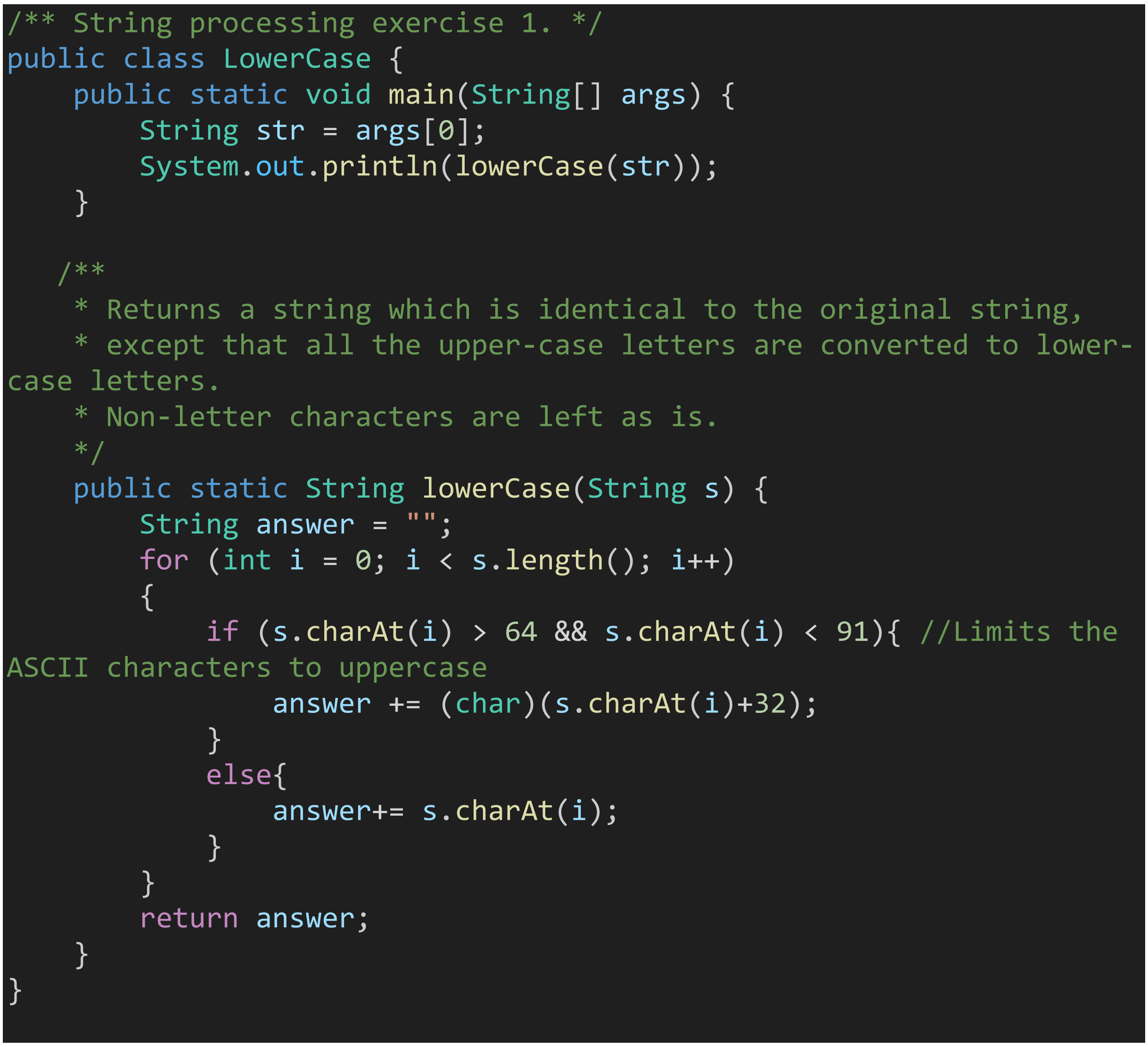


```java
/**
* Computes the periodical payment necessary to re-pay a given loan.
*/
public class LoanCalc {

    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);
    }

    /**
    * Uses a sequential search method  ("brute force") to compute an approximation
```



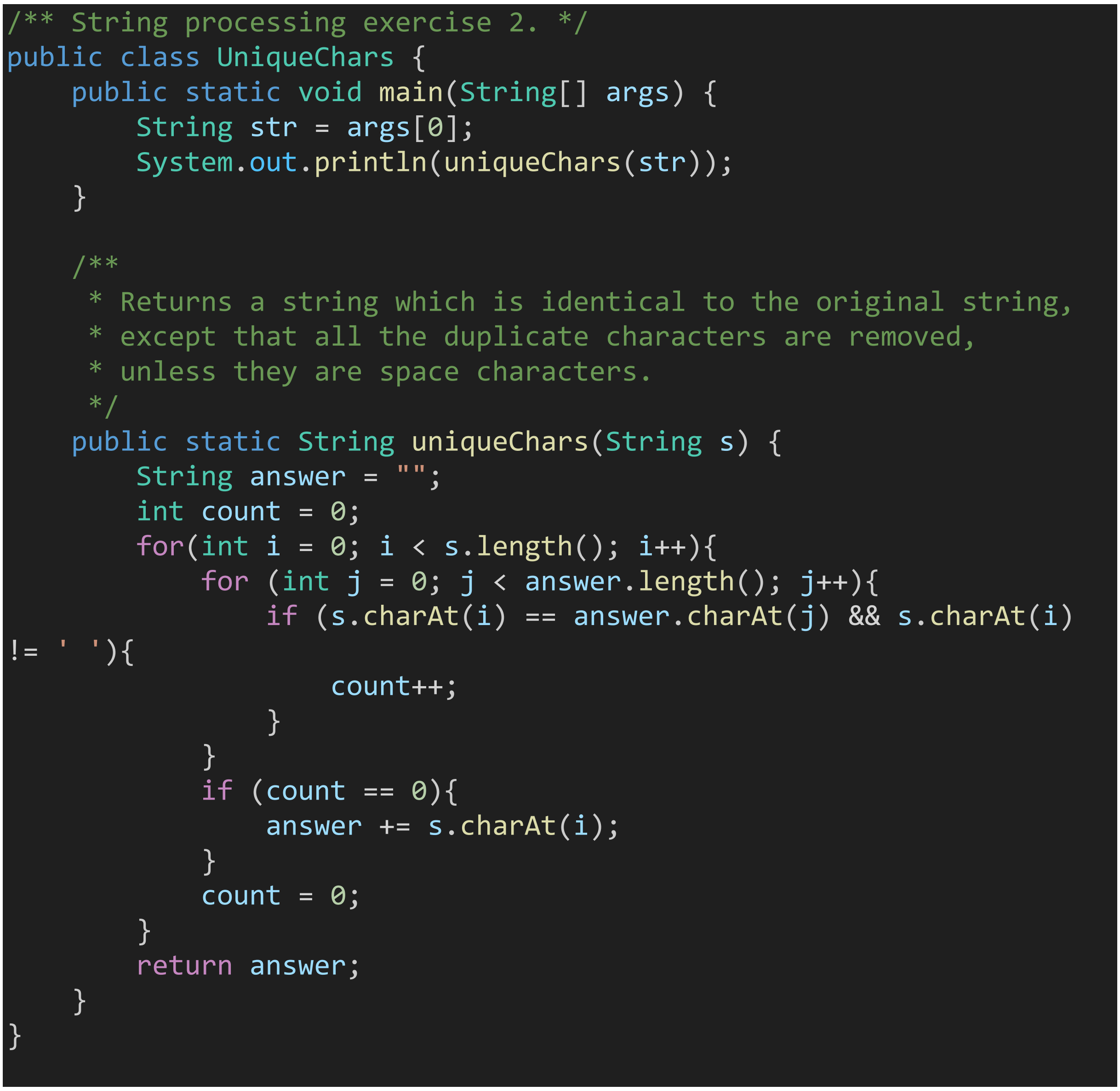
```
    * of the periodical payment that will bring the ending balance
of a loan close to 0.
    * Given: the sum of the loan, the periodical interest rate (as a
percentage),
    * the number of periods (n), and epsilon, a tolerance level.
    */
    // Side effect: modifies the class variable iterationCounter.
    public static double bruteForceSolver(double loan, double rate,
int n, double epsilon) {
        double g = loan/n;
        iterationCounter = 0;
        while (endBalance(loan, rate, n, g) > 0) {
            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) {
        iterationCounter = 0;
        double L = loan / n;
        double H = loan;  //(loan / n) + (loan * 5/100); is a more
optimized value for H, but uses less iteration than autograding
considers.
        double g = (L+H)/2;
        while (H-L > epsilon) {
            if (endBalance(loan, rate, n, g) * endBalance(loan,
rate, n, L) > 0){
                L = g;
                g = (L+H)/2;
            }
            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) {
        double balance = loan;
        for(int i= 1; i <= n; i++){
            balance = (balance - payment) * (1 + rate/100);
        }
        return balance;
    }
}
```

LowerCase.java



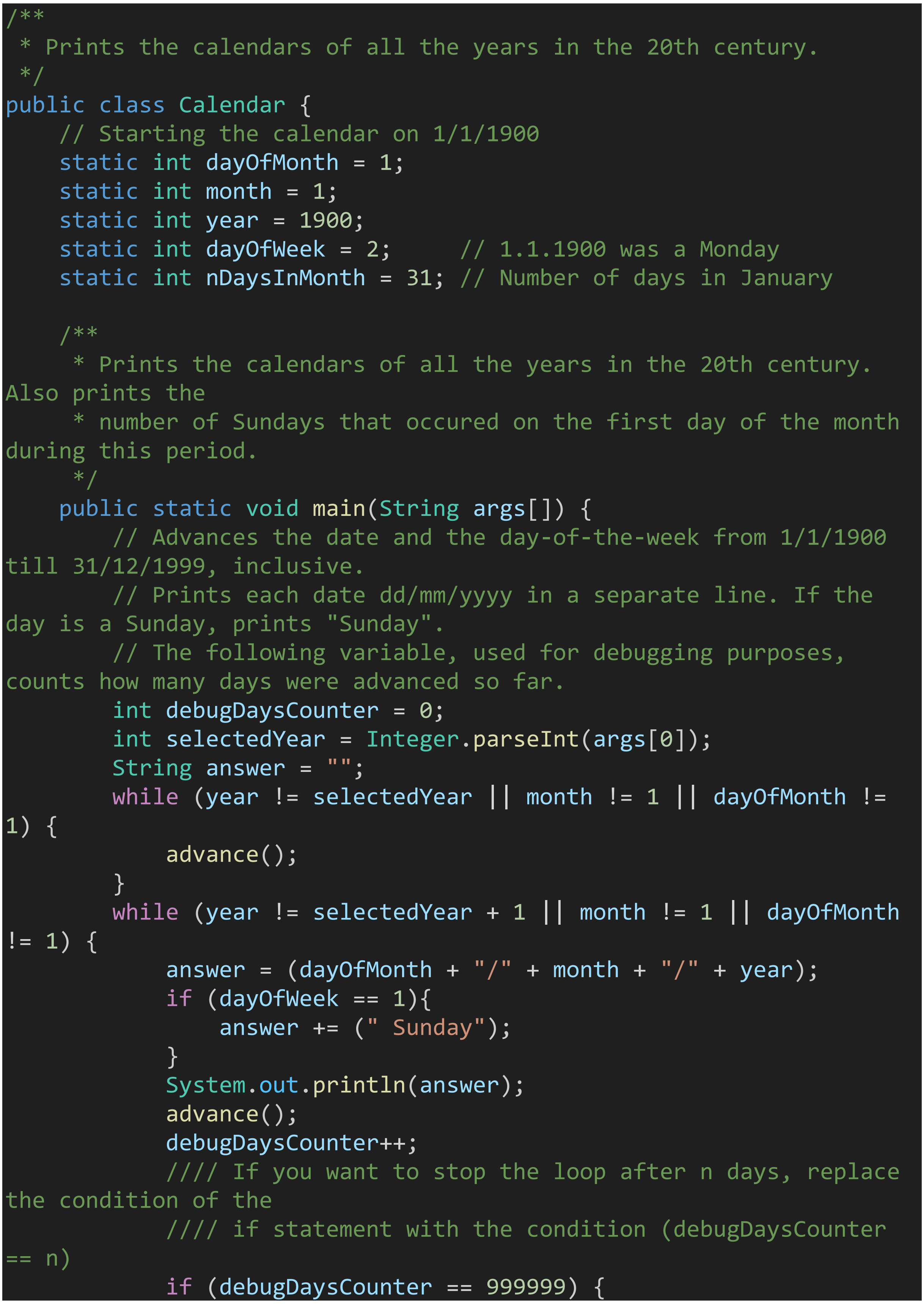
```java
/** String processing exercise 1. */
public class LowerCase {
    public static void main(String[] args) {
        String str = args[0];
        System.out.println(lowerCase(str));
    }

   /**
    * 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 answer = "";
        for (int i = 0; i < s.length(); i++)
        {
            if (s.charAt(i) > 64 && s.charAt(i) < 91){ //Limits the
ASCII characters to uppercase
                answer += (char)(s.charAt(i)+32);
            }
            else{
                answer+= s.charAt(i);
            }
        }
        return answer;
    }
}
```

UniqueChars.java

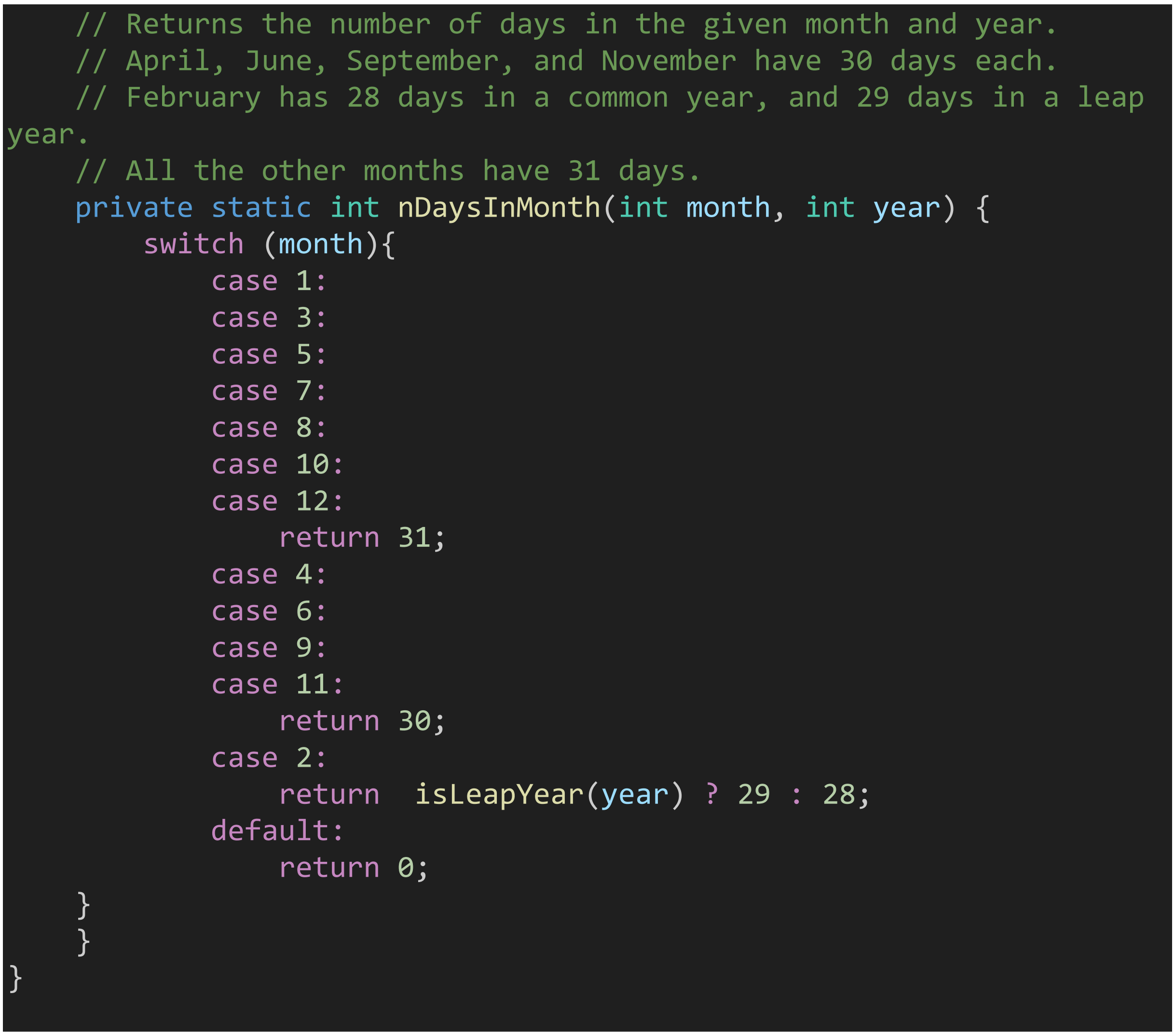


```
/** String processing exercise 2. */
public class UniqueChars {
    public static void main(String[] args) {
        String str = args[0];
        System.out.println(uniqueChars(str));
    }

    /**
     * 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 answer = "";
        int count = 0;
        for(int i = 0; i < s.length(); i++){
            for (int j = 0; j < answer.length(); j++){
                if (s.charAt(i) == answer.charAt(j) && s.charAt(i)
!= ' '){
                    count++;
                }
            }
            if (count == 0){
                answer += s.charAt(i);
            }
            count = 0;
        }
        return answer;
    }
}
```

Calendar.java



```
/**
 * 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;
        int selectedYear = Integer.parseInt(args[0]);
        String answer = "";
        while (year != selectedYear || month != 1 || dayOfMonth !=
1) {
            advance();
        }
        while (year != selectedYear + 1 || month != 1 || dayOfMonth
!= 1) {
            answer = (dayOfMonth + "/" + month + "/" + year);
            if (dayOfWeek == 1){
                answer += (" Sunday");
            }
            System.out.println(answer);
            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 == 999999) {
                System.out.println("Failed loop, debug activated");
                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() {
        if (dayOfWeek == 7){
            dayOfWeek = 1;
        }
        else{
            dayOfWeek++;
        }
        if (nDaysInMonth(month,year) == dayOfMonth ){
            if (month == 12){
                year++;
                month = 1;
            }
            else{
                month++;
            }
            dayOfMonth = 1;
            nDaysInMonth = nDaysInMonth(month,year);
        }
        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 % 400 != 0 && year % 100 == 0){
                return false;
            }
            else{
            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 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                return 31;
            case 4:
            case 6:
            case 9:
            case 11:
                return 30;
            case 2:
                return  isLeapYear(year) ? 29 : 28;
            default:
                return 0;
    }
    }
}
```