```
/**
 * 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);
   }

   /**
    * 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 -= payment;
         loan *= ((100 + rate) / 100);
      }
      return loan;
   }

   /**
    * 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) {
      iterationCounter = 0;
      double increment = 0.001;
      double g = loan / n;
      while (endBalance(loan, rate, n, g) > epsilon) {
```

```
            g += increment;

            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, 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;
    }
}
```

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

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

```
/**
 * 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

    public static void main(String args[]) {
        int inputYear = Integer.parseInt(args[0]);
        while (year < inputYear) {
            advance();
        }
        while (year < inputYear + 1) {
            log();
            advance();
        }
    }

    public static void log() {
        System.out.println(dayOfMonth + "/" + month + "/" + year + (dayOfWeek == 1 ? " Sunday" :
""));
    }

    // 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() {
        dayOfWeek++;
        dayOfWeek = dayOfWeek == 8 ? 1 : dayOfWeek;
        dayOfMonth++;
        if (dayOfMonth > nDaysInMonth) {
            dayOfMonth = 1;
            month++;
            if (month == 13) {
                month = 1;
                year++;
            }
            nDaysInMonth = nDaysInMonth(month, year);
        }
    }

    // Returns true if the given year is a leap year, false otherwise.
    public static boolean isLeapYear(int year) {
        // Checks if the year is divisible by 400
        boolean 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) {
```

```
        int days = 0;
        switch (month) {
            case 1:
            case 3:
            case 5:
            case 7:
            case 8:
            case 10:
            case 12:
                days = 31;
                break;
            case 2:
                days = isLeapYear(year) ? 29 : 28;
                break;
            default:
                days = 30;
        }
        return days;
    }
}
```