/\*\*

\* 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;

double finalBalance = endBalance(loan, rate, n, g);

boolean ifZero = (finalBalance <= 0);

iterationCounter = 0;

while (ifZero == false && g <= loan) {

g += epsilon;

finalBalance = endBalance(loan, rate, n, g);

ifZero = (finalBalance <= 0);

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;

double M = (L + H) / 2;

while (H - L > epsilon) {

// Checks if the product is negative or positive

// If positive - it means that the payment is too low so the low bound should be higher

if (endBalance(loan, rate, n, M) \* endBalance(loan, rate, n, L) > 0) {

L = M;

} else { // If negative - it means that the payment is too high than the high bound should be lower

H = M;

}

M = (L + H) / 2;

iterationCounter++;

}

return M;

}

/\*\*

\* 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 updatedPayLeft = loan;

for (int period = 0; period < n; period++){

updatedPayLeft = (updatedPayLeft - payment) \* (1 + rate / 100);

}

return updatedPayLeft;

}

}

/\*\* 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 inputString) {

String newLowerStr = "";

for (int charIndex = 0; charIndex < inputString.length(); charIndex++) {

char letter = inputString.charAt(charIndex);

if (letter >= 'A' && letter <='Z') {

letter += 32;

}

newLowerStr += letter;

}

return newLowerStr;

}

}

/\*\* 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 inputStr) {

String noDupStr = "";

for (int charIndex = 0; charIndex < inputStr.length(); charIndex++) {

char letter = inputStr.charAt(charIndex);

if (noDupStr.indexOf(letter) == -1 || letter == 32) {

noDupStr += letter;

}

}

return noDupStr;

}

}

/\*\*

\* 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".

int inputYear = Integer.parseInt(args[0]);

while (year < inputYear) {

advance();

}

while (year == inputYear) {

if (dayOfWeek != 1) {

System.out.println(dayOfMonth + "/" + month + "/" + year);

} else {

System.out.println(dayOfMonth + "/" + month + "/" + year + " Sunday");

}

advance();

}

}

// 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) {

month++;

} else {

month = 1;

year++;

}

nDaysInMonth = nDaysInMonth(month, year);

dayOfMonth = 1;

}

}

// Returns true if the given year is a leap year, false otherwise.

private static boolean isLeapYear(int year) {

boolean firstCond = year % 400 == 0;

boolean secondCond = year % 4 == 0 && year % 100 != 0;

boolean checkIfLeapYear = firstCond || secondCond;

return checkIfLeapYear;

}

// 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 monthDays = 0;

switch (month) {

case 1:

monthDays = 31;

break;

case 2:

monthDays = isLeapYear(year) ? 29 : 28;

break;

case 3:

monthDays = 31;

break;

case 4:

monthDays = 30;

break;

case 5:

monthDays = 31;

break;

case 6:

monthDays = 30;

break;

case 7:

monthDays = 31;

break;

case 8:

monthDays = 31;

break;

case 9:

monthDays = 30;

break;

case 10:

monthDays = 31;

break;

case 11:

monthDays = 30;

break;

case 12:

monthDays = 31;

break;

}

return monthDays;

}

}