// Quarter.cs

namespace Task3

{

public enum Quarter

{

First = 1,

Second,

Third,

Fourth

}

}

// Months.cs

namespace Task3

{

public enum Months

{

January,

February,

March,

April,

May,

June,

July,

August,

September,

October,

November,

December

}

}

// ApartmentInfo.cs

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Task3

{

public class ApartmentInfo

{

private string owner;

// TKey - number of month in the given quarter

// TValue - Tuple with two integers. Item1 - input indicator, Item2 - output indicator

private Dictionary<int, Tuple<int, int>> indicators;

private int apartmentNumber;

private Quarter quarter;

private double pricePerKiloWatt;

public string Owner { get => owner; }

public Dictionary<int, Tuple<int, int>> Indicators { get => indicators; }

public int ApartmentNumber { get => apartmentNumber; }

public Quarter Quarter { get => quarter; }

public double PricePerKiloWatt { get => pricePerKiloWatt; }

public ApartmentInfo(

string owner,

Dictionary<int, Tuple<int, int>> indicators,

int apartmentNumber,

Quarter quarter,

double pricePerKiloWatt

)

{

if (String.Compare(owner, "") == 0)

{

throw new ArgumentException("Owner must not be empty string.",

nameof(owner));

}

if (indicators.Count != 3)

{

throw new ArgumentException("Too less or too many elements in indicators.",

nameof(indicators));

}

if (apartmentNumber <= 0)

{

throw new ArgumentException("Invalid value for ApartmentNumber. It must be positive value.");

}

if (pricePerKiloWatt <= 0.0)

{

throw new ArgumentException("Invalid value for PricePerKiloWatt. It must be positive value.",

nameof(pricePerKiloWatt));

}

this.owner = owner;

this.indicators = indicators.ToDictionary(

entry => entry.Key,

entry => entry.Value

);

this.apartmentNumber = apartmentNumber;

this.quarter = quarter;

this.pricePerKiloWatt = pricePerKiloWatt;

}

public override string ToString()

{

StringBuilder sb = new();

sb.Append($"Owner: {owner}\n");

sb.Append($"Number: {ApartmentNumber}\n");

sb.Append($"Quarter: {Quarter}\n");

for (int i = 0; i < Indicators.Count; i++)

{

string month = ((Months)(((int)Quarter - 1) \* 3 + i)).ToString();

sb.Append($"\n{month}: \n");

sb.Append($"Input indicator: {Indicators[i + 1].Item1}\n");

sb.Append($"Output indicator: {Indicators[i + 1].Item2}\n");

sb.Append($"Electricity used: {Indicators[i + 1].Item2 - Indicators[i + 1].Item1} kW\n\n");

}

double overallUsed = indicators.Sum(indicator => indicator.Value.Item2 - indicator.Value.Item1);

sb.Append($"Overall electricity used: {overallUsed} kW\n");

double toPay = overallUsed \* pricePerKiloWatt;

sb.Append(String.Format("To pay: {0:F2} UAH\n", toPay));

sb.Append("-------------------------------------\n");

return sb.ToString();

}

}

}

// ElectricityAccounting.cs

using System;

using System.IO;

using System.Collections.Generic;

using System.Linq;

using System.Text;

namespace Task3

{

public class ElectricityAccounting

{

private int apartmentCount;

public double pricePerKiloWatt;

public Quarter Quarter { get; private set; }

public int ApartmentCount

{

get => apartmentCount;

set

{

if (value <= 0)

{

throw new ArgumentException("Apartment count must be positive value.", nameof(value));

}

apartmentCount = value;

}

}

public List<ApartmentInfo> ApartmentsInfo { get; private set; }

public double PricePerKiloWatt

{

get => pricePerKiloWatt;

private set

{

if (value <= 0.0)

{

throw new ArgumentException("Price per kiloWatt must be positive value.", nameof(value));

}

pricePerKiloWatt = value;

}

}

public ElectricityAccounting(string dataPath, double pricePerKiloWatt)

{

PricePerKiloWatt = pricePerKiloWatt;

ReadFromFile(dataPath);

}

public void ReadFromFile(string dataPath)

{

if (!File.Exists(dataPath))

{

throw new FileNotFoundException("File not found.", dataPath);

}

if (String.Compare(new FileInfo(dataPath).Extension, ".txt") != 0)

{

throw new ArgumentOutOfRangeException($"{nameof(dataPath)}", dataPath, "Only text files are supported (\*.txt).");

}

using (StreamReader reader = new(dataPath))

{

string line;

line = reader.ReadLine();

if (String.Compare(line, "") == 0 && line == null)

{

throw new FormatException("File has invalid format.");

}

string[] data = line.Split(" ", StringSplitOptions.RemoveEmptyEntries);

if (data.Length < 2)

{

throw new FormatException("File has invalid format.");

}

if (!Enum.TryParse(data[1], out Quarter quarter)

|| !Int32.TryParse(data[0], out int apartmentCount))

{

throw new FormatException("File has invalid format.");

}

this.Quarter = quarter;

this.ApartmentCount = apartmentCount;

data = reader.ReadToEnd().Split("\r\n");

if (data.Length != ApartmentCount)

{

throw new FormatException("Number of records is not equal to count of apartments.");

}

const int countOfDataPerLine = 5;

ApartmentsInfo = new List<ApartmentInfo>();

for (int i = 0; i < data.Length; ++i)

{

string[] accountingInfo = data[i].Split(' ', StringSplitOptions.RemoveEmptyEntries);

if (accountingInfo.Length != countOfDataPerLine)

{

throw new FormatException($"Records #{i + 1} is invalid.");

}

if (!Int32.TryParse(accountingInfo[0], out int apartmentNumber) || apartmentNumber <= 0)

{

throw new FormatException($"Invalid apartment number in record #{i + 1}");

}

string owner = accountingInfo[1];

Dictionary<int, Tuple<int, int>> dIndicators = new();

for (int j = 2; j < accountingInfo.Length; ++j)

{

string[] indicators = accountingInfo[j].Split('-');

if (indicators.Length != 2)

{

throw new FormatException("Every record must contain input and output indicator.");

}

if (!Int32.TryParse(indicators[0], out int inputIndicator)

|| !Int32.TryParse(indicators[1], out int outputIndicator)

|| inputIndicator < 0

|| outputIndicator < 0

|| inputIndicator > outputIndicator)

{

throw new FormatException($"Invalid indicator value in record #{i + 1}");

}

dIndicators.Add(j - 1, new Tuple<int, int>(inputIndicator, outputIndicator));

}

ApartmentsInfo.Add(new ApartmentInfo(

owner,

dIndicators,

apartmentNumber,

this.Quarter,

pricePerKiloWatt)

);

}

}

}

public string PrintAllApartments()

{

StringBuilder sb = new();

foreach (var apartmentInfo in ApartmentsInfo)

{

sb.Append(apartmentInfo.ToString());

sb.Append('\n');

}

return sb.ToString();

}

public string GetApartmentInfo(int apartmentNumber)

{

var result = ApartmentsInfo.Where(apartment => apartment.ApartmentNumber == apartmentNumber);

if (result.Count() == 0)

{

throw new ArgumentException("Failed to find apartment with given number.", nameof(apartmentNumber));

}

return result.First().ToString();

}

public string GetOwnerWithMostArrears()

{

return ApartmentsInfo

.OrderByDescending(apartmentInfo =>

apartmentInfo

.Indicators

.Sum(indicator => (indicator.Value.Item2 - indicator.Value.Item1) \* pricePerKiloWatt))

.First()

.Owner;

}

public int FindApartmentWithoutUsing()

{

var result = ApartmentsInfo.Where(

apartment => apartment

.Indicators

.Sum(indicators => indicators.Value.Item2 - indicators.Value.Item1) == 0

);

if (result.Count() == 0)

{

return -1;

}

return result.First().ApartmentNumber;

}

}

}

// MagicSquareOdd.cs

using System;

using System.Text;

namespace Task3

{

public class MagicSquareOdd

{

private int size;

public int Size

{

get => size;

set

{

if (value <= 0)

{

throw new ArgumentException($"{nameof(value)} is less or equal to 0.", $"{nameof(Size)} must be positive number.");

}

if (value % 2 == 0)

{

throw new ArgumentException($"{nameof(value)} is even number.", $"{nameof(Size)} must be odd number");

}

size = value;

}

}

public MagicSquareOdd(int size)

{

this.Size = size;

}

public int[,] BuildSquare()

{

int[,] square = new int[Size, Size];

int x = Size / 2, y = Size - 1;

for (int i = 1; i <= Size \* Size;)

{

if (x < 0 && y == Size)

{

x = 0;

y = Size - 2;

}

else

{

if (y == Size)

{

y = 0;

}

if (x < 0)

{

x = Size - 1;

}

}

if (square[x, y] != 0)

{

y -= 2;

x++;

continue;

}

else

{

square[x, y] = i;

i++;

}

x--;

y++;

}

return square;

}

public override string ToString()

{

StringBuilder sb = new();

int[,] square = BuildSquare();

for (int i = 0; i < square.GetLength(0); ++i)

{

for (int j = 0; j < square.GetLength(1); ++j)

{

sb.Append(square[i, j].ToString().PadRight(8));

}

sb.Append('\n');

}

return sb.ToString();

}

}

}

// Program.cs

using System;

using System.IO;

namespace Task3

{

class Program

{

static void Main(string[] args)

{

DemonstrateMethods();

}

static void DemonstrateMethods()

{

ElectricityAccounting accounting = null;

try

{

accounting = new(@"D:\Workspace\SigmaTasks\Task3\Task3\data.txt", 5);

}

catch (FileNotFoundException ex)

{

Console.WriteLine("Exception was caught!");

Console.WriteLine(ex.Message);

}

catch (ArgumentOutOfRangeException ex)

{

Console.WriteLine("Exception was caught!");

Console.WriteLine(ex.Message);

}

catch (FormatException ex)

{

Console.WriteLine("Exception was caught");

Console.WriteLine(ex.Message);

}

catch (ArgumentException ex)

{

Console.WriteLine("Exception was caught!");

Console.WriteLine(ex.Message);

}

int apartmentNumber = accounting.FindApartmentWithoutUsing();

if (apartmentNumber < 0)

{

Console.WriteLine("There is no apartment where electricity wasn't used.");

}

else

{

Console.WriteLine($"Found apartment where electricity wasn't used: {apartmentNumber}");

}

try

{

Console.WriteLine(accounting.GetApartmentInfo(110));

}

catch (ArgumentException ex)

{

Console.WriteLine("Exception was caught!");

Console.WriteLine(ex.Message);

}

Console.WriteLine(accounting.PrintAllApartments());

Console.WriteLine(accounting.GetOwnerWithMostArrears());

MagicSquareOdd magicSquare = new(15);

Console.WriteLine(magicSquare.ToString());

magicSquare.Size = 11;

Console.WriteLine(magicSquare.ToString());

}

}

}

// data.txt

5 2

10 Pavliv 1000-1100 1100-1220 1220-1400

15 Vovkanych 890-1020 1020-1080 1080-1080

27 Holub 0-0 0-0 0-0

43 Shevtsov 1000-1020 1020-1020 1020-1180

18 Ilchenko 2087-2100 2100-2100 2100-2345