class User

{

public string \_string;

private double \_compression;

private int \_position;

public void Compress(double compression)

{

\_compression \*= compression;

}

public void Move(int offset)

{

\_position += offset;

}

public User(int c, int p)

{

\_string = "";

\_compression = c;

\_position = p;

}

public User()

{

\_string = "";

\_compression = 1;

\_position = 0;

}

public void Write()

{

Console.WriteLine($"compression = {\_compression}, position = {\_position}");

}

public void DeleteSpaces(Action<User> op) => op(this);

public void DeletePunctuationMarks(Action<User> op) => op(this);

public int ReturnNumberOfSpaces(Func<User, int> op) => op(this);

public bool IsShorterThanTenCharacters(Predicate<User> op) => op(this);

public void AddQwerty(Action<User> op) => op(this);

}

class EventCallers

{

public delegate void delegateCompress(double compress);

public delegate void delegateMove(int offset);

public event delegateCompress? eventHandlerCompress;

public event delegateMove? eventHandlerMove;

public void EventCompress(double compress)

{

if (eventHandlerCompress != null)

{

eventHandlerCompress(compress);

}

}

public void EventMove(int offset)

{

if (eventHandlerMove != null)

{

eventHandlerMove(offseпt);

}

}

}

class **ProgramDelegatesEvents**

{

static void Main()

{

Action<User> ActionDeleteSpaces = (User user) =>

{

string result = "";

for (int i = 0; i < user.\_string.Length; i++) if (user.\_string[i] != ' ') result += user.\_string[i];

user.\_string = result;

Console.WriteLine(result);

};

Action<User> ActionDeletePunctuationMarks = (User user) =>

{

string result = "";

string marks = ".,?!-";

for (int i = 0; i < user.\_string.Length; i++)

{

bool mark = false;

for (int j = 0; j < marks.Length && !mark; j++) if (user.\_string[i] == marks[j]) mark = true;

if (!mark) result += user.\_string[i];

}

user.\_string = result;

Console.WriteLine(result);

};

Func<User, int> FuncReturnNumberOfSpaces = (User user) =>

{

int result = 0;

for (int i = 0; i < user.\_string.Length; i++) if (user.\_string[i] == ' ') result++;

return result;

};

Predicate<User> PredicateIsSHorterThanTenCharacters = (User user) =>

{

return user.\_string.Length < 10 ? true : false;

};

Action<User> ActionAddQwerty = (User user) =>

{

string qwerty = "QWERTY";

string result = "";

for (int i = 0, j = 0; i < user.\_string.Length; i++)

{

if (i % 3 == 0 && j < qwerty.Length) result += qwerty[j++];

result += user.\_string[i];

}

user.\_string = result;

Console.WriteLine(result);

};

User user = new();

user.\_string = "ir!u.i ulf,bvu ,?abupe.ivp hbefjfv";

Console.WriteLine(user.ReturnNumberOfSpaces(FuncReturnNumberOfSpaces));

user.DeleteSpaces(ActionDeleteSpaces);

user.DeletePunctuationMarks(ActionDeletePunctuationMarks);

if (user.IsShorterThanTenCharacters(PredicateIsSHorterThanTenCharacters)) Console.WriteLine("string is shorter than 10 characters");

else Console.WriteLine("string is longer than 9 characters");

user.AddQwerty(ActionAddQwerty);

User[] users = new User[5];

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

{

users[i] = new();

}

EventCallers eventsFirst = new();

EventCallers eventsSecond = new();

eventsFirst.eventHandlerCompress += users[1].Compress;

eventsFirst.eventHandlerCompress += users[2].Compress;

eventsFirst.eventHandlerCompress += users[3].Compress;

eventsFirst.eventHandlerMove += users[0].Move;

eventsFirst.eventHandlerMove += users[4].Move;

eventsFirst.EventCompress(0.4);

eventsFirst.EventMove(5);

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

{

users[i].Write();

}

Console.WriteLine();

eventsSecond.eventHandlerCompress += users[1].Compress;

eventsSecond.eventHandlerCompress += users[2].Compress;

eventsSecond.eventHandlerMove += users[0].Move;

eventsSecond.eventHandlerMove += users[1].Move;

eventsSecond.EventCompress(0.9);

eventsSecond.EventMove(-13);

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

{

users[i].Write();

}

}

}

class **ProgramLINQ**

{

static void Main()

{

// Task01

string[] months = { "January", "February", "March", "April", "May", "June", "July",

"August", "September", "October", "November", "December" };

IEnumerable<string> sumWinMonths = months.Where(n => Equals(n, "January") || Equals(n, "February")

|| Equals(n, "December") || Equals(n, "June") || Equals(n, "July") || Equals(n, "August"))

.Select(n => n);

Console.Write("N = ");

int N = int.Parse(Console.ReadLine());

IEnumerable<string> lengthEqualsN = months.Where(n => n.Length == N).Select(n => n);

IEnumerable<string> lengthMoreOrEquals4AndContainsU = months.Where(n => n.Length >= 4

&& n.IndexOf('u') != -1).Select(n => n);

IEnumerable<string> alphOrder = months.OrderBy(s => s);

foreach (string month in sumWinMonths) Console.WriteLine(month);

Console.WriteLine();

foreach (string month in lengthEqualsN) Console.WriteLine(month);

Console.WriteLine();

foreach (string month in alphOrder) Console.WriteLine(month);

//Task02

var rand = new Random();

List<Date> dateList = new();

for (int i = 0; i < 10; i++) dateList.Add(new Date(rand.Next(1945, 2076), rand.Next(1, 13),

rand.Next(1, 28)));

foreach(var n in dateList) Console.WriteLine(n);

Console.Write("year = ");

int year = int.Parse(Console.ReadLine());

//IEnumerable<Date> certainYearDates = dateList.Where(n => n.getYear() == year).Select(n => n);

IEnumerable<Date> certainYearDates = from d in dateList where d.getYear() == year select d;

foreach (Date n in certainYearDates) n.WriteDate();

Console.Write("month = ");

int iMonth = int.Parse(Console.ReadLine());

IEnumerable<Date> certainMonthDates = dateList.Where(n => n.getMonth() == iMonth).Select(n => n);

foreach (Date n in certainMonthDates) n.WriteDate();

Console.WriteLine("Date interval");

IEnumerable<Date> datesInterval = dateList.Where(n => 1900 <= n.getYear() && n.getYear() <= 2000)

.Select(n => n);

foreach (Date n in datesInterval) n.WriteDate();

Console.WriteLine("max date");

var sortedByDescending = dateList.OrderByDescending(n => n.getYear()).ThenByDescending(n => n.getMonth())

.ThenByDescending(n => n.getDay()).Select(n => n);

foreach (Date n in sortedByDescending)

{

n.WriteDate();

break;

}

Console.WriteLine("sorted");

foreach (Date n in sortedByDescending) n.WriteDate();

Console.Write("day = ");

int day = int.Parse(Console.ReadLine());

foreach (Date n in dateList)

{

if (n.getDay() == day)

{

n.WriteDate();

break;

}

}

//Task04

string[] names = { "Анна", "Станислав", "Ольга", "Сева" };

int[] key = { 1, 4, 5, 7 };

var sometype = names

.Join(

key,

w => w.Length,

q => q,

(w, q) => new

{

id = w,

name = string.Format("{0} ", q),

});

foreach (var item in sometype)

Console.WriteLine(item);

return;

}

}

static class Reflector

{

static string \_fileName = "text.txt";

public static void GetAssemblyName(Type type)

{

string str = "Assembly name: " + type.Assembly.ToString() + '\n';

File.AppendAllText(\_fileName, str);

}

public static void PublicConstructorsExist(Type type)

{

string str = "This type ";

bool found = false;

var constructors = type.GetConstructors();

foreach (var c in constructors)

{

if (c.IsPublic)

{

str += "has ";

break;

}

}

if (!found)

{

str += "doesn't have ";

}

str += "public constructors\n";

File.AppendAllText(\_fileName, str);

}

public static void GetPublicMethods(Type type)

{

var methods = type.GetMethods();

foreach (var m in methods)

{

if (m.IsPublic) File.AppendAllText(\_fileName, "Public method " + m.Name + '\n');

}

}

public static void GetFields(Type t)

{

var fields = t.GetFields();

var properties = t.GetProperties();

foreach (var f in fields)

{

File.AppendAllText(\_fileName, "Field " + f.Name + '\n');

}

foreach (var p in properties)

{

File.AppendAllText(\_fileName, "Property " + p.Name + '\n');

}

}

public static void GetInterfaces(Type t)

{

var interfaces = t.GetInterfaces();

foreach (var i in interfaces)

{

File.AppendAllText(\_fileName, "Interface " + i.Name + '\n');

}

}

public static void GetMethodsByParameter(Type t, Type p)

{

var methods = t.GetMethods();

foreach (var m in methods)

{

var pars = m.GetParameters();

foreach (var parm in pars)

{

if (parm.ParameterType == p)

{

File.AppendAllText(\_fileName, "Method with parameter " + p.Name + ": " + m.Name + "\n");

break;

}

}

}

}

public static void Invoke(A obj, string methodName, string[] pars)

{

Type type = obj.GetType();

var method = type.GetMethod(methodName);

string p1 = pars[0];

int p2 = int.Parse(pars[1]);

object[] parameters = new object[2];

parameters[0] = p1;

parameters[1] = p2;

method?.Invoke(obj, parameters);

}

public static T Create<T>(Type type) where T : class

{

object newObj = Activator.CreateInstance(type);

T ret = newObj as T;

return ret;

}

}

class A

{

public static void B(string p1, int p2)

{

Console.WriteLine($"type: {p1.GetType()} value: {p1}");

Console.WriteLine($"type: {p2.GetType()} value: {p2}");

}

public string str;

public A()

{

str = "qwerty";

}

}

class **ProgramReflection**

{

static string GenerateString()

{

string source = "qwertyuiopasdfghjklzxcvbnmQWERTYUIOPASDFGHJKLZXCVBNM 1234567890";

var rand = new Random();

int length = rand.Next(1, 20);

string result = "";

for (int i = 0; i < length; i++)

{

result += source[rand.Next(0, source.Length)];

}

return result;

}

static void Main()

{

Type type = typeof(string);

Reflector.PublicConstructorsExist(type);

Reflector.GetAssemblyName(type);

Reflector.GetPublicMethods(type);

Reflector.GetFields(type);

Reflector.GetInterfaces(type);

Reflector.GetMethodsByParameter(type, typeof(string));

//Task1-g

A a = new();

var pars = File.ReadAllLines("params.txt");

Reflector.Invoke(a, "B", pars);

pars[0] = GenerateString();

Random random = new Random();

pars[1] = random.Next().ToString();

Reflector.Invoke(a, "B", pars);

//Task2

Type typePlant = typeof(Plant);

Type typeCCollection = typeof(CCollection);

Reflector.PublicConstructorsExist(typePlant);

Reflector.GetPublicMethods(typeCCollection);

//Task3

A a1 = new();

Console.WriteLine($"a1: {a1.str}");

A a2 = Reflector.Create<A>(typeof(A));

Console.WriteLine($"a2: {a2.str}");

}

}

using System.IO.Compression;

using static System.Net.Mime.MediaTypeNames;

using System.Text;

class UserException : Exception

{

public string \_message;

public UserException(string message)

{

\_message = message;

}

}

static class TOVLog

{

static public void Write(string str)

{

File.AppendAllText("TOVlogfile.txt", $"{DateTime.Now}, {str}, " +

$"{new FileInfo("TOVlogfile.txt").FullName}\n");

}

}

static class TOVDiskInfo

{

static DriveInfo[] allDrives = DriveInfo.GetDrives();

static public void Write()

{

TOVLog.Write("Written disk info");

foreach (var d in allDrives)

{

Console.WriteLine("Drive name: {0}", d.Name);

Console.WriteLine("Drive type: {0}", d.DriveType);

if (!d.IsReady) continue;

Console.WriteLine("Volume Label: {0}", d.VolumeLabel);

Console.WriteLine("File system: {0}", d.DriveFormat);

Console.WriteLine("Root: {0}", d.RootDirectory);

Console.WriteLine("Total size: {0}", d.TotalSize);

Console.WriteLine("Free size: {0}", d.TotalFreeSpace);

Console.WriteLine("Available: {0}", d.AvailableFreeSpace);

}

}

}

static class TOVDirInfo

{

static public void Write(string path)

{

if (!Directory.Exists(path)) throw new UserException($"Directory {path} does not exist");

TOVLog.Write("Written directory info");

if (Directory.Exists(path))

{

Console.WriteLine("SubDir:");

string[] dirs = Directory.GetDirectories(path);

Console.WriteLine($"Directories amount: {dirs.Length}");

foreach (string s in dirs)

{

Console.WriteLine(s);

}

Console.WriteLine();

string[] files = Directory.GetFiles(path);

Console.WriteLine($"Amount of files: {files.Length}");

Console.WriteLine($"Created: {Directory.GetCreationTime(path)}");

Console.WriteLine("Files:");

foreach (string s in files)

{

Console.WriteLine(s);

}

}

}

}

static class TOVFileInfo

{

static public void Write(string path)

{

if (!File.Exists(path)) throw new UserException($"File {path} does not exist");

TOVLog.Write("Written file info");

FileInfo f = new FileInfo(path);

if (!f.Exists) throw new UserException($"File {path} doesn't exist");

Console.WriteLine($"File name: {f.FullName}");

Console.WriteLine($"Size: {f.Length}");

Console.WriteLine($"Extension: {f.Extension}");

Console.WriteLine($"Path: {f.DirectoryName}");

Console.WriteLine($"Created: {f.CreationTime}");

Console.WriteLine($"Edited: {f.LastWriteTime}");

}

}

static class TOVFileManager

{

static public void TaskAFirstHalf(string driveName)//имя диска задавать как С:\

{

TOVLog.Write($"Written disk {driveName} info");

var allDrives = DriveInfo.GetDrives();

bool driveFound = false;

foreach (var d in allDrives)

{

if (d.Name == driveName)

{

Console.WriteLine("папки");

foreach (var p in Directory.GetDirectories(driveName))

{

Console.WriteLine(p);

}

Console.WriteLine("файлы");

foreach (var p in Directory.GetFiles(driveName))

{

Console.WriteLine(p);

}

driveFound = true;

break;

}

}

if (!driveFound) throw new UserException($"Drive {driveName} does not exist");

}

static public void TaskASecondHalf()//работает

{

TOVLog.Write($"Created folder and file, copied, moved and deleted file");

if (Directory.Exists("TOVInspect")) throw new UserException($"Folder TOVInspect already exists");

Directory.CreateDirectory("TOVInspect");

File.WriteAllText("TOVdirinfo.txt", Directory.GetLastAccessTime("TOVInspect").ToString());

if (File.Exists("TOVdirinfoCOPY")) throw new UserException($"File TOVdirinfoCOPY.txt already exists");

File.Copy("TOVdirinfo.txt", "TOVdirinfoCOPY.txt");

File.Move("TOVdirinfoCOPY.txt", "TOVdirinfoRENAMED.txt");

File.Delete("TOVdirinfo.txt");

}

static public void TaskB(string path, string ext)//точно работает, но расширение должно быть с точкой

{

TOVLog.Write($"Copied files with extension {ext} to {path}");

if (Directory.Exists("TOVFiles")) throw new UserException($"Folder TOVFiles already exists");

Directory.CreateDirectory("TOVFiles");

IEnumerable<string> files = from f in Directory.GetFiles(path)

where new FileInfo(f).Extension ==

ext

select f; ;

foreach (var f in files)

{

FileInfo fi = new FileInfo(f);

if (File.Exists(@$"TOVFiles\{fi.Name}")) throw new UserException($"File TOVFiles\\{fi.Name} " +

$"already exists");

File.Copy($"{f}", Path.Combine("TOVFiles", fi.Name));

}

Directory.Move("TOVFiles", @"TOVInspect\TOVFiles");

}

static public void TaskC()//работает

{

TOVLog.Write($"Created zip-archive");

if (File.Exists("zip.zip")) throw new UserException("zip.zip already exists");

ZipFile.CreateFromDirectory(@"TOVInspect\TOVFiles", "zip.zip");

ZipFile.ExtractToDirectory("zip.zip", "new unzipped");

}

static bool Predicate(DateTime dt, string method = "year", bool interval = false, int start = 0, int end = 0)

{

int dtValue = 0;

switch (method)

{

case "year":

{

dtValue = dt.Year;

break;

}

case "month":

{

dtValue = dt.Month;

break;

}

case "day":

{

dtValue = dt.Day;

break;

}

case "hour":

{

dtValue = dt.Hour;

break;

}

case "minute":

{

dtValue = dt.Minute;

break;

}

case "second":

{

dtValue = dt.Second;

break;

}

default: break;//вставить новое исключение

}

if (interval) return (dtValue <= end && dtValue >= start);

else return (dtValue == start);

}

static public void Task07(string method = "year", bool interval = false, int start = 0, int end = 0)

{

TOVLog.Write($"Written lines based on {method} {start}");

if (!File.Exists("TOVlogfile.txt")) throw new UserException("Task07: TOVlogfile does not exist");

IEnumerable<string> lines = from l in File.ReadAllLines("TOVlogfile.txt")

where Predicate(DateTime.Parse(l.Substring(0, 19)), method, interval,

start, end)

select l;

IEnumerable<string> linesOfThisHour = from l in File.ReadAllLines("TOVlogfile.txt")

where Predicate(DateTime.Parse(l.Substring(0, 19)), "hour", false,

DateTime.Now.Hour)

select l;

Console.WriteLine("Task07 lines:");

foreach (string line in lines) { Console.WriteLine(line); }

Console.WriteLine("Lines of this hour:");

foreach (string line in linesOfThisHour) { Console.WriteLine(line); }

}

}

class **ProgramStreamClasses**

{

const string FIO = "TOV";

static void ClassNameStartsWithFIOCheck(Type type)

{

if (type.Name.Substring(0, 3) == FIO) throw new UserException("Class name doesn't start with FIO");

}

static void Main()//вроде пашет

{

try

{

TOVDirInfo.Write("new folder");

TOVDiskInfo.Write();

TOVFileInfo.Write("TOVlogfile.txt");

TOVFileManager.TaskAFirstHalf("D:\\");

TOVFileManager.TaskASecondHalf();

TOVFileManager.TaskB("new folder", ".txt");

TOVFileManager.TaskC();

Console.WriteLine("\nTask07");

TOVFileManager.Task07("day", false, 25);

}

catch (UserException e)

{

Console.WriteLine("ERROR: " + e.\_message);

}

}

}

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Text.Json;

using System.Threading.Tasks;

using System.IO;

using System.Runtime.Serialization;

using System.Runtime.Serialization.Formatters.Binary;

using System.Runtime.Serialization.Formatters.Soap;

using System.Xml.Serialization;

using System.Xml;

using System.Xml.Linq;

using System.Runtime.Serialization.Json;

using System.Text.Json.Serialization;

[AttributeUsage(AttributeTargets.Class)]

class AgeValidationAttribute : Attribute

{

public int Age { get;}

public AgeValidationAttribute() { }

public AgeValidationAttribute(int age) => Age = age;

}

[AgeValidation(18)]//18-это возраст атрибут же тоже что имеет(public AgeValidationAttribute(int age) => Age = age;

public class Person

{

public string Name { get;}

public int Age { get; set; }

public Person(string name, int age)

{

Name = name;

Age = age;

}

}

[DataContract]

[Serializable]

public class Ship

{

[DataMember]

public string \_Name { get; set; } = "DefaultName";

public override string ToString()

{

return $"Ship {\_Name}";

}

public Ship()

{

\_Name = "DefaultName";

}

}

[DataContract]

[Serializable]

public class Steamvessel : Ship

{

[DataMember]

public int \_Velocity { get; set; } = -1;

[XmlIgnore]

[JsonIgnore]

[field: NonSerialized]

public int \_PassengersCapacity { get; set; } = -1;

public Steamvessel(string name)

{

var rand = new Random();

\_Name = name;

\_Velocity = rand.Next(1, 100);

\_PassengersCapacity = rand.Next(1, 1000);

}

public Steamvessel()

{

var rand = new Random();

\_Name = "DefaultName";

\_Velocity = rand.Next(1, 100);

\_PassengersCapacity = rand.Next(1, 1000);

}

public override string ToString()

{

return $"Steamvessel {\_Name}, velocity: {\_Velocity}, capacity: {\_PassengersCapacity}";

}

}

class **ProgramSerialization**

{

static void Write(Steamvessel? obj, string str)

{

Console.WriteLine($"{str} {obj.\_Name}, скорость: {obj.\_Velocity}, " +

$"кол-во пассажиров: {obj.\_PassengersCapacity}");

}

static void Main()

{

//бинарный

Console.WriteLine("Binary");

//ser

BinaryFormatter binFormatter = new();

var Titanic = new Steamvessel("Titanic") { \_PassengersCapacity = 999, \_Velocity = 9 };

Write(Titanic, "было");

using (FileStream stream = new("ship.dat", FileMode.OpenOrCreate))

{

binFormatter.Serialize(stream, Titanic);

}

//deser

using (FileStream stream = new("ship.dat", FileMode.OpenOrCreate))

{

Steamvessel someShip = (Steamvessel)binFormatter.Deserialize(stream);

Write(someShip, "стало");

}

File.Delete("ship.dat");

//soap

Console.WriteLine("\nSOAP");

SoapFormatter soapFormatter = new();

//ser

using (FileStream stream = new("ship.soap", FileMode.OpenOrCreate))

{

//Steamvessel Olympia = new Steamvessel("Olympia") { \_PassengersCapacity = 999, \_Velocity = 9 };

Write(Titanic, "было");

soapFormatter.Serialize(stream, Titanic);

}

//deser

using (FileStream stream = new("ship.soap", FileMode.OpenOrCreate))

{

//Steamvessel someShip = (Steamvessel)soapFormatter.Deserialize(stream);

var someShip = (soapFormatter.Deserialize(stream) as Steamvessel)!;

Write(someShip, "стало");

}

File.Delete("ship.soap");

//xml

Console.WriteLine("\nXML");

Write(Titanic, "было");

XmlSerializer xs = new(typeof(Steamvessel));

using (StreamWriter sw = new("ship.xml"))

{

xs.Serialize(sw, Titanic);

}

using (StreamReader sr = new("ship.xml"))

{

var someShip = (xs.Deserialize(sr) as Steamvessel)!;

Write(someShip, "стало");

}

File.Delete("ship.xml");

//json

Console.WriteLine("\nJSON");

using (FileStream fs = new("ship.json", FileMode.OpenOrCreate))

{

JsonSerializer.Serialize(fs, Titanic);

Write(Titanic, "было");

}

// чтение данных

using (FileStream fs = new("ship.json", FileMode.OpenOrCreate))

{

Steamvessel? someShip = JsonSerializer.Deserialize<Steamvessel>(fs);

Write(someShip, "стало");

}

File.Delete("ship.json");

//collection

Console.WriteLine("\nCollection");

XmlSerializer xmlSerializer = new(typeof(List<Steamvessel>));

List<Steamvessel> xmlList = new();

xmlList.Add(new Steamvessel("Adm. Kuznetsov"));

xmlList.Add(new Steamvessel("Moskva"));

xmlList.Add(new Steamvessel("Gnevny"));

foreach (var s in xmlList) Write(s, "было");

//ser

using (FileStream topStream = new("list.xml", FileMode.Create))

{

xmlSerializer.Serialize(topStream, xmlList);

}

//deser

using (FileStream topStream = new("list.xml", FileMode.OpenOrCreate))

{

List<Steamvessel> someShips = (List<Steamvessel>)xmlSerializer.Deserialize(topStream)!;

foreach (var item in someShips)

{

Write(item, "стало");

}

}

//XPath

Console.WriteLine("\nXPath");

XmlDocument xmlDocXPath = new();

xmlDocXPath.Load("list.xml");

XmlNode xmlRoot = xmlDocXPath.SelectSingleNode("ArrayOfSteamvessel")!;

XmlNodeList nodeSteamvessel = xmlRoot.SelectNodes("Steamvessel")!;

foreach (XmlNode item in nodeSteamvessel)

{

Console.WriteLine($"{item.SelectSingleNode("\_Name")!.InnerText} " +

$"{item.SelectSingleNode("\_Velocity")!.InnerText}");

}

//Linq to XML

Console.WriteLine("\nLINQ to XML:");

XElement NewTank(string attrValue, string childElemValue)

{

XElement ret = new("TANK");

ret.Add(new XAttribute("model", attrValue));

ret.Add(new XElement("PRICE", childElemValue));

return ret;

};

XDocument xmlDoc = new();

XElement tanks = new("TANK\_PARK");

tanks.Add(NewTank("Maus", "999999989"));

tanks.Add(NewTank("Pz. Kpfw. 3E", "1000"));

tanks.Add(NewTank("T-62A", "223435"));

xmlDoc.Add(tanks);

xmlDoc.Save("tanksLinq.xml");

Console.WriteLine("Tanks more expensive than 1000 money:");

XDocument loadedDoc = XDocument.Load("tanksLinq.xml");

var tanksList = from t in loadedDoc.Descendants("TANK")

where int.Parse(t.Element("PRICE")!.Value) > 1000

select t.Attribute("model")!.Value;

foreach (var t in tanksList) { Console.WriteLine(t); }

}

}

class **ProgramPatterns**

{

static void Main()

{

try

{

StudentPrototype prototype = new ConcreteStudent("Oleg", 1111);

StudentPrototype clone = prototype.Clone();

TestFactory factory = new();

BuilderMath bm = new();

BuilderScience bs = new();

BuilderEnglish be = new();

Tutor.GetInstance().testList.Add(factory.CreateTest(bm));

Tutor.GetInstance().testList.Add(factory.CreateTest(bs));

Tutor.GetInstance().testList.Add(factory.CreateTest(be));

BuilderDelete deletor = new();

BuilderEdit editor = new();

BuilderPrint printer = new();

printer.DoSomething(-1, null, Tutor.GetInstance());

Context context = new(new Strategy2());

//creator.DoSomething(-1, mtf.CreateTest(), Tutor.GetInstance());

//creator.DoSomething(-1, stf.CreateTest(), Tutor.GetInstance());

//creator.DoSomething(-1, etf.CreateTest(), Tutor.GetInstance());

context.\_strategy.Execute(new(), Tutor.GetInstance());

TestHistory history = new();

history.History.Push(Tutor.GetInstance().testList[1].Savestate());

//deletor.DoSomething(-1, null, Tutor.GetInstance());

editor.DoSomething(1, new Test("some class", new List<QuestionDecorator>()), Tutor.GetInstance());

printer.DoSomething(-1, null, Tutor.GetInstance());

Tutor.GetInstance().testList[1].Restate(history.History.Pop());

printer.DoSomething(-1, null, Tutor.GetInstance());

Student student = new("dafault", 1111);

AccurateStudent accStudent = new("accurate student", 0000);

bool studentRegistered = false;

bool enteredSystem = false;

string input;

return;

}

catch (UserException e)

{

Console.WriteLine(e.\_message);

}

}

}

abstract class StudentPrototype

{

public string \_name;

public int \_password;

private Random random = new();

public StudentPrototype(string name, int password)

{

\_name = name;

\_password = password;

}

public abstract StudentPrototype Clone();

public void StartTesting(Tutor tutor)

{

List<double> grades = new();

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

{

double grade = 0;

for (int j = 0; j < tutor.testList[i].\_count; j++)

{

grade += AnswerQuestion(tutor, i, j);

}

grades.Add(grade / tutor.testList[i].\_count);

}

Console.Write("Grades: ");

foreach (var g in grades) Console.Write($"{g} ");

Console.WriteLine();

}

private double AnswerQuestion(Tutor tutor, int testIndex, int questionIndex)

{

int numOfAnswers = tutor.testList[testIndex].\_questions[questionIndex].\_numOfAnswers;

int numOfRightAnswers = tutor.testList[testIndex].\_questions[questionIndex].\_numOfRightAnswers;

List<int> answers = new();

List<int> availableAnswers = new();

for (int i = 0; i < numOfAnswers; i++) availableAnswers.Add(i);

for (int i = 0; i < numOfRightAnswers; i++)

{

int answer = random.Next(0, availableAnswers.Count);

answers.Add(availableAnswers[answer]);

availableAnswers.RemoveAt(answer);

}

return tutor.testList[testIndex].\_questions[questionIndex].GetGrade(answers);

}

}

class ConcreteStudent : StudentPrototype

{

public ConcreteStudent(string name, int password) : base(name, password) { }

public override StudentPrototype Clone()

{

return new ConcreteStudent(\_name, \_password);

}

}

interface IAbstractFactory

{

public Test CreateTest(Builder builder);

}

abstract class Builder

{

public string Class;

}

class BuilderCreate

{

public void DoSomething(int index, Test test, Tutor tutor)

{

tutor.testList.Add(test);

}

}

class BuilderMath : Builder

{

public string Class = "math";

}

class BuilderScience : Builder

{

public string Class = "science";

}

class BuilderEnglish : Builder

{

public string Class = "english";

}

class BuilderDelete

{

public void DoSomething(int index, Test test, Tutor tutor)

{

if (index > 0 && index < tutor.testList.Count)

{

tutor.testList.RemoveAt(index);

}

else throw new UserException("such index does not exist");

}

}

class BuilderPrint

{

public void DoSomething(int index, Test test, Tutor tutor)

{

foreach (var t in tutor.testList)

{

Console.Write($"Test {t.\_class}:\n");

foreach (var q in t.\_questions)

{

Console.WriteLine($"question: {q.\_text}, noa: {q.\_numOfAnswers}, nora: {q.\_numOfRightAnswers}");

}

}

}

}

class BuilderEdit

{

public void DoSomething(int index, Test test, Tutor tutor)

{

if (index > 0 && index < tutor.testList.Count)

{

tutor.testList[index] = test;

}

else throw new UserException("such index does not exist");

}

}

class TestFactory : IAbstractFactory

{

public Test CreateTest(Builder builder)

{

var random = new Random();

int numOfAnswers = 4;

List<QuestionDecorator> questions = new();

for (int j = 0; j < 3; j++)

{

int numOfRightAnswers = random.Next(1, numOfAnswers);

List<int> availableAnswers = new();

List<int> numsOfRightAnswers = new();

for (int k = 0; k < numOfAnswers; k++) availableAnswers.Add(k);

for (int k = 0; k < numOfRightAnswers; k++)

{

int answer = random.Next(0, availableAnswers.Count);

numsOfRightAnswers.Add(availableAnswers[answer]);

availableAnswers.RemoveAt(answer);

}

questions.Add(new("default text", numOfAnswers, numsOfRightAnswers));

}

return (new(builder.Class, questions));

}

}

//Strategy

using System.Collections;

interface IStrategy

{

public void Execute(Student student, Tutor tutor);

}

class Context

{

public IStrategy \_strategy;

public Context(IStrategy strategy)

{

\_strategy = strategy;

}

public Context()

{

\_strategy = new Strategy1();

}

}

class Strategy1 : IStrategy

{

public void Execute(Student student, Tutor tutor)

{

student.StartTesting(tutor);

}

}

class Strategy2 : IStrategy

{

public void Execute(Student student, Tutor tutor)

{

if (tutor.testList.Count == 0) return;

List<double> grades = new();

double grade = 0;

Aggregate aggregate = new(tutor.testList[0].\_questions);

Iterator iterator = aggregate.CreateIterator();

QuestionDecorator q = (QuestionDecorator)iterator.First();

Console.WriteLine($"Iterator start:\n{q.\_text}");

while (!iterator.IsDone())

{

q = (QuestionDecorator)iterator.Next();

Console.WriteLine(q.\_text);

}

for (int j = 0; j < tutor.testList[0].\_count; j++)

{

grade += Student.AnswerQuestion(tutor, 0, j);

}

grades.Add(grade / tutor.testList[0].\_count);

Console.Write("Grades: ");

foreach (var g in grades) Console.Write($"{g} ");

Console.WriteLine();

}

}

//Memento

public class TestMemento

{

public string \_class { get; private set; }

public List<QuestionDecorator> \_questions { get; private set; }

public int \_count { get; private set; }

public TestMemento(string @class, List<QuestionDecorator> questions)

{

\_class = @class;

\_questions = questions;

\_count = questions.Count;

}

}

class TestHistory

{

public Stack<TestMemento> History { get; private set; }

public TestHistory()

{

History = new Stack<TestMemento>();

}

}

//Iterator

class Aggregate

{

private readonly List<QuestionDecorator> \_items = new();

public Aggregate(List<QuestionDecorator> items)

{

\_items = items;

}

public Iterator CreateIterator()

{

return new Iterator(this);

}

public int Count

{

get { return \_items.Count; }

}

public object this[int index]

{

get { return \_items[index]; }

}

}

class Iterator

{

private readonly Aggregate \_aggregate;

private int \_current;

public Iterator(Aggregate aggregate)

{

this.\_aggregate = aggregate;

}

public object First()

{

return \_aggregate[0];

}

public object Next()

{

object ret = null;

\_current++;

if (\_current < \_aggregate.Count)

{

ret = \_aggregate[\_current];

}

return ret;

}

public object CurrentItem()

{

return \_aggregate[\_current];

}

public bool IsDone()

{

return \_current >= \_aggregate.Count - 1;

}

}

public sealed class Tutor

{

private Random random = new();

public List<Test> testList;

private static readonly Lazy<Tutor> Lazy = new(() => new Tutor());

private Tutor() { testList = new(); }

public static Tutor GetInstance() => Lazy.Value;

}

public class Test

{

public string \_class;

public List<QuestionDecorator> \_questions;

public int \_count;

public Test(string @class, List<QuestionDecorator> questions)

{

\_class = @class;

\_questions = questions;

\_count = questions.Count;

}

public TestMemento Savestate()

{

return new TestMemento(\_class, \_questions);

}

public void Restate(TestMemento tm)

{

\_class = tm.\_class;

\_questions = tm.\_questions;

\_count= tm.\_count;

}

}

class Student

{

public string \_name;

public int \_password;

public static Random random = new();

public Student(string name, int password)

{

\_name = name;

\_password = password;

}

public Student() { }

public void StartTesting(Tutor tutor)

{

List<double> grades = new();

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

{

double grade = 0;

for (int j = 0; j < tutor.testList[i].\_count; j++)

{

grade += AnswerQuestion(tutor, i, j);

}

grades.Add(grade / tutor.testList[i].\_count);

}

Console.Write("Grades: ");

foreach (var g in grades) Console.Write($"{g} ");

Console.WriteLine();

}

public static double AnswerQuestion(Tutor tutor, int testIndex, int questionIndex)

{

int numOfAnswers = tutor.testList[testIndex].\_questions[questionIndex].\_numOfAnswers;

int numOfRightAnswers = tutor.testList[testIndex].\_questions[questionIndex].\_numOfRightAnswers;

List<int> answers = new();

List<int> availableAnswers = new();

for (int i = 0; i < numOfAnswers; i++) availableAnswers.Add(i);

for (int i = 0; i < numOfRightAnswers; i++)

{

int answer = random.Next(0, availableAnswers.Count);

answers.Add(availableAnswers[answer]);

availableAnswers.RemoveAt(answer);

}

return tutor.testList[testIndex].\_questions[questionIndex].GetGrade(answers);

}

}

class AccurateStudent : Student

{

public AccurateStudent(string name, int password) : base(name, password) { }

public void StartTesting(Tutor tutor)

{

List<double> grades = new();

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

{

double grade = 0;

for (int j = 0; j < tutor.testList[i].\_count; j++)

{

Console.WriteLine($"Answering question {j} of {tutor.testList[i].\_count}");

grade += AnswerQuestion(tutor, i, j);

}

grades.Add(grade / tutor.testList[i].\_count);

}

Console.Write("Grades: ");

foreach (var g in grades) Console.Write($"{g} ");

Console.WriteLine();

}

}

public abstract class AbstractQuestion

{

public abstract double GetGrade(List<int> answers);

}

public class Question : AbstractQuestion

{

public string \_text;

public int \_numOfAnswers;

public int \_numOfRightAnswers;

public List<int> \_numsOfRightAnswers;

public Question(string text, int numOfAnswers, List<int> numsOfRightAnswers)

{

\_text = text;

\_numOfAnswers = numOfAnswers;

\_numOfRightAnswers = numsOfRightAnswers.Count;

\_numsOfRightAnswers = numsOfRightAnswers;

}

public Question() { }

public override double GetGrade(List<int> answers)

{

int rightAnswers = 0;

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

{

if (\_numsOfRightAnswers.Contains(answers[i])) rightAnswers++;

}

return rightAnswers / \_numOfRightAnswers;

}

}

public class QuestionDecorator :AbstractQuestion

{

protected Question question;

public string \_text

{

get

{

return question.\_text;

}

set

{

question.\_text = value;

}

}

public int \_numOfAnswers

{

get

{

return question.\_numOfAnswers;

}

set

{

question.\_numOfAnswers = value;

}

}

public int \_numOfRightAnswers

{

get

{

return question.\_numOfRightAnswers;

}

set

{

question.\_numOfRightAnswers = value;

}

}

private List<int> \_numsOfRightAnswers

{

get

{

return question.\_numsOfRightAnswers;

}

set

{

question.\_numsOfRightAnswers = value;

}

}

public QuestionDecorator(string text, int numOfAnswers, List<int> numsOfRightAnswers)

{

question = new(text, numOfAnswers, numsOfRightAnswers);

}

public QuestionDecorator()

{

question = new();

}

public override double GetGrade(List<int> answers)

{

if (question != null)

{

return question.GetGrade(answers);

}

else return 0;

}

}