МIНIСТЕРСТВО ОСВIТИ І НАУКИ УКРАЇНИ

ХАРКIВСЬКИЙ НАЦІОНАЛЬНИЙ

УНIВЕРСИТЕТ РАДIОЕЛЕКТРОНIКИ

Методичні вказівки

до лабораторних робіт з дисципліни

«СИСТЕМНЕ ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ»

для студентiв усіх форм навчання

напряму 6.050102 «Комп’ютерна інженерія»

Електронне видання

Затверджено

кафедрою ЕОМ

Протокол № \_\_\_ від \_\_\_\_\_\_\_\_\_

ХАРКIВ 2017

Методичні вказівки до лабораторних робіт з дисципліни «Системне програмне забезпечення» для студентів усіх форм навчання напряму 6.050102 «Комп’ютерна інженерія» [Електронне видання] / Упоряд. О.О. Фомічов. – Харків: ХНУРЕ, 2017. – 82 с.

Упорядник О.О. Фомічов

**ЛАБОРАТОРНА РОБОТА №1**

**ОЗНАЙОМЛЕННЯ З ОСНОВНИМИ ОСОБЛИВОСТЯМИ**

**МОВИ С# .NET**

* 1. **Мета роботи**

Ознайомлення з основними особливостями мови С# .NET.

* 1. **Вказівки з підготовки до виконання лабораторної роботи**

**Необхідно вивчити наступні теми:**

* консольний ввід-вивід даних;
* перетворення типів;
* оператори циклу, перебору та умовні оператори;
* конструктори та властивості типів;
* валідація даних і обробка виключень;
* робота з масивами.
  1. **Теоретичні відомості**

Найпростіша програма мовою C# .NET:

using System;namespace ConsoleSample{ class HelloClass { static void Main() { Console.WriteLine(“HelloWorld!”); Console.ReadKey(); } }}

**Консольний ввід-вивід даних:**

using System;namespace ConsoleSample   
{ class HelloClass   
 { static void Main()   
 { Console.Write(“Вывод строки без перевода на следующую”); Console.WriteLine(“Вывод строки с переводом на следующую”); // вывод строки, состоящей из набора аргументов Console.WriteLine(“{0} {1}”, “первый аргумент”,

“ второй аргумент” );

char c = Console.ReadKey(); // чтение введенного символа string str = Console.ReadLine(); // чтение введенной строки

}

}}

**Перетворення типів даних:**

using System;namespace ConsoleSample   
{ class HelloClass   
 { static void Main()   
 { string str = Console.ReadLine(); int i = Convert.ToInt32(str); // преобразование с Convert short x = Convert.ToInt16(str); try { double y = double.Parse(str); // преобразование с Parse() } catch(FromatException) {   
 Console.WriteLine(“Ошибка преобразования типов”); } } }}

**Умовний оператор:**

using System;namespace ConsoleSample{ class HelloClass   
 { static void Main()   
 { int a = 5, b = 6, c = 0; // с принимает значение большего из a и b if(a > b)   
 c = a; else   
 c = b; // тернарный аналог условного оператора c = a > b ? a : b; } }}

**Оператор перебору:**

static void Main()   
{ try { int a = Convert.ToInt32(Console.ReadLine()); switch(a) { // оператор перебора введенного значения case 0: case 1: Console.WriteLine(“Введен 0 или 1”); break; case 2: Console.WriteLine(“Введена 2”); break; default: Console.WriteLine(“Введено другое значение”); break;}} catch(FormatException) { // обработка ошибок преобразования Console.WriteLine(“Ошибка преобразования типов”); }

}} **Цикли та масиви:**

static void Main(){

int[] array = new int[5];

Random r = new Random();

for(int i=0; i < array.Length; ++i){// заполнение элементов массива случайными числами в диапазоне от 1 до 25array[i] = r.Next(1, 25);}int sum = 0;foreach(int item in array){sum += item; // определение суммы всех элементов массив массива}}

**Основні елементи класів:**

* поля – дані, що містяться в об’єктах класу;
* властивості – функціональні елементи, які надають доступ до полів класу;
* конструктори – методи, які викликаються при створенні об’єктів;
* методи – функції, що працюють з іншими членами класів;
* оператори – перевантажена елементарна дія з об’єктами даного класу. Наприклад: +, -, >=, == і т.і.

**Модифікатори доступу:**

* public – усі члени класу із даним модифікатором можна викликати у будь-якій частині програми;
* private – усі члени класу із даним модифікатором можна викликати тільки всередині даного класу;
* protected – усі члени класу із даним модифікатором можна викликати всередині даного класу і всередині всіх його дочірніх класів (нащадків);
* internal – обмежує можливості звернення до класу межами збірки, де він був оголошений.

**Поля і властивості у класах:**

class имя\_класса  
{  
тип\_доступа тип\_поля имя\_поля1; // объявление полей  
тип\_доступа тип\_поля имя\_поля2;

тип\_доступа тип\_свойства имя\_свойства1 // объявление свойств

{

get { return имя\_поля1; }

set { имя\_поля1 = value; }

}

тип\_доступа тип\_свойства имя\_свойства2  
{  
get { return имя\_поля2; }  
}

тип\_доступа тип\_свойства имя\_свойства3 { get; set; } // автосвойство  
}

**Методи і оператори класів:**

class имя\_класса{тип\_доступа имя\_класса() { // конструктор по-умолчаниюимя\_поля1 = 0;имя\_поля2 = “Строка”;}// инициализирующий конструктортип\_доступа имя\_класса(тип\_аргумента имя\_аргумента1, тип\_аргумента имя\_аргумента2) {имя\_поля1 = имя\_аргумента1;имя\_поля2 = имя\_аргумента2; }// копирующий конструктортип\_доступа имя\_класса(имя\_класса имя\_аргумента) {имя\_поля1 = имя\_аргумента.имя\_свойства1;имя\_поля2 = имя\_аргумента.имя\_свойства2;имя\_свойства3 = имя\_аргумента.имя\_свойства3; }}

* 1. **Порядок виконання роботи і методичні вказівки з її виконання.**

**Варіант 1. Кавовий автомат.**

У проекті передбачити розділення інтерфейсу та логіки типів, що створювалися під час виконання завдання. Передбачити створення виключень під час введення некоректних значень.

**Дані типу:**

* назва моделі автомату: не порожній рядок, немає пробілів;
* поточна кількість кави, молока, води та цукру;
* максимальна кількість кави, молока, води та цукру;
* каса автомату: позитивне дійсне число.

**Методи типу:**

* підготовка напою: реалізувати підтримку 4-5 типів напою (визначається перерахуванням і має вартість: тип напою і його вартість є аргументами методу);
* підготовка кількох напоїв: сукупність напоїв та оплата за них є вхідними аргументами;
* поповнення запас кави/молока/води/цукру;
* повернення каси автомату;
* вивід повідомлення при недостатній кількості кави/молока/води/цукру.

**Варіант 2. Банківський рахунок.**

У проекті передбачити розділення інтерфейсу та логіки типів, що створювалися під час виконання завдання. Передбачити створення виключень під час введення некоректних значень.

**Дані типу:**

* ПІБ власника: не може бути порожнім, тільки літери;
* номер рахунку: 8 символів, тільки цифри, немає пробілів;
* масив депозитів: депозит – перерахунок із 3-4 найменувань із різною відсотковою ставкою;
* сума на рахунку: тільки дійсні числа.

**Методи типу:**

* ініціалізуючий конструктор;
* відкриття депозиту: додавання нового депозиту до масива;
* нарахування відсотків по депозиту за рік;
* поповнення рахунку;
* зняття коштів з рахунку;
* розірвання виділеного депозиту: видалення депозиту;
* визначення загальної кількості коштів на депозитах;
* перевизначення методу ToString.

**Варіант 3. Профіль студента університету**

У проекті передбачити розділення інтерфейсу та логіки типів, що створювалися під час виконання завдання. Передбачити створення виключень під час введення некоректних значень.

**Дані типу:**

* ПІБ студента: не може бути порожнім рядком або містити цифри;
* номера залікової книжки: не може містити пробіли;
* курс: тільки ціле число;
* середній бал: дійсне число, залежить від оцінок з предметів;
* оцінки за 10 предметів за 100-бальною шкалою.

**Методи типу:**

* ініціалізуючий конструктор;
* встановлення оцінки за індексом предмета;
* отримання значення оцінки за значенням предмета: всі предмети представляються у перерахуванні з 10 елементів, у масиві предметів не може бути однакових елементів;
* перевизначення методу ToString;
* порівняння 2 студентів за оцінками: у результаті порівняння повертається масив різниці балів у предметах.

**Варіант 4. Співробітник підприємства**

У проекті передбачити розділення інтерфейсу та логіки типів, що створювалися під час виконання завдання. Передбачити створення виключень під час введення некоректних значень.

**Дані типу:**

* ПІБ не може бути порожнім рядком, не може містити цифри;
* посада: директор/керівник/інженер/різноробочий;
* заробітна плата: 50000/20000/10000/5000 – у залежності від посади;
* історія кар’єрних змін у компанії;
* назва відділу: містить хоча б одну букву і одну цифру.

**Методи типу:**

* ініціалізуючий конструктор;
* зміна посади;
* інкремент/декремент: підвищення/пониження посади;
* отримання заробітної плати;
* пошук посади/відділу у кар’єрній історії працівника;
* зміна відділу;
* перевантаження методу ToString
* порівняння співробітників за посадою;

**Варіант 5. Рахунок мобільного оператора**

У проекті передбачити розділення інтерфейсу та логіки типів, що створювалися під час виконання завдання. Передбачити створення виключень під час введення некоректних значень.

**Дані типу:**

* назва тарифу: визначається перерахуванням з 3-4 станів;
* вартість хвилини розмови;
* поточна сума на рахунку;
* номер телефону: містить 10 цифр;
* масив підключення послуги: визначається перерахуванням.

**Методи типу:**

* конструктор по-замовченню;
* ініціалізуючий конструктор;
* перехід на новий тарифний план: тариф є аргументом;
* дзвінок: вхідні аргументи – кількість хвилин та інший номер, вихідне значення – сума, що списана з рахунку за проведену розмову;
* підключення/відключення послуги: послуга має назву і ціну;
* поповнення рахунку;
* отримання поточного стану рахунку/історії дзвінків.

**Варіант 6. Планувальник завдань.**

У проекті передбачити розділення інтерфейсу та логіки типів, що створювалися під час виконання завдання. Передбачити створення виключень під час введення некоректних значень.

**Дані типу:**

* назва програми: не може бути порожнім рядком;
* масив завдань: завдання та їх складність визначаються перерахуванням з 4-5 станів;
* кількість співробітників;
* історія включень планувальника.

**Методи типу:**

* ініціалізуючий конструктор (усі поля типу)
* визначення часу виконання завдань: на основі внутрішніх даних об’єкта про співробітників, кількості і типах завдань;
* відображення історії запуску планувальника;
* очищення списку завдань;
* очищення історії запуску.
  1. **Зміст звіту.**

**Звіт повинен мати:**

* титульний лист із найменуванням лабораторної роботи і даними виконавця;
* мету роботи;
* завдання;
* текст програми з коментарями і поясненнями;
* результат роботи програми;
* висновки.
  1. **Контрольні запитання**

1. Що розуміється під терміном «.NET Framework»?
2. Чи залежать додатки, що розроблюються у «.NET», від платформи?
3. Чи можливо створення гетерогенних додатків у середовище «.NET»?
4. Що означає абревіатура «CLR»?
5. Чи є середа CLR багатомовною?
6. Наведіть узагальнений синтаксис оголошення змінної мовою C#.
7. Наведіть узагальнений синтаксис ініціалізації змінної мовою C#.
8. Яка дисципліна (варіант контролю) типів прийнята у мові C#.
9. Які основні категорії типів існують у мові C#?
10. Перелічіть п’ять простих типів мови C#?
11. Що розуміється під областю видимості змінної у мові C#?
12. Як позначається область видимості змінної у мові C#?
13. Як співвідносяться час життя змінної і область видимості?
14. Наведіть синтаксис умовного оператора у загальному вигляді. Проілюструйте його фрагментом програми мовою C#.
15. Наведіть синтаксис оператора вибору у загальному вигляді. Проілюструйте його фрагментом програми мовою C#.
16. Що розуміється під терміном «простір імен»?
17. У чому призначення простору імен у мові C#?
18. Завдяки якому механізму вдається уникнути колізій імен у мові C#?
19. Який простір імен використовує системна бібліотека .NET Framework?
20. Який простір імен використовує системна бібліотека C#?
21. У чому призначення директиви using?
22. Який символ використовується для вказання повного імені об’єкту у мові C#?
23. Наведіть синтаксис директиви using у загальному вигляді. Проілюструйте його фрагментом програми мовою C#.
24. Наведіть синтаксис опису простору імен у загальному вигляді. Проілюструйте його фрагментом програми мовою C#.

**ЛАБОРАТОРНА РОБОТА №2**

**ОЗНАЙОМЛЕННЯ З ОСНОВАМИ ОБ’ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ У C# ТА ТЕХНОЛГІЄЮ WINDOWS FORMS**

**2.1 Мета роботи**

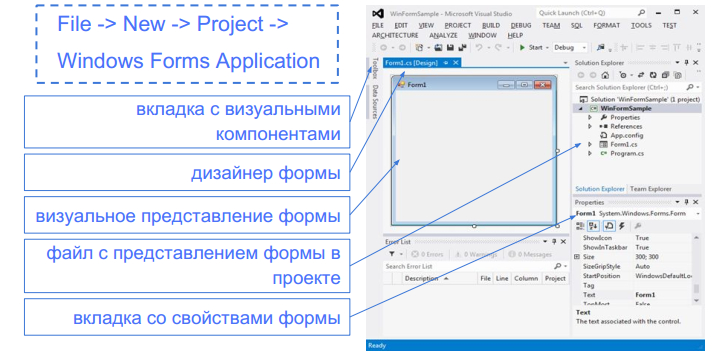
Ознайомлення з основами об’єктно-орієнтованого програмування у C# та технологією WINDOWS FORM.

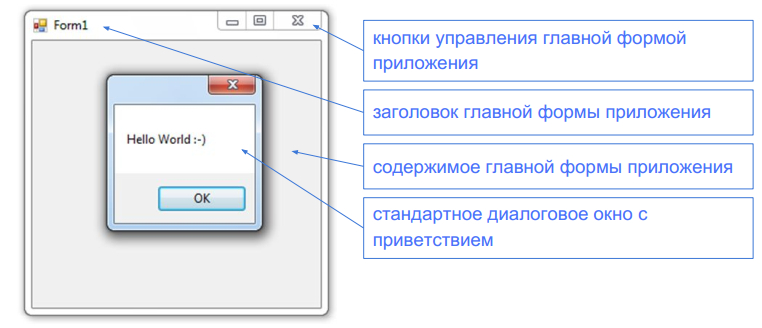
**2.2 Вказівки з підготовки до виконання лабораторної роботи**

Необхідно вивчити наступні теми:

* створення та використання класів;
* поліморфізм;
* перевантаження операторів;
* розробка WINDOWS FORM-додатку.
  1. **Теоретичні відомості**

**Найпростіший додаток на WINDOWS FORM**

****

****

**Program.cs**

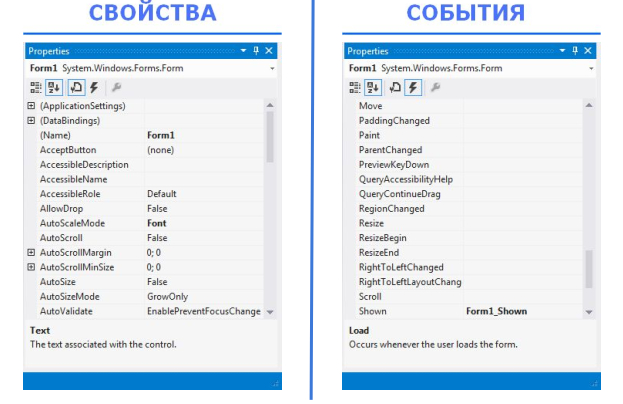
using System;using System.Windows.Forms;namespace WinFormSample{ static class Program { [STAThread] static void Main() { Application.EnableVisualStyles(); Application.SetCompatibleTextRenderingDefault(false); Application.Run(new Form1()); } } }

**Form1.cs**

using System;using System.Windows.Forms;namespace WinFormSample{ public partial class Form1 : Form { public Form1() { InitializeComponent (); } private void Form1\_Shown(object sender, EventArgs e) { MessageBox.Show(“Hello World :-)”); } } }

**Form1.Design.cs**

namespace WinFormSample { partial class Form1 { private System.ComponentModel.IContainer components = null; protected override void Dispose(bool disposing) { if(disposing && (components != null)){ components.Dispose(); } base.Dispose(disposing); } private void InitializeComponent(){ this.SuspendLayout(); this.AutoScaleDimensions = new System.Drawing.SizeF(6F, 13F); this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font; this.ClientSize = new System.Drawing.Size(284, 262); this.Name = "Form1"; this.Text = "Form1"; this.Shown += new System.EventHandler(this.Form1\_Shown); this.ResumeLayout(false); } } }

****

**Поліморфізм у С#**

public class Base { public virtual int Get() { return 1; }}public class First : Base { ... }  
public class Second : Base { public override int Get() { return 100; }}...Base obj = new First();Console.WriteLine(obj.Get()); // result = 1obj = new Second();Console.WriteLine(obj.Get()); // result = 100

* віртуальні методи та властивості можуть мати реалізацію у нащадку;
* абстрактні методи та властивості повинні мати реалізацію у нащадку;
* статичні методи і властивості не можуть бути віртуальними чи абстрактними;
* інтерфейс – чиста абстракція, що не має модифікаторів доступу і реалізації для метода, властивості чи події.

**Індексатори у класах**

Індексатори дозволяють звертатися до будь-якої функціональності об’єкта за індексом, аналогічно тому, як це відбувається у масивах.

public class Point2D { public int X { get; set; } public int Y { get; set; } public int this[int index] { get { if(index == 0) return X; return Y; } set { if(index == 0) X = value; Y = value; } }}

Розширюючі методи

Дозволяють додавати до існуючих типів, у тому числі і зачинені від наслідування, нову функціональність.

public static class RandomExtension{ public static double NextPercent(this Random obj) { return obj.Next(0, 100) + obj.NextDouble(); }}...Random obj = new Random();double x = obj.NextPercent();

* 1. **Порядок виконання роботи і методичні вказівки з її виконання.**

**Варіант 1. «Автомагазин» + віконний додаток.**

**Дані:**

* кількість відділів;
* назва;
* середній прибуток за місяць;
* загальна заробітна плата співробітників;
* витрати на закупку товару у виробників;
* кількість співробітників;
* адреса;
* кількість найменувань товару.

**Методи:**

* визначення прибутку за рік;
* додавання співробітників;
* метод ToString;
* визначення прибутку за місяць;
* звільнення співробітників;
* визначення податку за рік (ставка 17%);
* оператори ++ та – (для кількості співробітників);
* індексатор для прибутку за місяць, заробітної плати співробітників і витрат на закупку товару;
* реалізувати інтерфейс IComparable для порівняння магазинів.

**Варіант 2. «Студентський кампус» + віконний додаток.**

**Дані:**

* кількість кімнат;
* кількість персоналу;
* назва університету;
* кількість студентів;
* оплата за проживання;
* адреса.

**Методи:**

* збільшення кількості кімнат;
* виселення студентів;
* заселення студентів;
* визначення прибутку за певний період;
* ініціалізуючий конструктор;
* метод ToString;
* реалізація ICloneable для створення повної копії об’єкту;
* реалізація розширяючого методу (додавання їдальні до кампусу: +20% до прибутку за будь-який період, +5 осіб персоналу)

**Варіант 3. «Університет» + віконний додаток.**

**Дані:**

* кількість факультетів;
* кількість лабораторій;
* кількість студентів;
* кількість лекційних аудиторій;
* назва;
* кількість викладачів (на 1 викладача максимум по 10 студентів);
* кількість осіб персоналу (на 1 співробітника по 2 аудиторії).

**Методи:**

* зачислення/відрахування студентів;
* додавання ЛК/ЛБ аудиторій;
* конструктор ініціалізуючий;
* конструктор копіюючий;
* наймання/звільнення співробітників (персонал/викладач);
* реалізація інтерфейсу IEqualityComparer<T> (де Т – університет);
* оператор + (злиття 2 університетів);
* реалізація індексатора за кількістю лабораторій та лекційних аудторій.

**Варіант 4. «Завод» + віконний додаток.**

**Дані**:

* назва;
* кількість робочих;
* кількість цехів;
* заробітна плата робочого;
* заробітна плата майстра;
* кількість майстрів (1 майстер на 10 робочіх);
* прибуток за місяць від одного робочого;
* прибуток за місяць від одного майстра

**Методи:**

* наймання/звільнення робочого/майстра;
* копіюючий конструктор;
* визначення загальної заробітної плати усіх співробітників;
* реалізувати оператор + (злиття заводів);
* реалізувати інтерфейс IComparable для порівняння заводів;
* реалізувати розширюючий метод, який визначає прибуток, шо отримується від вкладення деякої суми у розвиток заводу (на наймання співробітників).

**Варіант 5. «Студія звукозапису» + віконний додаток.**

**Дані:**

* кількість співробітників;
* назва та адреса студії;
* вартість запису одного треку;
* час запису одного треку;
* заробітна плата одного співробітника;
* загальна заробітна плата співробітників студії;
* каса студії;
* кількість інструментів у студії;
* кількість кімнат (1 кімната – не менше 2 інструментів та 2 співробітників)

**Методи:**

* додавання/видалення кімнати;
* наймання/звільнення співробітників;
* оператор ++ (додавання кімнати з інструментами за гроші);
* оператор – (видалення кімнати студії без інструментів, безплатно);
* реалізація інтерфейсу ICloneable для створення точної копії об’єкта студії;
* індексатор по заробітній платі одного співробітника, загальної заробітної плати співробітників і касою студії.

**Варіант 6. «Морський порт» + віконний додаток.**

**Дані:**

* назва та адреса порту;
* кількість робочих;
* кількість одиниць техніки;
* вартість одної одиниці техніки;
* вартість обслуговування корабля;
* час обслуговування корабля;
* кількість причалів (на 1 причал – 5 одиниць техніки і 15 робочих).

**Методи:**

* розрахунок часу обслуговування вказаної кількості кораблів (розширюючий метод);
* розрахунок прибутків порту при заданій кількості кораблів;
* ініціалізуючий та копіюючий конструктори;
* оператор ++ (кількість причалів з технікою, без робочих);
* оператори >= та <= (за кількість функціонуючих причалів);
* реалізація інтерфейсу IEqualityComparer<T> (де Т – порт).
  1. **Зміст звіту.**Звіт повинен мати:
* титульний лист із найменуванням лабораторної роботи і даними виконавця;
* мету роботи;
* завдання;
* текст програми з коментарями і поясненнями;
* результат роботи програми;
* висновки.
  1. **Контрольні запитання**

1. Що розуміється під терміном «клас»?
2. Які елементи визначаються у складі класу?
3. Як співвідносяться поняття «клас» і «об’єкт»?
4. Що розуміється під терміном «члени класу»?
5. Які члени класу вам відомі?
6. Які члени класу містять код?
7. Які члени класу містять дані?
8. Перелічіть п’ять різновидів рівнів доступу членів класу специфічних для мови C#.
9. Що розуміється під терміном «конструктор»
10. Скільки конструкторів може містити клас мови C#?
11. Наведіть синтаксис опису класу у загальному вигляді. Проілюструйте його фрагментом програми мовою C#.
12. Які модифікатори типу доступу ви знаєте?
13. У чому полягають особливості доступу членів класу з модифікатором public?
14. У чому полягають особливості доступу членів класу з модифікатором private?
15. У чому полягають особливості доступу членів класу з модифікатором protected?
16. У чому полягають особливості доступу членів класу з модифікатором internal?
17. Яке ключове слово мови C# використовується при створенні об’єкту?
18. Наведіть синтаксис створення об’єкту у загальному вигляді. Проілюструйте його фрагментом програми мовою C#.
19. У чому призначення конструктору?
20. Чи кожний клас мови C# має конструктор?
21. Які замовчення для конструктору прийняті у мові C#?
22. Яким значенням ініціалізується за замовченням значення посилкового типу?
23. У якому випадку конструктор за замовченням не використовується?
24. Наведіть синтаксис конструктору класу у загальному вигляді. Проілюструйте його фрагментом програми мовою C#.
25. Що розуміється під терміном «деструктор»?
26. У чому призначення деструктора?
27. Наведіть синтаксис деструктора класу у загальному вигляді. Проілюструйте його фрагментом програми мовою C#.
28. Чи має деструктор параметри?
29. Чи повертає деструктор значення?
30. Що розуміється під терміном «наслідування»?
31. Яка класифікація об’єктів відповідає наслідуванню?
32. Що спільного має дочірній клас із батьківським?
33. У чому полягає відмінність між дочірнім та батьківським класами?
34. Наведіть синтаксис опису успадкування класів у загальному вигляді. Проілюструйте його фрагментом програми мовою C#.
35. Якому відношенню відповідає ієрархія класів?
36. Якому відношенню відповідає ієрархія обє’ктів?

**ЛАБОРАТОРНА РОБОТА №3**

**ОЗНАЙОМЛЕННЯ З ОСНОВНИМИ ВИДАМИ КОЛЕКЦІЙ**

**3.1 Мета роботи**

Ознайомлення з основними видами колекцій, створення користувацького інтерфейсу на основі Windows Forms

**3.2 Вказівки з підготовки до виконання лабораторної роботи**

Необхідно вивчити наступні теми:

* використання колекцій;
* створення користувацького інтерфейсу на основі Windows Forms;
* ООП.
  1. **Теоретичні відомості**

**Колекція**

Сукупність об’єктів, що застосовується для спрощення багатьох завдань програмування завдяки використанню готових розв’язків створення ряду типових, але трудомістких для розробки структур даних. .NET має велику кількість інтерфейсів та класів, де реалізуються різні типи колекцій.

**Позитивні якості колекцій:**

* простота роботи з групами об’єктів;
* використання універсального набору інтерфейсів;
* зручність використання під час роботи із циклами та LINQ;
* гнучкість розробки.

**Інтерфейси у колекціях:**

* IEnumerable – реалізується для підтримки роботи із оператором циклу foreach;
* IComparer – реалізується для порівняння двох об’єктів класу колекцій під час їхнього сортування;
* ICollection – реалізується усіма класами колекцій для забезпечення доступу до методів CopyTo(), GetEnumerator() та властивості Count;
* IList – реалізується для забезпечення індексації елементів, їхнього додавання та видалення;
* IDictionary – реалізується для здійснення доступу до об’єктів за ключем чи значенням;
* IDictionaryEnumerator – реалізується для підтримки роботи із оператором foreach у колекціях, які використовують доступ до об’єктів за ключем.

**ArrayList**

Задана колекція реалізує інтерфейс IList під час використання масива з динамічним зміненням його розміру.

**Основні методи:**

* AddRange(…) – додає елементи інтерфейсу у кінець;
* BinarySearch(…) – виконує пошук елементу по всьому списку колекції, повертає індекс;
* InsertRange(…) – вставляє елементи до колекції за індексом;
* LastIndexOf(…) – здійснює пошук вказаного елементу, повертає індекс останнього входження;
* RemoveRange(…) – видаляє діапазон елементів колекції;
* Sort(…) – сортування елементів у колекції;
* Reverse(…) – змінює порядок елементів колекції на зворотній;
* ToArray(…) – повертає елементи колекції у вигляді масива.

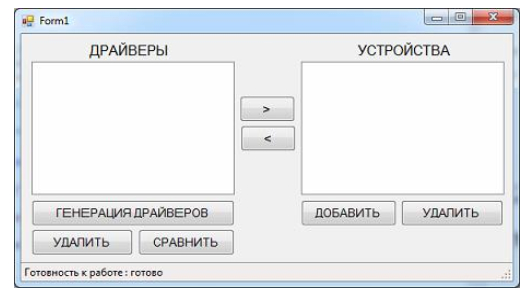
**Hashtable**

Колекція реалізує інтерфейс IDictionary і представляє набір пар типу «ключ-значення». Ключ створюється шляхом виклику методу GetHashCode() у зберігаємому елементі і не може приймати значення null.

**Властивості:**

* перевантаження методів GetHashCode() та Equals() у класі зберігаємого елементу;
* значення ключа не може приймати значення null;
* чутливість до регістру під час формування хеш-ключа;
* висока швидкість пошуку об’єктів;
* індексація об’єктів виключно за ключем.
  1. **Порядок виконання роботи і методичні вказівки з її виконання.**

**Варіант 1. «Диспетчер драйверів».**

****

**Класи:**

* диспетчер драйверів;
* драйвер;
* пристрій.

**Клас «Пристрій»**

**Дані:**

* назва;
* тип пристрою;
* драйвер, що використовується (об’єкт);
* готовність до роботи.

**Методи:**

* перевірка сумісності (із драйвером).

**Клас «Драйвер»**

**Дані:**

* назва;
* версія;
* підтримка ОС;
* тип пристрою.

**Методи:**

* перевірка сумісності (Dev, OC);
* оновлення версії.

**Оператори:**

* оператори == та != (за версією, типом пристрою та ОС).

**Клас «Диспетчер драйверів»**

**Дані:**

* словник драйверів (Hashtable/Dictionary, ключ-назва + версія);
* список пристроїв (ArrayList/List);
* готовність комп’ютера до роботи;
* тип ОС;
* кількість неактивних пристроїв.

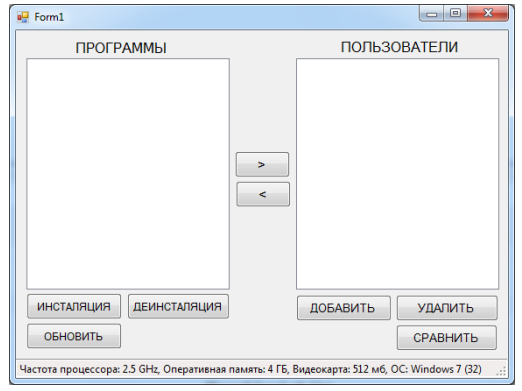
**Методи:**

* встановити драйвер на пристрій;
* видалити драйвер із пристрою;
* порівняти драйвери між собою;
* видалити пристрій;
* додати пристрій;
* видалити драйвери;
* завантажити драйвери (випадкову кількість, випадковим чином).

**Особливості:**

* під час 2х натискань на драйвер/пристрій – відкривається форма редагування;
* під час додавання пристрою – відкривається форма додавання;
* комп’ютер працює, якщо у ньому працюють пристрої типу: материнська плата, процесор, відеокарта, оперативна пам’ять, вінчестер, клавіатура;
* до комп’ютера можуть бути підключені пристрої наступних типів: миша, карта flash-пам'яті, CD / DVD-привід, монітор, принтер, сканер, мережева карта.

**Варіант 2. «Менеджер програм».**

****

**Класи:**

* менеджер програм;
* програма;
* користувач.

**Клас «Програма»**

**Дані:**

* назва;
* тип ОС;
* виробник;
* дата встановлення;
* версія;
* необхідна RAM;
* необхідна частота ЦП;
* займаний простір HDD.

**Методи:**

* оновлення версії програми;
* ініціалізуючий конструктор.

**Оператори:**

* оператори > та< для вимог ПО.

**Клас «Користувач»**

**Дані:**

* ім’я;
* пароль;
* список доступних програм;

**Методи:**

* автентифікація користувача ;

**Оператори:**

* оператори == та != (за кількістю та можливістю використання програм).

**Клас «Менеджер програм»**

**Дані:**

* список встановлених програм (ArrayList/List);
* словник користувача (Hashtable/Dictionary, ключ-ім’я + пароль);
* кількість програм;
* тип ОС;
* кількість користувачів;
* частота процесора;
* оперативна пам’ять;
* об’єм HDD.

**Методи:**

* дозволити/заборонити користувачу використання програми;
* порівняти дві вибрані програми між собою;
* порівняти 2 вибраних користувача між собою;
* оновлення версії програми;
* додавання/видалення програми;
* додавання/видалення користувача;
* змінення параметрів комп’ютера.

**Особливості:**

* під час подвійного натискання на елемент списку програм/користувачів відкривається форма їхнього редагування;
* під час додавання програми чи користувача відкриваються форми додавання;
* під час оновлення версії програми відбувається випадкова генерація параметрів класу програми (версія, необхідна RAM, частота ЦП і HDD збільшуються);
* під час подвійного натискання на рядок параметрів ПК визивається вікно редагування.

**Варіант 3. «Менеджер комп’ютерів у інтернет-кафе».**

**Класи:**

* менеджер комп’ютерів;
* задача;
* комп’ютер.

**Клас «Задача»**

**Дані:**

* час виконання;
* дата виконання;
* назва;
* виконано;
* додаткова інформація (назва програми чи ОС).

**Методи:**

* конструктор-копія;
* конструктор по замовченню;
* виконати задачу.

**Клас «Комп’ютер»**

**Дані:**

* назва;
* список призначених задач;
* кількість призначених задач;
* готовність до роботи

**Оператори:**

* оператори == та != (по всім даним, окрім назви).

**Клас «Менеджер комп’ютерів»**

**Дані:**

* список комп’ютерів (ArrayList/List);
* словник можливих задач (Hashtable/Dictionary, ключ – хеш-код);
* загальна кількість комп’ютерів;
* кількість робочих місць;
* наявність wi-fi роутера;
* пароль на wi-fi.

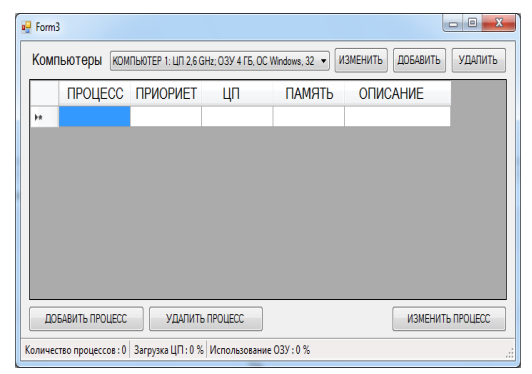
**Методи:**

* призначити задачу комп’ютеру – зняти задачу з комп’юетра;
* додати комп’ютер;
* видалити комп’ютер;
* встановлення wi-fi роутера;
* видалення wi-fi роутера;
* додавання нової задачі;
* видалення задачі;
* порівняння обраних комп’ютерів.

**Особливості:**

* під час подвійного натискання на елемент списку комп’ютерів/задач відкривається форма їхнього редагування;
* під час додавання комп’ютера або задачі відкриваються форми їхнього додавання;
* під час подвійного натискання на задачу в комп’ютері відбувається її виконання;
* під час виконання задачі, вона виділяється зеленим кольором.

**Варіант 4. «Диспетчер процесів на комп’ютері».**

****

**Класи:**

* диспетчер процесів;
* процес;
* комп’ютер.

**Клас «Процес»**

**Дані:**

* назва процесу;
* користувач;
* ЦП
* пам’ять;
* розташування;
* опис;
* пріорітет

**Методи:**

* ініціалізуючий конструктор;
* копіюючий конструктор

**Оператори:**

* оператори ++та – для зміни рівня пріоритету (від 0 до 4).

**Клас «Комп’ютер»**

**Дані:**

* ім’я;
* RAM;
* список процесів;
* частота ЦП;
* кількість процесорів.

**Методи:**

* змінення оперативної пам’яті;
* розгін/уповільнення процесора;
* додавання процесу;
* видалення процесу

**Клас «Менеджер комп’ютерів»**

**Дані:**

* список комп’ютерів (Hashtable);
* список процесу (Hashtable/Dictionary, ключ – назва процесу);
* ім’я адміністратора;
* пароль адміністратора.

**Методи:**

* додати комп’ютер;
* видалити комп’ютер;
* автентифікація адміністратора;
* додати процес у вибраний комп’ютер;
* видалити процес з вибраного комп’ютера;
* змінити процес на вибраному комп’ютері;
* змінити параметри вибраного комп’ютера.

**Особливості:**

* під час подвійного натискання у списку відкривається форма його редагування;
* під час додавання комп’ютера або процесу відкриваються форми додавання.

**Варіант 5. «Менеджер периферійних пристроїв »**

**Класи:**

* менеджер пристроїв;
* пристрій;
* комп’ютер.

**Клас «Пристрій»**

**Дані:**

* назва
* виробник
* тип пристрою (миша, клавіатура, HDD, Ram, Flash, принтер, сканер, колонки, телефон, мікрофон, навушники, монітор)
* тип порту (COM, USB, Micro USB);
* ознака підключення.

**Методи:**

* ініціалізуючий конструктор;
* копіюючий конструктор

**Оператори:**

* оператори true та false (після підключення).

**Клас «Комп’ютер»**

**Дані:**

* ідентифікатор;
* список пристроїв;
* кількість портів (COM, USB, Micro USB);
* кількість вільних портів.

**Методи:**

* підключити пристрій;
* відключити пристірй;
* ініціалізуючий конструктор.

**Клас «Менеджер пристроїв»**

**Дані:**

* список комп’ютерів (ArrayList/List);
* кількість пристроїв (SortedList);
* кількість пристроїв;
* кількість комп’ютерів;
* кількість пристроїв, що не використовуються.

**Методи:**

* додати комп’ютер;
* видалити комп’ютер;
* змінити комп’ютер;
* додати пристрій;
* видалити пристрій;
* змінити пристрій;
* підключення пристрою до комп’ютеру;
* відключення пристрою від комп’ютеру.

**Особливості:**

* під час подвійного натискання на пристрій чи комп’юетр у списку відкривається форма його редагування;
* під час додавання комп’ютера або пристрою відкриваються форми додавання.
  1. **Зміст звіту.**Звіт повинен мати:
* титульний лист із найменуванням лабораторної роботи і даними виконавця;
* мету роботи;
* завдання;
* текст програми з коментарями і поясненнями;
* результат роботи програми;
* висновки.
  1. **Контрольні запитання**

1. Що розуміється під терміном «поліморфізм? Якщо у класі для однієї назви/імені методу є множина різних реалізацій – це є поліморфізм?
2. Які частини класу можуть бути поліморфними?
3. Чи можуть оператори бути поліморфними? Якщо ні, то чому?
4. Що таке приховування методів?
5. Чим приховування відрізняється від поліморфізму?
6. Чим ітератор відрізняється від індексатора?
7. Як створити багатомірний індексатор?
8. Як змінити порядок виконання конструкторів у класі-спадкоємці?
9. Опишіть особливості статичного конструктору у C#.
10. Для чого потрібні розширюючі методи і як вони працюють?
11. Особливості герметизації класів/частин класу від наслідування.
12. Що таке інтерфейс у C#, чим інтерфейси відрізняються від класів?
13. Наведіть усі відмінності інтерфейсів від абстрактних класів, які ви знаєте.
14. Особливості роботи статичних класів.
15. Особливості наслідування у структурах.
16. У чому відмінність між контейнером і компонентом у Windows Form?
17. Перелічить основні переваги Windows Form?
18. Які базові класи або інтерфейси Windows Forms ви знаєте?
19. Визначення операторів класах, оператори явного/неявного перетворення типів.
20. Перелічить декілька контейнерних елементів у Windows Forms.

**ЛАБОРАТОРНА РОБОТА №4**

**ПРОГРАМНІ ІНТЕРФЕЙСИ ФАЙЛОВОГО ВВОДУ-ВИВІДУ ТА МАНІПУЛЯЦІЇ З ФАЙЛАМИ У ПЛАТФОРМІ .NET FRAMEWORK**

**4.1 Мета роботи**

Ознайомитися з програмними інтерфейсами, що призначені для маніпуляції з файлами і каталогами (створення копіювання, переміщення, видалення) та вводу-вивіду у мові C#.

**4.2 Вказівки з підготовки до виконання лабораторної роботи**

Необхідно вивчити наступні теми:

* програмні інтерфейси;
* створення, копіювання, переміщення та видалення каталогів;
  1. **Теоретичні відомості**

Простір імен System.IO в .NET - це область бібліотек базових класів, присвячена службам файлового введення-виведення, а також введення-виведення з пам'яті. Подібно будь-якому простору імен, в System.IO визначено набір класів, інтерфейсів, перерахувань, структур і делегатів, більшість з яких знаходяться в mscorlib.dll. На додаток до типів, що містяться всередині mscorlib.dll, в збірці System.dll визначені додаткові члени простору імен System.IO. Зверніть увагу, що всі проекти Visual Studio 2010 автоматично встановлюють посилання на обидві збірки.

Багато типів з простору імен System.IO зосереджені на програмних маніпуляціях фізичними каталогами і файлами. Додаткові типи надають підтримку читання і запису даних в рядкові буфери, а також області пам'яті. Нижче коротко описані основні (неабстрактне) класи, які дають уявлення про функціональності System.IO:

* **DriveInfo** – надає детальну інформацію відносно дискових пристроїв, які використовує дана машина;
* **Directory, DirectoryInfo** – використовується для маніпуляцій структурою каталогів машини. Тип **Directory** надає функціональність, користуючись статичними членами. Тип **DirectoryInfo** забезпечує аналогічну функціональність через дійсне об’єкте посилання;
* **FileSystemWatcher** – дозволяє відстежувати модифікації зовнішніх файлів в певному каталозі;
* **Path** – виконує операції над типами System.String, що містять інформацію про шлях до файлу або каталогу у незалежній від платформи манері;
* **File, FileInfo –** слугують для маніпуляцій множиною файлів даної машини. Тип File надає функціональність через статичні члени. Тип FileInfo забезпечує аналогічну функціональність через дійсне об’єктне посилання;
* **FileStream** – забезпечує довільний доступ до файлу (тобто, можливості пошуку) із даними, що представлені у вигляді потоку байтів;
* **MemoryStream –** забезпечує довільний доступ до даних, що зберігаються в пам’яті, а не у фізичному файлі;
* **BufferedStream -** надає тимчасове сховище для потоку байтів, які можуть потім бути перенесені до постійного сховища;
* **BinaryReader, BinaryWriter –** дозволяють зберігати та витягувати елементарні типи даних (целочисленные, булевские, строковые и т.п.) в двоичном виде;
* **StreamReader, StreamWriter -** використовуються для зберігання (і вилучення) текстової інформації з файлу. Ці класи не підтримують довільного доступу до файлу;
* **StringReader, StringWriter -** подібно класам StreamWriter / StreamReader, ці класи також працюють з текстовою інформацією. Однак що лежить в основі сховищем є строковий буфер, а не фізичний файл.

На додаток до цих конкретних типів класів в System.IO визначено кілька перерахувань, а також набір абстрактних класів (тобто Stream, TextReader і TextWriter), які визначають розділяється поліморфний інтерфейс для всіх спадкоємців.

Як не важко було помітити в наведеному вище списку, для подання файлів і папок використовуються по два класи. Який з них застосовувати - багато в чому залежить від того, скільки разів потрібно отримати доступ до даної папці або файлу.

* Класи Directory і File містять тільки статичні методи, а їх екземпляри ніколи не створюються. Для їх використання потрібно просто надати шлях до цікавого об'єкту файлової системи при кожному виклику методу-члена. Якщо потрібно виконати тільки одну операцію щодо папки або файлу, то застосування цих класів є найбільш ефективним підходом, оскільки дозволяє уникати накладних витрат, пов'язаних зі створенням примірників;
* Класи DirectoryInfo і FileInfo реалізують практично ті ж загальнодоступні методи, що і Directory і File, а також деякі загальнодоступні властивості і конструктори, але підтримують стан, а їхні члени не є статичними.

Спочатку потрібно буде зробити їх екземпляр, який потім асоціювати з певною папкою або файлом. Це означає, що застосування цих класів буде більш ефективним підходом, якщо необхідно виконувати безліч операцій щодо одного і того ж об'єкта. Справа в тому, що під час створення вони зчитують інформацію про аутентифікації та інші відомості про відповідному об'єкті файлової системи і більше не потребують її повторному читанні, скільки б методів не викликалося для даного об'єкта (екземпляра класу). На відміну від цього, класи, які не підтримують стан, потребують перевірки деталей, пов'язаних з файлом або папкою, при кожному виклику будь-якого методу.

Зверніть увагу на малюнку, що класи Directory і File безпосередньо розширюють System.Object, в той час як DirectoryInfo і FileInfo успадковуються від абстрактного класу **FileSystemInfo:**

Рисунок 4.1 – Ієрархія класів Directory, File, DirectoryInfo, FileInfo.

Класи DirectoryInfo і FileInfo успадкували значну частину своєї поведінки від абстрактного базового класу FileSystemInfo. Здебільшого члени класу FileSystemInfo використовуються для отримання загальних характеристик (таких як час створення, різні атрибути і т.д.) певного файлу або каталогу. У таблиці нижче перераховані деякі основні властивості, що представляють інтерес:

|  |  |
| --- | --- |
| Властивість | Призначення |
| **Attributes** | Отримує або встановлює асоційовані з поточним файлом атрибути, які представлені перерахуванням FileAttributes (доступний тільки для читання, зашифрований, прихований або стиснений) |
| **CreationTime** | Отримує або встановлює час створення поточного файлу або каталогу |
| **Exists** | Може використовуватися для визначення, чи існує даний файл або каталог |
| **Extension** | Витягує розширення файлу |
| **FullName** | Отримує повний шлях до файлу або каталогу |
| **LastAccessTime** | Отримує або встановлює час останнього доступу до поточного файлу або каталогу |
| **LastWriteTime** | Отримує або встановлює час останнього запису в поточний файл або каталог |
| **Name** | Отримує ім'я поточного файлу або каталогу |

У класі FileSystemInfo також визначено метод Delete (). Цей метод реалізується похідними типами для видалення файлу або каталогу з жорсткого диска. Крім того, метод Refresh () може бути викликаний перед отриманням інформації про атрибути, щоб забезпечити актуальність стану статистики про поточний файл або каталозі.

**Класи DirectoryInfo та FileInfo**

Клас FileInfo дозволяє отримувати подробиці щодо існуючих файлів на жорсткому диску (тобто час створення, розмір і атрибути) і призначений для створення, копіювання, переміщення і видалення файлів. До того ж до набору функціональності, успадкованої від FileSystemInfo, є деякі члени, унікальні для класу FileInfo, які описані нижче:

|  |  |
| --- | --- |
| **AppendText()** | Створює об'єкт StreamWriter і додає текст в файл |
| **СоруТо()** | Копіює існуючий файл в новий файл |
| **Create()** | Створює новий файл і повертає об'єкт FileStream для взаємодії з новоствореним файлом |
| **CreateText()** | Створює об'єкт StreamWriter, що записує новий текстовий файл |
| **Delete()** | Видаляє файл, до якого прив'язаний екземпляр FileInfo |
| **Directory** | Отримує екземпляр батьківського каталогу |
| **DirectoryName** | Отримує повний шлях до батьківського каталогу |
| **Length** | Отримує розмір поточного файлу або каталогу |
| **MoveTo()** | Переміщує вказаний файл в нове місце розташування, надаючи можливість вказати нове ім'я файлу |
| **Name** | Отримує ім'я файлу |
| **Open()** | Відкриває файл з різними привілеями читання / запису і спільного доступу |
| **OpenRead()** | Створює доступний тільки для читання об'єкт FileStream |
| **OpenText()** | Створює об'єкт StreamReader і читає з існуючого текстового файлу |
| **OpenWrite()** | Створює доступний тільки для запису об'єкт FileStream |

Зверніть увагу, що більшість методів класу FileInfo повертають специфічний об'єкт вводу-виводу (тобто FileStream і StreamWriter), який дозволяє почати читання і запис даних в асоційований файл в різноманітних форматах.

**Классы Directory и DirectoryInfo**

DirectoryInfo содержит набор членов, используемых для создания, перемещения, удаления и перечисления каталогов и подкаталогов. В дополнение к функциональности, предоставленной базовым классом (FileSystemInfo), DirectoryInfo предлагает ключевые члены, перечисленные ниже:

* **Create(), CreateSubdirectory()** – створює каталог (або набір підкаталогів) по заданому імені шляху;
* **Delete() –** видаляє каталог і весь його вміст;
* **GetDirectories() –** повертає масив об’єктів DirectoryInfo, що надає всі підкаталоги у поточному каталозі;
* **GetFiles() –** витягує масив об’єктів FileInfo, що надає множину файлів у заданому каталозі;
* **MoveTo() –** переміщує батьківській каталог даного каталогу;
* **Parent –** витягує батьківській каталог даного каталогу;
* **Root –** отримує кореневу частину шляху

Робота з типом DirectoryInfo починається з вказівки певного шляху в якості параметра конструктора. Якщо потрібно отримати доступ до поточного робочого каталогу (тобто каталогу виконується додатки), застосовуйте нотацію ".".

// Привязаться к текущему рабочему каталогу

DirectoryInfo dir1 = new DirectoryInfo(".");

// Привязаться к C:\Windows

DirectoryInfo dir2 = new DirectoryInfo(@"C:\Windows");

Об'єкт класу *DirectoryInfo* створюється в будь-якому випадку, незалежно від того, чи існує вказаний в конструкторі каталог чи ні. Однак, при спробі взаємодії з реально не існуючим каталогом виникне виключення *System.IO.DirectoryNotFoundException*.

Щоб вказати каталог, який поки ще не створено, спочатку доведеться викликати метод Create ():

// Привязаться к несуществующему каталогу, затем создать его

DirectoryInfo dir3 = new DirectoryInfo(@"C:\MyCode\Testing");

dir3.Create();

Клас Directory містить статичні члени, які повторюють функціональність об'єкта класу DirectoryInfo. Відмінністю є те, що члени Directory зазвичай повертають строкові дані замість строго типізованих об'єктів Filelnfo / DirectoryInfo.

Переміщення і видалення файлів або папок в класах FileInfo і DirectoryInfo забезпечують методи MoveTo () і Delete (). У класах File і Directory доступні аналогічні методи Move () і Delete (). До того ж класи FileInfo і File підтримують, відповідно, методи СоруТо () і Сору (). Однак для копіювання папок цілком методів не передбачено. Папку можна копіювати тільки за рахунок копіювання кожного міститься в ній файлу.

Простір імен System.IO включає клас DriveInfo. Подібно Directory.GetLogicalDrives (), статичний метод DriveInfo.GetDrives () дозволяє отримати імена дискових приводів машини.

Однак, на відміну від Directory.GetLogicalDrives (), DriveInfo надає безліч додаткових деталей (таких як тип приводу, доступне вільний простір і мітка тому), перерахованих в таблиці 4.1

Таблиця 4.1 – Члени класу DriveInfo

|  |  |
| --- | --- |
| Метод або властивість | Опис |
| GetDrives() | Статичний метод. Повертає масив рядків - імен дисков |
| DriveInfo(string name) | Конструктор |
| DriveType | Тип диска - перелічувальний тип зі значеннями Unknown, NoRootDirectory, Removable, Fixed, Network, CDRom, Ram |
| DriveFormat | Возвращает строку – тип файловой системы |
| Name | Отримання імені диска |
| VolumeLabel | Отримання і зміна мітки диска |
| TotalSize, TotalFreeSpace, AvailableFreeSpace | Загальний обсяг дискаОбъем свободного пространства  Объем доступного пространства |

**Читання та запис даних**

Читання і запис даних може виконуватися через спеціальні методи класу File, наприклад:

* ReadAllBytes() - відкриває вказаний файл, повертає виконавчі дані в вигляді масиву байт і потім закриває файл;
* ReadAllLines() - відкриває вказаний файл, повертає символьні дані у вигляді масиву рядків, потім закриває файл;
* ReadAllText() - відкриває вказаний файл, повертає символьні дані у вигляді System.String (), потім закриває файл;
* WriteAllBytes() - відкриває вказаний файл, записує в нього байтовий масив і закриває файл;
* WriteAllLines() - відкриває вказаний файл, записує в нього масив рядків і закриває файл;
* WriteAllText() - відкриває вказаний файл, записує в нього дані з зазначеного рядка і закриває файл.

Більш загальні можливості по введенню-висновку надають класи, що реалізують концепцію потоків введення-виведення. Ієрархія таких класів представлена на рис. 4.2.

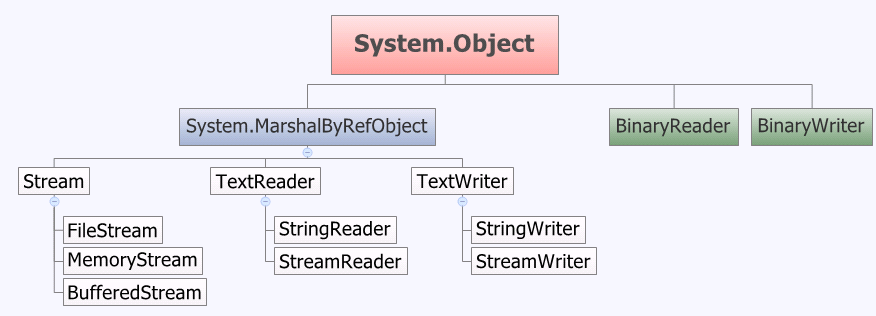


Рисунок 4.2 – Ієрархія потокових класів

Основним для потоків є клас System.IO.Stream. Він являє байтовий потік і є базовим для всіх інших класів потоків. Крім того, він є абстрактним класом, а це означає, що отримати екземпляр об'єкта класу Stream не можна.

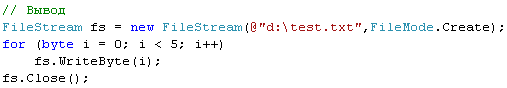
В абстрактному класі System.IO.Stream визначено набір членів, які забезпечують підтримку синхронного і асинхронного взаємодії зі сховищем (наприклад, файлом або областю пам'яті).

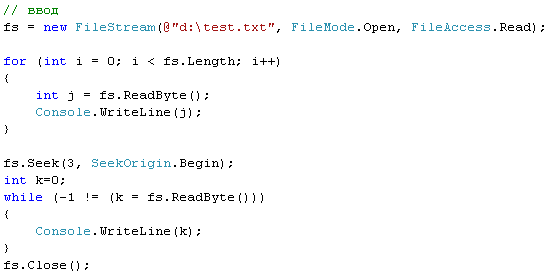
Концепція потоку не обмежена файловим введенням-виведенням. Точності заради слід зазначити, що бібліотеки .NET надають потоковий доступ до мереж, областям пам'яті та іншим абстракцій, пов'язаних з потоками.

Знову-таки, нащадки класу Stream представляють дані, як низькорівневі потоки байт, а безпосередня робота з низькорівневими потоками може виявитися досить загадковою. Деякі типи, успадковані від Stream, підтримують пошук (seeking), що означає можливість отримання і зміни поточної позиції в потоці. Щоб наблизитися до розуміння функціональності класу Stream, розглянемо список основних його членів, наведений нижче:

|  |  |
| --- | --- |
| **Метод, властивість** | **Опис** |
| CanRead CanWrite CanSeek | Визначення можливостей потоку |
| Read(), ReadByte() | Читання з потоку послідовності байтів (або одного байта) в буфер |
| Write(), WriteByte() | Запис послідовності байтів або одного байта в потік |
| Position | Отримання поточної позиції в потоці |
| Seek() | Зміна поточної позиції в потоці |
| Length  SetLength() | Отримання і зміна довжини потоку |
| Flush() | Очищення буфера виводу з оновленням пов'язаного джерела даних |
| Close() | Закриття потоку |

Робота з файловим потоком починається з його створення (шляхом виклику конструктора або з використанням методів класів File, FileInfo), далі виконується введення-виведення (Wirte (), WriteByte (), Read (), ReadByte ()) і закриття файлового потоку (Close ()).





Метод ReadByte () являє собою найпростіший спосіб для читання даних. Він бере один байт з потоку і призводить результат до типу int зі значенням в діапазоні від Про до 255. У разі досягнення кінця потоку він повертає -1.

Якщо необхідно, щоб за один раз читалося відразу безліч байтів, можна викликати метод Read (), який читає вказану кількість байтів в масив. Метод Read () повертає дійсну кількість прочитаних байтів; якщо повертається значення О, значить, був досягнутий кінець потоку. Нижче показаний приклад читання даних в масив байтів на ім'я ByteArray:

int nBytesRead = fs.Read(ByteArray, 0, nBytes);

У другому параметрі метод Read () приймає значення зсуву, де можна вказати, що масив має бути наповнений, починаючи не з першого, а з якогось іншого елемента. У третьому параметрі можна вказати, скільки байтів має читатися в масив.

Для виконання запису даних є два методи - WriteByte () і Write (). Метод WriteByte () дозволяє записувати по одному байту в потік:

byte NextByte = 100;

fs.WriteByte(NextByte);

Класи StreamWriter і StreamReader зручні у всіх випадках, коли потрібно читати або записувати символьні дані (наприклад, рядки). Обидва типи працюють за замовчуванням з символами Unicode; проте це можна змінити наданням правильно сконфигурированной посилання на об'єкт System.Text.Encoding. Щоб не ускладнювати приклад, припустимо, що кодування за замовчуванням Unicode цілком влаштовує.

Клас StreamReader, успадкований від абстрактного класу на ім'я TextReader. Базовий клас пропонує дуже обмежений набір функціональності кожному з його спадкоємців, зокрема - можливість читати і "заглядати" (peek) в символьний потік.

Клас StreamWriter успадковується від абстрактного базового класу на ім'я TextWriter. В цьому класі визначено члени, що дозволяють похідним типам записувати текстові дані в заданий символьний потік.

Методи класу StreamWriter аналогічні вже добре знайомим нам методам класу Console, зокрема:

* NewLine() - властивість, що визначає константу перекладу рядка для успадкованого класу письменника (за замовчуванням обмежувачем рядки в Windows є повернення каретки, за яким слід новий рядок (\ r \ n));
* Write() - метод запису даних в текстовий потік без додавання константи нового рядка;
* WriteLine() - метод запису даних в текстовий потік з додаванням константи нового рядка.

Клас BinaryWriter служить оболонкою, в яку полягає байтовий потік, керуючий висновком двійкових даних. Нижче наведено найбільш часто вживається конструктор цього класу: BinaryWriter (Stream output), де output позначає потік, в який виводяться записуються дані. Для запису в вихідний файл в якості параметра output може бути вказаний об'єкт, створюваний засобами класу FileStream. Якщо ж параметр output виявляється порожнім, то генерується виключення ArgumentNullException. А якщо потік, який визначається параметром output, не було відкрито для запису даних, то генерується виключення ArgumentException. По завершенні виведення в потік типу BinaryWriter його потрібно закрити. При цьому закривається і базовий потік.

Клас BinaryReader служить оболонкою, в яку полягає байтовий потік, керуючий введенням двійкових даних. Нижче наведено найбільш часто вживається конструктор цього класу: BinaryReader (Stream input) де input позначає потік, з якого вводяться зчитує дані. Для читання з вхідного файлу як параметр input може бути вказаний об'єкт, створюваний засобами класу FileStream.

Якщо ж потік, який визначається параметром input, не було відкрито для читання даних або виявився недоступним з інших причин, то генерується виключення ArgumentException. Правильно набравши з потоку типу BinaryReader його потрібно закрити. При цьому закривається і базовий потік.

* 1. **Порядок виконання роботи і методичні вказівки з її виконання.**

**Варіант 1.**

Дан файл цілих чисел. Знайти кількість містяться в ньому серій (тобто наборів послідовно розташованих однакових елементів). Наприклад, для файлу з елементами 1, 5, 5, 5, 4, 4, 5 результат дорівнює 4. Результат вивести в файл з таким же ім'ям, як і вихідний, але з розширенням «.out».

**Варіант 2.**

Дан файл цілих чисел. Створити новий файл цілих чисел, що містить довжини всіх серій вихідного файлу (серією називається набір послідовно розташованих однакових елементів, а довжиною серії - кількість цих елементів). Наприклад, для вихідного файлу з елементами 1, 5, 5, 5, 4, 4, 5 вміст результуючого файлу повинна бути наступним: 1, 3, 2, 1. Результат вивести в файл з таким же ім'ям, як і вихідний, але з розширенням «.out».

**Варіант 3.**

Дан файл дійсних чисел. Створити файл цілих чисел, що містить довжини всіх монотонних послідовностей елементів вихідного файлу. Наприклад, для вихідного файлу з елементами 1.7, 4.5, 3.4, 2.2, 8.5, 1.2 вміст результуючого файлу повинна бути наступним: 2, 3, 2, 2. Результат вивести в файл з таким же ім'ям, як і вихідний, але з розширенням «.out».

**Варіант 4.**

Дан файл дійсних чисел. Поміняти в ньому місцями мінімальний і максимальний елементи.

**Варіант 5.**

Дан файл цілих чисел з елементами A1, A2, ..., AN (N - кількість елементів у файлі). Замінити вихідне розташування його елементів на наступне:

A1, AN, A2, AN-1, A3, ....

**Варіант 6.**

Дан файл дійсних чисел. Замінити в файлі кожен елемент, крім початкового і кінцевого, на його середнє арифметичне з попереднім і наступним елементом. Запропонувати реалізацію, найбільш оптимальну за витратами пам'яті.

**Варіант 7.**

Дано три файли цілих чисел однакового розміру з іменами SA, SB, SC і рядок SD. Створити новий файл з ім'ям SD, в якому чергувалися б елементи вихідних файлів з одним і тим же номером:

A1, B1, C1, A2, B2, C2, ....

Запропонувати реалізацію для випадку, коли файли мають різну довжину.

**Варіант 8.**

Дано два файли дійсних чисел з іменами S1 і S2, елементи яких впорядковані за зростанням. Об'єднати ці файли в новий файл з ім'ям S3 так, щоб його елементи також виявилися впорядкованими за зростанням. Запропонувати реалізацію, найбільш оптимальну за вимогами до пам'яті.

**Варіант 9.**

Дана рядок S0, ціле число N (N <4) і N файлів цілих чисел з іменами S1, ... SN. Об'єднати їх вміст в новому файлі-архіві з ім'ям S0, використовуючи наступний формат: в першому елементі файлу-архіву зберігається число N, в наступних N елементах зберігається розмір (число елементів) кожного з вихідних файлів, а потім послідовно розміщуються дані з кожного вихідного файлу .

**Пояснення до завдань**

Всі завдання повинні бути виконані у вигляді додатку WindowsForms. Всі імена файлів запитуються через стандартні діалоги відкриття і збереження файлу. Для всіх варіантів завдань, де потрібно виконувати деякі дії з вмістом файлу, передбачити режим попереднього формування подібного файлу і заповнення його необхідними значеннями (можливо, з використанням генератора випадкових чисел).

* 1. **Зміст звіту**

Звіт повинен мати:

* титульний лист із найменуванням лабораторної роботи і даними виконавця;
* мету роботи;
* завдання;
* текст програми з коментарями і поясненнями;
* результат роботи програми;
* висновки.
  1. **Контрольні запитання**

1. Перелічіть основні типи, що об’явлені у простору імен System.IO.
2. Які функціональні можливості надають типи File та FileInfo? У чому їх схожість та відмінність?
3. Які функціональні можливості надають типи Directory та DirectoryInfo? У чому їх схожість та відмінність?
4. Вкажіть основні особливості методів MoveTo, Move, Сору та CopyTo.
5. Призначення та функціональні особливості FileStream.
6. Призначення та функціональні особливості типів StreamReader та StreamWriter. Необхідність використання методів Close ы Dispose.
7. Призначення та функціональні особливості типів StringWriter, StringReader. У чому принципова відмінність між даними типами?
8. Які функціональні можливості надає робота з файлами у бінарному режимі?
9. Перелічіть переваги та недоліки роботи із файлами у бінарному режимі, у зрівнянні з роботою із файлами у текстовому режимі.
10. Призначення і функціональні особливості типів BinaryWriter та BinaryReader.
11. Вкажіть основні функціональні можливості типу Path.
12. Поясніть функціональне призначення методів Seek та Flush у класі Stream.
13. Як підключитися до існуючого файлу із правами тільки для читання?
14. Для чого при створенні об’єкту файлового потоку вводу/вивіду рекомендується використання конструкції using?
15. Перелічіть основні типи у простору імен System.Security.AccessControl.

**ЛАБОРАТОРНА РОБОТА №5**

**ВИКОРИСТАННЯ LINQ TO OBJECTS, МЕХАНІЗМУ DATBINDING ТА СЕРИАЛІЗАЦІЇ ДАНИХ ДЛЯ ПОБУДОВИ ДОДАТКІВ**

**5.1 Мета роботи**

Метою роботи є дослідження можливостей, що надаються механізмом LINQ to data під час отримання даних із пов’язаних колекцій.

**5.2 Вказівки з підготовки до виконання лабораторної роботи**

Необхідно вивчити наступні теми:

* використання LINQ to data;
* вивчення можливостей представленого механізму;
  1. **Теоретичні відомості**

Запит являє собою вираз, який витягує дані з джерела даних. Запити зазвичай виражаються на спеціальній мові запитів. Згодом були розроблені різні мови для різних типів джерел даних, наприклад SQL для реляційних баз даних і XQuery для XML. Таким чином, розробники змушені вивчати нову мову запитів для кожного типу джерела даних або формату даних, який вони повинні підтримувати. LINQ спрощує ситуацію, пропонуючи однакову модель для роботи з даними в різних видах джерел і форматів даних. У запиті LINQ робота завжди здійснюється з об'єктами. Для запитів і перетворень даних в XML-документах, базах даних SQL, наборах даних ADO.NET, колекціях .NET і будь-яких інших форматах, для яких доступний постачальник LINQ, використовуються однакові базові шаблони кодування.

**Три частини операції запиту**

Всі операції запиту LINQ складаються з трьох різних дій:

* отримання джерела даних;
* створення запиту;
* виконання запиту.

У наступному прикладі показано вираз цих трьох частин операції запиту в вихідному коді. У прикладі в якості джерела даних для зручності використовується масив цілих чисел; тим не менш, ті ж принципи застосовні і до інших джерел даних. Частина, що залишилася розділу посилається на цей приклад.

class IntroToLINQ

{

static void Main()

{

// The Three Parts of a LINQ Query:

// 1. Data source.

int[] numbers = new int[7] { 0, 1, 2, 3, 4, 5, 6 };

// 2. Query creation.

// numQuery is an IEnumerable<int>

var numQuery =

from num in numbers

where (num % 2) == 0

select num;

// 3. Query execution.

foreach (int num in numQuery)

{

Console.Write("{0,1} ", num);

}

}

}

На следующем рисунке показана завершенная операция запроса. В LINQ выполнение запроса отличается от самого запроса; другими словами, создание переменной запроса само по себе не связано с получением данных.

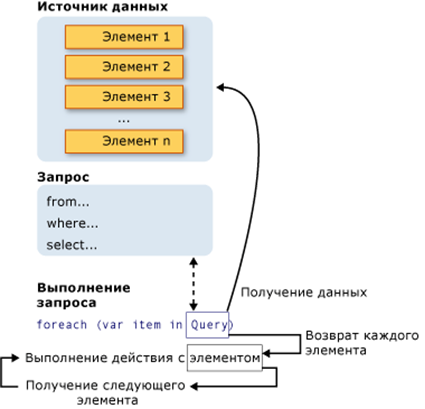
**Джерело даних**

У попередньому прикладі джерелом даних є масив, тому він неявно підтримує універсальний інтерфейс IEnumerable <T>. Це означає, що до нього можна виконувати запити з LINQ. Запит виконується в інструкцію foreach, і foreach для IEnumerable або IEnumerable <T>. Типи, які підтримують IEnumerable <T> або похідний інтерфейс як універсальний шаблон IQueryable <T> викликаються затребуваними типами.

Для запитуваної типу, який виступає в якості джерела даних LINQ, не потрібні зміни або спеціальна обробка.

**Запит**

Запит вказує, яку інформацію потрібно витягти з джерела або джерел даних. При необхідності, запит також вказує спосіб сортування, угруповання і формування цих відомостей перед поверненням. Запит зберігається в змінної запиту і инициализируется виразом запиту. Щоб спростити написання запитів, в C # з'явився новий синтаксис запиту.



Запит з першого прикладу повертає всі парні числа з масиву цілих чисел. Вираз запиту містить три пропозиції: from, where і select. (Якщо ви знайомі з SQL, зверніть увагу, що порядок пропозицій протилежний порядку в SQL.) Пропозиція from вказує джерело даних, пропозиція where застосовує фільтр, а пропозиція select вказує тип повертаються елементів.

**Виконання запиту**

**Відкладене виконання**

Як вже говорилося раніше, сама змінна запиту тільки зберігає команди запиту. Фактичне виконання запиту відкладається до виконання ітерації змінної запиту в операторі foreach. Цю концепцію називають відкладеним виконанням, вона показана в наступному прикладі:

// Query execution.

foreach (int num in numQuery)

{

Console.Write("{0,1} ", num);

}

Оператор foreach є також місцем, де беруться результати запиту. Наприклад, в попередньому запиті змінна ітерації num містить кожне (по черзі) значення в яку повертатимуть послідовності.

Так як сама змінна запиту ніколи не містить результатів запиту, її можна виконувати так часто, як необхідно. Наприклад, якщо база даних постійно оновлюється окремим додатком. У додатку можна створити один запит, який отримує останні дані, і його можна виконувати повторно з деяким інтервалом для вилучення кожного разу різних результатів.

**Примусове негайне виконання**

Запити, які виконують статистичні функції над діапазоном вихідних елементів, повинні спочатку виконати ітерацію цих елементів. Прикладами таких запитів є Count, Max, Average і First. Вони виконуються без явного оператора foreach, оскільки сам запит повинен використовувати foreach для повернення результату. Зверніть увагу, що такий тип запитів повертає одиночне значення, а не колекцію IEnumerable. Наступний запит повертає кількість парних чисел в початковому масиві.

var evenNumQuery =

from num in numbers

where (num % 2) == 0

select num;

int evenNumCount = evenNumQuery.Count();

Щоб примусово викликати негайне виконання будь-якого запиту і кешувати його результати, можна викликати метод ToList <TSource> або ToArray <TSource>.

List<int> numQuery2 =

(from num in numbers

where (num % 2) == 0

select num).ToList();

// or like this:

// numQuery3 is still an int[]

var numQuery3 =

(from num in numbers

where (num % 2) == 0

select num).ToArray();

Можна також примусово виконати запит, помістивши цикл foreach відразу після висловлення запиту. Однак виклик ToList або ToArray також кешируєт всі дані в одній колекції об'єктів.

**Прив’язка даних (data binding) -** це логічна асоціація між властивостями елементів призначеного для користувача інтерфейсу і властивостями деякого об'єкту. При цьому зміна значення властивості елемента користувальницького інтерфейсу, асоційованого в результаті прив'язки з деяким об'єктом, спричинить за собою автоматичну зміну відповідного властивості в даному об'єкті. Наприклад, в якості елемента користувальницького інтерфейсу використовується компонент TextBox, а в якості об'єкта - екземпляр класу Person, що має автосвойство Name типу string. В результаті прив'язки властивість textbox1.Text асоційоване з властивістю somePerson.Name. При зміні значення textbox1.Text користувачем, зміна значення somePerson.Name відбувається автоматично, тобто без додавання рядки somePerson.Name = textbox1.Text в код програми. Справедливо і зворотне перетворення, тобто якщо в ході роботи програми відбулася зміна значення somePerson.Name - автоматично змінюється значення textbox1.Text. Таким чином, використання механізму прив'язки даних дозволяє підвищити читаність коду і надійність програми, завдяки скороченню кількості рядків виду somePerson.Name = textbox1.Text, використовуваних кожен раз, коли відбувається перевизначення відповідних даних користувачем, або в ході роботи програми.

На прикладі об'єктів textbox1 і somePerson прив'язка даних здійснюється наступним способом:

textBox1.DataBindings.Add("Text", somePerson, "Name");

Метод Add (...) дозволяє організувати просту прив'язку даних. При цьому першим аргументом є назва властивості компонента користувальницького інтерфейсу (в даному випадку textBox1), яке буде асоційоване з даними; другим аргументом є джерело даних (в даному випадку somePerson); третім аргументом є назва асоційованого властивості, що знаходиться в джерелі даних (в даному випадку "Name"). При цьому обов'язковою умовою є збіг типів першого і третього аргументів (в даному випадку у них тип string). Таким чином, при використанні методу Add (...) відбувається встановлення зв'язку між властивістю Text об'єкта textBox1 і властивістю Name об'єкта somePerson. При цьому зміна значення властивості Text автоматично спричинить зміну властивості Name, і навпаки.

Використання механізму прив'язки даних особливо ефективно при роботі з наборами однотипних об'єктів, представлених у вигляді масиву, списку або будь-якого іншого способу угруповання об'єктів. Наприклад, в якості типу об'єктів використовується клас Person, представлений нижче:

public class Person

{

public string Name { get; set; } //имя человека

public int Age { get; set; } //возраст человека

public string Address { get; set; } //место проживания

public Person(string name, int age, string address)

{

Name = name;

Age = age;

Address = address;

}

}

У програмі використовується масив типу Person, елементи якого необхідно прив'язати до відповідних компонентів для користувача інтерфейсу (див. рисунок 1).

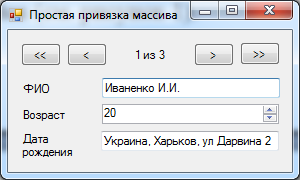


Рис.1 Проста прив’язка даних

При цьому для організації прив'язки даних до властивостей елементів призначеного для користувача інтерфейсу оголошується джерело даних (масив або список елементів типу Person) і об'єкт типу BindingSource, керуючий прив'язкою даних до компонентів інтерфейсу:

private Person[] data = new Person[3];

private BindingSource binding = new BindingSource();

Після створення об'єктів масив data асоціюється з об'єктом управління прив'язкою даних binding. Це відбувається наступним чином:

binding.DataSource = data;

Після асоціювання джерела даних з об'єктом управління прив'язкою, дані об'єкт (binding) може використовуватися як джерело даних при зв'язуванні з елементами призначеного для користувача інтерфейсу:

tbName.DataBindings.Add("Text", binding, "Name");

tbAge.DataBindings.Add("Value", binding, "Age");

tbBirth.DataBindings.Add("Text", binding, "Address");

де tbName - поле введення і відображення ПІБ, tbAge - поле введення і відображення віку, а tbBirth - поле введення і відображення дати народження об'єктів масиву.

Завдяки використанню об'єкта управління прив'язкою типу BindingSource з'являється можливість управління відображенням даних на компонентах призначеного для користувача інтерфейсу. У об'єкта binding класу BindingSource існує два властивості, що дозволяє керувати відображенням об'єктів, в разі, якщо джерелом даних є масив або список. Для отримання або перемикання індексу відображуваного об'єкта використовується властивість binding.Position. При цьому якщо встановити binding.Position = 2 - то на компоненти користувальницького інтерфейсу будуть відображати дані третього об'єкта з масиву, що є джерелом даних. Слід зазначити, що використання властивості Position є безпечним, оскільки відсутня можливість виходу за межі масиву, тобто навіть якщо встановити значення binding.Position = -2 - відображатися буде перший об'єкт в масиві, а якщо спробувати встановити значення, що перевищує загальну кількість елементів - властивість binding.Position автоматично прийме значення останнього елемента масиву. Крім того, з об'єкта binding можна отримати загальну кількість елементів в масиві або списку, що є джерелом даних. Для цього використовується властивість binding.Count.

Прив’язка даних ділиться на два види:

* проста прив’язка;
* складна прив’язка.

При простий прив'язці даних, відбувається асоціація між одним властивістю елемента управління і одним властивістю джерела даних. Крім того, при простий прив'язці в кожен вибільних момент часу роботи програми відбувається відображення властивість тільки одного об'єкта з джерела даних.

При складній прив'язці відбувається прив'язка облікового джерела даних цілком до елементу управління, тобто при цьому відбувається одночасне відображення декількох або всіх елементів джерела даних. Найбільш простим і поширеним прикладом складної прив'язки є використання елемента управління DataGridView.

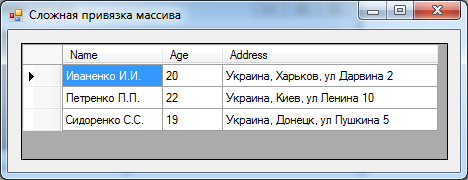


Рис.2 Складна прив’язка даних до елементу керування DataGridView

При складній прив'язці даних необхідність асоціювати властивості джерела даних з колонками або стовпцями таблиці відсутня. Асоціація джерела даних здійснюється відразу для всього компонента, при цьому назви колонок встановлюються автоматично відповідно до назв всіх відкритих автосвойств об'єктів з джерела даних. Прив'язка даних до елементу DataGridView проводиться таким чином:

List<Person> data = new List<Person>();

BindingSource binding = new BindingSource();

...

binding.DataSource = data;

dataGridView1.DataSource = binding;

Слід зазначити, що в разі зміни структури джерела даних - додавання нових або видалення елементів зі списку data, необхідно провести оновлення асоціативних зв'язків між елементом відображення і джерелом даних. Для цього використовується метод ResetBindings (...):

binding.ResetBindings(true);

Якщо в якості аргументу передається значення true - це означає, що проводиться оновлення прив'язки даних з урахуванням зміни кількості елементів в списку або масиві, який є джерелом даних. В іншому випадку відбувається тільки оновлення значень для елементів джерела даних.

**Серіалізація об’єкта -** процес перетворення поточного стану об'єкта в один зі стандартних форматів представлення даних, для подальшого збереження в потоці. Такими стандартними форматами представлення даних є: Binary, SOAP, XML. Для проведення сериализации об'єктів необхідно додати [Serializable] перед оголошенням їх класу.

Для проведення бінарної серіалізації даних необхідно:

1. Підключити відповідну бібліотеку:

using System.Runtime.Serialization.Formatters.Binary;

1. Створити екземпляр класу BinaryFormatter:

BinaryFormatter format = new BinaryFormatter();

1. Створити потік для запису результатів та провести серіалізацію даних:

using(FileStream file = new FileStream("file.bin",FileMode.Create))

{

format.Serialize(file, data);

}

Для проведення SOAP серіалізації необхідно:

1. Підключити відповідну бібліотеку

using System.Runtime.Serialization.Formatters.Soap;

1. Створити екземпляр класу SoapFormatter:

SoapFormatter format = new SoapFormatter();

1. Створити потік для запису результатів і провести серіалізацію даних:

using(FileStream file = new FileStream("file.txt",FileMode.Create))

{

format.Serialize(file, data.ToArray());

}

Для проведення XML серіалізації необхідно:

1. Підключити відповідну бібліотеку:

using System.Xml.Serialization;

1. Створити екземпляр класу XmlSerializer:

XmlSerializer format = new XmlSerializer(typeof(List<Person>));

1. Створити потік для запису результату та провести серіалізацію даних:

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

{

format.Serialize(file, data);

}

Слід зазначити, що формат SOAP підтримує роботу тільки з масивами, тобто в разі, якщо Серіалізуемое дані представлені у вигляді списку, або словника - для проведення SOAP-сериализации їх необхідно перетворити в масив. Важливою відмінністю XmlSerializer від інших класів, які використовуються для сериализации, є обов'язкове визначення типу Серіалізуемое даних в конструкторі примірника. Оскільки в даному прикладі використовувався список об'єктів типу Person - в конструкторі вказується typeof (List <Person>).

**Десеріалізація об’єкту -** процес, зворотний процесу сериализации, тобто це процес відновлення стану серіалізовані об'єкта з деякого стандартного формату даних.

Технічно процес десеріалізациі відрізняється від процесу сериализации тільки на етапі роботи з файловими потоками. Для читання серіалізовать об'єктів з потоку необхідно виконати наступні дії:

using(FileStream file = new FileStream("file.data", FileMode.Open))

{

data = format.Deserialize(file) as List<Person>;

binding.DataSource = data;

}

Використання механізмів сериализации і десеріалізациі даних має ряд переваг у порівнянні зі звичайними способами файлових операцій, оскільки розробнику не потрібно самостійно продумувати структуру і формат файлу при збереженні або завантаженнями. Таким чином, використання механізмів сериализации призводить до підвищення надійності роботи програми і читання коду.

* 1. **Порядок виконання роботи і методичні вказівки з її виконання.**

Необхідно реалізувати Windows Forms додаток, що виконує дії відповідно до варіанта завдання. Для візуалізації поточного стану колекції необхідно використовувати компонент DataGrid з встановленням прив'язки даних. Також необхідно реалізувати можливості збереження і відновлення інформації, що зберігається в додатку, за допомогою засобів сериализации мови С # в один з наступних форматів даних: xml / soap / binary / json.

**Варіант 1**. Додаток зберігає інформацію про викладачів, предметах і їх взаємозв'язку (зв'язок багато до багатьох, тобто один викладач може вести кілька предметів і кілька викладачів можуть вести один і той же предмет. Через діалогові вікна можна редагувати список викладачів і предметів, видаляючи або додаючи елементи списків. Через діалогове вікно також можна додавати зв'язку викладач-предмет в загальний список. Реалізувати сортування по зростанню і зменшенням за одну викладач і по стовпцю предмет.

**Варіант 2**. Додаток зберігає інформацію про студентів і їх групах (реалізується зв'язок багато до одного, тобто один студент належить тільки одній групі, але в групі студентів може бути будь-яку кількість). Через діалогові вікна можна редагувати інформацію про список груп (додавання, видалення, перейменування) і студента (додавання, видалення, перейменування, зміна групи). Реалізувати сортування на прізвище студента або номеру групи. Реалізувати фільтрацію виведення всіх студентів або тільки студентів обраної групи.

**Варіант 3**. Додаток зберігає інформацію про предметах, студентів та їх оцінках. Через діалогові вікна можна редагувати списки студентів (додавання, видалення, зміна прізвища) і предметів (додавання, зміна назви, видалення), додавання оцінок. Реалізувати сортування списку за прізвищем студента або назвою предмета. Реалізувати фільтрацію (відбір результатів запиту) з окремого предмету.

**Варіант 4**. Додаток зберігає інформацію про предметах, студентів та їх оцінках. Через Діалогове вікно можна редагувати списки студентів (додавання, видалення, зміна прізвища) і предметів (додавання, зміна назви, видалення), додавання оцінок. Реалізувати сортування списку за прізвищем студента або назвою предмета. Реалізувати сортування по успішності студентів.

**Варіант 5**. Додаток зберігає інформацію про співробітників і підрозділах підприємства. Через діалогові вікна можна редагувати інформацію про підрозділи (додавання, видалення, зміна назви) та співробітників (додавання інформації, видалення, зміна прізвища та / або заробітної плати). Вивести список всіх співробітників із зазначенням підрозділу і поточної латки з можливістю сортування по кожному полю по зростанню і зменшенням. Реалізувати можливість накладення фільтра, що дозволяє виводити список співробітників тільки одного підрозділу.

**Варіант 6**. Додаток зберігає інформацію про співробітників і підрозділах підприємства. Через діалогові вікна можна редагувати інформацію про підрозділи (додавання, видалення, зміна назви) та співробітників (додавання інформації, видалення, зміна прізвища та / або заробітної плати). Вивести список всіх співробітників із зазначенням підрозділу і поточної латки з можливістю сортування по кожному полю по зростанню і зменшенням. Реалізувати можливість виведення списку всіх підрозділів зазначенням кількості працюючих в ньому співробітників.

**Варіант 7**. Додаток зберігає інформацію про книгах і авторів у бібліотеці. Кожна книга характеризується автором (вибирається зі списку), назвою, видавництвом (вибирається зі списку), роком видання. Через діалогові вікна можна редагувати інформацію про авторів (додавання, видалення, зміна прізвища), видавництвах (додавання, видалення, зміна прізвища) і книгах. Вивести список всіх книг в повному форматі з можливістю сортування по автору, видавництву і / або році публікації. Реалізувати можливість фільтрації результатів по конкретному автору і / або видавництву.

**Варіант 8**. Додаток зберігає інформацію про книгах і авторів у бібліотеці. Кожна книга характеризується автором (вибирається зі списку), назвою, видавництвом (вибирається зі списку), роком видання. Через діалогові вікна можна редагувати інформацію про авторів (додавання, видалення, зміна прізвища), видавництвах (додавання, видалення, зміна прізвища) і книгах. Вивести список всіх книг в повному форматі з можливістю сортування по автору, видавництву і / або році публікації. Реалізувати можливість виведення статистичної інформації - скільки книг кожного учасника є в бібліотеці, яка кількість книг кожного видавництва є в бібліотеці.

**Варіант 9**. Додаток зберігає інформацію про аудіо-бібліотеці. Кожен запис характеризується автором (вибирається зі списку), жанром (вибирається зі списку), назвою альбому. Через діалогові вікна можна редагувати інформацію про авторів (додавання, видалення), жанрах (додавання, видалення) та записах (додавання, видалення). Вивести список всіх записів з можливістю сортування по автору, альбому або жанру. Реалізувати можливість фільтрації записів при виведенні за окремим автору, назві альбому і жанру.

* 1. **Зміст звіту**

Звіт повинен мати:

* титульний лист із найменуванням лабораторної роботи і даними виконавця;
* мету роботи;
* завдання;
* текст програми з коментарями і поясненнями;
* результат роботи програми;
* висновки.
  1. **Контрольні запитання**

1. Що розуміється під терміном прив’язки даних, для чого використовується прив’язка даних?
2. У чому відмінність складної прив’язки даних від простої?
3. Які типи даних використовуються для організації простої та складної прив’язки даних? У чому відмінності наведених типів даних?
4. Що розуміється під терміном серіалізація/десеріалізація даних?
5. Наведіть найбільш поширені формати даних, у яких можуть бути серіалізовані дані при використанні стандартних інструментів платформи .NET.
6. У чому принципова відмінність бінарного формату представлення об’єктів від інших форматів та способів серіалізації даних?
7. Поясніть призначення та основні функціональні можливості класу BindingSource.
8. Вкажіть призначення та основні функціональні можливості технології LINQ.
9. Наведіть основні способи створення LINQ-запитів. У чому відмінності між представленими способами створення запитів?
10. У чому відмінності між відкладеними та миттєвими LINQ-запитами?
11. Як LINQ-запит відкладеного виконання виконати миттєво?
12. Вкажіть основні переваги та недоліки прив’язки даних.
13. Вкажіть основні переваги та недоліки серіалізації даних.
14. У чому відмінність XML-серіалізації від інших способів серіалізації/десеріалізації інформації у .NET?
15. Вкажіть основні способи об’явлення породжуючого LINQ-запиту?

**ЛАБОРАТОРНА РОБОТА №6**

**БАГАТОПОТОЧНІСТЬ У С#**

**6.1 Мета роботи**

Метою роботи є ознайомлення з можливостями розробки найпростіших багатопоточних додатків у C#.

**6.2 Вказівки з підготовки до виконання лабораторної роботи**

Необхідно вивчити наступні теми:

* використання потоків;
* основні операції з потоками;
* класи багатопоточного програмування.
  1. **Теоретичні відомості**

**Поток(thread)**

***Поток(thread)*** являє собою незалежну послідовність інструкцій в програмі. Потоки грають важливу роль як для клієнтських, так і для серверних додатків. Наприклад, під час введення якогось коду C# у вікні редактора Visual Studio проводиться аналіз на предмет різних синтаксичних помилок. Цей аналіз здійснюється окремим фоновим потоком.

Кожна програма (процес) складається з ресурсів, таких як віконні дескриптори, файлові дескриптори і інші об'єкти ядра, має виділену область у віртуальній пам'яті і містить як мінімум один потік. Потоки плануються до виконання операційною системою. У будь-якого потоку є пріоритет, лічильник команд, який вказує на місце в програмі, де відбувається обробка, і стек, в якому зберігаються локальні змінні потоку. Стек у кожного потоку виглядає по-своєму, але пам'ять для програмного коду і куча поділяються серед всіх потоків, які функціонують усередині одного процесу. Це дозволяє потокам всередині одного процесу швидко взаємодіяти між собою, оскільки всі потоки процесу звертаються до однієї і тієї ж віртуальної пам'яті. Однак це також і ускладнює справу, оскільки дає можливість безлічі потоків змінювати одну і ту ж область пам'яті.

Потік може перебувати в одному з декількох станів. В цілому, потік може бути *потоком, що виконується; готовим до виконання*, як тільки він отримає час і ресурси ЦП; *припиненим*, тобто тимчасово що виконуються; *відновленим в подальшому; заблокованим* в очікуванні ресурсів для свого виконання; а також *закінченим*, коли його виконання закінчено і не може бути відновлено.

У середовищі .NET Framework визначені два різновиди потоків: **пріоритетний і фоновий**. За замовчуванням потік, що створюється автоматично стає пріоритетним, але його можна зробити фоновим. Єдина відмінність пріоритетних потоків від фонових полягає в тому, що фоновий потік автоматично завершується, якщо в його процесі зупинені всі пріоритетні потоки.

У зв'язку з організацією багатозадачності на основі потоків виникає потреба в особливого роду режимі, який називається синхронізацією і дозволяє координувати виконання потоків цілком певним чином. Для такої синхронізації в C # передбачена окрема підсистема.

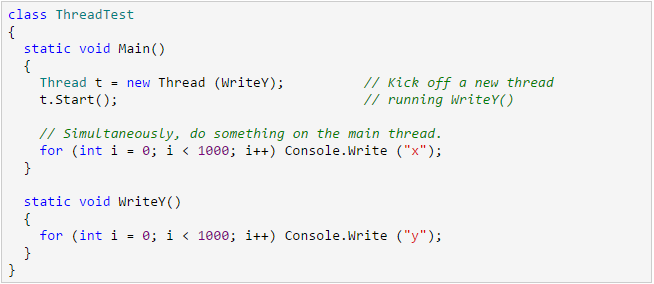
Всі процеси складаються хоча б з одного потоку, який зазвичай називають основним, оскільки саме з нього починається виконання програми. З основного потоку можна створити інші потоки.

Класи, що підтримують багатопотокове програмування, визначені в просторі імен **System.Threading**. Простір імен System.Threading містить різні типи, що дозволяють створювати багатопотокові програми. Мабуть, головним серед них є **клас Thread**, оскільки він представляє окремий потік. Клас Thread підтримує кілька методів рівня екземпляра, частина з яких описана в таблиці нижче. Скасування або призупинення активного потоку зазвичай вважається поганою ідеєю. Коли ви робите це, є шанс (хоча і невеликий), що потік може допустити "витік" своєї робочої навантаження, коли його турбують або переривають.

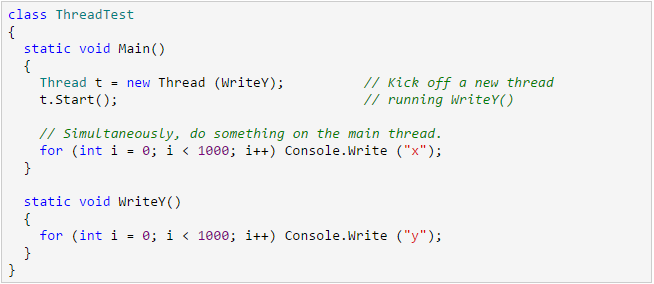
|  |  |
| --- | --- |
| **Елемент** | **Призначення** |
| **IsAlive** | Повертає булевське значення, що вказує на те, чи запущений потік (і ще не перерваний і не скасований) |
| **IsBackground** | Отримує або встановлює значення, що вказує, чи є даний потік "фоновим" (докладніше пояснюється далі) |
| **Name** | Дозволяє вам встановити дружнє текстове ім'я потоку |
| **Priority** | Отримує або встановлює пріоритет потоку, який може приймати значення з перерахування ThreadPriority |
| **ThreadState** | Отримує стан даного потоку, яким може бути присвоєно значення з перерахування ThreadState |
| **Abort()** | Інструктує CLR перервати потік, як тільки це буде можливо |
| **Interrupt()** | Перериває (тобто припиняє) поточний потік на заданий період очікування |
| **Join()** | Блокує викликає потік до тих пір, поки вказаний потік (той, в якому викликаний Join ()) ні завершиться |
| **Resume()** | Відновлює раніше припинений потік |
| **Start()** | Інструктує CLR запустити потік якомога швидше |
| **Suspend()** | Призупиняє потік. Якщо потік вже припинений, виклик Suspend () не дає ефекту |

**Основні операції з потоками**

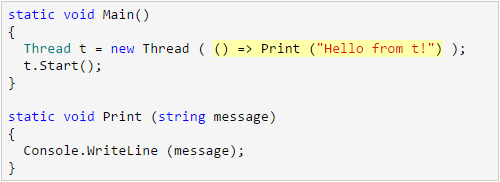
Створення потоку



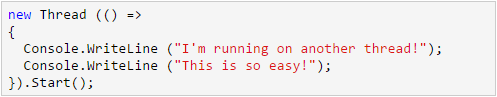
Переривання та очікування потоку



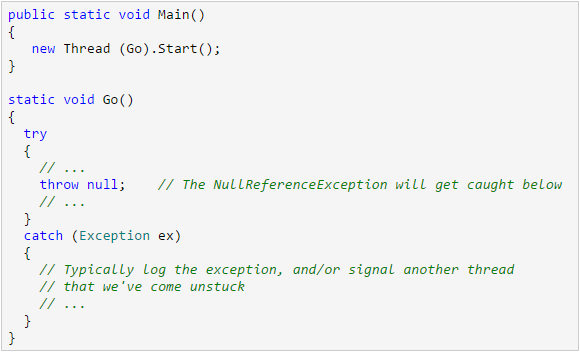
Передача даних потоку, лямбда-вирази у потоках



Під час опису метода можна користуватися повним описом λ-виразу декількома операторами.



Обробка виключень у потоках



* 1. **Порядок виконання роботи і методичні вказівки з її виконання.**

**Варіант 1.** Додаток моделює роботу магазину, що зберігає безліч товарів на складі. Кожен вид товару в магазині має назву, кількість на складі закупівельну ціну і ціну, за якою вона продається клієнтам магазину. Магазин може працювати у двох режимах. У першому режимі магазин відкритий для покупців, при цьому в окремому потоці протягом заданого часу відбувається генерація довільної кількості клієнтів, які купують різні товари, які видаляються зі складу магазину і додаючи гроші в касу магазину. У другому режимі магазин стає на переоблік, при цьому вся поточна інформація про суму, отриманої від продажу товарів, кількості клієнтів і місткості складу заноситься в колекцію і серіалізуются в XML. Крім того, в момент, коли магазин на переобліку відбувається підвезення товарів на склад магазину. При кожному підвезення товару з каси магазину віднімається сукупна вартість закупівельної ціни товару. Дані режими роботи магазину чергуються між собою, змінюючи один одного. Реалізувати можливість переглядати звіти про проміжках роботи магазину, витягнуті з XML за допомогою десеріалізациі.

**Варіант 2.** Додаток моделює роботу готелю, що зберігає інформацію про обмежену кількість номерів. Кожен номер характеризується кількістю спальних місць, вартістю за добу проживання і станом (зайнятий / вільний). При цьому в різний час готель працює по-різному: в сезон відпусток - він відкрита для туристів, в інший час - вона закрита для туристів. У сезон відпусток відбувається довільне заповнення готелю туристами (в окремому потоці в заданий проміжок часу), кожен турист довільним чином може вибирати номер на фіксовану кількість днів, або відмовлятися від послуг готелю. Після заповнення готелю туристами, потік клієнтів зупиняється до тих пір, поки в готелі не звільниться 10% від усієї кількості номерів. При цьому дані про заповнення номерів готелю туристами фіксуються і серіалізуются в XML. Реалізувати можливість відкриття / закриття готелю для туристів, а також можливість перегляду звітів з XML про завантаження готелю в різний час роботи.

**Варіант 3.** Додаток моделює роботу заводу, що містить інформацію про кількість цехів, сировини і готової продукції на складах. При цьому завод може працювати в декількох режимах: закупівля сировини, виробництво продукції, продаж продукції. При закупівлі завод витрачає гроші на придбання сировини, з якого буде виготовлена ​​продукція. Процес закупівлі відбувається у фоновому потоці і займає деякий час, а кожна партія сировини займає 5% від місця на складі. Процес виготовлення 1 одиниці продукції супроводжується расходном деякої фіксованої суми за експлуатацію обладнання з каси заводу, видаленні 1 одиниці сировини зі складу, займає деякий час і виконується в фоновому потоці. Процес продажу продукції відбувається у фоновому потоці, займає деякий час, призводить до видалення фіксованої кількості продукції зі складу і додаванню грошей в касу заводу. Перевезення 1 одиниці продукції зі складу при продажу також займає деякий час. В кінці виконання кожного їх перерахованих процесів відбувається збереження відповідного звіту про виконану роботу в XML. Реалізувати можливість управління процесами на заводі, а також перегляду звітів про виконану роботу з XML.

* 1. **Зміст звіту.**Звіт повинен мати:
* титульний лист із найменуванням лабораторної роботи і даними виконавця;
* мету роботи;
* завдання;
* текст програми з коментарями і поясненнями;
* результат роботи програми;
* висновки.

**6.6 Контрольні запитання**

1. Наведіть основні переваги та недоліки багато поточних додаткв?

2. Що входить до складу потоку?

3. Що таке TLS та для чого це використовується?

4. Опишіть проблему DeadLock кылькох потоків та основні засоби ії вирішення

5. Що таке путл потоків та для чого він використовується?

6. Наведіть основні видлики потоку, створеного за допомогою класу Thread та потоку, отриманого з ThredPool?

7. Наведіть основні засоби синхронізації потоків.

8. Наведіть основні відліки між foreground та background потоками?

9. Чи можливо використовувати структури для вирішення проблеми синхронізації потоків?

10. Опишіть послідовність виконання потоків у .NET

11. Опишіть функціональне призначення та особливості класу Monitor

12. У чому полягає основня відмінніть між класом Monitor та використанням команди lock?

13. Наведіть збіжності та відліки між класом Monitor та класом Mutex?

14. Чим поток, створений за допомогою BackgroundWorker, відрізняється від потоку, створеного за допомогою класу Thread?

15. Опишіть функціональне призначення та особливості класу Semaphore

**ЛАБОРАТОРНА РОБОТА №7**

**ОССНОВИ РОБОТИ З БАЗАМИ ДАНИХ У С# ІЗ ВИКОРИСТАННЯМ ТЕХНОЛОГІЇ ADO.NET**

**7.1 Мета роботи**

Метою роботи є ознайомлення з основними можливостями роботи з базами даних у С# із використання технології ADO.NET на прикладі підключення до БД MS SQL Server.

**7.2 Вказівки з підготовки до виконання лабораторної роботи**

Необхідно вивчити наступні теми:

* бази даних і роботу з ними;
* технологія ADO.NET;
  1. **Теоретичні відомості**

Для встановлення з'єднання з БД, виконання команд і отримання результатів використовується постачальник даних .NET Framework. Ці результати обробляються безпосередньо, поміщаються в DataSet, щоб у міру необхідності вони були доступні для користувачів разом з даними з декількох джерел, або розподіляються між рівнями. Джерела даних .NET Framework є спрощеними і створюють мінімальний рівень між джерелом даних і кодом, збільшуючи тим самим продуктивність без збитку для функціональних можливостей

**Постачальники даних у .NET Framework:**

* постачальник для SQL Server: надає доступ до даних для Microsoft SQL Server 7.0 або пізніших версій (простір імен System.Data.SqlClient);
* постачальник даних для OLE DB: надає доступ для джерел даних OLE DB (простір імен System.Data.OleDb);
* постачальник даних для OLE DB: надає доступ для джерел даних OLE DB (простір імен System.Data.OleDb);
* постачальник даних для Oracle: надає доступ для джерел даних Oracle, підтримує клієнтське програмне забезпечення версії 8.1.7 і старше (простір імен System.Data.OracleClient);
* постачальник даних для EntityClient: надає доступ до даних для додатків моделі Entity Data Model (простір імен System.Data.EntityClient)

**Постачальники даних у .NET Framework:**

* Connection: встановлює з'єднання з конкретним джерелом даних. Базовим класом для всіх об'єктів Connection є DbConnection;
* Connection: встановлює з'єднання з конкретним джерелом даних. Базовим класом для всіх об'єктів Connection є DbConnection;
* DataReader: зчитує з джерела даних однопрохідний потік даних тільки для читання. Базовим класом об'єктів DataReader є DbDataReader;
* DataAdapter: заповнює DataSet і виконує поновлення в джерелі даних. Базовим класом об'єктів DataAdapter є DbDataAdapter.

**IDataReader – интерфейс**

Дозволяє читати один або кілька потоків результуючих наборів тільки в напрямку вперед, виконуючи команду над джерелом даних. Він реалізується постачальниками даних .NET Framework, що звертаються до реляційних баз даних.

**Простір імен: System.Data**

Інтерфейси IDataReader і IDataRecord дозволяють класу-спадкоємця реалізувати клас DataReader, що дозволяє читати один або кілька потоків наборів результатів тільки в напрямку вперед. Додаток створює не безпосередньо екземпляр інтерфейсу IDataReader, а екземпляр класу, який є спадкоємцем IDataReader.

Класи, які є спадкоємцями IDataReader, повинні реалізовувати успадковані члени. Зазвичай вони також визначають додаткові члени для додавання функціональних можливостей, які залежать від постачальника даних.

Зміни, внесені в результуючий набір іншим процесом або потоком під час читання даних, можуть бути видимі користувачеві або класу, який реалізує інтерфейс IDataReader. Однак точна поведінка залежить від постачальника і від тимчасових залежностей.

Для досягнення однаковості серед постачальників даних .NET Framework успадковують класи слід іменувати в формі Prv "команда", де Prv - постійний префікс, який призначається всім класам в просторі імен певного постачальника даних .NET Framework. Наприклад, Sql є префіксом класу SqlDataAdapter в просторі імен System.Data.SqlClient.

**IDbCommand – інтерфейс**

Являє собою оператор SQL, виконуваний при підключенні до джерела даних і реалізований постачальниками даних .NET Framework, які мають доступ до реляційних баз даних.

**Простір імен: System.Data**

Інтерфейс IDbCommand дозволяє спадщини класу реалізовувати клас Command, який є інструкцією SQL, що виконується в джерелі даних. Додаток створює не безпосередньо екземпляр інтерфейсу IDbCommand, а екземпляр класу, який є спадкоємцем IDbCommand.

Класи, що успадковують IDbCommand, повинні реалізовувати всі успадковані елементи. Зазвичай вони також визначають додаткові елементи для додавання функціональних можливостей, які залежать від постачальника. Наприклад, інтерфейс IDbCommand визначає метод ExecuteNonQuery. У свою чергу, клас SqlCommand успадковує цей метод, а також визначає метод ExecuteXmlReader.

**Примітки для об’єктів, що реалізують**

Для досягнення однаковості серед постачальників даних .NET Framework успадковують класи слід називати в формі PrvClassname, де Prv - постійний префікс, який призначається всім класам в просторі імен певного постачальника даних .NET Framework. Наприклад, Sql є префіксом класу SqlCommand в просторі імен System.Data.SqlClient:

* SqlConnection - клас: надає відкрите підключення до бази даних SQL Server. Цей клас не може бути успадкований (простір імен: System.Data.SqlClient);
* SqlCommand - клас: представляє інструкцію Transact-SQL або збережену процедуру, виконувану над базою даних SQL Server. Цей клас не може бути успадкований (простір імен: System.Data.SqlClient);
* SqlDataReader - клас: надає можливість читання потоку рядків тільки в прямому напрямку з бази даних SQL Server.Данний клас не може успадковуватися (простір імен: System.Data.SqlClient);
* SqlParameter - клас: надає параметр для об'єкта SqlCommand і, при необхідності, його відображення на DataSet.Етот клас не може бути успадкований (простір імен: System.Data.SqlClient).

**Приклад:** використовується БД з ім'ям Univer і однією таблицею Students, наступної структури:

* int ID – ідентифікатор;
* string FIO – ПІБ студента;
* string Gruppa – группа;
* СКБД MS SQL

**Створення таблиці**

**using** System;

**using** System.Data.SqlClient;

**using** System.Data;

**using** System.Threading;

**namespace** Students

{

**class** Program

    {

**static** **void** Main(**string**[] args)

        {

*/\*Создание строки подключения*

*Data Source - имя сервера, по стандарту (local)\SQLEXPRESS*

*Initial Catalog - имя БД*

*Integrated Security= - параметры безопасности \*/*

**string** connStr = @"Data Source=(local)\SQLEXPRESS;

                           Initial Catalog=Test;

                           Integrated Security=True";

*/\*Здесь указано имя БД (для создания БД указывать не нужно)*

*для проверки того, создана ли данная БД*

*Создаем SqlConnection и передаем ему строку подключения \*/*

            SqlConnection conn = new SqlConnection(connStr);

**try**

            {

*//попытка подключения*

                conn.Open();

            }

**catch** (SqlException se)

            {

*// Если база не обнаружена, то создаем новую*

**if** (se.Number == 4060)

                {

                    Console.WriteLine("Подождите, идет создание БД");

*//закрытие соединения*

                    conn.Close();

*//переопредение обьекта с новой строкой подключения*

                    conn = new SqlConnection(@"Data

Source=(local)\SQLEXPRESS;Integrated Security=True");

*/\*Создание SqlCommand и передача запроса на создание БД*

*и объект типа SqlConnection \*/*

                    SqlCommand cmdCreateDataBase =

new SqlCommand(**string**.Format("CREATE DATABASE [{0}]",

"Test"), conn);

*//открываем подключение*

                    conn.Open();

*/\*Посылаем запрос к СУБД: в данном случае, в результате*

*запроса ничего не возвращается \*/*

                    Console.WriteLine("Посылаем запрос");

                    cmdCreateDataBase.ExecuteNonQuery();

*//закрываем подключение*

                    conn.Close();

*//задержка, нужна для того, чтоб БД успела создаться*

                    Thread.Sleep(5000);

                    conn = new SqlConnection(connStr);

*//открываем подключение*

                    conn.Open();

                }

            }

**finally**

            {

                Console.WriteLine("Соедение успешно произведено");

                conn.Close();

                conn.Dispose();

            }

        }

    }

}

**Вивід усіх записів з таблиці**

**using** System;

**using** System.Data.SqlClient;

**using** System.Data;

**namespace** Students

{

**class** Program {

**static** **void** Main(**string**[] args) {

*/\*Создание строковой переменной со строкой подключения*

*Data Source - имя сервера, по стандарту*

*(local)\SQLEXPRESS Initial Catalog - имя БД*

*Integrated Security=-параметры безопасности \*/*

**string** connStr = @"Data Source=(local)\SQLEXPRESS;

                           Initial Catalog=Test;

                           Integrated Security=True";

*/\*Здесь указывается имя БД для того, чтобы проверить,*

*может данная БД уже создана. Создание SqlConnection*

*и передаем конструктору строку подключения \*/*

            SqlConnection conn = new SqlConnection(connStr);

**try** { conn.Open(); }

**catch** (SqlException se) {

                Console.WriteLine("Ошибка подключения:{0}",

se.Message);

**return**;

            }

            Console.WriteLine("Соедение успешно произведено");

*/\*Создаем экземпляр класса SqlCommand и передаем запрос на*

*получение строк таблицы Students и объект типа SqlConnection \*/*

            SqlCommand cmd = new SqlCommand("Select \* From Students", conn);

*/\*Метод ExecuteReader() класса SqlCommand возвращает объект типа*

*SqlDataReader, с помощью которого происходит чтение всех строк,*

*возврашенных в результате выполнения запроса*

*CommandBehavior.CloseConnection - закрытие соединения после \*/*

**using** (SqlDataReader dr =

cmd.ExecuteReader(CommandBehavior.CloseConnection))

            {

*//цикл по всем столбцам полученной в результате запроса таблицы*

**for** (**int** i = 0; i < dr.FieldCount; i++)

*/\*Метод GetName() класса SqlDataReader позволяет получить имя*

*столбца по номеру, который передается в качестве параметра*

*и означает номер столбца в таблице (начинается с 0) \*/*

                Console.Write("{0}**\t**",dr.GetName(i).ToString().Trim());

**while** (dr.Read()) {

*/\*Метод GetValue() класса позволяет получить значение столбца*

*по номеру, который передается в качестве параметра и означает*

*номер столбца в таблице(начинается с 0) \*/*

Console.WriteLine("{0}**\t**{1}**\t**{2}",

dr.GetValue(0).ToString().Trim(),

                            dr.GetValue(1).ToString().Trim(),

                            dr.GetValue(2).ToString().Trim());

                }

              }

*//закрываем соединение*

                conn.Close();

                conn.Dispose();

                Console.WriteLine();

            }

         }

    }

**Додаваня запису до таблиці**

**using** System;

**using** System.Data.SqlClient;

**using** System.Data;

**namespace** Students

{

**class** Program

    {

**static** **void** Main(**string**[] args) {

**string** connStr = @"Data Source=(local)\SQLEXPRESS;

                           Initial Catalog=Test;

                           Integrated Security=True";

            SqlConnection conn = new SqlConnection(connStr);

**try** {

                conn.Open();

            }

**catch** (SqlException se) {

                Console.WriteLine("Ошибка подключения:{0}",se.Message);

**return**;

            }

            Console.WriteLine("Соедение успешно произведено");

*/\*Создание экземпляра класса  SqlCommand и передача запрос на*

*добавление строки в таблицу Students и объект SqlConnection \*/*

            SqlCommand cmd = new SqlCommand("Insert into Students" +

                "(ID,FIO,Grupa) Values (@ID,@FIO,@Grupa)", conn);

*/\*Работа с параметрами(SqlParameter), эта техника позволяет*

*уменьшить количество ошибок и повысить быстродействие, но*

*требует и больших усилий в написании кода \*/*

*//объявляем объект класса SqlParameter*

            SqlParameter param = new SqlParameter();

*//имя параметра*

            param.ParameterName = "@ID";

*//значение параметра*

            param.**Value** = 10;

*//тип параметра*

            param.SqlDbType = SqlDbType.**Int**;

*//параметр объекту класса SqlCommand*

            cmd.Parameters.**Add**(param);

*//переопределение объекта класса SqlParameter*

            param = new SqlParameter();

*//задаем имя параметра*

            param.ParameterName = "@FIO";

*//задаем значение параметра*

            param.**Value** = "Иванов Иван Иванович";

*//задаем тип параметра*

            param.SqlDbType = SqlDbType.Text;

*//передаем параметр объекту класса SqlCommand*

            cmd.Parameters.**Add**(param);

*//переопределяем объект класса SqlParameter*

            param = new SqlParameter();

*//задаем имя параметра*

            param.ParameterName = "@Grupa";

*//задаем значение параметра*

            param.**Value** = "4-2";

*//задаем тип параметра*

            param.SqlDbType = SqlDbType.Text;

*//передаем параметр объекту класса SqlCommand*

            cmd.Parameters.**Add**(param);

        Console.WriteLine("Вставляем запись");

**try** {

              cmd.ExecuteNonQuery();

            }

**catch** {

              Console.WriteLine("Ошибка запроса на добавление записи");

**return**;

            }

*//Вывод значения на экран*

            cmd = new SqlCommand("Select \* From Students", conn);

*/\*Метод ExecuteReader() возвращает объект типа SqlDataReader, с*

*помощью которого мы можем прочитать все строки, возвращенные в результате выполнения запроса CommandBehavior.CloseConnection – закрытие соединения после запроса \*/*

**using** (SqlDataReader dr =

cmd.ExecuteReader(CommandBehavior.CloseConnection)){

**for** (**int** i = 0; i < dr.FieldCount; i++)

                 Console.Write("{0}**\t**",dr.GetName(i).ToString().Trim());

                 Console.WriteLine();

**while** (dr.Read()) {

  Console.WriteLine("{0}**\t**{1}**\t**{2}",

dr.GetValue(0).ToString().Trim(),

                            dr.GetValue(1).ToString().Trim(),

                            dr.GetValue(2).ToString().Trim());

                 }

              }

                conn.Close();

                conn.Dispose();

                Console.WriteLine();

            }

         }

    }

**Видалення запису з таблиці**

**using** System;

**using** System.Data.SqlClient;

**using** System.Data;

**namespace** Students

{

**class** Program

    {

**static** **void** Main(**string**[] args)

        {

**string** connStr = @"Data Source=(local)\SQLEXPRESS;

                           Initial Catalog=Test;

                           Integrated Security=True";

            SqlConnection conn = new SqlConnection(connStr);

**try** { conn.Open(); }

**catch** (SqlException se) {

                Console.WriteLine("Ошибка подключения:{0}",

se.Message);

**return**;

            }

            Console.WriteLine("Соедение успешно произведено");

*/\*Создаем экземпляр SqlCommand и передаем запрос на*

*удаление строк таблицы Students, которые отвечают*

*условиям и объект типа SqlConnection \*/*

            SqlCommand cmd = new SqlCommand("Delete From Students" +

                    " where ID = @ID  and FIO = @FIO", conn);

            SqlParameter param = new SqlParameter();

            param.ParameterName = "@ID";

            param.**Value** = 10;

            param.SqlDbType = SqlDbType.**Int**;

            cmd.Parameters.**Add**(param);

            param = new SqlParameter();

            param.ParameterName = "@FIO";

            param.**Value** = "Иванов Иван Иванович";

            param.SqlDbType = SqlDbType.Text;

            cmd.Parameters.**Add**(param);

            Console.WriteLine("Удаляем запись");

**try** { cmd.ExecuteNonQuery(); }

**catch** {

                Console.WriteLine("Ошибка запроса на удаление");

                Console.WriteLine("Возможно запись уже удалена");

**return**;

            }

*//Вывод значения на экран*

            cmd = new SqlCommand("Select \* From Students", conn);

**using** (SqlDataReader dr =

cmd.ExecuteReader(CommandBehavior.CloseConnection)){

**for** (**int** i = 0; i < dr.FieldCount; i++)

                 Console.Write("{0}**\t**",dr.GetName(i).ToString().Trim());

                 Console.WriteLine();

**while** (dr.Read()) {

  Console.WriteLine("{0}**\t**{1}**\t**{2}",

dr.GetValue(0).ToString().Trim(),

                            dr.GetValue(1).ToString().Trim(),

                            dr.GetValue(2).ToString().Trim());

                 }

              }

                conn.Close();

                conn.Dispose();

                Console.WriteLine();

            }

         }

    }

**Зміна запису з таблиці**

**using** System;

**using** System.Data.SqlClient;

**using** System.Data;

**namespace** Students

{

**class** Program

    {

**static** **void** Main(**string**[] args)

        {

**string** connStr = @"Data Source=(local)\SQLEXPRESS;

                           Initial Catalog=Test;

                           Integrated Security=True";

            SqlConnection conn = new SqlConnection(connStr);

**try** { conn.Open(); }

**catch** (SqlException se) {

                Console.WriteLine("Ошибка подключения:{0}",

se.Message);

**return**;

            }

           Console.WriteLine("Соедение успешно произведено");

        SqlCommand cmd = new SqlCommand("Update Students" +

                " Set ID = @ID where FIO = @FIO", conn);

            SqlParameter param = new SqlParameter();

            param.ParameterName = "@ID";

            param.**Value** = 10;

            param.SqlDbType = SqlDbType.**Int**;

            cmd.Parameters.**Add**(param);

            param = new SqlParameter();

            param.ParameterName = "@FIO";

            param.**Value** = "Иванов Иван Иванович";

            param.SqlDbType = SqlDbType.Text;

            cmd.Parameters.**Add**(param);

  Console.WriteLine("Изменяем запись(и)");

**try** { cmd.ExecuteNonQuery(); }

**catch**

            {

                Console.WriteLine("Ошибка запроса на изменение записи(ей)");

**return**;

            }

*//Выводим значение на экран*

                cmd = new SqlCommand("Select \* From Students", conn);

**using** (SqlDataReader dr =

cmd.ExecuteReader(CommandBehavior.CloseConnection))

               {

**for** (**int** i = 0; i < dr.FieldCount; i++)

                       Console.Write("{0}**\t**",dr.GetName(i).ToString().Trim());

                   Console.WriteLine();

**while** (dr.Read())

                       {

**while** (dr.Read()) {

  Console.WriteLine("{0}**\t**{1}**\t**{2}",

dr.GetValue(0).ToString().Trim(),

                            dr.GetValue(1).ToString().Trim(),

                            dr.GetValue(2).ToString().Trim());

                 }

              }

                conn.Close();

                conn.Dispose();

                Console.WriteLine();

            }

         }

    }

**Видалення таблиці**

**using** System;

**using** System.Data.SqlClient;

**using** System.Data;

**namespace** Students

{

**class** Program {

**static** **void** Main(**string**[] args) {

**string** connStr = @"Data Source=(local)\SQLEXPRESS;

                           Initial Catalog=Test;

                           Integrated Security=True";

            SqlConnection conn = new SqlConnection(connStr);

**try** { conn.Open(); }

**catch** (SqlException se) {

                Console.WriteLine("Ошибка подключения:{0}", se.Message);

**return**;

            }

            Console.WriteLine("Соедение успешно произведено");

            SqlCommand cmdCreateTable = new SqlCommand("DROP TABLE Student",

conn);

**try** { cmdCreateTable.ExecuteNonQuery(); }

**catch** {

                Console.WriteLine("Ошибка при удалении таблицы");

**return**;

            }

            Console.WriteLine("Таблица удалена успешно");

            conn.Close();

            conn.Dispose();

        }

    }

}

* 1. **Порядок виконання роботи і методичні вказівки з її виконання.**

Необхідно реалізувати Windows Forms додаток, що виконує дії відповідно до варіанта завдання. Для візуалізації поточного стану колекції рекомендується використовувати компонент DataGrid. Реалізувати можливість збереження і відновлення інформації, що зберігається в додатку, за допомогою засобів ADO.NET мови С #.

**Варіант 1.** Додаток зберігає інформацію про викладачів, предметах і їх взаємозв'язку (зв'язок багато до багатьох, тобто один викладач може вести кілька предметів і кілька викладачів можуть вести один і той же предмет. Реалізувати можливість редагування списку викладачів і предметів. Реалізувати можливість додавання зв'язки викладач-предмет. Реалізувати можливість збереження, вилучення та зміни даних про предметах і викладачів в таблиці БД.

**Варіант 2.** Додаток зберігає інформацію про студентів і їх групах (реалізується зв'язок багато до одного, тобто один студент належить тільки одній групі, але в групі студентів може бути будь-яку кількість). Реалізувати можливість редагування інформації про список груп (додавання, видалення, перейменування) і студента (додавання, видалення, перейменування, зміна групи). Реалізувати можливість збереження, вилучення та зміни даних про студентів та групах в таблиці БД.

**Варіант 3.** Додаток зберігає інформацію про предметах, студентів та їх оцінках. Реалізувати можливість редагування списків студентів і предметів, додавання оцінок. Реалізувати можливість збереження, вилучення та зміни даних про предметах і студентах в таблиці БД.

**Варіант 4.** Додаток зберігає інформацію про співробітників і підрозділах підприємства. Реалізувати можливість редагування інформації про підрозділи і співробітників. Реалізувати можливість збереження, вилучення та зміни даних про підрозділи і співробітників в таблиці БД.

**Варіант 5.** Додаток зберігає інформацію про книгах і авторів у бібліотеці. Кожна книга характеризується автором, назвою, видавництвом і роком видання. Реалізувати можливість редагування інформації про авторів, видавництвах і книгах. Реалізувати можливість збереження, вилучення та зміни даних про книгах, видавництвах і авторів у таблиці БД.

**Варіант 6.** Додаток зберігає інформацію про аудіо-бібліотеці. Кожен запис характеризується автором, жанром, назвою і альбомом. Реалізувати можливість редагування інформації про авторів, жанрах і записах. Реалізувати можливість збереження, вилучення та зміни даних про авторів та аудіо-записи в таблиці БД.

* 1. **Зміст звіту.**Звіт повинен мати:
* титульний лист із найменуванням лабораторної роботи і даними виконавця;
* мету роботи;
* завдання;
* текст програми з коментарями і поясненнями;
* результат роботи програми;
* висновки.
  1. **Контрольні запитання**

1. Наведіть основні класи, що використовуються для пыдлкючення до баз даниз у ADO.NET

2. Для чого в ADO.NET використовується клас DataAdapter?

3. Наведить основні способи роботи з базами даних, які використовуються у C#

4. Чим технологія ADO.NET принципово відрізняється від технології Entity Framework?

5. Наведіть основні недоліки технології ADO.NET

6. Що таке команда в ADO.NET, де вона використовується?

7. Що таке транзакція, чим вона відрізняється від команди?

8. Для чого використовується клас Parameter при роботы з базами даних?

9. Що таке хранима процедура, як її створити в ADO.NET

10. Дайте визначення терміну ORM, наведіть основні існуючі ORM в .NET

**ЛАБОРАТОРНА РОБОТА №8**

**РОБОТА З РЕЄСТРОМ WINDOWS**

**7.1 Мета роботи**

Метою роботи є ознайомлення з основними можливостями роботи з реєстром ОС Windows.

**7.2 Вказівки з підготовки до виконання лабораторної роботи**

Необхідно вивчити наступні теми:

* Windows Registry
  1. **Теоретичні відомості**

У всіх версіях Windows, починаючи з Windows 95, системний реєстр є центральним сховищем всієї конфігураційної інформації, що стосується налаштування Windows, призначених для користувача переваг, а також встановленого програмного і апаратного забезпечення. Майже будь-яке комерційне програмне забезпечення в наші дні використовує реєстр для зберігання інформації про себе.

Компонентів СОМ теж потрібно зберігати інформацію про себе в реєстрі для того, щоб клієнти могли до них звертатися. Середа .NET Framework і пропонована в ній концепція установки з нульовим впливом (zeroimpact installation) робить реєстр трохи менш істотним для додатків, оскільки створювані з її допомогою збірки виходять повністю самодостатніми, тобто ніякої інформації до реєстру для них поміщати не потрібно, навіть якщо вони є розділяються.

До того ж в .NET Framework пропонується концепція ізольованого сховища (додатки можуть зберігати інформацію, що стосується кожного користувача, в файлах) і .NET Framework самостійно дбає про окремому зберіганні даних для кожного зареєстрованого в системі користувача.

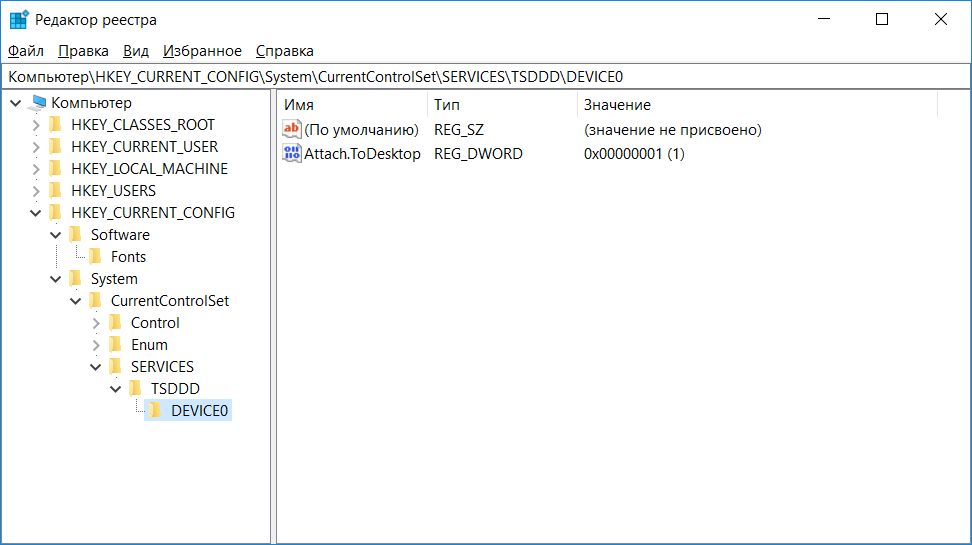
Той факт, що тепер додатки можуть встановлюватися за допомогою Windows Installer, звільняє розробників від необхідності проводити маніпуляції з реєстром, без яких установка додатків раніше була неможливою. Однак, незважаючи на це, залишається ймовірність, що розповсюджується додаток буде використовувати системний реєстр для збереження інформації про свою конфігурації.

Наприклад, якщо додаток повинно з'являтися в діалоговому вікні Add / Remove Programs (Установка й видалення програм) панелі управління, потрібно внести відповідні записи до реєстру. Системний реєстр також може знадобитися для забезпечення зворотної сумісності з успадкованим кодом.

Як і слід було очікувати від такої великої бібліотеки, як .NET, в ній поставляються класи, які дозволяють отримувати доступ до реєстру. Таких класів два - Registry і RegistryKey, і обидва вони знаходяться в просторі імен Microsoft.Win32. Перш ніж переходити до розгляду цих класів, давайте коротко ознайомимося зі структурою самого реєстру.

Реєстр має ієрархічну структуру, багато в чому схожу на структуру файлової системи. Зазвичай для перегляду і зміни вмісту реєстру застосовуються дві утиліти: regedit і regedt32. Утиліта regedit є стандартною і поставляється в усіх версіях Windows, починаючи з Windows 95. Утиліта regedt32 входить до складу Windows NT і Windows 2000, і в порівнянні з regedit менш дружня до користувача, але зате вона дозволяє отримувати доступ інформації, пов'язаної з безпекою, яку regedit переглядати не дозволяє.

У Windows Server 2003 ці утиліти були об'єднані в єдиний новий редактор, який називається просто regedit. В ході обговорень тут мається на увазі утиліта regedit, що поставляється в Windows 7, яку можна запустити, ввівши regedit в діалоговому вікні для запуску програм або в командному рядку.



На малюнку показано, як виглядає вікно regedit при запуску цієї утиліти. На цьому малюнку видно, що regedit має користувальницький інтерфейс в стилі "дерево / список", який схожий на інтерфейс провідника Windows і в точності відображає ієрархічну структуру самого реєстру. Однак незабаром будуть показані і деякі важливі відмінності:

У файлової системи вузли верхнього рівня представляють розділи дисків - З: \, D: \ і т.д. У реєстрі еквівалентом розділів дисків є розділи реєстру (registry hive). Змінювати існуючі розділи не допускається - вони є фіксованими і всього їх сім, хоча в інтерфейсі regedit видно тільки п'ять з них, які перераховані нижче:

HKEY\_CLASSES\_ROOT (HKCR) містить деталі, що стосуються типів файлів в системі (.txt, .doc і т.д.), і про те, які програми здатні відкривати файли кожного з цих типів. Також містить реєстраційну інформацію про всі компоненти СОМ (вона зазвичай займає найбільше місця в реєстрі, оскільки в Windows в даний час поставляється величезна кількість різноманітних СОМ-компонентів).

HKEY\_CURRENT\_USER (HKCU) містить деталі, що стосуються переваг користувача, який в поточний момент локально працює в системі. До числа цих переваг відносяться настройки робочого столу, змінні оточення, підключення до мереж і принтерів і інші параметри, які визначають зовнішній вигляд робочого середовища користувача.

HKEY\_LOCAL\_MACHINE (HKLM) являє собою величезний розділ, в якому містяться деталі, що стосуються всього встановленого на машині програмного і апаратного забезпечення. Також включає в себе розділ HKCU, який сам по собі не є окремим розділом, а просто зручним відображенням ключа реєстру HKLM / SOFTWARE / Classes.

HKEY\_USERS (HKUSR) містить деталі, що стосуються переваг всіх користувачів. Як не важко здогадатися, він теж містить розділи HKCU, які є просто відображенням відповідних ключів HKEY\_USERS.

HKEY\_CURRENT\_CONFIG (HKCF) містить деталі, що стосуються конфігурації встановленого на машині обладнання.

В інших двох кореневих розділах розміщується інформація, яка є тимчасовою і часто змінюється:

HKEY\_DYN\_DATA є загальним контейнером для будь-яких поточних даних, які потрібно зберігати де-небудь в реєстрі.

HKEY\_PERFORMANCE\_DATA містить інформацію, що стосується продуктивності функціонуючих в поточний момент додатків.

Всередині розділів знаходиться деревоподібна структура ключів реєстру. Кожен з них багато в чому нагадує папку або файл в файловій системі. Однак між ними є одна дуже важлива відмінність. У файлової системи розрізняються файли (в яких зберігаються дані) і папки (в яких головним чином містяться інші файли або папки), а в реєстрі присутні тільки ключі. У кожному ключі можуть міститися як дані, так і інші ключі.Если ключ содержит данные, то они представлены последовательностью значений. Каждое значение имеет ассоциированное с ним имя, тип данных и собственно данные. Вдобавок ключ имеет безымянное значение по умолчанию.

Всю цю структуру можна побачити, використовуючи утиліту regedit. На малюнку показано вміст ключа HKCU \ Console:

Записи в системному реєстрі можуть мати формат одного з трьох типів даних:

REG\_SZ (приблизно відповідає примірнику рядки в .NET, але ця подібність не точне, оскільки типи даних реєстру не є типами даних .NET);

REG\_DWORD (приблизно відповідає типу uint);

REG\_BINARY (масив байт).

Додаток, що передбачає збереження якихось даних в реєстрі, буде робити це за рахунок створення ряду ключів, причому зазвичай всередині ключа HKLM \ Software \ <Назва компанії>. Зверніть увагу, що в цих ключах зовсім не обов'язково повинні міститися будь-які дані. Часом сам факт існування ключа дозволяє додатку отримувати ті дані, які йому необхідні.

Як вже говорилося, доступ до реєстру дозволяють отримувати два класи з простору імен Microsoft.Win32: Registry і RegistryKey. Примірник RegistryKey представляє ключ реєстру. У цьому класі є методи для перегляду дочірніх ключів, для створення нових ключів, а також для читання і зміни значень в існуючих ключах, тобто можна виконувати все, що зазвичай потрібно робити з ключами реєстру, в тому числі встановлювати для них рівні безпеки. Саме цей клас застосовується для виконання більшої частини роботи з реєстром.

На відміну від нього, Registry є клас, який дозволяє отримувати ексклюзивний доступ до ключів реєстру для виконання простих операцій. Іншим призначенням класу Registry є надання примірників RegistryKey, що представляють ключі найвищого рівня, тобто розділи, які дозволяють здійснювати навігацію по реєстру. Надаються ці екземпляри через сім статичних властивостей з наступними іменами: ClassesRoot, CurrentConfig, CurrentUser, DynData, LocalMachine, PerformanceData і Users. Яким розділах відповідають ці властивості, має бути цілком очевидно.

Наприклад, для отримання примірника RegistryKey, що представляє ключ HKLM, потрібно написати такий код:

RegistryKey hklm = Registry.LocalMachine;

Процес отримання посилання на об'єкт RegistryKey називається відкриттям ключа. Хоча можна було б очікувати, що, оскільки реєстр має таку ж ієрархічну структуру, що і файлова система, методи, запропоновані класом RegistryKey, повинні бути схожими на ті, що реалізовані в DirectoryInfo, насправді це не так. Часто спосіб доступу до реєстру відрізняється від способу використання файлів і папок, і RegistryKey реалізує методи, що відображають це.

Найбільш очевидне відмінність пов'язана з тим, як відкривається ключ в певному місці реєстру. Клас Registry не має ніякого загальнодоступного конструктора, який можна було б використовувати, так само як не має і методів, які дозволяли б переходити до ключу на ім'я безпосередньо. Замість цього добиратися до потрібного ключа доведеться від самого верхнього розділу.

Якщо необхідно створити екземпляр об'єкта RegistryKey, то єдино можливий спосіб передбачає почати з відповідного статичного властивості класу Registry і звідти вже рухатися вниз. Тобто, наприклад, для читання даних з ключа HKLM \ Software \ Microsoft буде потрібно отримати на нього посилання в такий спосіб:RegistryKey hklm = Registry.LocalMachine;

RegistryKey hkSoftware = hklm.OpenSubKey("Software");

RegistryKey hkMicrosoft = hkSoftware.OpenSubKey("Microsoft");

Доступ до ключу подібним чином дозволяє здійснювати тільки читання. Якщо необхідно виконати запис якихось даних в ключ (в тому числі запис його значення, а також створення або видалення його безпосередніх дочірніх елементів), буде потрібно використовувати іншу перевизначену версію OpenSubKey. Вона приймає в другому параметрі значення типу bool, яке вказує, чи повинен доступ до ключа надаватися не тільки для читання, а й для запису. Наприклад, щоб отримати можливість змінити ключ Microsoft (маючи права системного адміністратора), буде потрібно написати наступний код:

RegistryKey hklm = Registry.LocalMachine;

RegistryKey hkSoftware = hklm.OpenSubKey("Software");

RegistryKey hkMicrosoft = hkSoftware.OpenSubKey("Microsoft", true);

До речі, оскільки в цьому ключі міститься інформація, яка використовується додатками Microsoft, в більшості випадків цей конкретний ключ модифікувати не варто.

Метод OpenSubKey () викликається у випадках, коли очікується, що ключ вже існує. Якщо ключа там не надається, OpenSubKey () повертає посилання null. Щоб створити ключ, необхідно використовувати метод CreateSubKey () (який автоматично надає доступ як для читання і запису до ключу через возвращаемую посилання):

RegistryKey hklm = Registry.LocalMachine;

RegistryKey hkSoftware = hklm.OpenSubKey("Software");

RegistryKey hkMine = hkSoftware.CreateSubKey("MyOwnSoftware");

Спосіб, яким працює метод CreateSubKey (), є досить цікавим. Він створює ключ, якщо його не існує, а якщо ключ існує, метод просто повертає екземпляр RegistryKey, який представляє існуючий ключ. Причина, по якій цей метод веде себе подібним чином, пов'язана з тим, як зазвичай використовується реєстр. У реєстрі в цілому містяться підлягають тривалому зберіганню дані на зразок конфігураційної інформації для Windows і різних додатків. Тому необхідність створювати ключ явно виникає не дуже часто.

Набагато частіше додатків потрібно перевіряти наявність деякою інформацією в реєстрі - іншими словами, створювати ключі, якщо вони ще не існують, і нічого не робити, якщо ключі вже там є. Метод CreateSubKey () просто ідеально дозволяє задовольнити таку потребу. На відміну від ситуації з FileInfo.Open (), в разі застосування CreateSubKey () випадкове видалення яких-небудь даних виключається. Якщо ж насправді необхідно видалити якісь ключі реєстру, потрібно викликати метод DeleteSubKey ().

З огляду на важливість реєстру для Windows, в цьому є сенс. Навряд чи комусь сподобається перспектива випадково вивести систему Windows з ладу, просто видаливши кілька важливих ключів під час налагодження самостійно написаного коду С #, який працює з системним реєстром.

Після виявлення підлягає читання або модифікації ключа можна за допомогою методу SetValue () або GetValue (), відповідно, встановити або витягти дані з нього. Обидва ці методу приймають в якості параметра рядок з ім'ям значення, але SetValue () також потрібно додатково передати в якості параметра посилання на об'єкт, який містить деталі, що стосуються значення. Оскільки за визначенням цей параметр повинен являти собою посилання на об'єкт, він насправді може бути посиланням на будь-який клас. За типом наданого класу метод SetValue () і буде визначати тип для значення - REG\_SZ, REG DWORD або REG\_BINARY. наприклад:

RegistryKey hkMine = HkSoftware.CreateSubKey("MyOwnSoftware") ;

hkMine.SetValue("MyStringValue", "Hello World");

hkMine.SetValue("MyIntValue", 20);

У цьому коді для ключа встановлюються два значення: MyStringValue з типом REG\_SZ і MyIntValue з типом REG\_DWORD.

**8.4. Варіанти завдань**

**Варіант 1** - Розробити додаток WindowsForms, яке після натискання на кнопку може змінювати колір робочого столу на вказаний. Передбачити можливість повернення початкового кольору робочого столу (при виході з програми або при виборі відповідного пункту меню)

**Варіант 2** - Розробити додаток WindowsForms, призначене для редагування текстових документів. Ім'я шрифту може встановлюватися через відповідний діалог і зберігатися / вилучатись з реєстру.

**Варіант 3** - Розробити додаток WindowsForms, призначене для редагування текстових документів. Розмір шрифту може встановлюватися через відповідний діалог і зберігатися / вилучатись з реєстру.

**Варіант 4** - Розробити додаток WindowsForms, призначене для редагування текстових документів. Список останніх 3 відкритих файлів зберігається і витягується з реєстру.

**Варіант 5** - Розробити додаток WindowsForms, призначене для редагування текстових документів. Кількість запусків програми зберігається в реєстрі. При кожному запуску програми виводити діалог із зазначенням кількості запусків.

**Варіант 6** - Розробити додаток WindowsForms, призначене для редагування текстових файлів. Дата і час першого запуску програми зберігається в реєстрі. У діалоговому вікні «Про програму» виводити дату і час першого запуску програми

**Варіант 7** - Розробити додаток WindowsForms, що дозволяє приховувати зазначені диски (A, B, C, D) при відображенні їх в Провіднику і вікнах відкриття / Збереження. Передбачити режим повернення початкового значення відповідного ключа реєстру.

**Варіант 8** - Розробити додаток Windows Forms, що дозволяє обмежувати список запускаються Windows додатків. Передбачити режим повернення початкового значення відповідного ключа реєстру.

**Варіант 9** - Розробити додаток WindowsForms, що дозволяє вивести інформацію про версію і виробника BIOS і типі центрального процесора (гілка реєстру HKLM / HARDWARE / DESCRIPTION)