МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ
імені ІГОРЯ СІКОРСЬКОГО»

**Д.В. Настенко**

**А.Б. Нестерко**

**Г.О. Труніна**

# Основи об’єктно-орієнтованого програмування

# лабораторний практикум

*Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського як навчальний посібник для здобувачів ступеня бакалавра за спеціальністю 141 «Електроенергетика, електротехніка та електромеханіка» освітня програма «Управління, захист та автоматизація енергосистем»*

Навчальний посібник

Київ
КПІ ім. Ігоря Сікорського
2022

Основи об'єктно-орієнтованого програмування. Лабораторний практикум [Електронний ресурс]: навч. посіб. для студ. спеціальності 141 «Електроенергетика, електротехніка та електромеханіка» / КПІ ім. Ігоря Сікорського; уклад.: Д.В. Настенко, А.Б. Нестерко, Г.О. Труніна – Електронні текстові данні (1 файл: 647 КБ, pdf). – Київ: КПІ ім. Ігоря Сікорського, 2022. – 60с.
Реєстр. № 21/22-750

*Гриф надано Методичною радою КПІ ім. Ігоря Сікорського (протокол № 6 від 24.06.2022 р) за поданням Вченої ради Факультету електроенерготехніки та автоматики (протокол № 9 від 17.05.2022 р.)*

# ОСНОВИ ОБ'ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

## лабораторний практикум

| | |
|---|---|
| Укладачі: | *Настенко Дмитро Васильович*, ст.викл., *Нестерко Артем Борисович*, к.т.н., доц., *Труніна Ганна Олексіївна*, к.т.н., ст.викл. |
| Відповідальний редактор: | *Яндульський О.С.*, д.т.н., проф. |
| Рецензент: | *Кацадзе Т.Л., к.т.н., доц.* КПІ ім.Ігоря Сікорського, ФЕА, кафедра електричних мереж |

Посібник містить матеріали 10-ти занять, які включають в себе теоретичні відомості та приклади для вирішення інженерних завдань у сфері енергопостачання за допомогою алгоритмічної мови C#: навчає розробляти програмні проекти для комп’ютерів на основі використання технології об’єктно-орієнтованого програмування. Для кожного заняття приведено індивідуальні завдання для студентів, виконання яких дозволить закріпити знання з використанням сучасної алгоритмічної мови програмування C#.

**ЗМІСТ**

# ВСТУП

Розвиток економіки, промисловості, науки і техніки, сфери освіти сьогодні значною мірою залежить від масового запровадження та використання обчислювальної техніки. Це вимагає підготовки і перепідготовки фахівців з програмування і використання персональних комп'ютерів.

Вибір мови програмування C# пояснюється такими чинниками:

- простотою і природністю основних конструкцій мови, що дозволяє швидко її освоїти і створювати алгоритмічно складні програми;
- можливістю використання розвинених засобів подання структур даних, що забезпечує зручність роботи як з числовою, так і з символьною інформацією;
- відповідністю принципам об'єктно-орієнтованого програмування, що робить програми наочними;
- наявністю бібліотеки процедур і функцій для роботи як з текстовою, так і з графічною інформацією, що дозволяє створювати досить складні програми.

# ВИМОГИ БЕЗПЕКИ ПІД ЧАС РОБОТИ В ЛАБОРАТОРІЇ ОБЧИСЛЮВАЛЬНОЇ ТЕХНІКИ

- Ці правила є обов'язковими для всіх студентів та інших осіб, які працюють в лабораторіях постійно чи тимчасово.
- Перевірка знань цих правил проводиться:
    - обслуговуючого персоналу – завідуючим лабораторією;
    - студентів – викладачами, які є керівниками лабораторних робіт.
- Після перевірки знань цих правил, кожен з тих, хто працює в лабораторії ставить свій підпис в спеціальному журналі. Без цього підпису ніхто до роботи в лабораторії не повинен бути допущеним.
- Дотримання правил з техніки безпеки повинно гуртуватися на високій свідомості всіх, хто працює в лабораторіях. Кожен, хто помітить порушення правил, а також несправність, яка являє собою небезпеку для людей і обладнання, повинен сповістити про це керівника.
- Робота в лабораторії по виконанню конкретного завдання може проводитися тільки після ретельного ознайомлення студентів з обладнанням і роботою, та чіткого уявлення про те, які елементи установки будуть під напругою та дотик до яких є небезпечними у стані роботи.
- Забороняється виконання ремонтних робіт на обладнанні, яке знаходиться під напругою.
- Забороняється наближатися або торкатися до струмоведучих частин, що обертаються, або усувати несправності без відключення установок.
- Забороняється проводити переключення в схемі, яка знаходиться під напругою.
- Перевірку наявності напруги дозволяється проводити тільки за допомогою вольтметра.
- Апарати управління та вимірювальні прилади слід розташовувати так, щоб було зручно вести спостереження за приладами, не перегинаючись через проводи та апарати.
- У випадку виникнення будь-яких несправностей необхідно негайно

вимкнути живленим установки та сповістити керівника занять про це.

- Кнопки управління, рубильники встановлювати в легкодоступних місцях для швидкого виключення схеми.
- У випадку припинення подачі електроенергії в лабораторію всі установки в лабораторії обов'язково вимикаються на робочих місцях.
- В лабораторіях категорично забороняється:
    - Палити в усіх приміщеннях, крім спеціально відведених для цього місць;
    - Прокладати без дозволу постійні та тимчасові лінії;
    - Користуватися побутовими електронагрівальними приладами;
    - Користуватися зіпсованим електрообладнанням, саморобними запобіжниками, провідниками із зіпсованою ізоляцією та саморобними електросвітильниками;
    - Проводити в непристосованих приміщеннях обмивку та фарбування деталей горючими рідинами та фарбниками;
    - Зберігати паливно-мастильні матеріали, хімікати та інші горючі речовини;
    - Загромаджувати проходи в лабораторіях;
- Всі, хто працює в лабораторії повинні знати, де знаходиться аптечка з медикаментами для наданим першої допомоги.
- При ураженні людини електричним струмом треба негайно вимкнути напругу, надати першу допомогу та покликати лікаря.
- Порятунок осіб, які постраждали, залежить від тою, як швидко вони будуть звільнені від електричного струму та як швидко їм буде надано першу допомогу.
- Першу допомогу необхідно надати негайно на місці події.
- Переносити людину, яка постраждала в інше місце необхідно тільки в тих випадках, коли небезпека продовжує загрожувати або надання допомоги на місці неможливе.
- При відсутності у постраждалого дихання, серцебиття, пульсу ніколи не

треба ставити під сумнів необхідність першої допомоги, тому що при ураженні електричним струмом смерть часто буває несправжньою. Тільки лікар може дати висновок про смерть постраждалого.

- До приїзду лікаря постраждалому необхідно надати допомогу і провести штучне дихання з дотриманням всіх правил надання першої допомоги.
- Про випадок негайно треба сповістити керівництво кафедри, деканату та інституту.
- Недотримання цих вимог не дозволяється. Якщо розпорядження суперечить діючим правилам, необхідно надати роз'яснення з приводу неухильною виконання цих правил і довести це до відома керівництва.

# ВИМОГИ ДО ВИКОНАННЯ ЛАБОРАТОРНИХ РОБІТ

Підготовка до кожної роботи проводиться студентами самостійно в поза аудиторний час. Студенти ознайомлюються із теоретичними відомостями, пишуть необхідні програми відповідно до отриманого індивідуального завдання на роботу.

Усі лабораторні роботи виконуються на мові програмування C#. Під час занять студенти проводять тестування написаних програм, тобто запускають їх в інтегрованому середовищі розробки на персональних комп'ютерах, займаються налагодженням і виконують необхідні розрахунки. Після виконання роботи студент оформляє звіт, який складається з таких розділів:

1. Назва, тема, мета роботи, стислі теоретичні відомості
2. Індивідуальне завдання
3. Текст програми
4. Результати виконання програми

Робота оформлюється в друкованому вигляді на папері стандартного формату А4.

Під час захисту роботи необхідно відповісти на контрольні питання і вміти пояснити роботу програми.

# 1. ТЕМА. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ. ЛАБОРАТОРНІ РОБОТИ №1 ТА №2

**Мета роботи** – знайомство з механізмом перевантаження операторів.

## 1.1. Теоретичні відомості

Перевантаження операторів дозволяє дати визначення операторам для класів створених користувачем. Головна перевага механізму перевантаження операторів в тому, що він дозволяє використовувати природній синтаксис при написанні програм. Що приводить до зменшення витрат на написання програм.

Синтаксис перевантаження:

```
public static <тип_значення> operator <унарний_оператор>
(<тип_параметру> <назва_параметру>)

public static <тип_значення> operator <бінарний_оператор> (
      <тип_параметру1> <назва_параметру1>,
      <тип_параметру2> <назва_параметру2>
)
```

тут

`тип_значення` – тип результату дії оператора;

`унарний_оператор` – один з операторів **+, -, !, ~, ++, --, true, false;**

`бінарний_оператор` – один з операторів **+, -, *, /, %, &, |, ^, <<, >>, ==, !=, >, <, >=, <=;**

`тип_параметру` – тип параметру в операторі;

`назва_параметру` – тип параметру в операторі.

Приклад перевантаження:

```
using System;
public class Point{
    double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    // при додаванні крапок утворюється нова,
    // координати якої рівні сумі відповідних координат
    public static Point operator +(Point a, Point b){
        return new Point(a.x+b.x,a.y+b.y);
    }
    // при множенні крапки на число утворюється нова,
    // координати якої рівні добутку відповідних
```

```
    // координат на число
    public static Point operator *(Point a, double d) {
        return new Point(a.x * d, a.y * d);
    }
    // переозначимо метод ToString()
    public override string ToString() {
        return String.Format("Крапка({0},{1})", this.x, this.y);
    }
}
class UsingOperatorOverload {
    static void Main(string[] args) {
        Point p1 = new Point(5, 8);
        Point p2 = new Point(7, 12);
        Console.WriteLine(p1 + "+" + p2 + "=" + (p1 + p2));
        Console.WriteLine(p1 + "*" + 5 + "=" + (p1 * 5));
        Console.ReadKey();
    }
}
```

Ця програма виведе на екрані:

```
Крапка(5,8)+Крапка(7,12)=Крапка(12,20)
Крапка(5,8)*5=Крапка(25,40)
```

Те що в цій програмі перевантажено метод `ToString()`, дозволяє спростити вивід за допомогою методу `Console.WriteLine`, так як метод `ToString()` використовується за замовчуванням при перетворенні об'єкту в рядок.

Круглі дужки використоуються у виразах `(p1 + p2)` та `(p1 * 5)` для того, щоб спочатку виконались дії над об'єктами крапка, а вже потім перетворення в рядок за допомогою методу `ToString()`.

Слід зазначити, що оператори **==** та **!=**, **>** та **<**, **>=** та **<=** потрібно перевантажувати тільки парами.

Для прикладу розглянемо порівняння крапок:

```
    public static bool operator ==(Point a, Point b) {
        if(a.x == b.x && a.y == b.y) {
            return true;
        } else {
            return false;
        }
    }
    public static bool operator !=(Point a, Point b) {
        return !(a == b);
    }
```

Як бачимо з цього прикладу, при перевантаженні оператора "недорівнює", можна скористатися тим, що він обернений до оператора "дорівнює". Також

можна спростити код оператора "дорівнює", скориставшись тим що вираз `(a.x == b.x && a.y == b.y)` - повертає значення булівського типу:

```
    public static bool operator ==(Point a, Point b) {
        return a.x == b.x && a.y == b.y;
    }
```

Текст програми повністю:

```
using System;
public class Point{
    double x, y;
    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public override string ToString() {
        return String.Format("Крапка({0},{1})", this.x, this.y);
    }

    public static bool operator ==(Point a, Point b) {
        return a.x == b.x && a.y == b.y;
    }
    public static bool operator !=(Point a, Point b) {
        return !(a == b);
    }
}
class OperatorOverload {
    static void Main(string[] args) {
        Point p1 = new Point(5, 8);
        Point p2 = new Point(7, 12);
        Console.WriteLine(p1 + " та " + p2 +
            (p1 == p2 ? " - рівні " : " - не рівні"));
        Console.ReadKey();
    }
}
```

Результат роботи програми:

```
Крапка(5,8) та Крапка(7,12)- не рівні
```

Крім вищезгаданих операторів можна перевантажити оператор [], але для цього використовують спеціальну властивість – ***індексатор***. Індексатор можна використовувати для доступу до різних полів об’єкта або полів-масивів, оголошуються вони за допомогою ключового слова `this` та квадратних дужок `[]`.

Приклад використання індексатору для різних полів:

```
using System;
public class Car {              //клас машина
    public string color;        //колір
    public string model;        //модель
```

```
    public int yearBuilt;       //рік випуску
    // індексатор
    public string this[int i] {
        // властивість на читання
        get {
            switch(i) {
                case 0:
                    return model;
                case 1:
                    return color;
                case 2:
                    return yearBuilt.ToString();
                default:
                    return null;
            }
        }
        // властивість для запису
        set {
            switch(i) {
                case 0:
                    model=value;
                    break;
                case 1:
                    color=value;
                    break;
                case 2:
                    yearBuilt = Convert.ToInt32(value);
                    break;
                default:
                    break;
            }
        }
    }
}
class IndexatorOverload {
    static void Main(string[] args) {
        Car c = new Car();
        c[0] = "Audi";
        c[1] = "Green";
        c[2] = "2000";
        for(int i = 0; i < 3; i++) {
            Console.WriteLine(c[i]);
        }
        Console.ReadKey();
    }
}
```

На екрані буде наступне:

```
Audi
Green
2000
```

В цій програмі використання індексатора дозволяє поставити у відповідність кожному полю класа індекс і звертатися до полів по імені об’єкта за допомогою цього індекса.

Приклад використання індексатору для масиву:

```
using System;
public class Vector {
    int[] v;
    // конструктор
    public Vector(int size) {
        v = new int[size];
    }
    // метод
    public int Length() {
        if(v == null) {
            return 0;
        } else {
            return v.Length;
        }
    }
    // індексатор
    public int this[int i] {
        get { return v[i]; }
        set { v[i] = value; }
    }
}
class Program {
    static void Main(string[] args) {
        Vector vect = new Vector(5);
        for(int i = 0; i < vect.Length(); i++) {
            vect[i] = i * 2;
            Console.WriteLine("vect[{0}] = {1}", i, vect[i]);
        }
        Console.ReadKey();
    }
}
```

Програма виведе на екран:

```
vect[0] = 0
vect[1] = 2
vect[2] = 4
vect[3] = 6
vect[4] = 8
```

Як бачимо з цього прикладу, використання індексатору дозволяє звертатися до елементів поля масиву `v`, як до елеметів об’єкту `vect`.

## 1.2. Порядок виконання робіт

Ознайомитись із теоретичним матеріалом.

1. Вибрати варіанти першого та другого завдання згідно номеру в групі.
2. Написати програму на мові C#, яка б реалізувала вказані дії за допомогою класів та перевантаження операторів.
3. В розробці програмі використати проект Windows Forms.
4. Початкові данні вводити за допомогою меню форми з файлів.
5. Результати роботи програми виводити в елемент керування типу RichTextBox.
6. Доповнити приклади перевантаженням методу `ToString()`.
7. В першому та другому завданні створити необхідні конструктори.
8. В другому завданні навести приклад використання індексатора.
9. В другому завданні перевіряти допустимі значення і генерувати необхідні винятки.
10. Налагодити програму на комп'ютері.
11. Підготувати звіт по роботі.

## 1.3. Лабораторна робота №1. Унарні та бінарні оператори

1. Додавання, множення та порівняння комплексних чисел.
2. Віднімання, ділення та порівняння комплексних чисел.
3. Ділення комплексного на дійсне, ділення та порівняння комплексних чисел.
4. Ділення комплексного на дійсне, множення та порівняння комплексних чисел.
5. Додавання, ділення та порівняння комплексних чисел.
6. Віднімання, множення та порівняння комплексних чисел.
7. Ділення комплексного на дійсне, додавання та порівняння комплексних чисел.
8. Ділення комплексного на дійсне, віднімання та порівняння комплексних чисел.
9. Унарний мінус, множення та порівняння комплексних чисел.
10. Унарний мінус, ділення та порівняння комплексних чисел.
11. Множення комплексного на дійсне, додавання та порівняння комплексних чисел.
12. Множення комплексного на дійсне, віднімання та порівняння комплексних чисел.
13. Множення комплексного на дійсне, множення та порівняння комплексних чисел.
14. Множення комплексного на дійсне, ділення та порівняння комплексних чисел.

Для 15 варіант 1, для 16 варіант 2 і т.д.

## 1.4. Лабораторна робота №2. Застосування індексаторів.

1. Додавання (поелементне) матриць.
2. Скалярний добуток векторів.
3. Додавання векторів.
4. Порівняння векторів.
5. Віднімання (поелементне) матриць.
6. Множення вектора на число.
7. Порівняння матриць.
8. Віднімання векторів.
9. Множення (поелементне) матриці на число.
10. Ділення (поелементне) матриці на число.

Для 11 та 21 варіант 1, для 12 та 22 варіант 2 і т.д.

## 1.5. Контрольні запитання.

1. Для чого потрібно перевантаження операторів?
2. Які оператори можна перевантажувати?
3. Яка особливість перевантаження операторів порівняння?
4. Що таке індексатор?
5. Які області застосування індексаторів?

# 2. ТЕМА. ВИВЧЕННЯ УСПАДКУВАННЯ ТА ПОЛІМОРФІЗМУ. ЛАБОРАТОРНІ РОБОТИ №3, №4 ТА №5.

**Мета роботи** – знайомство з двома основами об'єктно-орієнтованого програмування – успадкуванням та поліморфізмом.

## 2.1. Теоретичні відомості

***Успадкування*** – це властивість класу породжувати нащадків. Тобто: клас-нащадок (дочірній клас) успадковує від базового(батьківського) частину полів і методів, та може доповнюватись новими полями та методами.

***Поліморфізм*** – це властивість споріднених об'єктів вирішувати подібні проблеми різними способами.

Щоб один клас успадковував інший використовують наступний синтаксис:

```
class <ім'я_класу_нащадку> : <ім'я_базового_класу>
```

Для прикладу розглянемо базовий клас двовимірних графічних об'єктів:

```
public class Object2D {
    public int x, y; //поля-координати
    //конструктор
    public Object2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
    //метод для "відображення"
    public void Draw() {
        Console.WriteLine("Об'єкт з координатами {0},{1}",
          x, y);
    }
    //метод для "переміщення"
    public void Move(int dx, int dy) {
        x += dx;
        y += dy;
        Draw();
    }
}
```

маючи цей клас, не важко створити нові – крапку та коло:

```
// клас крапка
public class Point : Object2D {
    public Point(int x, int y)// конструктор Point посилається
        : base(x, y) {        // на батьківський конструктор –
                              // Object2D
        //тут може бути додатковий код
    }
}
```

тут ключове слово **base** означає, що при створенні об'єкту крапка спочатку викликається базовий конструктор `Object2D`, після чого в середині

конструктора `Point` можна виконати унікальні для цього класу дії. Що наочніше демонструється в класі – коло.

```
// клас коло
public class Circle : Object2D {
    public int r; //нове поле - радіус
    public Circle(int x, int y, int r)
        : base(x, y) {
        this.r = r;
    }
}
```

В цьому класі введено додаткове поле для радіусу – `r`, і значення поля радіус – задається у власному конструкторі `Circle` після виконання батьківського конструктора `Object2D`.

Обидва ці класи автоматично успадковують методи `Draw()` і `Move(int dx, int dy)` батьківського класу, і результат роботи цих методів ні чим не відрізнятиметься від визову аналогічних методів для об'єкту `Object2D`.

Проте більш цікавим є випадок коли методи відображення (`Draw()`) для кола та точки відрізнятимуться від базових. При цьому при оголошенні батьківського метода повинно використовуватись ключове слово `virtual` (можливий), а при оголошенні методу для дочірнього класу ключове слово – `override` (відхиляти). Наведемо повний приклад:

```
using System;
using System.Collections.Generic;
using System.Text;
public class Object2D {
    public int x, y; //поля-координати
    //конструктор
    public Object2D(int x, int y) {
        this.x = x;
        this.y = y;
    }
    //метод для "відображення"
    public virtual void Draw() {
        Console.WriteLine("Об'єкт з координатами {0},{1}",
          x, y);
    }
    //метод для "переміщення"
    public void Move(int dx, int dy) {
        x += dx;
        y += dy;
        Draw();
    }
}
// клас крапка
public class Point : Object2D {
    public Point(int x, int y)//конструктор Point посилається на
        : base(x, y) {        // батьківський конструктор -
```

```
                                // Object2D
    }
    public override void Draw() {
        //повністю перекриває батьківський метод
        Console.WriteLine("Крапка з координатами {0},{1}",
          x, y);
    }
}
// клас коло
public class Circle : Object2D {
    public int r; //нове поле - радіус
    public Circle(int x, int y, int r)
        : base(x, y) {
        this.r = r;
    }
    public override void Draw() {
        //спочатку викликає батьківський метод
        base.Draw();
        //потім доповнює його
        Console.WriteLine("це коло радіусом {0}", r);
    }
}
class UsingOverloadedObjects {
    static void Main() {
        Point p0 = new Point(10,20);
        Point p1 = new Point(1, 40);
        Circle c0 = new Circle(15, 17, 10);
        p0.Draw();
        p1.Draw();
        c0.Draw();
        Console.ReadKey();
    }
}
```

Ця програма виведе на екран наступну інформацію:

```
Крапка з координатами 10,20
Крапка з координатами 1,40
Об'єкт з координатами 15,17
це коло радіусом 10
```

Крім того, з усіма об’єктами нащадками базового класу можна спільно працювати, як з об’єктами базового класу. Для прикладу залишимо класи `Object2D`, `Point` та `Circle` без змін перепишемо тільки клас `UsingOverloadedObjects`:

```
class UsingOverloadedObjects {
    static void Main() {
        Point p0 = new Point(10,20);
        Point p1 = new Point(1, 40);
        Circle c0 = new Circle(15, 17, 10);
```

```
        //створено масиву вказівників на об'єкти Object2D
        Object2D[] objects = new Object2D[3];
        //спочатку всі елементи цього масиву рівні null
        //заповнимо його посиланнями на створені об'єкти
        objects[0] = p0;
        objects[1] = p1;
        objects[2] = c0;
        //тепер над ними можна виконувати групові дії
        //відображення
        foreach(Object2D o in objects) {
            o.Draw();
        }
        Console.WriteLine("\n\rпісля зміщення\n\r");
        //зміщення
        foreach(Object2D o in objects) {
            o.Move(5, -4);
        }
        Console.ReadKey();
    }
}
```

Результатом роботи цієї програми буде:

```
Крапка з координатами 10,20
Крапка з координатами 1,40
Об'єкт з координатами 15,17
це коло радіусом 10

після зміщення

Крапка з координатами 15,16
Крапка з координатами 6,36
Об'єкт з координатами 20,13
це коло радіусом 10
```

Слід зазначити, що взагалі всі класи .Net є нащадками базового класу **System.Object.** Це наслідування відбувається неявно, без буд-яких вказівок. Від **System.Object** всі класи успадковують ряд корисних методів, серед яких:

- `public Type` **`GetType`**`()` – повертає об'єкт класу `Type`, що містить детальну інформацію поточний об'єкт (до якого класу відноситься, чи є об'єкт масивом, чи являється екземпляром класу або перерахуванням і т. д.
- `public virtual string` **`ToString`**`()` – повертає представлення об'єкту у вигляді рядка (для більності об'єктів – це ім'я класу, а для чисел значення у вигляді рядку), може перевантажуватись.
- `public virtual bool` **`Equals`**`(Object obj)` для порівняння поточного об'єкту з іншими), може перевантажуватись, повертає `true`, якщо об'єкти рівні.
- `public static bool` **`Equals`**`(Object objA, Object objB)` – для

порівняння двох різних об’єктів.

**Оператори `is` та `as`.**

Оператор **`is`** – використовується для перевірки сумісності змінної або об’єкта одного типу з іншим типом даних і повертає значення **`true`** або **`false`**.

Наприклад:

```
int i;
if(i is IComparable){
    Console.WriteLine("Підтримує IComparable");
}
```

Дасть ствердну відповідь.

Оператор **`as`** – використовується для перетворення змінної одного типу в інший сумісний тип. Якщо типи не сумісні поверне **`null`**. Тому наступний фрагмент коду:

```
Point p0 = new Point(1, 2);
IConvertible c = p0 as IConvertible;
if(c == null) {
    Console.WriteLine("Не підтримує IConvertible");
}
```

Видасть:

```
Не підтримує IConvertible
```

**Елемент керування PropertyGrid.**

**PropertyGrid** – надає інтерфейс для перегляду та зміни властивостей об’єкту. Для задання об’єкту, властивості якого необхідно переглянути, використовується властивість, необхідно прописати посилання на нього у властивості **SelectedObject** елементу керування **PropertyGrid.** Наприклад:

```
this.propertyGrid1.SelectedObject = this.button1;
```

дозволяє переглянути та змінити властивості кнопки на формі (рис 2.1).

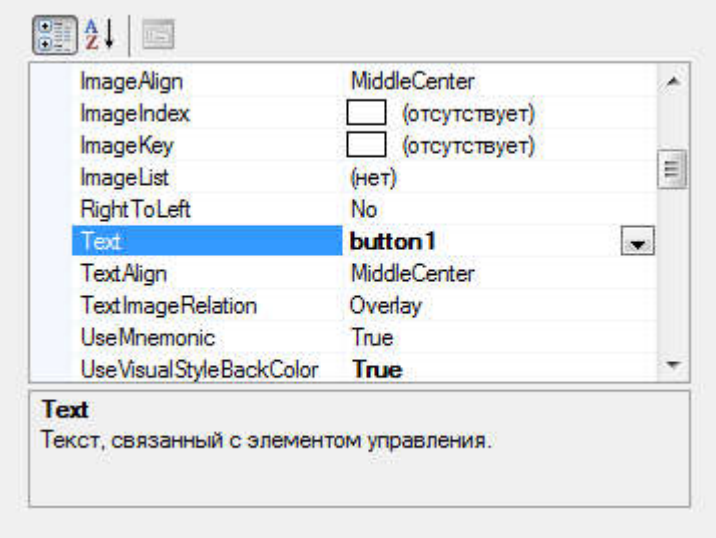


Рисунок 2.1 – Властивості кнопки в PropertyGrid

**Модифікатори abstract, new, sealed**

Модифікатор **abstract** вказує на те, що елемент має відсутню або неповну реалізацію. Його можна використовувати з класами, методами, та властивостями.

При оголошенні класу модифікатор abstract дозволяє вказати, що клас може бути тільки базовим класом для інших класів. Створювати об'єкти абстрактного класу не можна.

Члени класу (методи та властивості), помічені як абстрактні або включені в абстрактний клас, повинні бути реалізовані класами-нащадками абстрактного класу. Абстрактні методи неявно являються віртуальними методами. Оголошення абстрактних методів допустимо тільки в абстрактних класах. Так як при оголошенні абстрактних методів реалізація відсутня, то не потрібні і фігурні дужки для тіла методу, натомість в кінці оголошення ставиться крапка с комою (;).

Приклад:

```
abstract class BaseClass   // Abstract class
{
    protected int _x = 100;
    protected int _y = 150;
    public abstract void AbstractMethod();// Abstract method
    public abstract int X    { get; set; }
    public abstract int Y    { get; set; }
}
```

Модифікатор **new** явним чином приховує члени (методи та властивості), успадковані від базового класу. При приховуванні успадкованого члену класу його похідна версія замінює версію базового класу. Члени класу можна приховувати без використання ключового слова new, але в цьому випадку з'явиться попередження компілятора. У випадку ж використання new для явного приховування члену, попередження не з'явиться.

Пример:

```
public class BaseClass // Базовий клас
{
    public int x;
    public void Method() { }
}

public class DerivedClass : BaseClass // Похідний клас
{
    new public void Method() { }
}
```

При застосуванні до класу модифікатор **sealed** забороняє іншим класам ставати нащадком даного. Також модифікатор **sealed** можна використовувати для метода або властивості, що перевантажує віртуальний метод або властивість базового класу. Це заборонить класам-нащадкам перевантажувати певні віртуальні методи або властивості.

Приклад:

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}
class Y : X
{
    sealed protected override void F() {
Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2");
}
}
class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("C.F");
}

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2");
}
}
```

## 2.2. Порядок виконання робіт

## 2.3. Лабораторна робота №3. Ієрархія класів.

1. Ознайомитись із теоретичним матеріалом.
2. Вибрати варіант завдання згідно номеру в групі.
3. У консольному додатку побудувати ієрархію класів згідно варіанту.
4. Навести приклади перевантаження батьківських методів в класах-нащадках. Для цього у базового класу повинно бути 2-3 методи для перевантаження.
5. Базовий клас зробити абстрактним.
6. Продемонструвати роботу ключових virtual та override.
7. Перевірити роботу модифікаторів new та sealed.
8. Налагодити програму на комп'ютері.
9. Забезпечити вивід результатів роботи програми на екран комп'ютера.
10. Підготувати звіт по роботі.

Індивідуальні завдання

1. Трансформатор, трансформатор струму, силовий трансформатор, автотрансформатор.
2. Студент, викладач, людина, доцент, декан.
3. Пошкодження, трифазне кз, двохфазне кз, однофазне кз на землю, двофазне кз на землю.
4. Держава, республіка, монархія, королівство, демократія.
5. Службовець, персона, робітник, інженер, головний інженер.
6. Деталь, механізм, виріб, вузол, гвинт.
7. Організація, компанія, банк, страхова компанія, енергетична компанія.
8. Газета, журнал, книга, видання, підручник.
9. Тест, іспит, випробування, залік, диференційний залік.
10. Їжа, продукт, товар, хліб, послуга.
11. Квитанція, накладна, документ, рахунок, чек.
12. Автомобіль, поїзд, транспорт, експрес, автобус.
13. Електричний двигун, двигун, двигун внутрішнього згоряння, дизель, реактивний двигун.
14. Тварина, риба, ссавець, птах, примат.
15. Корабель, пароплав, вітрильник, фрегат, атомохід.

## 2.4. Лабораторна робота №4. Створення графічних класів.

1. Ознайомитись із теоретичним матеріалом.
2. Створити проект Windows Forms.
3. В проекті створити клас Object2D

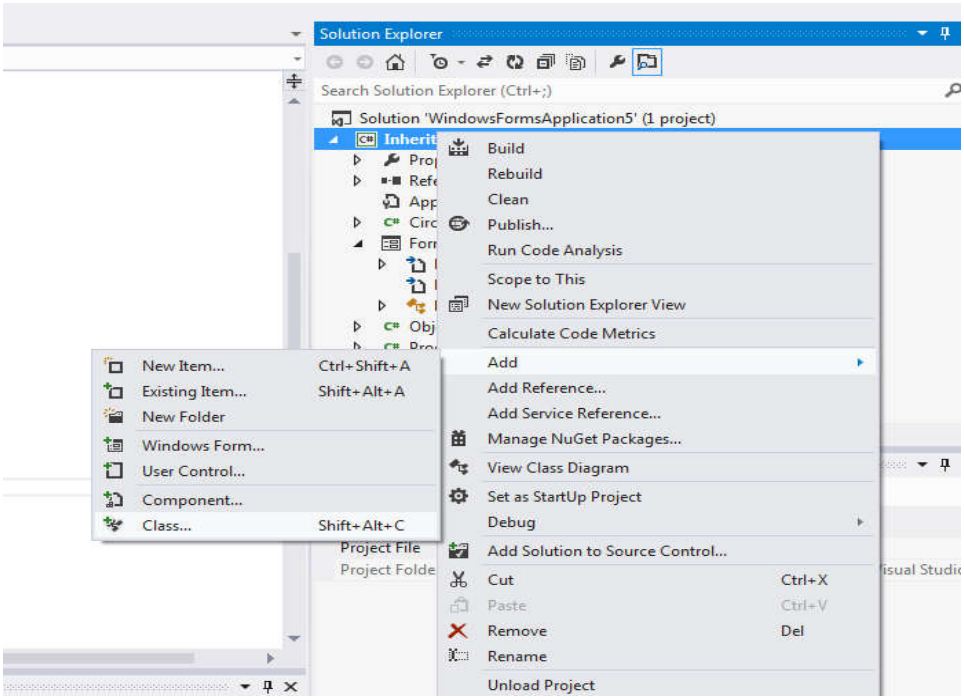


Рисунок 2.2 – Створення класу.

4. В цьому класі створити поля:

```
protected int x, y;
protected Color color;
```

5. Створити для Object2D властивості для полів, конструктор для заповнення значень полів та методи:

```
public virtual void Draw(Graphics gr){}

public void Move(int dx, int dy) {
    x += dx; y += dy;
}
```

6. Використовуючи Object2D як базовий клас, створити ієрархію класів еліпс, круг, прямокутник, квадрат, кожен в окремому файлі. В кожному з класів створити необхідний конструктор та перевантажити метод `Draw(Graphics gr)` для малювання зафарбованого об'єкту методами `Graphics`.
7. Продемонструвати роботу класів з відображенням їх на формі, для цього:
   а) Додати в класі форми поле типу список:

```
private List<Object2D> objects2D = new List<Object2D>();
```

   б) До класу форми додати метод для створення 2-3 об'єктів для кожного класу. Наприклад:

```
private void InitObjects() {

}
```

   Виклик методу додати в конструктор форми поруч з `InitializeComponent();`
   в) Прописати відображення об'єктів в події `Paint`, наприклад:

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
        Graphics gr = e.Graphics;
        foreach (Object2D obj in objects2D) {
            obj.Draw(gr);
        }
}
```

8. Налагодити програму на комп'ютері.
9. Забезпечити вивід результатів роботи програми на екран комп'ютера.
10. Підготувати звіт по роботі.

## 2.5. Лабораторна робота №5. Використання елементу курування PropertyGrid.

1. Додати **PropertyGrid** на форму.
2. Для всіх класів нащадків **Object2D** описати метод, який визначатиме чи потрапила крапка на об'єкт чи ні.
3. За допомогою цього метода, виводити властивості поточного об'єкту при натисканні на ньому лівої клавіши миши.
4. Зауваження: Щоб приховати зайві властивості від відображення в **PropertyGrid**, перед ними необхідно додати атрибут `[System.ComponentModel.Browsable(false)]`
   Наприклад для кола:

```
public int R {
    get { return r; }
    set { r = value; base.A = r; base.B = r; }
}

[System.ComponentModel.Browsable(false)]
public override int A {
    get {
        return r;
    }
    set {
        R = value;
    }
}
```

## 2.6. Контрольні запитання.

1. Дайте означення понять успадкування та поліморфізм.
2. Який синтаксис успадкування у мові C# ?
3. Коли використовується ключове слово **`base`**?
4. За допомогою яких ключових слів реалізується механізм поліморфізму?
5. Яким чином можна з дочірнього класу можна звертатись до батьківських методів?
6. Який об’єкт є базовим для всіх класів в середовищі .Net?
7. Які методи є спільними для всіх класів?

# 3. ТЕМА. ВИКОРИСТАННЯ СПИСКІВ, СТВОРЕННЯ ВЛАСНИХ СПИСКІВ. ЛАБОРАТОРНА РОБОТА №6

**Мета роботи** – знайомство з колекціями платформи .Net, навчитися використовувати динамічні масиви, бітові масиви та різні види списків.

## 3.1. Теоретичні відомості

**Поняття списків, черг та стеків. Класи List, Queue та Stack.**

**Лінійний список (List)** в інформатиці та програмуванні визначається як екземпляр абстрактного типу даних, що формалізує концепцію впорядкованої множини елементів. Це означає, що він складається з вузлів, кожен з яких містить як власні дані, так і посилання на наступний або/і попередній вузол (рис 1.1).

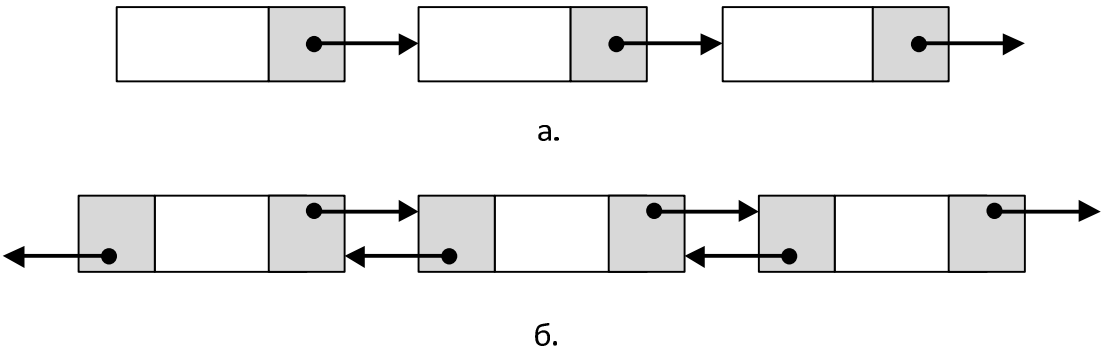


Рисунок 1.1 - Лінійні списки: а. однозв’язний, б. двозв’заний

**Стек (Stack)** в інформатиці та програмуванні – різновид лінійного списку, структура даних, яка працює за принципом (дисципліною) "останнім прийшов -- першим пішов" (LIFO, last in, first out). Всі операції (наприклад, видалення елементу) в стеку можна проводити тільки з одним елементом, який знаходиться на верхівці стеку та був введений в стек останнім.

Стек можна розглядати як певну аналогію до стопки тарілок, з якої можна взяти верхню, і на яку можна покласти верхню тарілку (інша назва стеку - "магазин", за аналогією з принципом роботи магазину в автоматичній зброї) рис. 1.2.

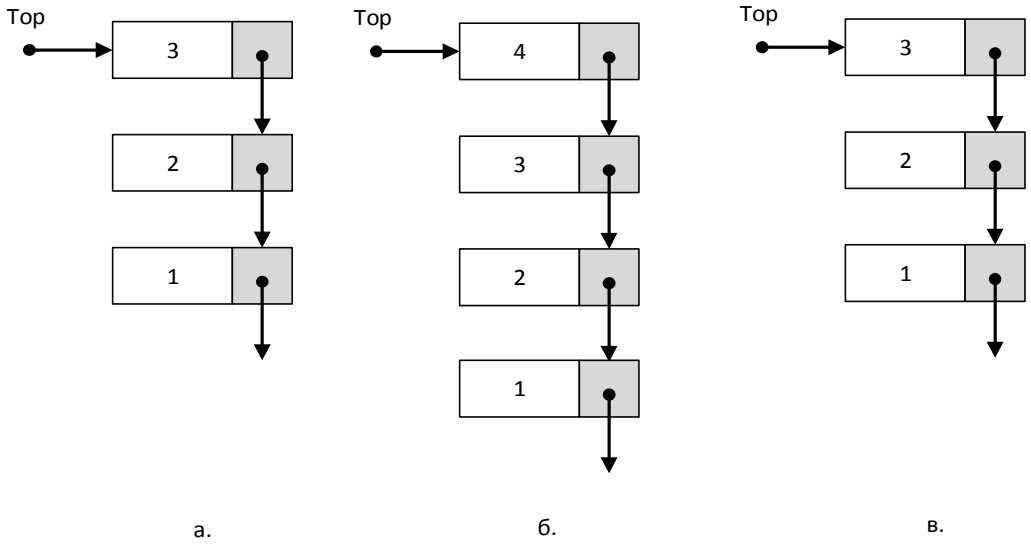


Рисунок 1.2 - Стек: а. початковий, б. додано елемент, в. вилучено елемент

На цьому рисунку Top – означає змінну, що вказує на вершину (перший елемент) списку.

**Черга (Queue [kju:])** в програмуванні — динамічна структура даних, що працює за принципом "перший прийшов - перший пішов" (англ. FIFO — first in, first out). У черги є голова (англ. head) та хвіст (англ. tail). Елемент, що додається до черги, опиняється в її хвості. Елемент, що видаляється з черги, знаходиться в її голові.

Така черга повністю аналогічна звичній "базарній" черзі, в якій хто перший встав в неї, той першим буде обслуженим (але, на відміну від реальної черги, імовірність пройти поза чергою виключена)

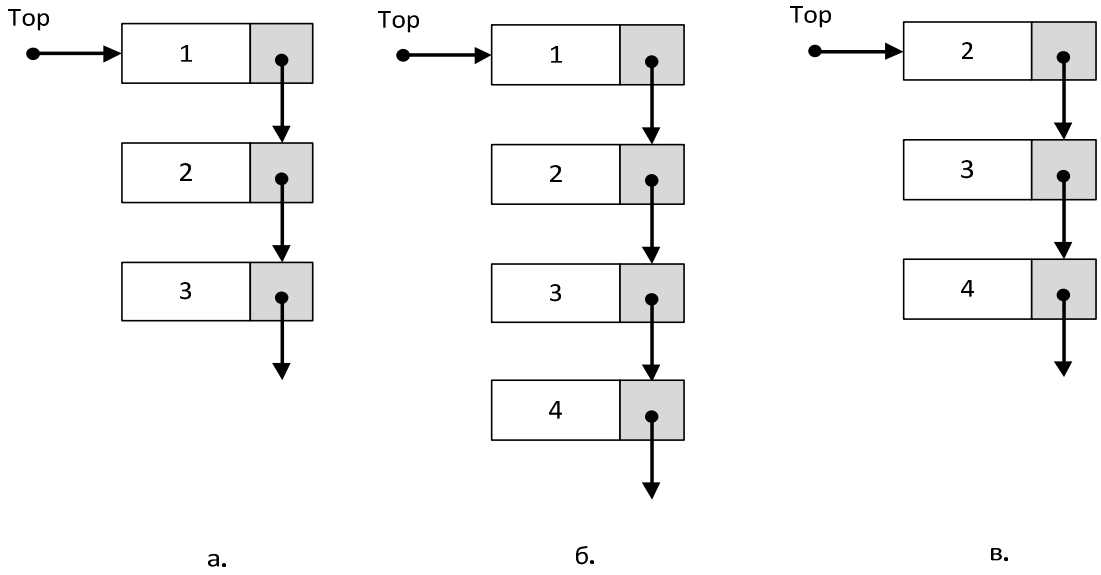

Рисунок 1.3 - Черга: а. початковий, б. додано елемент, в. вилучено елемент

Простір імен **System.Collections.Generic** містить інтерфейси та класи, що визначають універсальні колекції, які дозволяють користувачам створювати суворо типізовані колекції, що забезпечують підвищену продуктивність і безпеку типів порівняно з неуніверсальними строго типізований колекціями.

**List<T>** - клас, представляє строго типізований список об'єктів, доступних за індексом. Підтримує методи для пошуку за списком, виконання сортування та інших операцій зі списками.

**Queue<T>** - клас, представляє колекцію об'єктів, засновану на принципі "першим надійшов - першим обслуговано".

**Stack<T>** - клас, представляє колекцію змінного розміру примірників того ж довільного типу, що має тип "останнім прийшов - першим вийшов" (LIFO).

***Зауваження:*** Позначення **<T>** - означає, що треба в кутових дужках замість **T** вказати конкретний тип даних з яким буде працювати клас (Наприклад: **List<int> -** для роботи з цілими, **Queue<string>** - для роботи з рядками).

Приклад використання класу System.Collections.Generic.List:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace TestList {
    class Program {
        static void Main(string[] args) {
            List<string> myList = new List<string>();
            // Список пустий
            Console.WriteLine(
                 "\nCapacity:{0}", myList.Capacity
                 );

            // Заповнимо список
            myList.Add("Один");
            myList.Add("Два");
            myList.Add("Три");
            myList.Add("Чотири");
```

```
            // Вивід результату
            Console.WriteLine();
            foreach (string str in myList) {
                Console.WriteLine(str);
            }

            Console.WriteLine(
                 "\nCapacity:{0}", myList.Capacity
                 );
            Console.WriteLine(
                 "Count: {0}", myList.Count
                 );
            // Вставка в задану позицію
            Console.WriteLine(
                 "\nInsert(2, \"П'ять\")"
                 );
            myList.Insert(2, "П'ять");

            Console.WriteLine(
                 "\nCapacity:{0}", myList.Capacity
                 );
            Console.WriteLine(
                 "Count: {0}", myList.Count
                 );

            Console.WriteLine();
            foreach (string str in myList) {
                Console.WriteLine(str);
            }

            // Очистка списку
            myList.Clear();
            Console.WriteLine("\nClear()");
            Console.WriteLine(
                 "Capacity: {0}", myList.Capacity
                 );
            Console.WriteLine(
                 "Count: {0}", myList.Count
                 );
            Console.ReadKey();
        }
    }
}
```

Результат роботи програми:

```
Capacity:0
```

```
Один
Два
Три
Чотири

Capacity:4
Count: 4

Insert(2, "П'ять")

Capacity:8
Count: 5

Один
Два
П'ять
Три
Чотири

Clear()
Capacity: 8
Count: 0
```

Реалізуємо аналогічний клас List<T> з методом Insert самостійно. Для збереження даних у вузлах створимо клас Elem:

```
// Клас для збереження вузлів
class Elem {
    //поле даних
    internal T value;
    //посилання на наступний вузол
    internal Elem next;
    //коструктор
    internal Elem(T value, Elem next) {
            this.value = value;
            this.next = next;
    }
}
```

Він буде внутрішнім класом по відношенню до класу List<T>. В ньому заведено два поля `value` – для збереження даних типу T, та `next` – для посилання на наступний вузол в списку.

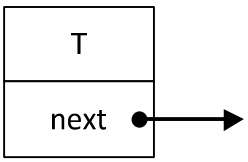


Рисунок 1.4 – Клас Elem.

Клас List<T>: матиме поле Top – для збереження вершини списку, реалізацію метода Insert(int pos, T obj) – для вставки об’єкта T в задану позицію pos, та перевантаження методу ToString() – для відображення вмісту списку.

В методі Insert можливо три випадки:

1. Список пустий (Top == null), тоді треба створити новий елемент, і вказати на нього Top Рис.1.5.
2. Список  не пустий, але вставка в початок (pos == 0) Рис.1.6.
3. Список не пустий і вставка в середину списку Рис 1. 7.

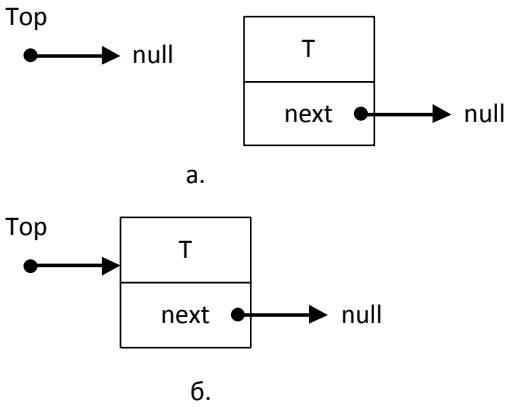


Рисунок 1.5 – Вставка в пустий список: а. до вставки,

б. після вставки

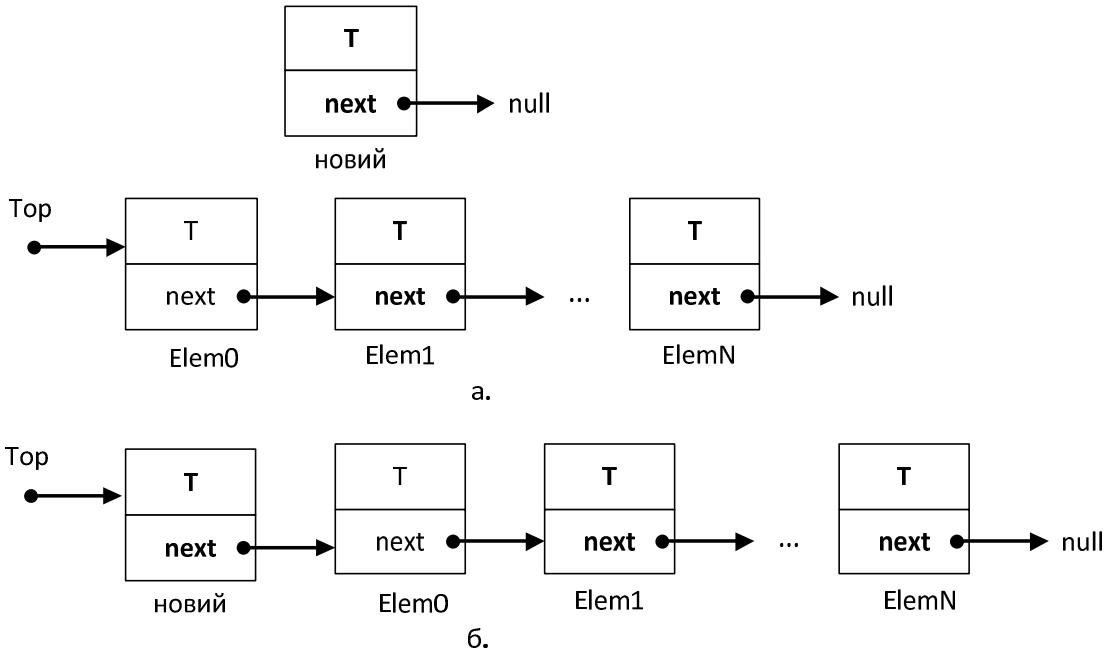


Рисунок 1.6 – Вставка в початок: а. до вставки,

б. після вставки

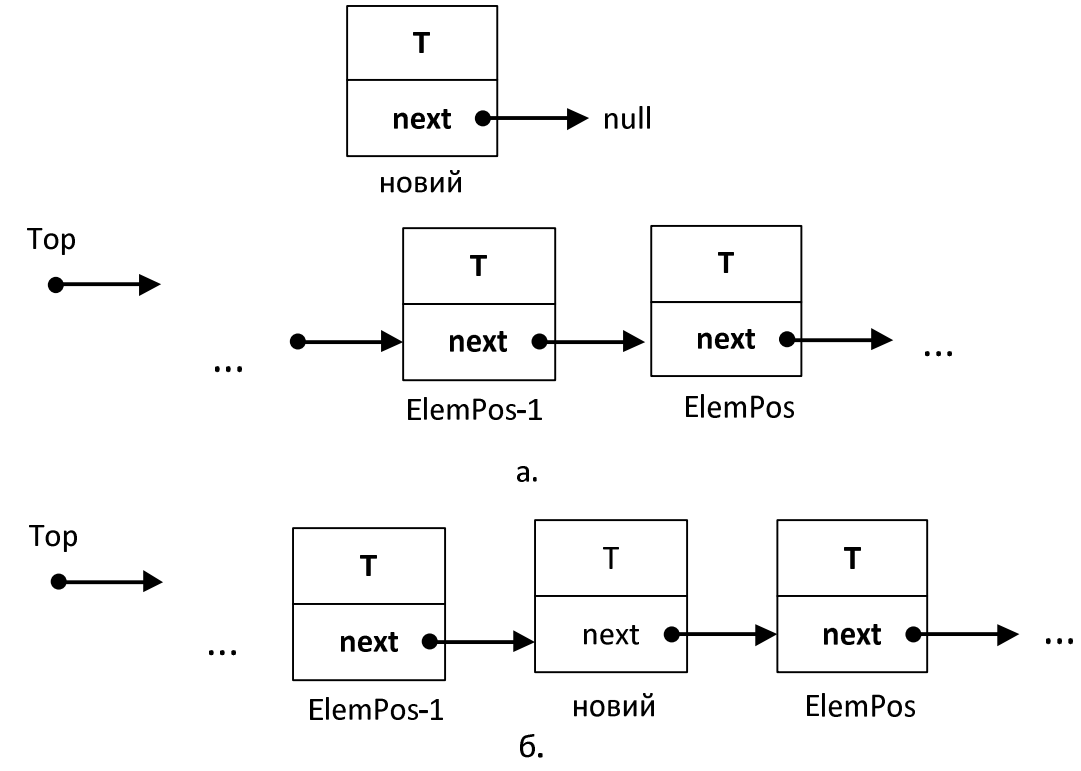


Рисунок 1.7 – Вставка в середину: а. до вставки,

б. після вставки

Програмна реалізація методу Insert.

```
// Метод вставки в задану позицію
public void Insert(int pos, T obj) {
```

```
            Elem e = Top;
            // Якщо список пустий або вставка в початок
            if (e == null || pos <= 0) {
                Top = new Elem(obj, Top);
                return;
            }
            // Пропускаємо pos - 1 елемент
            for (int i = 0; i < pos - 1; i++) {
                if (e.next != null)
                    e = e.next;
                else
                    break;
                }
            // Вставка в позицію pos
            e.next = new Elem(obj, e.next);
        }
```

В методі об’єднано випадки, якщо список пустий та вставка в початок.

Наведемо текст програми повністю:

```
using System;
using System.Text;

namespace MyList {
    class List<T> {
        // Клас для збереження вузлів
        class Elem {
            //поле даних
            internal T value;
            //посилання на наступний вузол
            internal Elem next;
            //коструктор
            internal Elem(T value, Elem next) {
                this.value = value;
                this.next = next;
            }
        }
        // Вершина списку
        Elem Top;
        // Метод вставки в задану позицію
        public void Insert(int pos, T obj) {
            Elem e = Top;
            // Якщо список пустий або вставка в початок
```

```
            if (e == null || pos <= 0) {
                Top = new Elem(obj, Top);
                return;
            }
            // Пропускаємо pos - 1 елемент
            for (int i = 0; i < pos - 1; i++) {
                if (e.next != null)
                    e = e.next;
                else
                    break;
            }
            // Вставка в позицію pos
            e.next = new Elem(obj, e.next);
        }
        // Перевантаження методу ToString()
        public override string ToString() {
            string s = String.Empty;
            for (Elem e = Top; e != null; e = e.next) {
                s += e.value.ToString() + "\n";
            }
            return s;
        }
    }
    class Program {
        static void Main(string[] args) {
            List<int> list = new List<int>();
            list.Insert(0, 0);
            list.Insert(0, 1);
            list.Insert(0, 2);
            Console.WriteLine("Було");
            Console.Write(list);
            list.Insert(1, 3);
            list.Insert(2, 4);
            Console.WriteLine("Стало");
            Console.Write(list);
            Console.ReadKey();
        }
    }
}
```

Результат роботи програми:

```
Було
2
1
0
```

```
Стало
2
3
4
1
0
```

## 3.2. Порядок виконання роботи

## 3.3. Лабораторна робота №6

1. Ознайомитись із теоретичним матеріалом (література, помічник F1 та бібліотека MSDN).
2. Запустити Visual Studio. Пуск/Програми/Microsoft Visual Studio/ Microsoft Visual Studio, або за допомогою ярлику на робочому столі.
3. Створити в Visual Studio новий проект File/New/Project…/ Visual C#/ ConsoleApplication та задати йому нове ім’я.
4. Дослідити властивості та методи стандартних класів List<T>, Queue<T>, Stack<T>. Реалізувати власні класи Queue та Stack з методами додавання та вилучення елементу, індексатором на читання та перевантаженням методу **`string ToString()`**. Причому парні номери в списку групи реалізують Queue<T>, а непарні – Stack<T>.
5. Навести приклад використання створеного класу в програмі.
6. Підготувати звіт по роботі.

Зауваження: для того щоб повернути значення за замовчуванням для типу T, можна використати ключове слово **`default`**, а саме **`default(T)`**.

Детальніше про це https://docs.microsoft.com/ru-ru/previous-versions/visualstudio/visual-studio-2012/xwth0h0d%28v%3dvs.110%29

## 3.4. Контрольні запитання.

1. Що таке список, черга та стек?
2. Який простір імен містить класи List<T>, Queue<T>, Stack<T>?
3. Що означає позначення <T>?
4. Які основні методи додавання та вилучення елементу в стеку?
5. Які основні методи додавання та вилучення елементу у черги?
6. Для чого використовується ключове слово **`default?`**

# 4. ТЕМА. ВИКОРИСТАННЯ БЛОКУ TRY…CATCH…FINALLY. ГЕНЕРАЦІЯ ВИНЯТКІВ. ЛАБОРАТОРНА РОБОТА №7

**Мета роботи** – навчитися обробляти помилки виконання, а також навчитись генерувати власні винятки.

## 4.1. Теоретичні відомості

**Винятки. Обробка винятків.**

**Винятки** генеруються, коли метод не може успішно виконуватись, інша назва – помилка виконання. Функції обробки винятків на мові C# допомагають обробляти будь-які непередбачувані або виняткові ситуації, що відбуваються при виконанні програми. При обробці винятків використовуються ключові слова **try**, **catch** і **finally**:

- Блок **try** – містить програмний код в якому може виникнути виняток;
- Блок **catch** – використовується, для обробки виняткових ситуацій, при відсутності винятків – програма в цей блок не заходить;
- Блок **finally** – виконується завжди, в незалежності від того було виключення чи ні. Зазвичай використовується для звільнення відкритих ресурсів (файлів, потоків, тощо).

```
try {
    // Блок коду, призначений для
    // обробки винятків.
} catch (ExcepType1 exOb) {
    // Обробник винятку типу ExcepType1
} catch (ExcepType2 exOb) {
    // Обробник винятку типу ExcepType2.
} finally {
    // Код завершення обробки винятку
}
```

Винятки можуть генеруватися середовищем CLR, платформою .NET Framework, зовнішніми бібліотеками, або кодом програми. Винятки створюються за допомогою ключового слова **throw**.

У багатьох випадках виняток може ініціюватися не методом, викликаним безпосередньо кодом, а іншим методом, розташованим нижче в стеку викликів. Коли це відбувається, середовище CLR виконує відкат стека в пошуках методу з

блоком catch для певного типу винятку. При виявленні першого такого блоку catch цей блок виконується. Якщо середу CLR не знаходить відповідного блоку catch де-небудь в стеку викликів, вона завершує процес і відображає користувачеві повідомлення.

Приклад обробки:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Sem6Lab4_00 {
    class Program {

        public int Test(int j) {
            try {
                int i = 1;
                i /= j;
                Console.WriteLine("OK");
                return 1;
            } catch(Exception ex) {
                Console.WriteLine("Exeption : "
                   + ex.Message);
                return 0;
            } finally {
                Console.WriteLine("Finaly");
            }
        }

        static void Main() {
            Program p = new Program();
            int i = p.Test(2);
            Console.WriteLine("Result = {0}", i);
            Console.WriteLine("----------------");
            i = p.Test(0);
            Console.WriteLine("Result = {0}", i);
            Console.ReadKey();
        }
    }
}
```

```
OK
Finaly
Result = 1
----------------
Exeption : Attempted to divide by zero.
Finaly
```

```
Result = 0
```

**Властивості винятків.**

Винятки мають наступні властивості:

1. Всі винятки – є нащадками класу **System.Exception**.
2. Слід використовувати блок **try** для розміщення в ньому операторів, виконання яких може призвести до винятків.
3. При виникненні винятку в блоці **try** потік керування негайно переходить до першого відповідного **try** обробника винятків, присутнього в стеку викликів. У мові C # ключове слово **catch** використовується для визначення обробника винятків.
4. Якщо обробника для певного винятку не існує, виконання програми завершується з повідомленням про помилку (рис. 4.1).

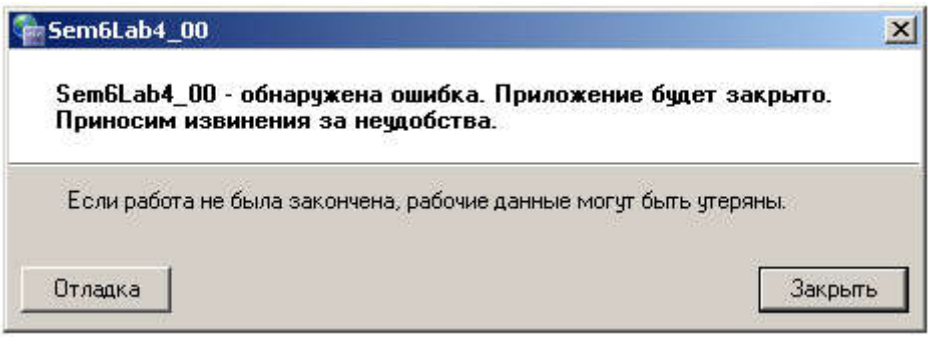


Рисунок 4.1 – Форма з повідомленням про виняток

5. Не перехоплюйте виняток, якщо його не можна обробити. Якщо потрібно, при перехопленні **System.Exception**, можна знову ініціювати цей же виняток, використавши ключового слова **throw** в кінці блоку **catch**.
6. Якщо в блоці **catch** визначена змінна винятку, її можна використовувати для отримання додаткової інформації про тип події винятку.
7. Винятки можуть явно генеруватися програмної за допомогою ключового слова **throw**.
8. Об'єкти винятки містять докладні відомості про помилку, такі як стан стека викликів і текстовий опис помилки.
9. Код в блоці **finally** виконується, навіть при використанні ключового слова return в попередньому коді. Блок **finally** використовується для звільнення ресурсів, наприклад для закриття потоків або файлів, відкритих в блоці **try**.
10.

**Генерація винятків**

Оператор **throw** використовується для повідомлення про випадки аномальних ситуацій (виключень) в ході виконання програми, після ключового слова **throw** потрібно вказати об'єкт-нащадок класу **System.Exception**, що буде описувати поточне виключення.

Наприклад:

```
using System;

class Program {
    private double temperature;

    void SetTemperature(double t) {
        if (t < -50 || t > 50) {
            throw new Exception(
    "Температура повинна бути в діапазоні [-50; 50]"
             );
        }
        temperature = t;
    }

    static void Main(string[] args) {
        Program p = new Program();
        p.SetTemperature(100);
    }
}
```

В документації MSDN не рекомендується викликати System.Exception, System.SystemException, System.NullReferenceException або System.IndexOutOfRangeException з власного програмного коду. Тому наведений вище приклад краще переписати наступним чином:

```
using System;

class Program {
    private double temperature;
    void SetTemperature(double t) {
        if (t < -50 || t > 50) {
            throw new ArgumentException(
"Температура повинна бути в діапазоні [-50; 50]", "t"
             );
        }
        temperature = t;
```

```
    }
    static void Main(string[] args) {
        Program p = new Program();
        p.SetTemperature(100);
    }
}
```

**Повторна генерація виключень**

Виключення, що перехоплено в одному блоці catch, може бути повторно згенеровано в іншому блоці, для того щоб бути перехопленим у зовнішньому блоці catch. Для повторної генерації виключення достатньо вказати оператор throw без згадування об'єкту виключення. Не слід забувати, що коли виключення генерується повторно, то воно знову не перехоплюється одним й тим же блоком catch, а передається у зовнішній блок catch. Напиклад:

```
using System;


class Program {
    void Div(int x, int y) {
        try {
            int result = x / y;
        } catch (DivideByZeroException) {
            Console.WriteLine("Ділення на нуль");
            throw;
        }
    }

    static void Main() {
        Program p = new Program();
        try {
            p.Div(5, 0);
        } catch (DivideByZeroException) {
            Console.WriteLine("Помилка в програмі");
        }
        Console.ReadKey();
    }
}
```

## 4.2. Порядок виконання роботи

## 4.3. Лабораторна робота №7

1. Ознайомитись із теоретичним матеріалом.
2. Знайти та описати властивості та методи класу **`System.Exception`**.
3. Написати приклад програми по обробці різних типів виключень. Програма повинна бути проектом Windows Form і завантажувати початкові данні з файлу. При завантаженні передбачити обробку таких виключень: відсутність файлу, ділення на нуль, індекс поза межами масиву.
4. Навести приклади генерації власних виключень.
5. Налагодити програму на комп'ютері.
6. Забезпечити вивід результатів роботи програми на екран комп'ютера.
7. Підготувати звіт по роботі.

## 4.4. Контрольні запитання

1. Для чого використовуються блоки **try**, **catch** і **finally**?
2. Які з цих блоків є необов'язковими?
3. За допомогою якого ключового слова можна згенерувати виняток?
4. Нащадками якого класу є всі винятки?
5. Що відбудеться з програмою якщо в ній трапиться необроблений виняток?

# 5. ТЕМА. ПОНЯТТЯ ДЕЛЕГАТІВ. СТВОРЕННЯ ТА ВИКОРИСТАННЯ ДЕЛЕГАТІВ. ЛАБОРАТОРНА РОБОТА №8

**Мета роботи** – ознайомитися з поняттям делегатів та навчитися використовувати делегати.

## 5.1. Теоретичні відомості

### Загальні відомості про делегати

**Делегат -** це тип, що являє собою посилання на методи з певним списком параметрів (сигнатура) і типом, що повертається. При створенні екземпляра (примірника) делегата цей примірник можна пов'язати з будь-яким методом з сумісною сигнатурою і типом, що повертається. В результаті метод можна викликати (активувати) за допомогою примірника делегата [1].

Делегати також можна використовувати використовуються для передачі методів в якості аргументів до інших методів.

Всі делегати є об'єктами типу System.Delegate або System.MulticastDelegate, який є похідним від першого [11].

Уже до закінчення розробки середовища виконання, її архітектори прийшли до висновку, що доцільно використовувати тільки клас MulticastDelegate, а клас Delegate є надлишковим. Але, побоюючись серйозних архітектурних нестиковок, розробники були змушені залишити клас Delegate в складі загальної бібліотеки класів.

Різниця між цими класами полягає в тому, що екземпляри першого (делегати) можуть зберігати лише одне посилання на метод, а примірник другої можуть містити відразу кілька посилань на методи. Завдяки цьому, можна приєднувати до одного делегату кілька методів, кожен з яких при єдиному зверненні до делегату буде викликатися по ланцюжку. Таким чином, з програми буде видно лише один делегат, за яким ховається кілька методів.

Ця можливість дуже зручна для підтримки подій, оскільки дозволяє без використання додаткових механізмів приєднати до події декілька функцій обробників. Фактично, делегат являє собою об'єкт - чорний ящик, що приховує

в своїх надрах покажчики на функції. Важливо зрозуміти, що делегати, по суті справи, нічим не відрізняються від звичайних об'єктів користувача. Головна їхня особливість полягає лише в тому, що вони мають підтримку з боку середовища виконання. Про їхні властивості, на відміну від звичайних об'єктів, знають навіть компілятори, що мають зручні спеціальні сервіси для роботи з ними.

**Оголошення класів – делегатів**

Не дивлячись на те, що всі делегати є нащадками класу MulticastDelegate, оголошення делегатів простим наслідуванням від системного класу – неприпустиме. Наступний код:

```
public class NewDelegate : MulticastDelegate {

}
```

викличе помилку при компіляції:

**'NewDelegate' cannot derive from special class 'System.MulticastDelegate'**

Що означає : 'NewDelegate' не може бути спадкоємцем спеціального класу 'System. MulticastDelegate'.

Для оголошення делегатів в C# існує спеціальна конструкція з ключовим словом **delegate**:

**[<модифікатор рівня доступу>] delegate <тип результату> <ім'я класу> (<список аргументів>);**

Приклади:

```
public delegate void NewDelegate(string s);
public delegate int CalcDelegate(int a, int b);
```

Таким чином створюються два різних класи делегатів NewDelegate і CalcDelegate, що мають різні набори аргументів та різні типи результату.

**Створення екземплярів (об'єктів) для делегатів**

Об'єкт делегату зазвичай створюється шляхом вказування імені методу, для якого делегат у подальшому буде слугувати оболонкою, або за допомогою анонімного методу. Після створення екземпляра делегата, виклик методу

виконаний в делегаті, передається делегатом у цей метод. Параметри, що передаються делегату при виконанні, передаються в метод, а значення, що повертається (якщо воно є) повертається делегатом в об’єкт, який його викликав. Ця процедура називається викликом делегату.

Приклад створення об’єкту делегата:

```
using System;
using System.Collections.Generic;
using System.Text;

// клас делегат
public delegate void NewDelegate(string s);

class Program {
    // метод
    void WriteMessage(string str) {
        Console.WriteLine("Message : " + str);
    }

    static void Main(string[] args) {
        Program prog = new Program();

        // створення об'єкту делегата
        // з використанням конструктора
        NewDelegate del0 =
             new NewDelegate(prog.WriteMessage);

        // створення об'єкту делегата
        // без використання конструктора
        NewDelegate del1 = prog.WriteMessage;

        // виклик першого об'єкту делегата
        del0("ОК");
        // виклик другого об'єкту делегата
        del1("ТАК");
        Console.ReadKey();
    }
}
```

Де `NewDelegate` - це клас делегат, del0 та del1 - об’єкти делегати, що створюються, `WriteMessage` - метод, на який посилаються обидва об’єкти. Відповідно, після створення об’єкта делегата можна звертатися до методів, на які

він посилається. Виглядає це так.

```
// виклик першого об'єкту делегата
del0("ОК");
// виклик другого об'єкту делегата
del1("ТАК");
```

Результат роботи програми буде наступний:

```
Message : ОК
Message : ТАК
```

***Зауваження:*** Делегату може бути присвоєний довільний метод, що відповідає класу делегату, з довільного доступного класу або структури. Цей метод може бути, **як екземплярним** (що відноситься до об’єкту), **так і статичним**.

Приклад присвоювання статичного методу:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace MyDelegate01 {

    // клас делегат
    public delegate void NewDelegate(string s);

    class Program {
        static void Main(string[] args) {
            // створення об'єкту делегата
            // для статичного методу
           NewDelegate del0 = Console.WriteLine;

            // виклик об'єкту делегата
            del0("Test");
            Console.ReadKey();
        }
    }
}
```

**Багатоадресні делегати**

Як вже вказувалося раніше, нащадки класу MulticastDelegate можуть містити відразу кілька посилань на методи. Делегати, що містять посилання на декілька методів - називаються **багатоадресними**.

Багатоадресні делегати містять список призначених делегатів, кожен з яких містить посилання на свій метод. При виклику багатоадресний делегат викликає по черзі всі делегати зі списку. Делегати в списку повинні бути одного й того ж самого типу (класу).

Для роботи з багатоадресними делегатами визначено операції додавання та віднімання делегатів (+, -, + =, - =).

Для наочності наведемо приклад, що демонструє роботу з операціями над делегатами.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace MyDelegate02 {
    // клас делегат
    public delegate void NewDelegate(string s);

    class Program {
        // метод
        void WriteMessage(string str) {
            Console.WriteLine("Message : " + str);
        }

        static void Main(string[] args) {
            Program prog = new Program();

            // створення об'єкту делегата
            // для статичного методу
            NewDelegate del0 = prog.WriteMessage;
            NewDelegate del1 = Console.WriteLine;

            // додавання делегатів
            NewDelegate del2 = del0 + del1;
            Console.WriteLine("del2 = del0 + del1");
            // виклик об'єкту делегата
            del2("Test");
            Console.WriteLine("----------------");
            // додамо, ще одного делегата
            Console.WriteLine(
                 "del2 += prog.WriteMessage"
                 );
            del2 += prog.WriteMessage;
```

```
            del2("Test");
            Console.WriteLine("----------------");
            // віднімемо делегата
            Console.WriteLine(
                 "del2 -= Console.WriteLine"
                 );
            del2 -= Console.WriteLine;
            del2("Test");

            Console.ReadKey();
        }
    }
}
```

Після виконання на екрані отримаємо:

```
del2 = del0 + del1
Message : Test
Test
----------------
del2 += prog.WriteMessage
Message : Test
Test
Message : Test
----------------
del2 -= Console.WriteLine
Message : Test
Message : Test
```

**Делегати в якості аргументів**

Екземпляри делегатів - це об’єкти, отже вони можуть передаватися в якості параметрів в інші методи.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace MyDelegate03 {
    // клас делегат
    public delegate void NewDelegate(string s);

    class Program {

        // метод для параметру
        void WriteMessage(string str) {
            Console.WriteLine("Message : " + str);
```

```
        }

        // метод з параметром делегатом
        void FullMessage(
             int num, string text, NewDelegate del
             ) {
            del("№ " + num.ToString() + " " + text);
        }

        static void Main(string[] args) {
            Program prog = new Program();

            // виклик методу та передача параметру
            prog.FullMessage(
                 101, "Global Error", prog.WriteMessage
                 );

            Console.ReadKey();
        }
    }
}
```

Результат роботи цієї програми:

```
Message : № 101 Global Error
```

**Анонімні методи**

Анонімні методи – це методи, що не мають власної назви. Вони в основному використовуються для створення екземплярів делегатів. Визначення анонімних методів починається з ключового слова delegate, після якого слідує необов'язковий список параметрів у круглих дужках і тіло методу в фігурних дужках.

Наприклад:

```
delegate void Del(string s);
Del nd = delegate(string str){Console.WriteLine(str);};
```

## 5.2. Порядок виконання роботи

## 5.3. Лабораторна робота №8

1. Ознайомитись із теоретичним матеріалом.
2. Навести власні приклади використання делегатів з різними сигнатурами (наборами параметрів та значень, що повертаються).
3. Навести приклади застосування операторів додавання та віднімання для делегатів.
4. Навести приклад використання делегатів в якості аргументу.
5. Перевірити роботу анонімних методів.
6. Налагодити програму на комп’ютері.
7. Підготувати звіт по роботі.

## 5.4. Контрольні запитання

1. Що таке делегат в C#?
2. Нащадками яких класів є делегати?
3. Синтаксис оголошення делегатів C#.
4. Які типи методів можна присвоювати делегатам?
5. Що означає термін – багатоадресні делегати?
6. Які операції допустимі для багатоадресних делегатів?
7. Що таке анонімні методи?

# 6. ТЕМА. ПРИНЦИПИ РОБОТИ ПОДІЙ. ОТРИМАННЯ ПОДІЙ. ЛАБОРАТОРНІ РОБОТИ №9 ТА №10

**Мета роботи** – познайомитись з подіями в C#. Навчитися застосовувати події в програмних додатках.

## 6.1. Теоретичні відомості

**Обробники подій в C #**

**Події** це особливий тип багатоадресних делегатів, які можна викликати тільки з класу або структури, в якій вони оголошені (клас видавця). Якщо на подію підписані інші класи або структури, їхні методи обробників подій будуть викликані коли клас видавця ініціює подію.

Події дозволяють класу або об'єкту повідомляти інші класи чи об'єкти про виникнення будь-яких ситуацій. Клас, що відправляє (або викликає) подію, називається **видавцем**, а класи, які отримують (або оброблюють) подія, називаються **передплатниками**.

Події оголошуються за допомогою ключового слова **`event`**. Синтаксис оголошення події – наступний:

```
[модифікатори] event [тип-делегат] [ім'я події];
```

Наприклад:

```
// оголошуємо клас-делегат
public delegate void MyEventHandler();
// оголошуємо подію типу MyEventHandler
public event MyEventHandler MyEvent;
```

**Властивості подій**

Події мають наступні властивості [6]:

1. Видавець визначає момент виклику події, передплатник визначає - яка дія виконується у відповідь.
2. Подія може мати декількох передплатників. Передплатник може оброблювати декілька подій від декількох видавців.
3. Події, що не мають передплатників ніколи не виникають.
4. Події зазвичай використовуються для повідомлення про дії користувачів,

такі як натискання кнопок або вибір пункту меню в графічних інтерфейсах користувача.

5. Якщо у події кілька передплатників, то при її виконанні виклик обробників подій виконується асинхронно (т.т. послідовно).
6. В бібліотеці класів .NET Framework, базовим класом для всіх поді є делегат EventHandler, а для аргументів подій базовий клас - EventArgs.

**Використання подій**

Розглянемо роботу з подіями на прикладі струмової відсічки. Нехай в нас є пристрій релейного захисту **Device**, який спрацьовує кожного разу як значення поточного струму **Current** досягне 10. І є вимикач, який вимикає лінію **Switch**. Відповідно **Device** – це видавець, а **Switch** – буде передплатником.

```
using System;
//
// Клас Пристрій
//
public class Device {
    // поле де записується поточний струм
    private double current = 0;

    // делегат
    public delegate void Funct(double d);
    // подія спрацювання відсічки
    public event Funct CutOff;

    // властивість, що моделює зняття показника струму
    // та спрацювання при стумі, що більше 10
    public double Current {
        get { return current; }
        set {
            current = value;
            if (value > 10 && CutOff != null) {
                CutOff(value);
            }
        }
    }
}
//
// Клас Вимикач
```

```
//
public class Switch {
    // положення вимикача, true - включений
    public bool On = true;
    // вимикання
    public void Switch_Off(double p) {
        if (On) {
            Console.WriteLine(
             ">>>Відключено при струмі {0:f3}<<<", p
             );
            On = false;
        }
    }
}
//
// Основна програма
//
public class Program{
    static void Main() {
        // пристрій
        Device dev = new Device();
        // вимикач
        Switch swt = new Switch();

        // підключаємо спрацювання до події відсічка
        dev.CutOff += swt.Switch_Off;

        // моделюємо зміну значення струму
        // випадковим чином
        Random rnd = new Random();

        for (int i = 0; i < 20; i++) {
            // зміна струму
            double current = rnd.Next(7, 11)
                 + rnd.NextDouble();
            // зняття показника пристроєм
            dev.Current = current;
            Console.WriteLine(
             "Струм = {0,7:f3} Вимикач {1}",
             current, swt.On ? "On" : "Off"
             );
        }
        Console.ReadKey();
    }
}
```

Приклад результату роботи програми:

```
Струм =   8,401 Вимикач On
Струм =   9,095 Вимикач On
Струм =   8,382 Вимикач On
Струм =   9,560 Вимикач On
Струм =   7,275 Вимикач On
Струм =   7,123 Вимикач On
Струм =   8,505 Вимикач On
Струм =   7,456 Вимикач On
Струм =   9,798 Вимикач On
Струм =   9,441 Вимикач On
Струм =   9,289 Вимикач On
>>>Відключено при струмі 10,808<<<
Струм =  10,808 Вимикач Off
Струм =  10,366 Вимикач Off
Струм =   9,490 Вимикач Off
Струм =  10,610 Вимикач Off
Струм =   8,920 Вимикач Off
Струм =   7,459 Вимикач Off
Струм =  10,839 Вимикач Off
Струм =   8,361 Вимикач Off
Струм =   8,406 Вимикач Off
```

## 6.2. Порядок виконання робіт

## 6.3. Лабораторна робота №9. Створення подій.

1. Ознайомитись із теоретичним матеріалом.
2. Навести власний приклад події з предметної області, що відноситься до вашої спеціальності.
3. Скласти відповідну програму на базі класів та подій C#.
4. Налагодити програму на комп’ютері.
5. Підготувати звіт по роботі.

## 6.4. Лабораторна робота №10. Події для класів графічних об’єктів

1. Для класу **Object2D** з Заняття №2 – створити власну подію **MouseClick**, подібну до відповідної події елемента керування форми.
2. Для виклику події створити метод OnMouseClick, що перевірятиме, чи попала миша в середину об’єкта і викликатиме подію **MouseClick**.

3. В формі створити обробник події **MouseClick**, що викликатиме події для вибраних об’єктів класу **Object2D**.
4. Налагодити програму на комп’ютері.
5. Підготувати звіт по роботі.

## 6.5. Контрольні запитання

1. Що таке події в C#?
2. Поясніть значення термінів видавець та передплатник відносно подій?
3. Синтаксис оголошення подій.
4. Які основні властивості подій?

# ЛІТЕРАТУРА

1. Настенко, Д. В. Об’єктно-орієнтоване програмування. Частина 1. Основи об’єктно-орієнтованого програмування на мові C# [Електронний ресурс] : навчальний посібник для бакалаврів напряму підготовки 6.050701 «Електротехніка та електротехнології» програми професійного спрямування «Системи управління виробництвом та розподілом електроенергії» / Д. В. Настенко, А. Б. Нестерко ; НТУУ «КПІ». – Електронні текстові дані (1 файл: 931,2 Кбайт). – Київ : НТУУ «КПІ», 2016. – 76 с. – Назва з екрана. https://ela.kpi.ua/handle/123456789/16671

2. Обчислювальна техніка та програмування. Конспект лекцій. Частина 1 [Електронний ресурс] : навчальний посібник для студентів спеціальності 141 «Електроенергетика, електротехніка та електромеханіка» / КПІ ім. Ігоря Сікорського ; уклад.: Г. О. Труніна, Д. В. Настенко, А. Б. Нестерко. – Електронні текстові данні (1 файл: 3,28 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2020. – 117 с. – Назва з екрана. https://ela.kpi.ua/handle/123456789/39004

3. Обчислювальна техніка та програмування. Лабораторні роботи. Частина 1 [Електронний ресурс] : навчальний посібник для студентів спеціальності 141 «Електроенергетика, електротехніка та електромеханіка» / КПІ ім. Ігоря Сікорського; уклад.: А. Б. Нестерко, Д. В. Настенко, Г. О. Труніна. – Електронні текстові данні (1 файл: 1,99 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2020. – 83 с. – Назва з екрана. https://ela.kpi.ua/handle/123456789/39020

4. Обчислювальна техніка та програмування. Домашня контрольна робота. Частина 1 [Електронний ресурс] : навчальний посібник для студентів спеціальності 141 «Електроенергетика, електротехніка та електромеханіка» / КПІ ім. Ігоря Сікорського ; уклад.: Д. В. Настенко, Г. О. Труніна, А. Б. Нестерко – Електронні текстові данні (1 файл: 1,31 Мбайт). – Київ : КПІ ім. Ігоря Сікорського, 2020. – 17 с. – Назва з екрана. https://ela.kpi.ua/handle/123456789/39019

5. C# docs - get started, tutorials, reference. | Microsoft Docs [Електронний ресурс] URL: https://docs.microsoft.com/en-us/dotnet/csharp/

6. Events (C# Programming Guide) . | Microsoft Docs [Електронний ресурс] URL: https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/events/