



ФУНДАМЕНТАЛЬНІ КОНЦЕПЦІЇ ООП. ІНКАПСУЛЯЦІЯ

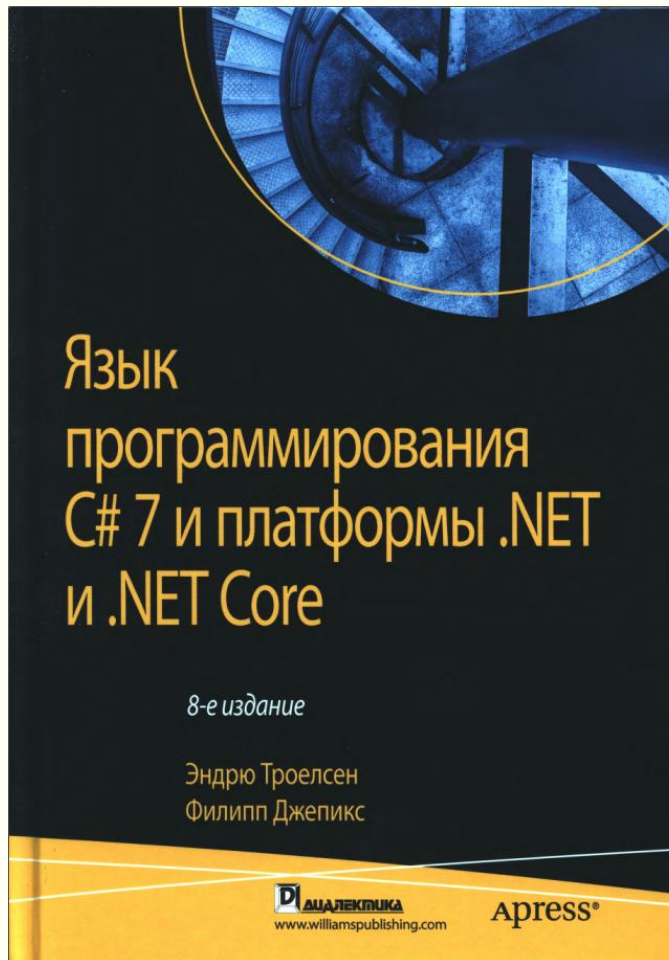
Лекція 03
Об'єктно-орієнтоване програмування



План лекції

- Базова робота з класами.
- Інкапсуляція та приховування даних у мові С#.
- Конструювання об'єктів.

Література до теми





БАЗОВА РОБОТА З КЛАСАМИ

Питання 3.1.

Знайомство з типом класу в C#


Создание проекта


Последние шаблоны проектов


- Консольное приложение (.NET Core) C#
- Консольное приложение (.NET Framework) C#
- Веб-приложение ASP.NET Core C#


Поиск шаблонов (ALT+“B”) [Очистить все](#)

C# Все платформы Все типы проектов

**Консольное приложение (.NET Core)**
Проект для создания приложения командной строки, которое может выполняться в среде .NET Core в Windows, Linux и Mac OS.
C# Linux macOS Windows Консоль

**Веб-приложение ASP.NET Core**
Шаблоны проектов для создания веб-приложений ASP.NET Core и веб-API в Windows, Linux и macOS с использованием .NET Core или .NET Framework. Создавайте веб-приложения с использованием Razor Pages и MVC, а также одностраничные приложения с использованием Angular, React и React + Redux.
C# Linux macOS Windows Облако Служба Веб

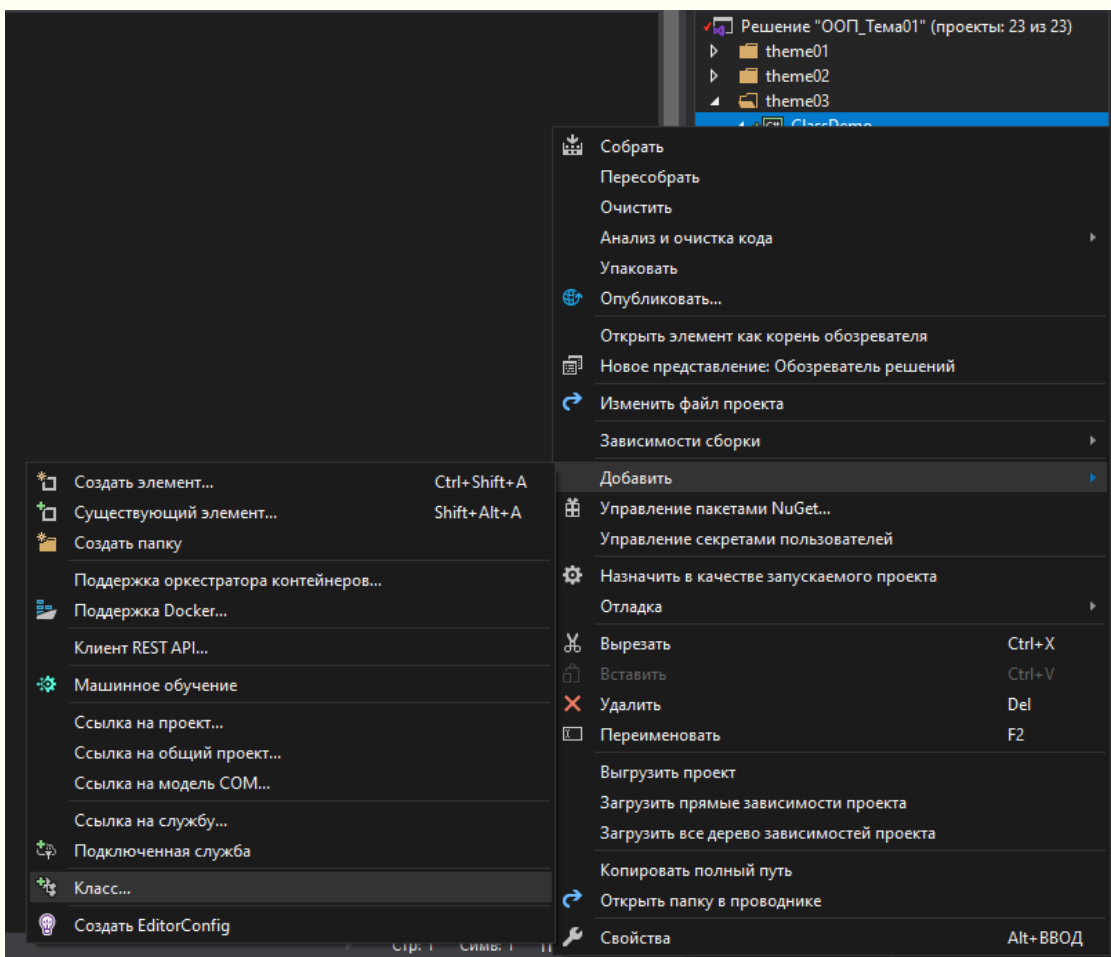
**Приложение Blazor**
Шаблоны проектов для создания приложений Blazor, которые выполняются на сервере в приложении ASP.NET Core или в браузере в WebAssembly (wasm). Эти шаблоны можно использовать для создания веб-приложений с полнофункциональными динамическими пользовательскими интерфейсами (UI).
C# Linux macOS Windows Облако Веб

**Библиотека классов (.NET Standard)**
Проект для создания библиотеки классов, предназначенной для .NET Standard.
C# Android iOS Linux macOS Windows Библиотека

[Назад](#) [Далее](#)

- Тип класу – найбільш фундаментальна програмна конструкція. є тип класу.
 - Формально клас - це визначений користувачем тип, який складається з полів даних (змінних-членів) і членів, що оперують цими даними (конструкторів, властивостей, методів, подій і т.п.).
 - Всі разом поля даних класу представляють *"стан" екземпляра класу* (інакше званого *об'єктом*).
- Потужність об'єктно-орієнтованих мов, подібних C#, полягає в їх здатності групувати дані і пов'язану з ними функціональність у визначенні класу, що дозволяє моделювати програмне забезпечення на основі сутностей реального світу.

Додавання класу в проект



```
1 using System;
2 using System.Collections.Generic;
3 using System.Text;
4
5 namespace ClassDemo
6 {
7     class Car
8     {
9     }
10 }
```

- Нехай об'єкти Car (автомобілі) повинні мати поле даних типу int, що представляє поточну швидкість, і поле даних типу string для представлення назви автомобіля.

Клас Car

```
class Car
{
    // стан об'єкта Car
    public string petName;
    public int currSpeed;

    // функціональність (поведінка) Car
    public void PrintState()
    {
        Console.WriteLine("{0} рухається зі швидкістю {1} км/год.", petName, currSpeed);
    }

    public void SpeedUp(int delta)
    {
        currSpeed += delta;
    }
}
```

- Поля даних класу рідко повинні визначатися з модифікатором `public`.
 - Щоб забезпечити цілісність даних стану, набагато краще оголошувати дані **закритими** (*private*) або **захисними** (*protected*) і дозволяти контрольований доступ до даних.

Метод Main() з управляючого класу

```
using System;
using System.Text;

namespace ClassDemo {
    class Program
    {
        static void Main(string[] args)
        {
            Console.OutputEncoding = Encoding.UTF8;
            // розміщення в пам'яті та конфігурація об'єкту типу Car
            Car myCar = new Car();
            myCar.petName = "Лола";
            myCar.currSpeed = 50;

            // збільшуємо швидкість автомобіля та виводимо новий стан
            for (int i = 0; i <= 10; i++)
            {
                myCar.SpeedUp(2);
                myCar.PrintState();
            }

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

14.09.2020

```
F:\csbc-github\oop-theory-repo\TimeStamp\ClassDemo\bin\
Лола рухається зі швидкістю 52 км/год.
Лола рухається зі швидкістю 54 км/год.
Лола рухається зі швидкістю 56 км/год.
Лола рухається зі швидкістю 58 км/год.
Лола рухається зі швидкістю 60 км/год.
Лола рухається зі швидкістю 62 км/год.
Лола рухається зі швидкістю 64 км/год.
Лола рухається зі швидкістю 66 км/год.
Лола рухається зі швидкістю 68 км/год.
Лола рухається зі швидкістю 70 км/год.
Лола рухається зі швидкістю 72 км/год.
```


Розміщення об'єктів за допомогою ключового слова new

- Щоб коректно створити об'єкт, потрібно визначити та розмістити в його пам'яті:

```
Car myCar = new Car();  
myCar.petName = "Лола";  
myCar.currSpeed = 50;
```

- Зазвичай бажано привласнити осмислені значення полів об'єкту перед тим, як працювати з ним.
 - тип Car вимагає присвоювання значень полів petName і currSpeed.
 - нерідко класи складаються з декількох десятків полів, небажано писати 20 операторів ініціалізації для всіх 20 елементів даних такого класу.
- в C # підтримується механізм **конструкторів**, які дозволяють встановлювати стан об'єкта в момент його створення.
 - Конструктор - це спеціальний метод класу, який викликається неявно при створенні об'єкта з використанням ключового слова new.
 - Конструктор ніколи не має значення, що повертається (навіть void) і завжди іменується ідентично імені класу, об'єкт якого він конструює.

Конструювання об'єкта за допомогою стандартного конструктора

- Кожен клас C# забезпечується стандартним конструктором, який при необхідності може бути перевизначений.
 - За визначенням стандартний конструктор ніколи не сприймає аргументів.
 - Після розміщення нового об'єкта в пам'яті стандартний конструктор гарантує ініціалізацію всіх полів даних відповідними стандартними значеннями.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    // Вызов стандартного конструктора.
    Car chuck = new Car();
    // Выводит "Chuck is going 10 MPH."
    chuck.PrintState();
    ...
}
```

```

class Car {
    public string petName;
    public int currSpeed;

    // спеціальний стандартний конструктор
    public Car() {
        petName = "Лада-седан Баклажан";
        currSpeed = 40;
    }

    // спеціальний конструктор з одним параметром
    // інший параметр отримує стандартне значення int - 0
    public Car(string name) {
        petName = name;
    }

    // спеціальний конструктор з двома параметрами
    // дозволяє встановити повний стан об'єкта типу Car
    public Car(string name, int speed) {
        petName = name;
        currSpeed = speed;
    }

    // функціональність Car
    public void PrintState() {
        Console.WriteLine("{0} рухається зі швидкістю {1} км/год.", petName, currSpeed);
    }

    public void SpeedUp(int delta) {
        currSpeed += delta;
    }
}

```

Визначення спеціальних конструкторів

- клас Car має перевантажений конструктор, щоб надати кілька способів створення об'єкта під час оголошення.

Створення об'єктів за допомогою спеціальних конструкторів

```
class Program
{
    static void Main(string[] args)
    {
        Console.OutputEncoding = Encoding.UTF8;

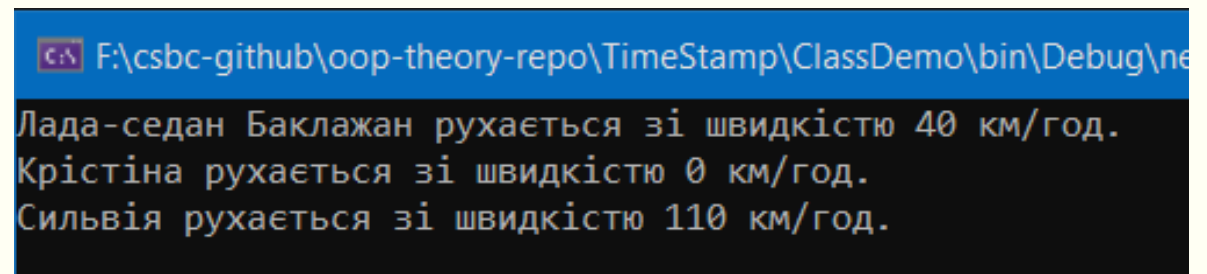
        Car renaultZoe = new Car();
        renaultZoe.PrintState();

        Car plymouthFury = new Car("Крістіна");
        plymouthFury.PrintState();

        Car nissan = new Car("Сильвія", 110);
        nissan.PrintState();

        Console.ReadKey();
    }
}
```

- Поява спеціальних конструкторів викликає видалення стандартного конструктора.
 - щоб дозволити користувачеві об'єкта створювати екземпляр типу за допомогою стандартного конструктора, а також спеціального конструктора, знадобиться явно перевизначити стандартний конструктор.
 - в переважній більшості випадків реалізація стандартного конструктора класу навмисно залишається порожньою, оскільки все, що потрібно - це створення об'єкта зі стандартними значеннями всіх полів.
 - Явно заданий стандартний конструктор:
 - `public Car() { }`



```
F:\csbc-github\oop-theory-repo\TimeStamp\ClassDemo\bin\Debug\net
Ладаседан Баклажан рухається зі швидкістю 40 км/год.
Крістіна рухається зі швидкістю 0 км/год.
Сильвія рухається зі швидкістю 110 км/год.
```

Роль ключового слова `this`

- забезпечує доступ до поточного екземпляру класу.
 - Одне з можливих застосувань - вирішувати неоднозначність контексту, яка може виникнути, коли вхідний параметр названий так само, як поле даних конкретного класу.

```
class Motorcycle
{
    public int driverIntensity;

    // Новые члены для представления имени водителя.
    public string name;
    public void SetDriverName(string name)
    { name = name; }
    ...
}
```

- Хоча цей код нормально компілюється, Visual Studio відобразить попередження про те, що змінна присвоюється сама собі!

```
public void SetDriverName(string name)
{
    this.name = name;
}
```

Необов'язковість this

- якщо неоднозначності немає, то ви не зобов'язані використовувати ключове слово `this`, коли класу потрібно звертатися до власних даних або членам.
 - Крім невеликого виграшу від використання `this` в неоднозначних ситуаціях, це ключове слово може бути корисно при реалізації членів (IntelliSense).

```
class Motorcycle
{
    public int driverIntensity;
    public string driverName;
    public void SetDriverName(string name)
    {
        // Эти два оператора функционально эквивалентны.
        driverName = name;
        this.driverName = name;
    }
    ...
}
```

```

class Motorcycle
{
    public int driverIntensity;
    public string driverName;

    // Связывание конструкторов в цепочку.
    public Motorcycle()
    {
        Console.WriteLine("In default ctor");
    }
    public Motorcycle(int intensity)
        : this(intensity, "")
    {
        Console.WriteLine("In ctor taking an int");
    }
    public Motorcycle(string name)
        : this(0, name)
    {
        Console.WriteLine("In ctor taking a string");
    }

    // Это 'главный конструктор', выполняющий всю реальную работу.
    public Motorcycle(int intensity, string name)
    {
        Console.WriteLine("In master ctor ");
        if (intensity > 10)
        {
            intensity = 10;
        }
        driverIntensity = intensity;
        driverName = name;
    }
    ...
}

```

Побудова ланцюжка викликів конструкторів з використанням this

- Інше застосування ключового слова **this** полягає в проектуванні класу, що використовує прийом під назвою **зчеплення конструкторів** або **побудова ланцюжка конструкторів**.
 - Цей шаблон проектування корисний, коли є клас, який визначає кілька конструкторів.
 - в кожному конструкторі робиться перевірка, що рівень потужності не перевищує 10.
- Зрозуміліший підхід передбачає призначення конструктора, який приймає максимальну кількість аргументів, як "головного конструктора", з реалізацією всередині нього необхідної логіки перевірки достовірності.
 - Решта конструкторів зможуть використовувати ключове слово **this**, щоб передати вхідні аргументи головному конструктору і при необхідності надати будь-які додаткові параметри.

Робота програми

- Метод Main():

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Class Types *****\n");
    // Создание Motorcycle.
    Motorcycle c = new Motorcycle(5);
    c.SetDriverName("Tiny");
    c.PopAWheely();
    Console.WriteLine("Rider name is {0}", c.driverName); // вывод имени гонщика
    Console.ReadLine();
}
```

- Вивід програми:

```
***** Fun with Class Types *****
In master ctor
In ctor taking an int
Yeeeeeeee Haaaaaaeeww!
Yeeeeeeee Haaaaaaeeww!
Yeeeeeeee Haaaaaaeeww!
Yeeeeeeee Haaaaaaeeww!
Yeeeeeeee Haaaaaaeeww!
Yeeeeeeee Haaaaaaeeww!
Yeeeeeeee Haaaaaaeeww!
Rider name is Tiny
```


Ключове слово static

- Клас C # може визначати будь-яку кількість статичних членів, які оголошуються з використанням ключового слова static.
 - При цьому відповідний член повинен викликатися безпосередньо на рівні класу, а не на змінної, що зберігає посилання на об'єкт.
- статичні члени - це елементи, задумані (проектувальником класу) як загальні, так що немає потреби створювати екземпляр класу перед їх викликом.
 - найчастіше їх можна виявити всередині "обслуговуючих класів".
 - обслуговуючий клас - це такий клас, який підтримує стан на рівні об'єктів і не створюється за допомогою ключового слова new.
 - Замість цього обслуговуючий клас відкриває всю функціональність у вигляді членів рівня класу (тобто статичних).
- Метод System.Console.WriteLine() не викликається на рівні об'єкта:

```
// Ошибка! WriteLine() не является методом уровня объекта!  
Console c = new Console();  
c.WriteLine("I can't be printed...");  
  
// Правильно! WriteLine() - статический метод.  
Console.WriteLine("Thanks...");
```

Ключове слово `static`

- статичні члени можуть перебувати не тільки в обслуговуючих класах; вони можуть бути частиною будь-якого визначення класу.
 - статичні члени переміщують заданий елемент на рівень класу, а не об'єкта.
- Ключове слово `static` може бути застосовано до наступних конструкцій:
 - дані класу;
 - методи класу;
 - властивості класу;
 - конструктор;
 - все визначення класу.
- Коли визначаються дані рівня екземпляра, відомо, що при кожному створенні нового об'єкта цей об'єкт підтримує власну незалежну копію таких даних.
 - На противагу цьому, якщо визначені статичні дані класу, ця пам'ять розділяється всіма об'єктами відповідної категорії.

Демонстрація

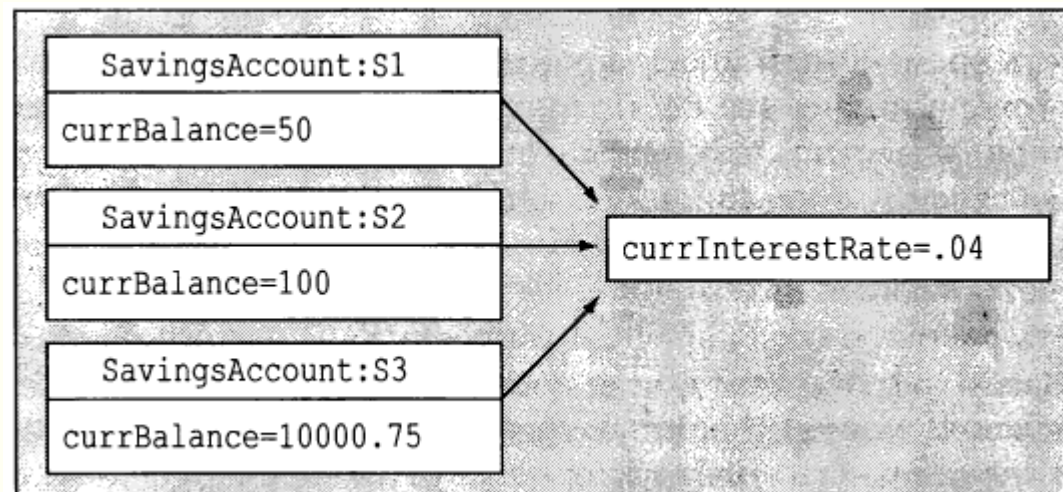
```
using System;

namespace StaticDataAndMembers
{
    class SavingsAccount
    {
        // дані рівня екземпляру
        public double currBalance;
        // статичний елемент даних - депозитний %
        public static double currInterestRate = 0.04;

        public SavingsAccount(double balance)
        {
            currBalance = balance;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            SavingsAccount s1 = new SavingsAccount(50);
            SavingsAccount s2 = new SavingsAccount(100);
            SavingsAccount s3 = new SavingsAccount(10000.75);
            Console.ReadKey();
        }
    }
}
```

- При створенні об'єктів SavingsAccount пам'ять під поле currBalance виділяється для кожного об'єкта.
 - можна було б створити п'ять різних об'єктів SavingsAccount, кожен з власним унікальним балансом.
 - Більш того, якщо ви зміните баланс на якомусь одному рахунку, це не вплине на інші об'єкти.
 - З іншого боку, статичні дані розподіляються одноразово і поділяються всіма об'єктами того ж самого класу.



Доповнення класу статичними методами

```
class SavingsAccount
{
    // дані рівня екземпляру
    public double currBalance;
    // статичний елемент даних - депозитний %
    public static double currInterestRate = 0.04;

    public SavingsAccount(double balance)
    {
        currBalance = balance;
    }

    // статичні члени для встановлення/отримання %ї ставки
    public static void SetInterestRate(double newRate)
    {
        currInterestRate = newRate;
    }

    public static double GetInterestRate()
    {
        return currInterestRate;
    }
}
```

- при створенні нових екземплярів класу SavingsAccount значення статичних даних не скидається, оскільки CLR виділяє для них місце в пам'яті тільки один раз.

Ілюстрація проблем з конструюванням статичних полів

```
class SavingsAccount
{
    // дані рівня екземпляру
    public double currBalance;
    // статичний елемент даних - депозитний %
    public static double currInterestRate;

    public SavingsAccount(double balance)
    {
        currInterestRate = 0.04;
        currBalance = balance;
    }

    // статичні члени для встановлення/отримання %ї ставки
    public static void SetInterestRate(double newRate)
    {
        currInterestRate = newRate;
    }

    public static double GetInterestRate()
    {
        return currInterestRate;
    }
}
```

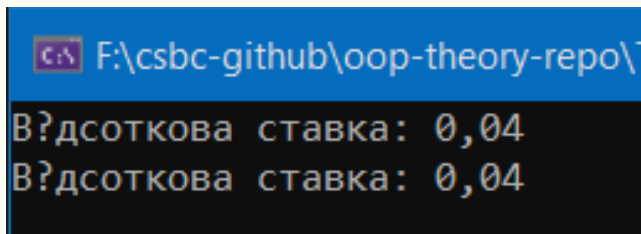
```
class Program
{
    static void Main(string[] args)
    {
        SavingsAccount s1 = new SavingsAccount(50);
        Console.WriteLine("Відсоткова ставка: {0}",
            SavingsAccount.GetInterestRate());

        // спробуємо змінити відсоткову ставку
        SavingsAccount.SetInterestRate(0.08);

        SavingsAccount s2 = new SavingsAccount(100);
        Console.WriteLine("Відсоткова ставка: {0}",
            SavingsAccount.GetInterestRate());

        Console.ReadKey();
    }
}
```

- При виконанні методу Main () виявляється, що змінна currInterestRate буде скидатися при кожному створенні нового об'єкта SavingsAccount, завжди повертаючись до значення 0.04.



```
F:\csbc-github\oop-theory-repo\T
Відсоткова ставка: 0,04
Відсоткова ставка: 0,04
```

Ілюстрація проблем з конструюванням статичних полів

```
class SavingsAccount
{
    // дані рівня екземпляру
    public double currBalance;
    // статичний елемент даних - депозитний %
    public static double currInterestRate = 0.04;

    public SavingsAccount(double balance)
    {
        currBalance = balance;
    }

    // статичні члени для встановлення/отримання %ї ставки
    public static void SetInterestRate(double newRate)
    {
        currInterestRate = newRate;
    }

    public static double GetInterestRate()
    {
        return currInterestRate;
    }
}
```

- Один із способів правильного встановлення статичного поля – використання синтаксису ініціалізації.
 - Однак що, якщо значення статичних даних потрібно отримати під час виконання?
 - Наприклад, в типовому банківському додатку значення змінної, що представляє процентну ставку, має бути прочитано з бази даних або зовнішнього файлу.
- Вирішення подібних завдань вимагає контексту методу (такого як конструктор), щоб можна було виконати оператори коду.
 - Саме тому в C# передбачена можливість визначення статичного конструктора, який дозволяє безпечно встановлювати значення статичних даних.

Статичний конструктор

```
class SavingsAccount
{
    public double currBalance;
    public static double currInterestRate;
    public SavingsAccount(double balance)
    {
        currBalance = balance;
    }
    // Статический конструктор.
    static SavingsAccount()
    {
        Console.WriteLine("In static ctor!");
        currInterestRate = 0.04;
    }
    ...
}
```

- Кілька цікавих моментів стосовно статичних конструкторів:
 - В окремому класі може бути визначено тільки один статичний конструктор. Іншими словами, статичний конструктор можна перевантажувати.
 - Статичний конструктор не має модифікатора доступу і не може приймати параметрів.
 - Статичний конструктор виконується тільки один раз, незалежно від того, скільки об'єктів окремого класу створюється.
 - Виконавча система викликає статичний конструктор, коли створює екземпляр класу або перед першим зверненням до статичного члену цього класу.
 - Статичний конструктор виконується перед будь-яким конструктором рівня екземпляра.

Визначення статичних класів

```
// Статические классы могут содержать только статические члены!  
static class TimeUtilClass  
{  
    public static void PrintTime()  
    { Console.WriteLine(DateTime.Now.ToShortTimeString()); }  
    public static void PrintDate()  
    { Console.WriteLine(DateTime.Today.ToShortDateString()); }  
}
```

```
static void Main(string[] args)  
{  
    Console.WriteLine("***** Fun with Static Classes *****\n");  
    // Это работает нормально.  
    TimeUtilClass.PrintDate();  
    TimeUtilClass.PrintTime();  
    // Ошибка компиляции! Создавать экземпляр статического класса нельзя!  
    TimeUtilClass u = new TimeUtilClass ();  
  
    Console.ReadLine();  
}
```

- Ключевое слово `static` допускається також застосовувати прямо на рівні класу.
 - Коли клас визначений як статичний, його екземпляри можна створювати з використанням ключового слова `new`, і він може включати в себе тільки члени або поля даних, помічені ключовим словом `static`.
 - клас (або структуру), який відкриває тільки статичну функціональність, часто називають обслуговуючим.
 - При проектуванні обслуговуючого класу рекомендується застосовувати ключове слово `static` до визначення класу.
- слід врахувати, що клас, який не містить нічого крім статичних членів і / або константних даних, і не потребує виділення пам'яті.



ДЯКУЮ ЗА УВАГУ!

Наступне питання: Інкапсуляція та приховування даних у мові C#
