



ІНКАПСУЛЯЦІЯ ТА ПРИХОВУВАННЯ ДАНИХ У МОВІ C#

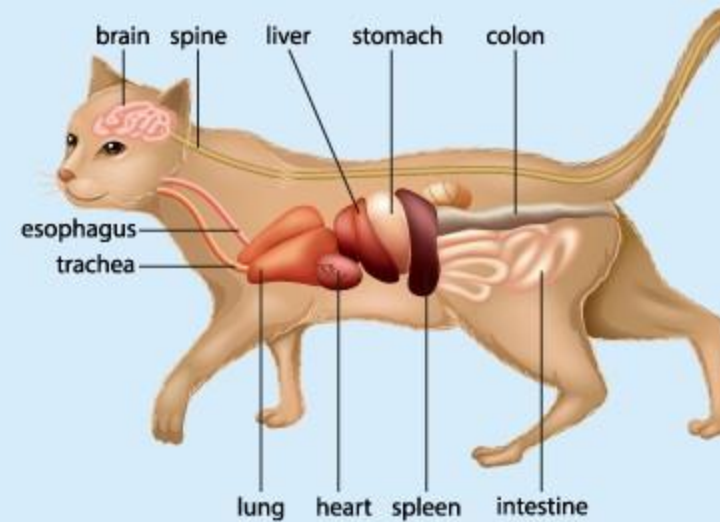
Питання 3.2.

Роль інкапсуляції

Понятно, что делать с объектом
(кормить, что же еще)



Не понятно, как именно
взаимодействовать с объектом
(слишком много деталей)



- Клас інкапсулює дані та операції з цими даними.

Різниця між класом і структурою

КЛАС	СТРУКТУРА
Посилальні типи.	Значимі типи.
Всі посилальні типи розташовуються в кучі.	Всі значимі типи розташовуються в стеку.
Виділення <i>великого обсягу</i> пам'яті для посилального типу дешевше, ніж для значимого типу.	Виділення (Allocation) та вивільнення (de-allocation) для значимого типу загалом дешевші в порівнянні з посилальним типом.
Клас має необмежені можливості.	Структура має обмежені можливості.
Загалом використовуються в крупних програмах.	Використовуються в невеликих програмах.
Можуть містити конструктор / деструктор (фіналізатор).	Структури можуть містити лише параметричний або статичний конструктори.
Використовують ключове слово <code>new</code> при конструюванні об'єктів.	Можуть створювати екземпляр з або без ключового слова <code>new</code> .
Клас може успадковуватись від іншого класу.	Структура не допускає наслідування з іншої структури або класу.
Члени класу можуть бути захищеними.	Члени структури не можуть бути захищеними.
Члени-функції класу можуть бути віртуальними або абстрактними.	Члени-функції структур не можуть бути віртуальними або абстрактними.
Два поля класу можуть містити посилання на один об'єкт, а кожна дія над одним полем впливатиме на інше.	Кожне поле структури містить власну копію даних (за винятком <code>ref</code> та <code>out</code> параметрів), жодна дія над одним полем не впливає на інше.

Роль інкапсуляції



Внутренняя реализация **скрыта**,
шанс что-то сломать – минимальный



Внутренняя реализация **открыта**,
любое вмешательство опасно

- З ідеєю інкапсуляції програмної логіки тісно пов'язана ідея *захисту даних*.
 - Працює на основі *модифікаторів доступу*.

Служби інкапсуляції C#

- Внутрішні дані об'єкта не повинні бути безпосередньо доступні через екземпляр об'єкта.
 - Замість цього дані класу визначаються як закриті.
 - Якщо викликає код бажає змінити стан об'єкта, то повинен робити непрямо через відкриті методи.
- Приклад проблеми з відкритими даними: змоделюємо клас Book з одним відкритим полем типу int (максимальне значення – 2147483647).

```
class Book
{
    public int numberOfPages;
}
```

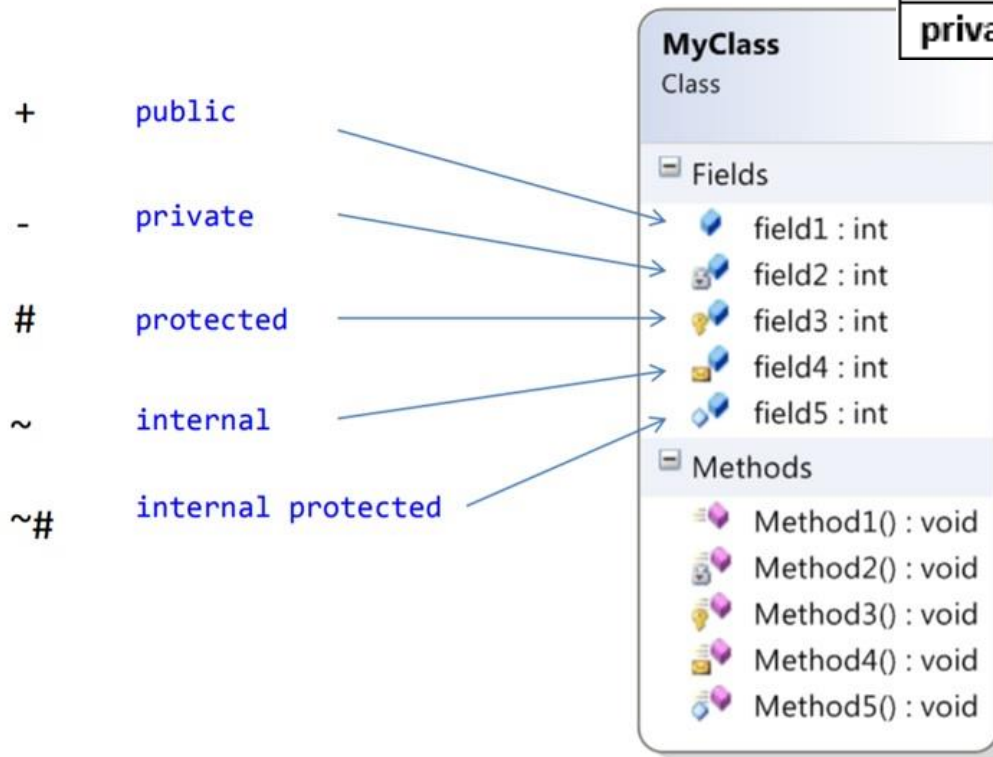
- відкриті поля не дають можливості перехоплювати помилки, пов'язані з подоланням верхніх (або нижніх) логічних меж.
- Якщо в поточній системі встановлено бізнес-правило, з якого випливає, що книга повинна мати від 1 до 1000 сторінок, його доведеться забезпечити програмно.

```
static void Main(string[] args)
{
    Book miniNovel = new Book();
    miniNovel.numberOfPages = 30000000;
}
```


Модифікатори доступу

- Доступні модифікатори:
 - private protected починаючи з C# 7.2

	Same Assembly			Different Assembly	
	Same class	Derived class	Other class	Derived class	Other class
private	✓	✗	✗	✗	✗
protected	✓	✓	✗	✓	✗
internal	✓	✓	✓	✗	✗
protected internal	✓	✓	✓	✓	✗
public	✓	✓	✓	✓	✓
private protected	✓	✓	✗	✗	✗



- За умовчанням члени типів є неявно закритими (private) та неявно внутрішніми (internal).
 - `class Radio { }`
- Щоб дозволити іншим частинам програми звертатись до членів об'єкта, їх слід відмітити як відкриті (public).
 - Також зручно при побудові бібліотек, коли класи чи їх частини відкриваються зовнішнім збіркам.
 - `public class Radio { }`

Роль інкапсуляції

- Інкапсуляція надає спосіб збереження цілісності даних про стан об'єкта.
 - Замість визначення відкритих полів (які легко призводять до пошкодження даних), необхідно виробити звичку визначення закритих даних, управління якими здійснюється опосередковано, із застосуванням одного з двох головних прийомів:
 - визначення пари відкритих методів доступу і зміни;
 - визначення відкритої властивості .NET.
- Ідея полягає в тому, що добре інкапсульований клас повинен захищати свої дані і приховувати подробиці свого устрою від цікавих очей з зовнішнього світу.
 - Це часто називають програмуванням чорного ящика.
 - Перевага такого підходу: об'єкт може вільно змінювати внутрішню реалізацію будь-якого методу.
 - За рахунок забезпечення незмінності сигнатури методу, робота існуючого коду, який використовує цей метод, не порушується.

Інкапсуляція з використанням традиційних методів доступу і зміни

```
class Employee
{
    // Поля даних.
    private string empName;
    private int empID;
    private float currPay;

    // Конструкторы.
    public Employee() {}
    public Employee(string name, int id, float pay)
    {
        empName = name;
        empID = id;
        currPay = pay;
    }

    // Методы.
    public void GiveBonus(float amount)
    {
        currPay += amount;
    }

    public void DisplayStats()
    {
        Console.WriteLine("Name: {0}", empName);
        Console.WriteLine("ID: {0}", empID);
        Console.WriteLine("Pay: {0}", currPay);
    }
}
```

- Буде побудований досить повний клас, що моделює звичайного співробітника.
 - поля класу Employee визначені із застосуванням ключового слова private.
 - поля empName, empID і currPay безпосередньо через об'єктну змінну не доступні.
- Традиційний підхід (дуже поширений в Java) передбачає визначення методів доступу (метод get) і зміни (метод set).
 - Роль методу get полягає в поверненні зухвалому коду значення лежать в основі статичних даних.
 - Метод set дозволяє викликаючому коду змінювати поточне значення статичних даних, що лежать в основі, за умови дотримання бізнес-правил.

```
static void Main(string[] args)
{
    // Ошибка! Невозможно напрямую обращаться
    // к закрытым полям объекта!
    Employee emp = new Employee();
    emp.empName = "Marv";
}
```


Інкапсуляція поля

```
class Employee
{
    // Поля данных.
    private string empName;
    ...

    // Метод доступа (метод get) .
    public string GetName()
    {
        return empName;
    }

    // Метод изменения (метод set) .
    public void SetName(string name)
    {
        // Перед присваиванием проверит входное значение.
        if (name.Length > 15)
            // Ошибка! Имя должно иметь меньше 16 символов!
            Console.WriteLine("Error! Name must be less than 16 characters!");
        else
            empName = name;
    }
}
```

- Закрите поле `empName` інкапсульоване з використанням двох відкритих методів `GetName()` і `SetName()`.
 - Для подальшої інкапсуляції даних в класі `Employee` знадобиться додати ряд додаткових методів (наприклад, `GetID()`, `SetID()`, `GetCurrentPay()`, `SetCurrentPay()`).
 - Кожний метод, що змінює дані, може мати в собі кілька рядків коду для перевірки додаткових бізнес-правил.
 - Хоч можна вчинити саме так, для інкапсуляції даних класу в C# пропонується зручна альтернативна нотація.

```
class Employee
{
    // Поля данных.
    private string empName;
    private int empID;
    private float currPay;
    // Свойства.
    public string Name
    {
        get { return empName; }
        set
        {
            if (value.Length > 15)
                Console.WriteLine("Error! Name must be less than 16 characters!");
            else
                empName = value;
        }
    }
    // Можно было бы добавить дополнительные бизнес-правила для установки
    // этих свойств, но в данном примере в этом нет необходимости.
    public int ID
    {
        get { return empID; }
        set { empID = value; }
    }
    public float Pay
    {
        get { return currPay; }
        set { currPay = value; }
    }
    ...
}
```

Інкапсуляція за допомогою властивостей .NET

- в мовах .NET є кращий спосіб інкапсуляції даних за допомогою властивостей.
 - Властивість C # складається з визначень контекстів get (метод доступу) і set (метод зміни), вкладених безпосередньо в контекст самої властивості.
- Зверніть увагу, що властивість вказує тип даних, який вона інкапсулює, як тип значення.
 - Крім того, на відміну від методу, у визначенні властивості не використовуються дужки (навіть порожні).

set властивості

- В контексті set властивості використовується лексема value, яка представляє вхідний значення, що привласнюється властивості викликає кодом.
 - Ця лексема не є справжнім ключовим словом C #, а являє собою те, що називається контекстним ключовим словом.
 - Коли лексема value знаходиться всередині контексту set, вона завжди позначає значення, що присвоюється викликаючим кодом, і завжди має тип, що співпадає з типом самої властивості.
- Тому властивість Name може перевірити допустиму довжину string:

```
public string Name
{
    get { return empName; }
    set
    {
        // Здесь value имеет тип string.
        if (value.Length > 15)
            Console.WriteLine("Error! Name must be less than 16 characters!");
        else
            empName = value;
    }
}
```

set властивості

- При наявності цих властивостей викликає коду здається, що він має справу з відкритим елементом даних: однак "за лаштунками" при кожному зверненні викликається коректний блок get або set, зберігаючи інкапсуляцію:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Encapsulation *****\n");
    Employee emp = new Employee("Marvin", 456, 30000);
    emp.GiveBonus(1000);
    emp.DisplayStats();

    // Установка и получение свойства Name.
    emp.Name = "Marv";
    Console.WriteLine("Employee is named: {0}", emp.Name);
    Console.ReadLine();
}
```

Додаємо вік робітника

```
class Employee
{
    ...
    // Новое поле и свойство.
    private int empAge;
    public int Age
    {
        get { return empAge; }
        set { empAge = value; }
    }

    // Обновленные конструкторы.
    public Employee() {}
    public Employee(string name, int id, float pay)
        :this(name, 0, id, pay){}
    public Employee(string name, int age, int id, float pay)
    {
        empName = name;
        empID = id;
        empAge = age;
        currPay = pay;
    }

    // Обновленный метод DisplayStats() теперь учитывает возраст
    public void DisplayStats()
    {
        Console.WriteLine("Name: {0}", empName);
        Console.WriteLine("ID: {0}", empID);
        Console.WriteLine("Age: {0}", empAge);
        Console.WriteLine("Pay: {0}", currPay);
    }
}
```

- Властивості (на противагу методам доступу і зміни) також полегшують маніпулювання типами, оскільки здатні реагувати на внутрішні операції C #.
 - припустимо, що тип класу Employee має внутрішню закриту змінну-член, що зберігає вік співробітника.
- Створюємо об'єкт та враховуємо дні народження:
 - припустимо, що створений об'єкт Employee по імені joe.
 - Необхідно, щоб в день народження співробітника вік збільшувався на 1 рік.
- Використовуючи традиційні методи set / get, довелося б написати код на кшталт:
 - Employee joe = new Employee();
joe.SetAge(joe.GetAge() + 1);
- Однак якщо empAge інкапсулюється через властивість Age, можна записати простіше:

```
Employee joe = new Employee();
joe.Age++;
```

Використання властивостей всередині визначення класу

- Властивості, а особливо їх частина set - це загальноприйняте місце для розміщення бізнес-правил класу.
 - У даний час клас Employee має властивість Name, яке гарантує довжину імені не більше 15 символів.
 - Решта властивості (ID, Pay і Age) також можуть бути оновлені з додаванням відповідної логіки.
- У даний час головний конструктор не перевіряє вхідні строкові дані на допустимий діапазон, тому можна було б змінити його таким чином:

```
public Employee(string name, int age, int id, float pay)
{
    // Это может оказаться проблемой...
    if (name.Length > 15)
        Console.WriteLine("Error! Name must be less than 16 characters!");
    else
        empName = name;
    empID = id;
    empAge = age;
    currPay = pay;
}
```

- Властивість Name і основний конструктор виконують одну і ту ж перевірку помилок!
- Щоб спростити код і розмістити всю перевірку помилок в центральному місці, для установки і отримання даних всередині класу розумно завжди використовувати властивості.


```

class Employee
{
    // Поля данных.
    private string empName;
    private int empID;
    private float currPay;
    private int empAge;

    // Конструкторы.
    public Employee() { }
    public Employee(string name, int id, float pay)
        :this(name, 0, id, pay){}
    public Employee(string name, int age, int id, float pay)
    {
        Name = name;
        Age = age;
        ID = id;
        Pay = pay;
    }

    // Методы.
    public void GiveBonus(float amount)
    { Pay += amount; }

    public void DisplayStats()
    {
        Console.WriteLine("Name: {0}", Name);
        Console.WriteLine("ID: {0}", ID);
        Console.WriteLine("Age: {0}", Age);
        Console.WriteLine("Pay: {0}", Pay);
    }

    // Свойства остаются прежними...
    ...
}

```

Оновлення конструктора та класу в цілому

```

public Employee(string name, int age, int id, float pay)
{
    // Уже лучше! Используйте свойства для установки данных класса
    // Это сократит количество дублированных проверок ошибок.
    Name = name;
    Age = age;
    ID = id;
    Pay = pay;
}

```

Властивості лише для зчитування та лише для запису

- При інкапсуляції даних може знадобитися налаштувати властивість, доступну тільки для зчитування.
 - Для цього потрібно просто опустити блок set.
 - Аналогічно, якщо потрібно створити властивість, доступну тільки для запису, слід опустити блок get.

```
public string SocialSecurityNumber
{
    get { return empSSN; }
}
```

```
public Employee(string name, int age, int id, float pay, string ssn)
{
    Name = name;
    Age = age;
    ID = id;
    Pay = pay;
    // Теперь это невозможно, поскольку свойство
    // предназначено только для чтения!
    SocialSecurityNumber = ssn;
}
```

- Якщо не планується робити властивість доступною як для зчитування, так і для запису, єдиний вибір передбачає використання всередині логіки конструктора лежить в основі змінної-члена empSSN:

```
public Employee(string name, int age, int id, float pay, string ssn)
{
    ...
    empSSN = ssn;
}
```

Автоматичні властивості

- У деяких випадках потрібно реалізувати тільки прості отримання або присвоєння значення і ніякої іншої логіки.

- У підсумку виходить великий обсяг коду

```
// Тип Car, использующий стандартный синтаксис свойств.  
class Car  
{  
    private string carName = "";  
    public string PetName  
    {  
        get { return carName; }  
        set { carName = value; }  
    }  
}
```

- Щоб спростити процес інкапсуляції даних полів, можна застосовувати синтаксис автоматичних властивостей.

- цей засіб перекладає роботу по визначенню закритого підтримуючого поля і пов'язаної властивості C# на компілятор, використовуючи вдосконалення синтаксису:

```
class Car  
{  
    // Автоматические свойства!  
    public string PetName { get; set; }  
    public int Speed { get; set; }  
    public string Color { get; set; }  
}
```

Автоматичні властивості

- Під час компіляції тип буде оснащений автоматично згенерованим полем і відповідною реалізацією логіки get / set.
 - Назва автоматично згенерованого закритого підтримуючого поля в кодовій базі C # не є видимою.
 - Єдиний спосіб поглянути на нього – скористатися інструментом [ildasm.exe](#).
- Створювати автоматичні властивості лише для зчитування чи лише для запису не можна

```
// Свойство только для чтения? Ошибка!  
public int MyReadOnlyProp { get; }  
  
// Свойство только для записи? Ошибка!  
public int MyWriteOnlyProp { set; }
```

Автоматичні властивості

```
class Car
{
    // Автоматические свойства!
    public string PetName { get; set; }
    public int Speed { get; set; }
    public string Color { get; set; }

    public void DisplayStats()
    {
        Console.WriteLine("Car Name: {0}", PetName);
        Console.WriteLine("Speed: {0}", Speed);
        Console.WriteLine("Color: {0}", Color);
    }
}
```

- При використанні об'єкта, визначеного з автоматичними властивостями, можна присвоювати і отримувати значення, використовуючи очікуваний синтаксис властивостей:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Automatic Properties *****\n");

    Car c = new Car();
    c.PetName = "Frank";
    c.Speed = 55;
    c.Color = "Red";

    Console.WriteLine("Your car is named {0}? That's odd...",
        c.PetName);

    c.DisplayStats();
    Console.ReadLine();
}
```

Автоматичні властивості

- будьте обережні, якщо синтаксис автоматичної властивості застосовується для упаковки змінної іншого класу, тому що приховане поле посилального типу також буде встановлено в стандартне значення, тобто null.
 - Якщо безпосередньо звернутися до MyAuto, то під час виконання згенерується виняток посилання на null,
 - Змінній-члену типу Car, яка лежить в основі, не було присвоєно новий об'єкт.

```
class Garage
{
    // Скрытое поддерживающее поле int установлено в 0!
    public int NumberOfCars { get; set; }
    // Скрытое поддерживающее поле Car установлено в null!
    public Car MyAuto { get; set; }
}
```

```
static void Main(string[] args)
{
    ...
    Garage g = new Garage();
    // Нормально, выводится стандартное значение, равное 0.
    Console.WriteLine("Number of Cars: {0}", g.NumberOfCars);
    // Ошибка времени выполнения! Поддерживающее поле в данный момент равно null!
    Console.WriteLine(g.MyAuto.PetName);
    Console.ReadLine();
}
```


Потрібний спеціальний конструктор (-и)

```
class Garage
{
    // Скрытое поддерживающее поле установлено в 0!
    public int NumberOfCars { get; set; }

    // Скрытое поддерживающее поле установлено в null!
    public Car MyAuto { get; set; }

    // Для переопределения стандартных значений, присвоенных скрытым
    // поддерживающим полям, должны использоваться конструкторы.
    public Garage()
    {
        MyAuto = new Car();
        NumberOfCars = 1;
    }
    public Garage(Car car, int number)
    {
        MyAuto = car;
        NumberOfCars = number;
    }
}
```

- Враховуючи, що закриті підтримуючі поля створюються під час компіляції, в синтаксисі ініціалізації полів C# не можна безпосередньо розміщувати екземпляр посиляльного типу за допомогою new.
 - Це повинно робитися конструкторами класу, що забезпечить створення об'єкта безпечним чином.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Automatic Properties *****\n");

    // Создать автомобиль.
    Car c = new Car();
    c.PetName = "Frank";
    c.Speed = 55;
    c.Color = "Red";
    c.DisplayStats();

    // Поместить автомобиль в гараж.
    Garage g = new Garage();
    g.MyAuto = c;

    // Вывод количества автомобилей в гараже.
    Console.WriteLine("Number of Cars in garage: {0}", g.NumberOfCars);

    // Вывод названия автомобиля.
    Console.WriteLine("Your car is named: {0}", g.MyAuto.PetName);

    Console.ReadLine();
}
```

Синтаксис ініціалізатора об'єктів

- Щоб полегшити процес створення і запуску об'єкта, в С# пропонується синтаксис *ініціалізатора об'єкта*.
 - За допомогою цього механізму можна створити нову об'єктну змінну і присвоїти значення безлічі властивостей і / або відкритих полів в декількох рядках коду.
 - Він більш читабельний та зручний у багатопоточних додатках:

```
public class Student
{
    public int StudentID { get; set; }
    public string StudentName { get; set; }
    public int Age { get; set; }
    public string Address { get; set; }
}
```

```
class Program
{
    static void Main(string[] args)
    {
        Student std = new Student() { StudentID = 1,
                                     StudentName = "Bill",
                                     Age = 20, Address = "New York"
                                   };
    }
}
```

```
IList<Student> studentList = new List<Student>() {
    new Student() { StudentID = 1, StudentName = "John" } ,
    new Student() { StudentID = 2, StudentName = "Steve" } ,
    new Student() { StudentID = 3, StudentName = "Bill" },
    new Student() { StudentID = 3, StudentName = "Bill" },
    new Student() { StudentID = 4, StudentName = "Ram" } ,
    new Student() { StudentID = 5, StudentName = "Ron" }
};
```

Виклик спеціальних конструкторів за допомогою синтаксису ініціалізації

```
public enum PointColor
{ LightBlue, BloodRed, Gold }

class Point
{
    public int X { get; set; }
    public int Y { get; set; }
    public PointColor Color { get; set; }
    public Point(int xVal, int yVal)
    {
        X = xVal;
        Y = yVal;
        Color = PointColor.Gold;
    }
    public Point(PointColor ptColor)
    {
        Color = ptColor;
    }
    public Point()
        : this(PointColor.BloodRed) { }
    public void DisplayStats()
    {
        Console.WriteLine("[{0}, {1}]", X, Y);
        Console.WriteLine("Point is {0}", Color);
    }
}
```

```
Point goldPoint = new Point(PointColor.Gold) { X = 90, Y = 20 };
goldPoint.DisplayStats();
```

- якщо тип Point надає конструктор, що дозволяє викликаючому коду встановити колір (через спеціальне перелічення PointColor), комбінація спеціальних конструкторів та синтаксису ініціалізації об'єкта стає зрозумілою.
 - За допомогою цього нового конструктора можна створити золоту точку на позиції (90, 20)

Синтаксис ініціалізації об'єкта vs традиційний підхід

- Перевага синтаксису ініціалізації об'єктів в тому, що він в основному скорочує обсяг коду (припускаючи відсутність відповідного конструктора).

```
// Создать и инициализировать Rectangle.  
Rectangle myRect = new Rectangle  
{  
    TopLeft = new Point { X = 10, Y = 10 },  
    BottomRight = new Point { X = 200, Y = 200 }  
};
```

```
// Традиционный подход.  
Rectangle r = new Rectangle();  
Point p1 = new Point();  
p1.X = 10;  
p1.Y = 10;  
r.TopLeft = p1;  
Point p2 = new Point();  
p2.X = 200;  
p2.Y = 200;  
r.BottomRight = p2;
```

Робота з даними константних полів

```
namespace ConstData
{
    class MyMathClass
    {
        public const double PI = 3.14;
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Fun with Const *****\n");
            Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
            // Ошибка! Нельзя изменять константу!
            MyMathClass.PI = 3.1444;

            Console.ReadLine();
        }
    }
}
```

```
class MyMathClass
{
    // Попытка установить PI в конструкторе?
    public const double PI;
    public MyMathClass()
    {
        // Ошибка!
        PI = 3.14;
    }
}
```

- У C# є ключове слово `const` для визначення константних даних, які ніколи не можуть змінюватися після початкового встановлення.
 - це корисно при визначенні наборів відомих значень, логічно прив'язаних до конкретного класу або структури, для використання в додатках.
 - звернення до константних даних, визначених у класі `MyMathClass`, здійснюється з використанням префікса у вигляді назви класу (тобто `MyMathClass.PI`).

Початкове значення константи завжди має бути вказано в момент її визначення.

- Причина цього обмеження в тому, що значення константних даних має бути відомо під час компіляції.
- Конструктори ж викликаються під час виконання.

Поля, що допускають тільки зчитування

- Близько до поняття константних даних лежить поняття полів, доступних тільки для зчитування (не плутати з властивостями тільки для зчитування).
 - Подібно констант, поля тільки для читання не можуть бути змінені після початкового присвоювання.
 - Однак, на відміну від констант, значення, що присвоюється такому полю, може бути визначено під час виконання, і тому може бути присвоєно в контексті конструктора, але ніде більше.
 - Це може бути дуже корисно в ситуаціях, коли значення поля невідомо аж до моменту виконання, але потрібно гарантувати, що воно не буде змінюватися після початкового присвоювання.
 - Наприклад, за умови, що для отримання значення необхідно прочитати зовнішній файл

```
class MyMathClass
{
    // Поля только для чтения могут присваиваться
    // в конструкторах, но нигде более.
    public readonly double PI;
    public MyMathClass ()
    {
        PI = 3.14;
    }
}
```

```
class MyMathClass
{
    public readonly double PI;
    public MyMathClass ()
    {
        PI = 3.14;
    }

    // Ошибка!
    public void ChangePI ()
    { PI = 3.14444; }
}
```


Статичні поля, що допускають тільки зчитування

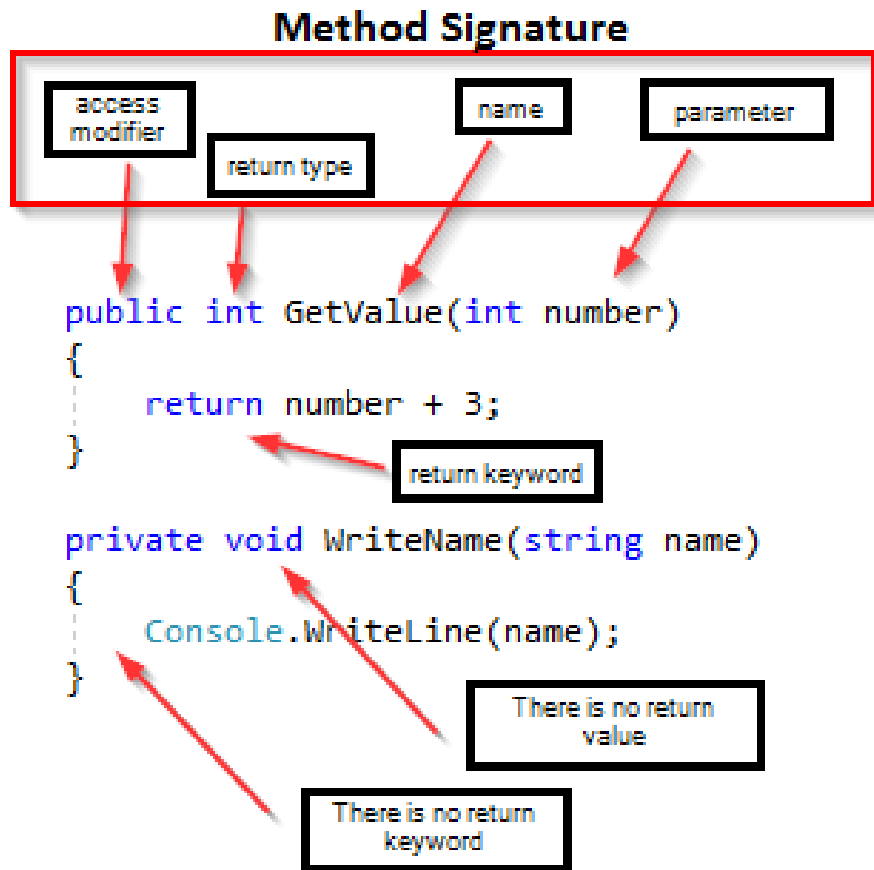
- На відміну від константних полів, поля, що допускають тільки зчитування, не є неявно статичними.
 - якщо необхідно представити PI на рівні класу, то для цього знадобиться явно застосувати ключове слово `static`.
 - Проте якщо значення статичного поля, доступного тільки для зчитування, невідоме до моменту виконання, можна вдатися до використання статичного конструктора,

```
class MyMathClass
{
    public static readonly double PI = 3.14;
}

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Const *****");
        Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
        Console.ReadLine();
    }
}
```

```
class MyMathClass
{
    public static readonly double PI;
    static MyMathClass()
    { PI = 3.14; }
}
```

Опис поведінки об'єкта за допомогою методів



- Метод складається з сигнатури (заголовку) та тіла методу.
- Сигнатура може включати:
 - Модифікатор доступу (access modifier)
 - Тип вихідного значення (return type)
 - Назву
 - Перелік параметрів у дужках
- За умовчанням, передача змінної посилального типу в метод здійснює передавання копії посилання, а не реальних даних.
 - Передача за посиланням дозволяє змінювати і зберігати змінені значення параметрів членів функцій, методів, властивостей, індексаторів, операторів і конструкторів у викликаючому середовищі.

Передача аргументів у методи



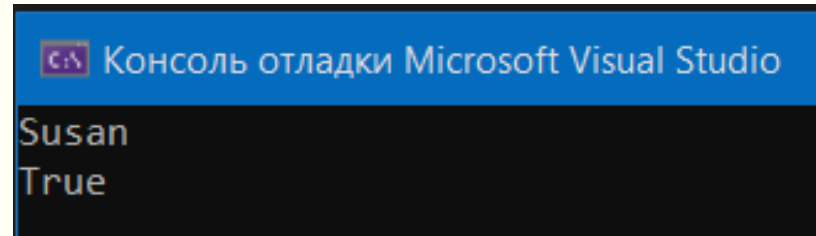
- У C # аргументи можуть передаватися параметрам або **за значенням**, або **за посиланням**.
 - Щоб передати параметр за посиланням, маючи намір змінити значення, використовуйте ключове слово **ref** або **out**.
 - Щоб передати за посиланням, маючи намір запобігти копіювання, але не зміні значення, використовуйте модифікатор **in**.
- Модифікатори параметрів методу:
 - **ref** вказує на те, що переданий параметр **може змінюватись** у методі;
 - **in** задає, що переданий параметр **не може бути зміненим** методом.
 - **out** вказує, що переданий параметр **повинен змінюватись** методом.
- **ref** та **in** вимагають попередньої ініціалізації параметру до передачі в метод.
 - Модифікатор **out** не вимагає цього та зазвичай не ініціалізується до моменту використання в методі.

```
using System;
```

```
namespace ParameterPassing
```

```
{  
    public class Student  
    {  
        public string Name { get; set; }  
        public bool Enrolled { get; set; }  
    }  
  
    class Program  
    {  
        static void Enroll(Student student)  
        {  
            student.Enrolled = true; // змінює змінну student, передану ззовні методу.  
            student = new Student(); // не змінює передану ззовні змінну student, а створює нове посилання.  
                                     // оскільки student тепер має нове посилання, зміни далі в методі  
                                     // більше не впливають на зовнішню змінну.  
            student.Enrolled = false; // змінює локальну версію student в методі  
        }  
  
        static void Main(string[] args)  
        {  
            var student = new Student  
            {  
                Name = "Susan",  
                Enrolled = false  
            };  
  
            Enroll(student);  
            Console.WriteLine(student.Name);  
            Console.WriteLine(student.Enrolled);  
        }  
    }  
}
```

Передача посилального типу



Консоль отладки Microsoft Visual Studio

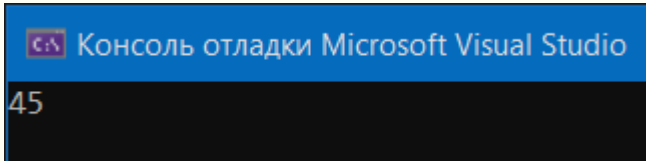
Susan
True

Ключове слово ref

- Використовується в 4 різних контекстах:
 - У сигнатурі методу та виклику методу для передачі аргументів за посиланням.
 - У сигнатурі методу для повернення значення для викликаючого методу за посиланням.
 - У тілі члена класу, щоб вказати, що посилальне вихідне значення зберігається локально як посилання, яке викликаючий об'єкт має намір змінювати (загальніше – локальна змінна отримує доступ до іншого значення за посиланням).
 - В оголошенні структури, щоб оголосити ref struct або readonly ref struct.

```
static void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}
```

```
static void Main(string[] args)
{
    int number = 1;
    Method(ref number);
    Console.WriteLine(number);
}
```



- При передачі ref-параметра в методі всі дії з ним всередині методу виконуються напряму над переданим аргументом.
 - Не плутайте передачу параметру за посиланням з передачею посилального типу: модифікатор ref застосовується і для значимих, і для посилальних типів даних. Упаковки (boxing) значимого типу не відбувається.
 - Зверніть увагу на необхідну ініціалізацію змінної перед її передачею в якості аргументу з модифікатором ref.
 - out-параметри цього не потребують.

Модифікатори в сигнатурах членів класу

- Члени класу не можуть мати сигнатури, які відрізняються лише на ref, in чи out.
 - Проте методи можна перевантажувати, коли в одного з них ref-, in- або out-параметр, а інший має аналогічний значимий параметр

```
class CS0663_Example
{
    // Compiler error CS0663: "Cannot define overloaded
    // methods that differ only on ref and out".
    public void SampleMethod(int i) { }
    // public void SampleMethod(out int i) { }
    public void SampleMethod(ref int i) { }
}
```

- Властивості не є змінними, вони – методи, тому не можуть мати модифікатор ref.
- Не можна використовувати ключові слова ref, in та out для методів такого виду:
 - асинхронних методів (позначених модифікатором async).
 - методів-ітераторів (включають інструкції yield return або yield break).

Повернення даних за посиланням

- Це значення, які метод повертає викликаючому об'єкту (caller) за посиланням.
 - Таким чином, викликаючий об'єкт може змінювати значення, яке буде повертатись методом, що впливатиме на стан самого об'єкта.
 - Щоб викликаючий об'єкт міг змінити стан об'єкту через метод, він повинен явно оголошуватись як *посилальна локальна змінна (ref local)*
- Вихідне посилальне значення (Reference return values, ref returns) визначається за допомогою ключового слова ref:
 - У сигнатурі методу: `public ref decimal GetCurrentPrice()`
 - У return-інструкції: `return ref DecimalArray[0];`

■ Приклад:

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

Посилальні локальні змінні

- Посилальна локальна змінна не може ініціалізуватись не-ref вихідним значенням.
 - `ref decimal estValue = ref Building.GetEstimatedValue();`
- Отримати доступ за посиланням можна таким же чином:
 - `ref VeryLargeStruct reflocal = ref veryLargeStruct;`
 - Приклад показує визначення посилальної локальної змінної, яка використовується для посилання на значення
- В обох прикладах ключове слово `ref` повинно використовуватись в обох місцях. Інакше компілятор згенерує помилку CS8172: "Cannot initialize a by-reference variable with a value."

Оператор ref-присвоєння та посилальні локальні змінні в foreach (починаючи з C# 7.3)

```
static void Display(double[] s)
{
    Console.WriteLine(string.Join(" ", s));
}

static void Main(string[] args)
{
    int number = 1;
    Method(ref number);
    Console.WriteLine(number);

    double[] arr = { 0.0, 0.0, 0.0 };
    Display(arr);

    ref double arrayElement = ref arr[0];
    arrayElement = 3.0;
    Display(arr);

    arrayElement = ref arr[arr.Length - 1];
    arrayElement = 5.0;
    Display(arr);
}
```

```
■ public static void Main()
{
    Span<int> storage = stackalloc int[10];
    int num = 0;
    foreach (ref int item in storage)
    {
        item = num++;
    }

    foreach (ref readonly var item in storage)
    {
        Console.Write($"{item} ");
    }
    // Вивід:
    // 0 1 2 3 4 5 6 7 8 9
}
```

Загальний приклад

```
public class Book {  
    public string Author;  
    public string Title;  
}
```

```
public class BookCollection {  
    private Book[] books = { new Book { Title = "Call of the Wild, The", Author = "Jack London" },  
                              new Book { Title = "Tale of Two Cities, A", Author = "Charles Dickens" }  
    };  
    private Book nobook = null;  
  
    public ref Book GetBookByTitle(string title)  
    {  
        for (int ctr = 0; ctr < books.Length; ctr++)  
        {  
            if (title == books[ctr].Title)  
                return ref books[ctr];  
        }  
        return ref nobook;  
    }  
  
    public void ListBooks()  
    {  
        foreach (var book in books) {  
            Console.WriteLine($"{book.Title}, by {book.Author}");  
        }  
        Console.WriteLine();  
    }  
}
```

```
static void Main(string[] args)  
{  
    var bc = new BookCollection();  
    bc.ListBooks();  
  
    ref var book = ref bc.GetBookByTitle(  
        "Call of the Wild, The");  
  
    if (book != null)  
        book = new Book {  
            Title = "Republic, The",  
            Author = "Plato"  
        };  
    bc.ListBooks();  
}
```

Консоль отладки Microsoft Visual Studio

```
Call of the Wild, The, by Jack London  
Tale of Two Cities, A, by Charles Dickens  
  
Republic, The, by Plato  
Tale of Two Cities, A, by Charles Dickens
```

Порівняння ref і out

REF KEYWORD	OUT KEYWORD
Необхідно ініціалізувати параметри перед передачею з ref.	Не обов'язково ініціалізувати параметри перед передачею з out.
Не обов'язково ініціалізувати значення параметру до повернення в calling-метод.	Необхідно ініціалізувати значення параметру до повернення в calling-метод.
Передача значення через ref-параметр корисна, коли викликаний метод також потребує зміни значення цього переданого параметру.	Оголошення параметру за допомогою out-параметру корисне, коли метод повертає кілька значень.
Коли використовується ключове слово ref, дані можна передавати в обох напрямках.	Коли використовується ключове слово out, дані передаються тільки в одному напрямку.

Модифікатор in

- Крім вихідних параметрів з модифікатором out, метод може використовувати вхідні параметри з модифікатором in.
 - Модифікатор in вказує, що даний параметр буде передаватися в метод за посиланням, проте всередині методу його значення параметра не можна буде змінити.

```
static void GetData(in int x, int y, out int area, out int perim)
{
    // x = x + 10; не можна змінити значення параметра x
    y = y + 10;
    area = x * y;
    perim = (x + y) * 2;
}
```



ДЯКУЮ ЗА УВАГУ!

Наступне питання: Конструювання об'єктів.