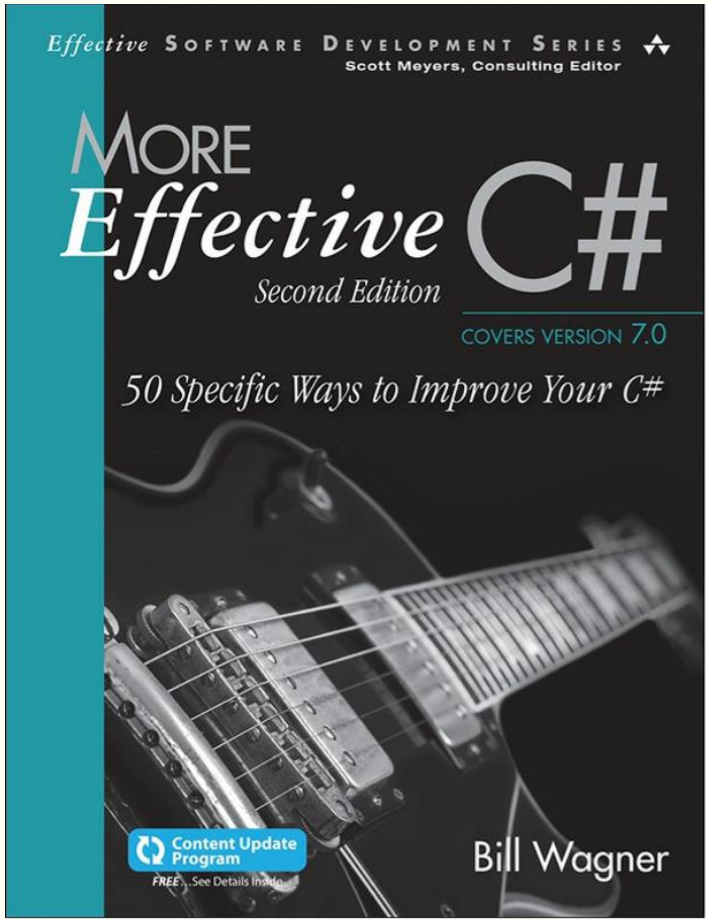
ПРАКТИКИ СТВОРЕННЯ КЛАСІВ ТА ОБ'ЄКТІВ

Питання 2.3.

Література до питання



1. Використовуйте властивості замість полів

- Властивості завжди були частиною мови C#.
  - На геттер і сеттер можна накладати різні обмеження доступу.
  - Автовластивості зменшують ручний набір коду, включаючи readonly-властивості.
  - Властивості дають можливість створювати інтерфейс взаємодії, як з даними, проте мати всі переваги методів.
- .NET за умовчанням вважає, що Ви користуєтесь властивостями, а не відкритими полями, зокрема в бібліотеках прив'язування даних (data binding libraries у WPF, Windows Forms, Web Forms тощо).
  - Механізм прив'язування даних використовує рефлексію для знаходження іменованої властивості в типі:
  - textBoxCity.DataBindings.Add("Text", address, nameof(City));
  - Код прив'язує властивість Text графічного елементу управління textBoxCity до властивості City.

1. Використовуйте властивості замість полів

- Оскільки властивості реалізуються з методами, спрощується додавання підтримки багатопоточного виконання.
  - Можна покращити реалізацію методів доступу get і set, забезпечуючи синхронізований доступ

```
public class Customer
{
        private object syncHandle = new object();
        private string name;
        public string Name {
            get { lock (syncHandle) return name; }
            set {
                if (string.IsNullOrEmpty(value))
                    throw new ArgumentException(
                        "Name cannot be blank", nameof(Name));
                lock (syncHandle) name = value;
            }
        }
        ...
}
```

1. Використовуйте властивості замість полів

- Методи доступу до властивості (аксесори) – два окремих методи, які компілюються з вашим тип.
  - Можна задати різні модифікатори доступу для get- і set-аксесорів у властивості C#.
  - Це дає ще більший контроль над видимістю тих елементів даних, які розкриваються за допомогою властивостей:
-

```
public class Customer
{
    public virtual string Name { get; protected set; }
    // решту реалізації пропущено
}
```

1. Використовуйте властивості замість полів

- Якщо тип має містити індексовані елементи як частину свого інтерфейсу, можна використовувати індексатори (параметризовані властивості).
  - Вони корисні при створенні властивості, яка повертає елементи в послідовності:

```
public int this[int index] {
    get => theValues[index];
    set => theValues[index] = value;
}
private int[] theValues = new int[100]; // доступ до індексатора: int val = someObject[i];
```

- Індексатори підтримуються мовою так, як і окремі значення: до них можна застосовувати алгоритми верифікації чи виконувати внутрішні обчислення.
  - Індексатори можуть бути віртуальними чи абстрактними, оголошуватись в інтерфейсах, бути доступними тільки для зчитування або для читання/запису.
  - Індексатори можуть використовувати нецілочисельні параметри для визначення відображень:

```
public Address this[string name] {
       get => addressValues[name];
       set => addressValues[name] = value;
}
private Dictionary<string, Address> addressValues;
```

1. Використовуйте властивості замість полів

- При роботі з багатовимірними масивами можна створити багатовимірні індексатори з однаковими або різними типами на осях:

```
public int this[int x, int y] => ComputeValue(x, y);
public int this[int x, string name] => ComputeValue(x, name);
```

- Зауважте, що всі індексатори оголошуються з ключовим словом this.
  - Іменувати індексатор у C# не можна, тому кожний індексатор повинен мати різні списки параметрів для уникнення неоднозначності.
  - Майже всі можливості властивостей застосовуються до індексаторів, проте неможливо створити неявні індексатори, як можна зробити з властивостями.
- Хоч властивості та члени-дані є сумісними джерелами даних, вони бінарно несумісні.
  - Очевидно, що при заміні публічного поля еквівалентною публічною властивістю, необхідно перекомпілювати весь код, що використовує публічне поле.
  - Одна з цілей мови – можливість релізу однієї оновленої збірки без перекомпіляції всього додатку.
  - Для згаданої заміни порушується бінарна сумістність, що ускладнює розгортання оновленого додатку в цілому.

1. Використовуйте властивості замість полів

- Може виникнути питання відносної продуктивності властивостей та полів.
  - Доступ до властивостей не буде швидшим, ніж до полів, проте може бути не повільнішим.
  - JIT-компілятор вбудовує виклики методів, зокрема й аксесорів властивості. При цьому продуктивність доступу стане однаковою.
  - Навіть якщо аксесор властивості не було вбудовано, реальною різницею продуктивності можна знехтувати. Ціна одного виклику функції вимірювана в дуже небагатьох ситуаціях.
- Властивості – це методи, які можна переглядати з викликаючого коду, як і дані.
  - Геттер не повинен вносити видимі побічні ефекти.
  - Сеттери змінюють стан, що користувач повинен мати змогу побачити.
- Аксесори властивості також мають користувацькі очікування своєї роботи.
  - Доступ до властивості аналогічний доступу до поля з можливими незначними відмінностями продуктивності.
  - Аксесори мають не виконувати тривалих обчислень чи здійснювати кросдодаткові виклики (на зразок запитів до БД) або інші тривалі операції, несумісні з користувацькими уявленнями щодо аксесору властивості.
- Застосовуйте властивості за потреби розкриття окремих даних та індексатори для послідовностей чи словників.
  - Усі члени-дані ***повинні*** бути приватними, що надаватиме підтримку прив'язування даних та спрощуватиме зміни реалізації методів в майбутньому.

2. Віддавайте перевагу авто-властивостям для змінюваних даних

- При додавання доступних даних у клас аксесори властивостей часто будуть простими обгортками (wrappers) над полями даних.
  - У такому випадку підвищуємо читабельність коду за допомогою авто-властивостей:
  - public string Name { get; set; }
- Компілятор створює backing field, використовуючи згенероване компілятором ім’я.
  - Можна навіть застосовувати сеттер властивості для зміни значення backing field.
  - Оскільки назва backing field генерується компілятором, навіть всередині даного класу необхідно викликати аксесор властивості, а не змінювати backing field напряму.
  - Це не проблема, оскільки виклик аксесора виконує ту ж роботу, а згенерований аксесор є простим оператором присвоєння та, найімовірніше, буде вбудований (inlined).
  - Runtime-поведінка авто-властивості така ж, як і runtime-поведінка при доступі до backing field, навіть з точки зору продуктивності.

```
public string Name {    get; protected set; } // або
public string Name {    get;    internal set; } // або
public string Name {    get; protected internal set; } // або
public string Name {    get;    private set; } // або
// задане ЛИШЕ в конструкторі:
public string Name { get; }

public class BaseType {
    public virtual string Name { get; protected set; }
}

public class DerivedType : BaseType
{
    public override string Name
    {
        get => base.Name;
        protected set {
            if (!string.IsNullOrEmpty(value))
                base.Name = value;
        }
    }
}
```

- Автоматичні властивості підтримують ті ж модифікатори, що і відповідні їм явні властивості.
  - Основна перевага – краща читабельність класів.
- При створенні віртуальної авто-властивості породжений клас не має доступу до згенерованого компілятором backing store.
  - Проте заміщення (override) дозволяє отримати доступ до реалізацій методів get і set властивості з батьківського класу.
- Маємо додаткові переваги від застосування авто-властивостей.
  - Коли неявна реалізація заміняється на конкретну у зв'язку з валідацією даних чи іншими діями, ці зміни будуть бінарно сумісними(binary-compatible) для класу.

```
// original version
public class Person {
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public override string ToString() => $"{FirstName} {LastName}";
}
// Later updated for validation
public class Person {
    public Person(string firstName, string lastName)    {
        // leverage validation in property setters:
        this.FirstName = firstName;
        this.LastName = lastName;
    }
    private string firstName;
    public string FirstName    {
        get => firstName;
        set {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException( "First name cannot be null or empty");
            firstName = value;
        }
    }

    private string lastName;
    public string LastName {
        get => lastName;
        private set {
            if (string.IsNullOrEmpty(value))
                throw new ArgumentException("Last name cannot be null or empty");
            lastName = value;
        }
    }

    public override string ToString() => $"{FirstName} {LastName}";
}
```

- Раніше більшість розробників напряму отримували доступ до backing field для змін всередині класу.
  - Така практика «розмазувала» валідацію та перевірку помилок по файлу.
  - Кожна зміна backing-поля авто-властивості викликає (можливо, закритий) аксесор властивості.
  - Аксесор автовластивості перетворюється в аксесор звичайної властивості, в який записується вся логіка валідації в одному місці.
- Важливе обмеження автовластивостей: неможливо застосовувати на типах, декорованих трибутом Serializable.
  - Персистентне файлове сховище залежить від імені згенерованого компілятором поля в основі.
  - Немає гарантії незмінності назви цього поля.
- Незважаючи на ці обмеження, автовластивості прискорюють розробку, роблять код більш читабельним.

3. Віддавайте перевагу незмінюваності для значимих типів

- Незмінювані типи прості: після створення вони константні.
  - При валідації параметрів конструювання об’єкта він весь час існування буде в коректному стані.
  - Змінити внутрішній стан об’єкта на некоректний неможливо, що позбавляє потреби в перевірці помилок після конструювання.
- Незмінювані типи є потокобезпечними: багато читачів можуть отримати доступ до одного контенту.
  - Якщо внутрішній стан неможливо змінити, відпадає можливість неузгодженості даних при роботі потоків.
- Незмінювані типи можуть безпечно експортуватись з об’єктів.
  - Викликаюча сторона (caller) не може змінювати внутрішній стан ваших об’єктів.
- Незмінювані типи краще працюють у хешованих колекціях.
  - Значення, що повертає Object.GetHashCode(), повинно не залежати від об’єкта (instance invariant); це завжди так для незмінюваних типів.

3. Віддавайте перевагу незмінюваності для значимих типів

- На практиці дуже складно зробити всі типи незмінюваними (immutable).
- Розбивайте (Decompose) свої типи на структури, що мають природну форму єдиної сутності.
  - Наприклад, тип Address не формує таку сутність: адреса компонується з кількох пов'язаних полів.
  - Зміна одного поля, ймовірно, зачепить і інші поля.
  - Тип «customer» не є атомарним: він мститиме багато частин інформації – адресу, ПІБ, телефони та ін.
  - Кожна з незалежних частин може змінитись: клієнт переїде, змінить номер, прізвище тощо.
  - Подібний тип повинен будуватись за допомогою композиції незмінюваних типів: адреси, ПІБ, колекції телефонних номерів тощо.
  - Атомарні типи описують єдині сутності: природною є заміна всього вмісту атомарного типу.
  - Винятком може бути зміна одного з полів всередині такого типу.

3. Віддавайте перевагу незмінюваності для значимих типів

```
// Mutable Address structure.
public struct Address {
    private string state;
    private int zipCode;
    // Rely on the default system-generated constructor.
    public string Line1 { get; set; }
    public string Line2 { get; set; }
    public string City { get; set; }
    public string State { get => state; set {
                ValidateState(value); state = value; } }
    public int ZipCode { get => zipCode; set {
               ValidateZip(value); zipCode = value; } }
    // other details omitted.
}
// Example usage:
Address a1 = new Address();
a1.Line1 = "111 S. Main";
a1.City = "Anytown";
a1.State = "IL";
a1.ZipCode = 61111;
// Modify:
a1.City = "Ann Arbor"; // Zip, State invalid now.
a1.ZipCode = 48103; // State still invalid now.
a1.State = "MI"; // Now fine.
```

- Типова реалізація адреси – змінювана.
  - Зміна внутрішнього стану означає можливе порушення інваріантів об’єкта (принаймні тимчасово).
  - Після створення поля City об’єкт a1 перейшов у некоректний (invalid) стан.
  - Змінене місто більше не відповідає штату та індексу.
  - Код виглядає досить нешкідливим, проте уявіть його запуск у багатопоточній програмі.
  - Будь-яке перемикання контексту після зміни міста, проте до зміни штату буде нести потенційну загрозу неузгодженого доступу до даних для іншого потоку.
- Навіть в однопоточному коді можуть виникнути проблеми.
  - Уявіть, що індекс (ZIP code) був некоректним, що призвело до винятку.
  - Для виправлення доведеться додати суттєвий валідаційний блок коду до структури адреси.
  - Потокобезпечність вимагатиме значні перевірки синхронізованості потоків для кожного аксесора властивості.
  - Замість нагромадження перевірочного коду (за потреби, щоб Address була структурою) зробіть її незмінюваною.
  - Розпочніть із заміни всіх полів екземпляру на read-only для зовнішніх користувачів.

3. Віддавайте перевагу незмінюваності для значимих типів

```
public struct Address
{    // remaining details elided
    public string Line1 { get; }
    public string Line2 { get; }
    public string City { get; }
    public string State { get; }
    public int ZipCode { get; }
    public Address(string line1, string line2,
                   string city, string state,
                   int zipCode):this()
    {
        Line1 = line1;
        Line2 = line2;
        City = city;
        ValidateState(state);
        State = state;
        ValidateZip(zipCode);
        ZipCode = zipCode;
    }
}
```

- Тепер маємо незмінюваний тип з публічним інтерфейсом.
  - Щоб зробити його корисним, потрібно додати всі потрібні конструктори для повної ініціалізації структури Address.
- Використання незмінюваного типу вимагає дещо іншу послідовність викликів для зміни стану.
  - Створюється новий об'єкт, а не змінюється вже існуючий:

```
// Create an address:
Address a2 = new Address("111 S. Main", "", "Anytown", "IL", 61111);
// To change, re-initialize:
a2 = new Address(a1.Line1, a1.Line, "Ann Arbor", "MI", 48103);
```

- Значення a1 знаходиться в одному з 2 станів: початкове розташування в Anytown або оновлене – в Ann Arbor.
  - Ви не змінюєте існуючу адресу для утворення тимчасово некоректного стану, як у попередньому прикладі.

3. Віддавайте перевагу незмінюваності для значимих типів

```
// Almost immutable: there are holes that would
// allow state changes.
public struct PhoneList {
    private readonly Phone[] phones;
    public PhoneList(Phone[] ph) { phones = ph; }
    public IEnumerable<Phone> Phones { get {
        return phones; }
    }
}

Phone[] phones = new Phone[10];
// initialize phones
PhoneList pl = new PhoneList(phones);
// Modify the phone list:
// also modifies the internals of the (supposedly)
// immutable object.
phones[5] = Phone.GeneratePhoneNumber();
```

- Подібні тимчасові стани існують лише протягом виконання конструктора Address і не видимі ззовні конструктора.
  - Як тільки Address-об’єкт сконструйовано, його значення зафіксоване надалі.
  - Він захищений від винятків з попереднього прикладу: якщо виняток буде викидатись під час конструювання нового Address-об’єкту, оригінальне значення a1 залишиться незмінним.
  - Для створення незмінюваного типу необхідно забезпечити відсутність дірок, які дозволятимуть клієнтам змінювати внутрішній стан об’єкта.
- Значимі типи не підтримують derived типи, тому немає потреби захищатись від того, що породжені типи будуть змінювати поля базової структури.
  - Проте потрібно стежити за будь-яким полем незмінюваного типу, яке має змінюваний посилальний тип.
  - При реалізації власних конструкторів необхідно створювати захисну копію даних змінюваного типу.
  - Дані приклади передбачають, що Phone – незмінюваний значимий тип, оскільки ми потурбувались лише щодо незмінюваності значимих типів.

3. Віддавайте перевагу незмінюваності для значимих типів

- Масив – посилальний тип: тут використовується всередині структури PhoneList.
  - Розробники можуть змінити вашу незмінювану структуру через іншу змінну, яка посилається на те ж місце в пам'яті.
  - Для усунення такої можливості потрібно створювати захисну копію масиву (змінюваного типу).
  - Ще одна альтернатива – використання класу ImmutableArray з простору імен System.Collections.Immutable:

```
public struct PhoneList {
    private readonly ImmutableList<Phone> phones;
    public PhoneList(Phone[] ph) { phones = ph.ToImmutableList(); }
    public IEnumerable<Phone> Phones => phones;
}
```

3. Віддавайте перевагу незмінюваності для значимих типів

- Складність типу диктує 3 стратегії ініціалізації вашого незмінюваного типу:
  - (1) Структура Address визначила 1 конструктор, щоб ініціалізувати адресу. Визначення достатнього набору конструкторів часто є найпростішим підходом.
  - (2) Можна створити фабричні методи (factory methods) для ініціалізації структури. Фабрики спрощують створення спільних значень: у .NET Framework таку стратегію використовує тип Color для ініціалізації системних кольорів. Статичні методи Color.FromKnownColor() і Color.FromName() повертають копію значення кольору, який представляє поточне значення із заданої колірної системи.
  - (3) Можна створити змінюваний клас-компаньйон для тих об'єктів, у яких багатокрокові операції необхідні для повного конструювання незмінюваного типу.
    - Приклад: клас StringBuilder для створення рядків, використовуючи кілька операцій.
    - Після всіх дій з побудови рядка, з типу StringBuilder виділяється (retrieve) незмінюваний рядок (string).
- Незмінювані типи простіше програмувати та підтримувати.
  - Не створюйте наосліп get- та set-методи для кожної властивості вашого типу.
  - Першим вибором для типів, які зберігають дані, мають бути незмінювані, атомарні значимі типи.
  - З цих сутностей можна запросто сконструювати складніші структури.

4. Перевіряйте, що 0 – коректне значення для значимих типів

```
public enum Planet
{   // Explicitly assign values.
    // Default starts at 0 otherwise.
    Mercury = 1,
    Venus = 2,
    Earth = 3,
    Mars = 4,
    Jupiter = 5,
    Saturn = 6,
    Uranus = 7,
    Neptune = 8
}

// First edition included Pluto.
Planet sphere = new Planet();
var anotherSphere = default(Planet);
```

- За умовчанням система ініціалізації .NET задає всім об'єктам нульові значення.
  - There is no way for you to prevent other programmers from creating an instance of a value type that is initialized to all 0s.
- Створюйте значення за умовчанням для свого типу.
  - Окремий випадок – перелічення.
  - Ніколи не створюйте enum, який не містить 0 у якості коректного вибору.
  - Усі перелічення породжені від System.ValueType.
  - Перелічувані значення починаються з 0, проте таку поведінку можна змінити.
- Як sphere, так і anotherSphere = 0, що не є коректним значенням.
  - Будь-який код, який покладається на факт початку перелічуваних значень з 0, не буде працювати коректно.
  - При створенні власних значень для enum забезпечте 0 як одне з них.
  - Якщо використовуєте бітові маски в переліченні, визначте 0 як відсутність усіх інших характеристик.

4. Перевіряйте, що 0 – коректне значення для значимих типів

- У даному випадку доводиться явно ініціалізувати значення:
  - **`Planet sphere2 = Planet.Mars;`**
  - Це ускладнює побудову інших значимих типів, які містять даний тип:

```
public struct ObservationData
{
    private Planet whichPlanet; //what am I looking at?
    private double magnitude; // perceived brightness.
}
```

  - Користувачі, які створюють новий об'єкт ObservationData, будуть створювати некоректне поле Planet:
  - **`ObservationData d = new ObservationData();`**
  - Потрібно зробити 0 валідним станом.

4. Перевіряйте, що 0 – коректне значення для значимих типів

```
public enum Planet
{
    None = 0,
    Mercury = 1,
    Venus = 2,
    Earth = 3,
    Mars = 4,
    Jupiter = 5,
    Saturn = 6,
    Uranus = 7,
    Neptune = 8
}

public struct ObservationData
{
    Planet whichPlanet; //what am I looking at?
    double magnitude; // perceived brightness.
    ObservationData(Planet target, double mag) {
        whichPlanet = target;
        magnitude = mag;
    }
}
```

- За можливості обирайте краще значення за умовчанням, ніж 0.
  - Перелічення Planet не має очевидного значення за умовчанням: обирати довільну планету за умовчанням немає сенсу.
  - У такому випадку використовуйте 0 для неініціалізованого значення, яке може бути оновленим пізніше:
- **`Planet sphere = new Planet();`**
  - Тепер sphere має значення None, що вплине і на структуру ObservationData.
  - Новостворені об'єкти ObservationData мають магнітуду 0 і target = None.
  - Проте пам'ятайте, що конструктор за умовчанням все ще видимий, і частина користувачів зможе створити ініціалізований системою варіант.
  - Це все ще невеликий збій, оскільки нічого не спостерігати беззмістовно.
- Можна вирішити проблему, зробивши з ObservationData клас, проте при створенні enum-а ви не можете змусити інших розробників дотримуватись цих правил.

4. Перевіряйте, що 0 – коректне значення для значимих типів

- Також розглянемо кілька специфічних правил при використанні перелічень як прапорців.
  - enum-и, які використовують атрибут Flags, мають завжди задавати значення None = 0:
  - ```
    [Flags]
    public enum Styles {    None = 0,    Flat = 1,    Sunken = 2,    Raised = 4, }
    ```
- Багато розробників застосовують прапорцеві перелічення з побітовим I.
  - 0 завжди спричиняє суттєві проблеми з бітовими прапорцями.
  - Наступний тест ніколи не працюватиме, якщо Flat має значення 0:
  - ```
    Styles flag = Styles.Sunken;
    if ((flag & Styles.Flat) != 0) // Never true if Flat == 0.
        DoFlatThings();
    ```
  - Якщо використовуєте атрибут Flags, забезпечте, щоб 0 був коректним значенням, яке означає «відсутність усіх прапорців».

4. Перевіряйте, що 0 – коректне значення для значимих типів

- Інша поширена проблема ініціалізації включає значимі типи, які містять посилання. Поширений приклад – рядки:

```
public struct LogMessage {
    private int ErrLevel;
    private string msg;
}
LogMessage MyMessage = new LogMessage();
```

- Об'єкт MyMessage містить null-посилання у своєму полі msg.
  - Способу задати іншу ініціалізацію немає, проте можна локалізувати проблему за допомогою властивостей:

```
public struct LogMessage {
    private int ErrLevel;
    private string msg;
    public string Message {
        get => msg ?? string.Empty;
        set => msg = value;
    }
}
```

5. Забезпечте поведінку властивостей, подібну до поведінки даних

- Розробники, які використовують ваші типи, припускають, що властивості ведуть себе в тій же манері, що й члени-дані.
  - Наприклад, розробники-клієнти вважають, що послідовні виклики геттера без інших інструкцій, що втручаються в значення властивості, надаватиме однакову відповідь:
  - ```
    int someValue = someObject.ImportantProperty;
    Debug.Assert(someValue == someObject.ImportantProperty);
    ```
  - Багатопоточна робота може порушити очікування, незалежно від того, використовуєте ви властивості чи поля.
  - Проте в інших випадках повторювані виклики однієї властивості мають повертати те ж значення.
- Також розробники-користувачі вашого типу не очікують, що аксесори властивості виконують багато роботи.
  - Геттер ніколи не повинен виконувати обчислювально затратні операції.
  - Аналогічно, сеттери, найімовірніше, виконують певну валідацію, проте жодних затратних викликів.
  - Розробники розглядають ваші властивості як дані, тому робота з ними повинна бути швидкою.
    ```

5. Забезпечте поведінку властивостей, подібну до поведінки даних

- Часто доступ до властивостей відбувається в циклі при роботі з колекціями:
- for (int index = 0; index < myArray.Length; index++)
  - Чим довший масив, тим більше доступів до властивості Length.
  - Якщо щоразу перераховувати всі елементи масиву, ніхто не буде використовувати цикли – програма значно сповільниться.
- Відповідати очікуванням розробників-клієнтів не дуже складно:
  - (1) використовуйте автовластивості. Їх характеристики близькі до прямого доступу до даних.
  - Проте наявність поведінки у властивостей не завжди проблема: валідація в сеттерах відповідає очікуванням розробників.
  - Геттери також часто виконують деякі обчислення перед поверненням значення. Наприклад, обчислення відстані швидке:
  - :

```
public class Point {
    public int X { get; set; }
    public int Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);
}
```

```
public class Point
{
    private int xValue;
    public int X
    {
        get => xValue;
        set
        {
            xValue = value;
            distance = default(double?);
        }
    }

    private int yValue;
    public int Y {
        get => yValue;
        set
        {
            yValue = value;
            distance = default(double?);
        }
    }
    private double? distance;
    public double Distance {
        get {
        if (!distance.HasValue)
            distance = Math.Sqrt(X * X + Y * Y);
        return distance.Value;
        }
    }
}
```

5. Забезпечте поведінку властивостей, подібну до поведінки даних

- Проте якщо Distance виявляється вузьким місцем, можна кешувати відстань після першого її обчислення.
  - Звісно, потрібно виключати (invalidate) кешоване значення при зміні принаймні одного зі значень (або зробити Point незмінюваним типом).

5. Забезпечте поведінку властивостей, подібну до поведінки даних

- Якшо обчислення вихідного значення геттера властивості набагато затратніше, краще переосмислити публічний інтерфейс класу.

```
// Поганий дизайн властивості. Тривала операція в геттері
public class MyType {
    // lots elided
    public string ObjectName => RetrieveNameFromRemoteDatabase();
}
```

- Користувачі не очікують при доступі до властивості циклічні звернення до віддаленого сховища чи викидання винятку.
  - Ваша реалізація типу повинна залежати від шаблону застосування цього типу.
  - Можливим виходом є кешування:

```
public class MyType {
    // lots elided
    private string objectName;
    public string ObjectName => (objectName != null) ? objectName : RetrieveNameFromRemoteDatabase();
}
```

5. Забезпечте поведінку властивостей, подібну до поведінки даних

- Даний підхід реалізовано в класі Lazy<T>. Попередній код заміняється на:

```
private Lazy<string> lazyObjectName;
public MyType() {
    lazyObjectName = new Lazy<string>(() => RetrieveNameFromRemoteDatabase());
}
public string ObjectName => lazyObjectName.Value;
```

- Добре працює, коли властивість ObjectName потрібна тільки час від часу.
  - Результати отримання значення зберігаються тоді, коли вони не потрібні, проте перше звернення до цієї властивості отримує додаткове навантаження.
  - Якщо цей тип майже завжди використовує властивість ObjectName та коректно кешує її назву, можна завантажувати значення в конструктор та застосовувати кешоване значення як вихідне значення властивості.
  - Представлений код також передбачає, що ObjectName можна безпечно кешувати.
  - Якщо інші частини програми чи інші процеси в системі змінюють віддалене сховище з назвою об’єкта, такий дизайн порушується.

5. Забезпечте поведінку властивостей, подібну до поведінки даних

- Операції отримування (pulling) даних з віддаленої БД та подальше зберігання змін назад у віддалену БД достатньо поширене.
  - Користувацькі очікування – виконання цих операцій у методах, назви яких відповідають цим діям.
  - Таким очікуванням відповідає дана версія коду:

```
// краще рішення: використовувати методи
// для управління кешованими значеннями
public class MyType
{
    public void LoadFromDatabase()
    {
        ObjectName = RetrieveNameFromRemoteDatabase();
        // other fields elided.
    }
    public void SaveToDatabase()    {
        SaveNameToRemoteDatabase(ObjectName);
        // other fields elided.
    }
    // lots elided
    public string ObjectName { get; set; }
}
```

5. Забезпечте поведінку властивостей, подібну до поведінки даних

- Не лише геттери можуть порушити очікування користувачів.
  - Нехай ObjectName – це read-write властивість. Якщо сеттер записав значення у віддалену БД, він порушує очікування користувачів:

```
public class MyType
{
    // lots elided
    private string objectName;
    public string ObjectName {
        get {
            if (objectName == null)
                objectName = RetrieveNameFromRemoteDatabase();
            return objectName;
        }
        set {
            objectName = value;
            SaveNameToRemoteDatabase(objectName);
        }
    }
}
```

5. Забезпечте поведінку властивостей, подібну до поведінки даних

- Така додаткова робота в сеттері порушує кілька користувацьких припущень:
  - Розробники клієнтського коду не очікують, щоб сеттер здійснював звернення до віддаленої БД.
    - Це займе більше часу, ніж очікується. Також існує шанс відмови багатьма неочікуваними способами.
    - Налагоджувачі можуть автоматично викликати геттери ваших властивостей, щоб відобразити їх значення.
    - Якщо геттери кидають винятки, забирають багато часу або змінюють внутрішній стан, це ускладнить ваші сеанси налагодження.
    - Властивості встановлюють інші очікування для розробників-клієнтів, ніж методи.
  - Розробники клієнтського коду очікують, щоб властивості виконувались швидко та забезпечували представлення стану об’єкта.
    - Вони очікують, що властивості будуть схожими на поля даних як за поведінкою, так і за характеристиками продуктивності.
    - Коли ви створюєте властивості, що порушують ці припущення, вам слід змінити загальнодоступний інтерфейс, щоб створити методи, які представляють операції, що не відповідають очікуванням користувачів щодо властивостей.
    - Ця практика дозволяє повернути призначення властивостей – забезпечувати представлення стану об’єкта.

ДЯКУЮ ЗА УВАГУ!

Наступна тема: Фундаментальні концепції ООП. Взаємодія класів