.NET

CONTENTS

C# PROGRAMMING: INTERMEDIATE

(Integrative-Generative AI Edition)

Student Price Book Center

คำนำ

ในยุคดิจิทัลที่เทคโนโลยีและซอฟต์แวร์มีบทบาทสำคัญในทุกภาคส่วนของชีวิต การพัฒนาทักษะการเขียนโปรแกรมจึงกลายเป็นสิ่งจำเป็นสำหรับนักพัฒนาซอฟต์แวร์ทุกคน โดยเฉพาะภาษา C# ที่ถือเป็นหนึ่งในภาษายอดนิยมสำหรับการพัฒนาแอปพลิเคชันหลากหลายรูปแบบ ตั้งแต่แอปพลิเคชันบนเดสก์ท็อป เว็บเซอร์วิส ไปจนถึงเกมและระบบคลาวด์ ภาษา C# มีความสามารถครบถ้วนและรองรับเทคนิคการเขียนโปรแกรมทั้งแบบเชิงวัตถุและแบบฟังก์ชัน ทำให้เหมาะกับผู้ที่ต้องการพัฒนาซอฟต์แวร์คุณภาพสูงและยั่งยืน

หนังสือเล่มนี้มีจุดประสงค์เพื่อเป็นคู่มือสำหรับผู้ที่มีพื้นฐานภาษา C# อยู่แล้วและต้องการก้าวเข้าสู่ระดับกลางอย่างมั่นคง ผ่านการเรียนรู้แนวคิดสำคัญและเทคนิคเชิงลึกในหลายหัวข้อที่จำเป็น เช่น การเขียนโปรแกรมเชิงวัตถุขั้นกลาง (Intermediate OOP) ที่ครอบคลุมการสืบทอดคลาส (Inheritance) อย่างละเอียด, การประยุกต์ใช้ Polymorphism ด้วย virtual, override, abstract และ interface รวมถึงการเข้าใจความแตกต่างระหว่าง Abstract Class และ Interface พร้อมทั้งการจัดการ Encapsulation ด้วย Access Modifiers อย่างถูกต้อง ซึ่งจะช่วยให้ผู้อ่านสามารถออกแบบโครงสร้างโปรแกรมได้มีประสิทธิภาพและยืดหยุ่นมากขึ้น

นอกจากนี้ หนังสือยังเจาะลึกการจัดการข้อผิดพลาด (Exception Handling) ด้วยเทคนิค try-catch-finally การสร้าง Exception แบบกำหนดเอง (Custom Exception) และการใช้ throw keyword เพื่อให้โปรแกรมมีความทนทานและสามารถจัดการกับเหตุการณ์ที่ไม่คาดฝันได้อย่างเหมาะสม ในส่วนของการสืบค้นและประมวลผลข้อมูล LINQ (Language Integrated Query) ก็ได้รับการอธิบายอย่างละเอียด พร้อมตัวอย่างการใช้คำสั่ง query ต่าง ๆ เช่น where, select, orderBy, groupBy, join และ let รวมถึง Lambda Expression และการใช้ delegate อย่าง Func และ Action ที่ช่วยเพิ่มความยืดหยุ่นและความกระชับให้กับโค้ด

หนังสือยังให้ความสำคัญกับการทำงานกับไฟล์และระบบไฟล์ (File Handling) โดยสอนการอ่านและเขียนไฟล์ผ่านคลาสพื้นฐานอย่าง File, StreamReader และ StreamWriter รวมถึงการจัดการเส้นทางไฟล์และโฟลเดอร์ด้วย Path, Directory และ FileInfo เพื่อให้ผู้อ่านสามารถจัดการข้อมูลในระบบไฟล์ได้อย่างมีประสิทธิภาพและปลอดภัย สุดท้าย บทที่เกี่ยวกับพื้นฐานของ Asynchronous Programming จะช่วยให้เข้าใจแนวคิด async/await, การใช้ Task และ Task รวมถึงวิธีการจัดการงานหลายตัวพร้อมกัน ซึ่งจำเป็นสำหรับการพัฒนาแอปพลิเคชันที่ต้องการตอบสนองรวดเร็วและรองรับงานแบบขนาน

เนื้อหาในเล่มถูกออกแบบให้เหมาะกับนักพัฒนาที่ต้องการยกระดับทักษะของตนเองอย่างเป็นระบบและครบถ้วน โดยทุกบทจะมีคำอธิบายเชิงลึกและตัวอย่างโปรแกรมบูรณาการที่ช่วยให้ผู้อ่านได้ทดลองเขียนและเข้าใจแนวคิดในทางปฏิบัติอย่างแท้จริง ผู้อ่านสามารถนำความรู้เหล่านี้ไปต่อยอดพัฒนาแอปพลิเคชันที่มีความซับซ้อนและรองรับการทำงานจริงในสภาพแวดล้อมต่าง ๆ ได้อย่างมั่นใจ

หวังเป็นอย่างยิ่งว่าหนังสือเล่มนี้จะเป็นเครื่องมือสำคัญที่จะช่วยให้คุณก้าวข้ามขั้นพื้นฐานและก้าวเข้าสู่การเป็นนักพัฒนา C# มืออาชีพที่มีความเข้าใจลึกซึ้ง พร้อมรับมือกับโจทย์ที่ท้าทายในโลกการพัฒนาซอฟต์แวร์ยุคใหม่ได้อย่างมั่นใจและมีประสิทธิภาพ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

บทที่ 6
OOP ขั้นกลาง
(Intermediate OOP)

เนื้อหา

- OOP ขั้นกลาง
- Inheritance (การสืบทอดคลาส) — เชิงลึก
- อธิบาย Inheritance (การสืบทอดคลาส) ใน C# แบบ เชิงลึก
- Polymorphism ใน C# ที่ครอบคลุมทั้ง virtual, override, abstract และ interface
- Encapsulation ใน C# โดยเน้นเรื่อง Access Modifiers
- คำอธิบายเชิงลึก เรื่อง Abstract Class vs Interface ใน C#
- ตัวอย่างโปรแกรมบูรณาการ

บทนำบทที่ 6: OOP ขั้นกลาง

การเขียนโปรแกรมเชิงวัตถุ (Object-Oriented Programming หรือ OOP) เป็นหัวใจสำคัญในการพัฒนาซอฟต์แวร์ที่มีโครงสร้างและจัดการได้ง่ายขึ้น บทนี้จะพาผู้อ่านก้าวสู่ความเข้าใจเชิงลึกของแนวคิด OOP ในระดับกลางที่ช่วยให้การออกแบบโปรแกรมมีความยืดหยุ่นและสามารถขยายได้อย่างมีประสิทธิภาพ เริ่มต้นด้วยหัวข้อการสืบทอดคลาส (Inheritance) ซึ่งช่วยให้เราสามารถสร้างคลาสลูกที่รับคุณสมบัติและพฤติกรรมจากคลาสแม่ได้ ช่วยลดความซ้ำซ้อนของโค้ดและเพิ่มความสามารถในการนำกลับมาใช้ใหม่ ในบทนี้จะอธิบายการสืบทอดใน Kotlin อย่างละเอียด พร้อมตัวอย่างการใช้งานจริง

ต่อมาคือการทำงานของ Polymorphism ที่รวมถึงการใช้ virtual, override, abstract และ interface ซึ่งช่วยให้โปรแกรมสามารถจัดการกับวัตถุหลายรูปแบบได้อย่างยืดหยุ่น บทนี้จะอธิบายแนวคิดและวิธีใช้งานฟีเจอร์เหล่านี้ใน Kotlin อย่างเข้าใจง่ายและชัดเจน นอกจากนี้ Encapsulation หรือการห่อหุ้มข้อมูลเป็นอีกหนึ่งแนวคิดที่สำคัญที่ช่วยควบคุมการเข้าถึงข้อมูลผ่านตัวกำหนดสิทธิ์ (Access Modifiers) เช่น public, private, protected และ internal เพื่อรักษาความปลอดภัยและความถูกต้องของข้อมูล บทนี้ยังเปรียบเทียบความแตกต่างระหว่าง Abstract Class กับ Interface ซึ่งเป็นพื้นฐานสำคัญในการออกแบบสถาปัตยกรรมโปรแกรมอย่างมืออาชีพ

OOP ขั้นกลาง

- Inheritance (การสืบทอดคลาส)

- Polymorphism (virtual, override, abstract, interface)
- Encapsulation (Access modifiers: public, private, protected, internal)
- Abstract Class vs Interface

เยี่ยม — ต่อไปนี้ผมจะอธิบาย **บทที่ 6: OOP ขั้นกลาง (Intermediate OOP in C#)** อย่างละเอียด ครอบคลุมแนวคิด หลักการ โค้ดตัวอย่างเต็มรูปแบบ ข้อควรระวัง และแนวทางปฏิบัติที่ดี เพื่อให้คุณใช้งานได้จริงทันที □

Inheritance (การสืบทอดคลาส)

แนวคิดสั้น ๆ: การสืบทอดคือการให้คลาสใหม่ (derived/subclass) รับคุณสมบัติและพฤติกรรมจากคลาสเดิม (base/superclass) เพื่อ reuse โค้ดและขยายฟังก์ชัน

จุดสำคัญใน C#:

- C# รองรับ *single class inheritance* — คลาสหนึ่งสืบทอดได้จากคลาสเดียวเท่านั้น แต่สามารถ implement หลาย interface ได้
- ใช้ syntax: class Dog : Animal { ... }
- base ใช้เรียกคอนสตรัคเตอร์หรือเมธอดของฐาน
- sealed ป้องกันการสืบทอดต่อ
- คอนสตรัคเตอร์ของ base ถูกเรียกก่อน derived (เรียงจากบนลงล่าง)

ตัวอย่าง (เต็มไฟล์):

```
using System;

namespace OOPExamples
{
    // Base class
    public class Animal
    {
        public string Name { get; }

        public Animal(string name)
        {
            Name = name;
            Console.WriteLine("Animal constructed: " + name);
        }

        public virtual void MakeSound()
        {
```

```
            Console.WriteLine($"{Name} makes a sound.");
        }
    }

    // Derived class
    public class Dog : Animal
    {
        public Dog(string name) : base(name)
        {
            Console.WriteLine("Dog constructed: " + name);
        }

        public override void MakeSound()
        {
            Console.WriteLine($"{Name} says: Woof!");
        }
    }

    class Program
    {
        static void Main()
        {
            Animal a = new Dog("Bingo");
            a.MakeSound(); // Polymorphic call -> Dog.MakeSound
        }
    }
}
```

ผลลัพธ์เมื่อรัน:

Animal constructed: Bingo

Dog constructed: Bingo

Bingo says: Woof!

ข้อควรระวัง: ห้ามเรียกเมธอด virtual ที่ overridden ในคอนสตรัคเตอร์ของ base เพราะ derived ยังไม่ถูกตั้งค่าเต็มที่ — อาจเกิดพฤติกรรมไม่คาดคิด

Polymorphism (Polymorphism: virtual, override, abstract, interface)

ความหมาย: Polymorphism = หลายรูปแบบ — ในบริบท OOP หมายถึง *การเรียกเมธอดเดียวกันโดยพฤติกรรมต่างกัน ขึ้นกับชนิดจริงของอ็อบเจ็กต์ (runtime)*

ประเภทสำคัญ:

1. **Runtime polymorphism (Overriding)** — ใช้ virtual ที่ base และ override ที่ derived
2. **Compile-time polymorphism (Overloading)** — same method name but different signature (ไม่ใช่ OOP runtime polymorphism)
3. **Abstract methods** — บังคับให้ derived ต้อง implement
4. **Interface** — contract; ใส่ method signatures, implement โดยหลายคลาสได้

virtual / override / sealed / new

- virtual — เปิดให้ override
- override — แทนที่ implementation ของ base
- sealed override — ห้ามคลาสถัดไป override อีก
- new — ซ่อนเมธอดของ base (method hiding) — ต่างจาก override เพราะเป็น compile-time binding ขึ้นกับ type ของตัวแปร

ตัวอย่างเปรียบเทียบ override vs new:

```
using System;

class Base
{
    public void SayHello() => Console.WriteLine("Base.SayHello");
    public virtual void SayHi() => Console.WriteLine("Base.SayHi");
}

class Derived : Base
{
    public new void SayHello() => Console.WriteLine("Derived.SayHello"); // hides
    public override void SayHi() => Console.WriteLine("Derived.SayHi"); // overrides
}

class Program
{
    static void Main()
    {
```

```
        Base b = new Derived();
        b.SayHello(); // calls Base.SayHello (hiding -> bound by compile-time type)
        b.SayHi();    // calls Derived.SayHi (virtual -> runtime dispatch)
    }
}
```

ผลลัพธ์:

```
Base.SayHello
Derived.SayHi
```

interface polymorphism (ใช้งานได้กว้าง):

```
public interface IShape { double Area(); }
public class Circle : IShape { /*...*/ }
public class Rectangle : IShape { /*...*/ }

List<IShape> shapes = new() { new Circle(2), new Rectangle(3,4) };
foreach (var s in shapes) Console.WriteLine(s.Area()); // เรียก polymorphically
```

abstract

- abstract class อาจมีทั้ง method ที่มี body และ method ที่ไม่มี (abstract)
- abstract method ถูก implement โดย derived (ถ้า derived ไม่ abstract)

Encapsulation (การห่อหุ้ม) — Access Modifiers: public, private, protected, internal (และที่ควรรู้เพิ่มเติม)

แนวคิด: ซ่อนรายละเอียดภายใน (implementation) และเปิดเฉพาะสิ่งที่จำเป็นให้ผู้ใช้ (interface) เพื่อควบคุม state และป้องกันการใช้งานผิดพลาด

Modifiers ที่สำคัญใน C#:

- public — เข้าถึงได้ทุกที่
- private — เข้าถึงได้ภายในคลาสเดียว
- protected — เข้าถึงได้ในคลาสและ derived classes
- internal — เข้าถึงได้ภายใน assembly เดียวกัน
- protected internal — เข้าถึงได้ถ้าเป็น derived **หรือ** ใน assembly เดียวกัน
- private protected (C# 7.2+) — เข้าถึงได้ใน derived แต่เฉพาะถ้าอยู่ใน assembly เดียวกัน

ตัวอย่างแนวทางการ Encapsulation (BankAccount):

```
using System;

public class BankAccount
```

```
{
    private decimal _balance; // field private

    public string Owner { get; } // public read-only property

    public decimal Balance => _balance; // read-only outside

    public BankAccount(string owner, decimal initial)
    {
        Owner = owner;
        _balance = initial;
    }

    public void Deposit(decimal amount)
    {
        if (amount <= 0) throw new ArgumentException("amount must be > 0");
        _balance += amount;
    }

    public void Withdraw(decimal amount)
    {
        if (amount <= 0) throw new ArgumentException("amount must be > 0");
        if (amount > _balance) throw new InvalidOperationException("Insufficient funds");
        _balance -= amount;
    }
}

class Program
{
    static void Main()
    {
        var acc = new BankAccount("Alice", 100);
        acc.Deposit(50);
        acc.Withdraw(30);
```

```
        Console.WriteLine(acc.Balance); // 120
        // acc._balance = 1000; // ไม่สามารถเข้าถึงโดยตรง -> encapsulation สำเร็จ
    }
}
```

การเปิดเผย Collections อย่างปลอดภัย:

- อย่าส่ง List<T> แบบ mutable ออกจากคลาสโดยตรง ถ้าต้องการเพียงอ่าน ให้ส่ง IReadOnlyList<T> หรือ AsReadOnly() หรือ clone ก่อนส่ง

Properties vs Fields:

- ใช้ private fields + public properties เพื่อใส่ validation, lazy init หรือ logic ใน getter/setter
- public fields ถือว่าไม่ดีในเชิง encapsulation

Abstract Class vs Interface — แตกต่าง/เมื่อใดควรใช้

สรุปสั้น ๆ ก่อนลงรายละเอียด:

- **Interface** = ข้อตกลง (contract) — บอก *อะไรที่ต้องทำ* (method signatures, properties, events). สามารถ implement ได้หลาย interface. (ตั้งแต่ C# 8+ interface สามารถมี default implementations แต่ยังไม่มี field instance)
- **Abstract class** = คลาสฐานที่อาจมีทั้ง implementation และ abstract members — ใช้เมื่อมี *พฤติกรรมที่ใช้ร่วมกัน* หรือ *state (fields)* ที่ต้องแชร์ระหว่าง subclasses. คลาสนี้สามารถมีคอนสตรัคเตอร์และตัวเก็บสถานะได้ แต่สามารถสืบทอดได้แค่คลาสเดียว

ตารางสรุป (สำคัญ):

- *Instances:* interface ไม่สามารถ instantiate; abstract class ก็ไม่สามารถ instantiate
- *Fields:* interface ไม่มี instance fields; abstract class มีได้
- *Constructors:* interface ไม่มี; abstract class มี
- *Multiple inheritance:* class สามารถ implement หลาย interface แต่สืบทอดได้จาก class เดียว (abstract หรือไม่)
- *Default implementation:* (C# 8+) interface อนุญาต default method bodies แต่ยังมีข้อจำกัดและควรใช้ระมัดระวัง

ตัวอย่างเปรียบเทียบ:

```
public interface ILogger
{
    void Log(string message);
}

public abstract class RepositoryBase
```

```
{
    protected string _connectionString;

    protected RepositoryBase(string connectionString)
    {
        _connectionString = connectionString;
    }

    public abstract void Save(object entity); // forced to implement
    public virtual void OpenConnection() { /* default implementation */ }
}
```

เมื่อใช้อะไรดี?

- ใช้ **interface** เมื่อ: คุณต้องการบังคับ contract ที่หลากหลายคลาสสามารถ implement ได้ (เช่น IDisposable, IEnumerable<T>, IRepository), ต้องการความยืดหยุ่นและรองรับการทดสอบ (mocking)
- ใช้ **abstract class** เมื่อ: มี logic/fields ร่วมที่ต้องแชร์, ต้องการคอนสตรัคเตอร์, หรือเมื่อคุณต้องการให้มี default behavior ที่ subclasses สามารถ reuse

ข้อพิจารณาพิเศษ:

- หาก API ของคุณอาจขยายในอนาคตและคุณต้องการให้ทุก implementor ไม่ต้องเปลี่ยนมาก — interface แบบมี default implementations อาจช่วย แต่ยังต้องระวังการออกแบบ (อาจทำให้ความหมายของ interface เปลี่ยน)
- สำหรับ library/public API: มักแนะนำให้เริ่มเป็น interface เพื่อความยืดหยุ่น แล้วถ้าจำเป็นค่อยให้ abstract base class เป็น helper/adapter ที่ implement interface และเตรียม implementation พื้นฐาน

ตัวอย่างรวม (Abstract + Interface + Polymorphism + Encapsulation)

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

public interface IPayable
{
    decimal CalculatePay();
}
```

```
public abstract class Employee
{
    public string Name { get; }
    protected Employee(string name) => Name = name;
    public abstract decimal CalculateBonus();
    public virtual void PrintInfo() => Console.WriteLine($"Employee: {Name}");
}

public class SalariedEmployee : Employee, IPayable
{
    private decimal _salary; // encapsulated field

    public SalariedEmployee(string name, decimal salary) : base(name) => _salary = salary;

    public override decimal CalculateBonus() => _salary * 0.1m;
    public decimal CalculatePay() => _salary;
    public override void PrintInfo() => Console.WriteLine($"Salaried: {Name}, Salary {_salary}");
}

class Program
{
    static void Main()
    {
        List<Employee> emps = new List<Employee>
        {
            new SalariedEmployee("Alice", 5000),
            new SalariedEmployee("Bob", 7000)
        };

        foreach(var e in emps)
        {
            e.PrintInfo();
            Console.WriteLine("Bonus: " + e.CalculateBonus());
            if (e is IPayable p) Console.WriteLine("Pay: " + p.CalculatePay());
```

```
        }
    }
}
```

ข้อควรระวัง & Best Practices (โดยย่อ)

- อย่าเรียก virtual method จากคอนสตรัคเตอร์ของ base (อาจเกิดการเรียก method ของ derived ขณะที่ derived ยังไม่ถูกสร้างสมบูรณ์)
- ถ้าไม่อยากให้คลาสถูกสืบทอดต่อ ให้ใช้ sealed
- ถ้าต้องการซ่อนเมธอดของ base ใช้ new แต่เข้าใจผลต่างจาก override
- Prefer **properties** (with private setter or readonly) แทน public fields
- ใช้ **interface** สำหรับ contract (testability + multiple implementations)
- ใช้ **abstract class** เมื่อมี state หรือ behavior ที่ต้องแชร์
- จำกัดการเข้าถึง (encapsulation): ให้ mínimos visibility ที่จำเป็น (private หรือ internal แทน public ถ้าเป็นไปได้)
- สำหรับ collections: คืนเป็น IReadOnlyList<T> หรือ IEnumerable<T> เพื่อป้องกันการแก้ไขภายนอก

Inheritance (การสืบทอดคลาส) — เชิงลึก

1 — Inheritance (การสืบทอดคลาส) — เชิงลึก

- C# รองรับ **single class inheritance** (คลาสเดียวเท่านั้น) แต่อนุญาต multiple interface implementation
- ลำดับการสร้างอ็อบเจ็กต์: **Base constructor ⟶ Derived constructor**
- base(...) ใช้เรียกคอนสตรัคเตอร์ของฐาน; base.Method() เรียกเมธอดของ base จาก derived
- sealed ปิดการสืบทอดของ **คลาส** (sealed class C { }) หรือ **เมธอด** (public sealed override void M() { }) — ประโยชน์ทั้งเชิงออกแบบ (ป้องกันการขยาย) และเชิงประสิทธิภาพ (JIT อาจ devirtualize/inlining ได้ง่ายขึ้น)

ข้อควรระวังสำคัญ:

อย่าเรียก virtual / override method ที่ **ขึ้นกับตัวแปรของ derived** ภายในคอนสตรัคเตอร์ของ base — derived ยังไม่ถูก initialised สมบูรณ์ จึงอาจได้ค่า null หรือพฤติกรรมไม่คาดคิด (ตัวอย่างด้านล่าง)

ตัวอย่าง (ปัญหา virtual ใน constructor):

```
using System;
class Base {
    public Base() {
```

```
        Initialize(); // Dangerous: derived override may run before derived ctor
    }
    protected virtual void Initialize() { Console.WriteLine("Base init"); }
}

class Derived : Base {
    private readonly string _name;
    public Derived(string name) {
        _name = name; // executed AFTER base ctor
    }
    protected override void Initialize() {
        // _name is still null here -> NullReferenceException if used
        Console.WriteLine(_name?.Length ?? -1);
    }
}

class Program {
    static void Main() {
        var d = new Derived("hello"); // may print -1 or throw, bug source
    }
}
```

2 — Polymorphism (virtual / override / abstract / interface) — เชิงลึก

- **runtime polymorphism** เกิดจาก virtual/override (dynamic dispatch) — การเลือก implementation เกิดที่ runtime ตามชนิดจริงของอ็อบเจ็กต์
- new ทำ **method hiding** — binding ตาม type ของตัวแปร (compile-time) — มักเป็นแหล่งสับสน
- abstract บังคับให้ derived ต้อง implement; abstract class สามารถมี state/fields และบาง method ที่มี body ได้
- **interface** = contract; ให้ความยืดหยุ่นสูง (และ seit C# 8: มี default implementation ได้ — ระวังผลข้างเคียงในการออกแบบ API)

override vs new — ตัวอย่างชัดเจน

```
class Base {
    public virtual void V() => Console.WriteLine("Base.V");
    public void H() => Console.WriteLine("Base.H");
```

```
}

class Derived : Base {
    public override void V() => Console.WriteLine("Derived.V");
    public new void H() => Console.WriteLine("Derived.H");
}

Base b = new Derived();
b.V(); // Derived.V    (virtual -> runtime dispatch)
b.H(); // Base.H       (hiding -> compile-time dispatch)
```

explicit interface implementation — ใช้เมื่อต้องการซ่อนเมธอดจาก public API หรือเมื่อสอง interface ขัดกัน:

```
interface IA { void M(); }
interface IB { void M(); }

class C : IA, IB {
    void IA.M() => Console.WriteLine("IA.M");
    void IB.M() => Console.WriteLine("IB.M");
    public void M() => Console.WriteLine("Public M");
}
```

เรียก ((IA)c).M() หรือ ((IB)c).M() เพื่อเลือก implementation

interface default implementations (C# 8+)

- อนุญาตให้ใส่ method body ใน interface — ช่วยไม่ทำให้ breaking change เมื่อเพิ่ม method ใหม่ แต่ทำให้ API design ซับซ้อน — ใช้ด้วยความระมัดระวัง (library/public API ควรพิจารณาอย่างรอบคอบ)

delegates / generics variance (covariance / contravariance) — สำคัญสำหรับ polymorphism กับ generic interfaces:

- out T = **covariant** (สามารถใช้ IEnumerable<Derived> กับ IEnumerable<Base>)
- in T = **contravariant** (เช่น IComparer<Base> สามารถใช้เป็น IComparer<Derived>)
- ตัวอย่าง:

```
IEnumerable<Dog> dogs = new List<Dog>();
IEnumerable<Animal> animals = dogs; // allowed: covariance (out)

Action<Animal> eatAnimal = a => Console.WriteLine("eat");
```

Action<Dog> eatDog = eatAnimal; // allowed: contravariance (in)

3 — Encapsulation (Access modifiers) — ละเอียดและเทคนิค

Modifiers:

- public — ทุกที่
- private — ภายใน class เท่านั้น
- protected — class + derived classes (ทุก assembly)
- internal — ภายใน **assembly** เดียวกัน
- protected internal — (OR) ถ้า derived **หรือ** อยู่ใน assembly เดียวกัน
- private protected (C# 7.2+) — (AND) derived **และ** อยู่ใน assembly เดียวกัน

ตัวอย่าง InternalsVisibleTo (ใช้กับ unit testing):

// AssemblyInfo.cs (หรือบนไฟล์ C# ใดก็ได้ที่คอมไพล์ใน assembly)

using System.Runtime.CompilerServices;

[assembly: InternalsVisibleTo("MyProject.Tests")]

internal class InternalService { internal void Foo() { } }

Properties vs Fields

- ให้ใช้ private fields + public properties (หรือ public readonly property) เพื่อใส่ validation, lazy init, encapsulation
- init-only property (C# 9) — ช่วยสร้าง immutable objects ที่อ่านได้ภายนอก แต่กำหนดค่าได้ใน object initialiser

Collections exposure

- อย่าส่ง List<T> mutable ออกไป — ให้ส่ง IReadOnlyList<T>, IEnumerable<T> หรือ AsReadOnly() หรือ clone
- Example:

private List<Order> _orders = new();

public IReadOnlyList<Order> Orders => _orders;

Thread-safety & Immutability

- ถ้าต้องแชร์ระหว่าง threads ให้พิจารณา immutable objects (records), locks, Concurrent collections
- readonly fields + immutable types เป็นวิธีปลอดภัยและง่ายที่สุด

4 — Abstract Class vs Interface — แบบลงลึก (เมื่อใด ใช้อย่างไร)

สรุปความต่างเชิงเทคนิค

- Interface: ไม่มี instance fields; ไม่มี constructor; ตั้งแต่ C#8 สามารถมี default impl; รองรับ multiple inheritance

- Abstract class: มี state, constructor, protected helper methods; สืบทอดได้แค่คลาสเดียว

แนวทางเลือก

- หากต้องการ **contract/polymorphism** และต้องการให้ implementors หลากหลาย — เลือก interface
- หากมี **behavior ร่วม/fields** ที่ต้องแชร์เป็น default — เลือก abstract class (หรือให้ interface + helper/adapter abstract class ผสมกัน)
- API design tip: ถ้าเป็น public library ให้ expose interface เป็นหลัก แล้ว provide abstract base class หรือ default implementation เป็น convenience

Pattern ที่สัมพันธ์กัน

- Template Method ⟶ ใช้ abstract class (กำหนด skeleton ของอัลกอริทึม แล้วเรียก abstract steps)
- Strategy ⟶ ใช้ interface (แลกเปลี่ยนพฤติกรรมได้ใน runtime)

5 — Design Principles ที่เกี่ยวข้อง & Pitfalls (LSP, ISP, SOLID)

- **Liskov Substitution Principle (LSP)** — Derived ควรใช้ได้แทน Base โดยไม่เปลี่ยนสัญญา (preconditions/postconditions)
    - Classic violation: Rectangle/Square ถ้าทำเป็น subclass จะทำให้สมมติฐานของ caller (set Width แล้ว set Height) ผิดพลาด
- **ISP (Interface Segregation)** — แบ่ง interface ให้เล็ก อ่อนตัว ไม่บังคับ method ที่ไม่จำเป็นให้ implementor
- **Dependency Inversion / DI** — โค้ดควรขึ้นกับ abstraction (interface/abstract), ไม่ใช่ concrete class เพื่อทดสอบง่ายและลด coupling

ตัวอย่าง LSP violation (Rectangle/Square):

```
public class Rectangle {
    public virtual int Width { get; set; }
    public virtual int Height { get; set; }
    public int Area() => Width * Height;
}
public class Square : Rectangle {
    public override int Width { set { base.Width = base.Height = value; } }
    public override int Height { set { base.Width = base.Height = value; } }
}
void Test(Rectangle r) {
    r.Width = 5;
    r.Height = 4;
```

```
    Console.WriteLine(r.Area()); // expected 20 — but Square gives 16 or 25 => LSP violation
}
```

แก้: แยก IShape (read-only) หรือใช้ composition แทน inheritance

6 — Runtime / CLR / Performance notes (สิ่งที่นักพัฒนาควรรู้)

- คำสั่ง IL: callvirt มักถูกใช้เพื่อเรียก instance method เพราะมันทำ null-check และ handle virtual dispatch — แต่ JIT อาจ optimize (devirtualize) ให้เป็น direct call ถ้ารู้ชนิด concrete (เช่น sealed type)
- virtual calls มี overhead เล็กน้อย (indirection ผ่าน vtable) — แต่ JIT สามารถ inline/devirtualize ในบางกรณี (sealed types, single implementation)
- **boxing** เกิดเมื่อ value type (struct) ถูก cast เป็น object หรือ interface — หลีกเลี่ยงใน hotspot
- การเรียกผ่าน **interface** อาจมี overhead เทียบกับ direct class call แต่ปกติไม่เป็นปัญหานอก hotspot
- ใช้ sealed หรือ sealed override ในจุดที่แน่นอนถ้าต้องการช่วย JIT inlining และลด overhead

7 — Modern C# features ที่เกี่ยวข้องกับ OOP

- **records** (C# 9+) — ออกแบบเพื่อ immutable data objects, ให้ value-based equality, รองรับ inheritance ของ record types
- **init-only properties** (C# 9) — ช่วยทำ immutable initialisation
- **default interface methods** (C# 8) — อธิบายแล้ว แต่ระวังเมื่อออกแบบ public API

8 — ตัวอย่างโค้ดเชิงลึก (ใช้งานได้ใน Console app)

8.1 — ตัวอย่างแสดงปัญหา virtual ใน constructor และแนวทางแก้

```
using System;
namespace DeepOOPExamples {
    class Base {
        public Base() {
            Console.WriteLine("Base ctor: calling Initialize()");
            Initialize(); // dangerous
        }
        protected virtual void Initialize() {
            Console.WriteLine("Base.Initialize");
        }
    }

    class Derived : Base {
```

```
        private readonly string _name;
        public Derived(string name) {
            _name = name;
            Console.WriteLine("Derived ctor finished, name set to: " + _name);
        }
        protected override void Initialize() {
            // _name not set yet -> bug
            Console.WriteLine("Derived.Initialize -> name length = " + (_name?.Length ?? -999));
        }
    }

    class Program {
        static void Main() {
            Console.WriteLine("Creating Derived...");
            var d = new Derived("Hello");
        }
    }
}
```

แก้ไขแนวปฏิบัติ:

- อย่าเรียก virtual methods ใน base ctor; ถ้าจำเป็น ให้เปลี่ยนเป็น non-virtual protected method หรือใช้ factory/static Initialize pattern หรือให้ derived เรียก initialization เองจาก constructor ของมัน

8.2 — Explicit interface + default interface method + hiding/override demo

```
using System;

public interface ILogger {
    void Log(string msg);
    // Default implementation (C#8+)
    void LogWarning(string msg) => Log("WARN: " + msg);
}

public class ConsoleLogger : ILogger {
    public void Log(string msg) => Console.WriteLine(msg);
    // inherits default LogWarning unless we override
```

```
}

public interface ISecret {
    void Do();
}
public class Multiple : ISecret {
    void ISecret.Do() => Console.WriteLine("ISecret.Do (explicit)");
    public void Do() => Console.WriteLine("Public Do()");
}

class Program {
    static void Main() {
        ILogger l = new ConsoleLogger();
        l.Log("Hello");          // ConsoleLogger.Log
        l.LogWarning("Be careful"); // default impl in interface

        Multiple m = new Multiple();
        m.Do();                  // Public Do()
        ((ISecret)m).Do();       // explicit impl
    }
}
```

9 — Best practices / Practical checklist (รวบรัด)

- เริ่มจาก sealed (หรือ non-inheritable) และเปิดให้สืบทอดเมื่อจำเป็น (prefer sealed-by-default)
- ใช้ interface สำหรับ contract และ testability; ใช้ abstract เมื่อมีการแชร์ state/behavior
- อย่าเรียก virtual methods ใน constructor ของ base
- ใช้ IReadOnlyList<T> / IEnumerable<T> เมื่อ expose collection เพื่อป้องกัน mutation
- ให้ visibility แบบ **minimal** ที่ทำงานได้ (private > protected > internal > public)
- ปฏิบัติตาม LSP — ถ้ารู้สึกต้องเปลี่ยนพฤติกรรมของ base แสดงว่าอาจต้อง refactor (composition หรือ new contracts)
- ใช้ explicit interface implementation เพื่อซ่อนเมธอดเมื่อจำเป็น
- ระวัง boxing เมื่อทำงานกับ struct + interfaces
- เพิ่ม sealed override บน methods ที่ไม่ต้องการให้ override เพิ่มเพื่อความชัดเจนและช่วย JIT

อธิบาย Inheritance (การสืบทอดคลาส) ใน C# แบบ เชิงลึก

1. แนวคิดของ Inheritance

Inheritance คือการสร้าง **คลาสใหม่ (Derived Class / Subclass)** โดยนำคุณสมบัติและพฤติกรรม (Fields, Properties, Methods) มาจาก **คลาสเดิม (Base Class / Superclass)**

- เป้าหมายหลัก: **ลดการซ้ำซ้อนของโค้ด (code reuse)** และทำให้โค้ดมีโครงสร้างแบบ **hierarchy**
- ใน C#, การสืบทอดเป็น **Single Inheritance** (คลาสหนึ่งสืบทอดได้เพียงคลาสเดียว) แต่สามารถ **Implement ได้หลาย Interface**

2. หลักการทำงาน

เมื่อคลาส B สืบทอดจากคลาส A:

1. คลาส B จะได้รับ **Fields**, **Properties**, **Methods** ที่เป็น public และ protected ของคลาส A
2. คลาส B สามารถ:
    - ใช้ฟังก์ชันเดิมจากคลาส A ได้
    - เพิ่มสมาชิกใหม่ของตัวเอง
    - **Override** เมธอดที่เป็น virtual หรือ abstract
3. สามารถเรียก **Constructor ของ Base Class** ผ่าน base(...)
4. สมาชิกที่เป็น private ใน Base Class จะไม่สามารถเข้าถึงโดยตรงใน Subclass ได้

3. Syntax การประกาศ Inheritance

```
class BaseClass
{
    public void ShowMessage()
    {
        Console.WriteLine("Message from BaseClass");
    }
}

class DerivedClass : BaseClass
{
    public void ShowAnotherMessage()
```

```
    {
        Console.WriteLine("Message from DerivedClass");
    }
}
```

DerivedClass : BaseClass หมายถึง DerivedClass สืบทอดคุณสมบัติจาก BaseClass

---

4. ตัวอย่างโปรแกรมเต็มไฟล์

```
using System;

// Base Class
public class Animal
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Animal(string name, int age)
    {
        Name = name;
        Age = age;
    }

    public void Eat()
    {
        Console.WriteLine($"{Name} is eating.");
    }

    public void Sleep()
    {
        Console.WriteLine($"{Name} is sleeping.");
    }
}

// Derived Class
public class Dog : Animal
```

```
{
    public string Breed { get; set; }

    public Dog(string name, int age, string breed)
        : base(name, age) // เรียก constructor ของ Animal
    {
        Breed = breed;
    }

    public void Bark()
    {
        Console.WriteLine($"{Name} the {Breed} is barking!");
    }
}

// Program Entry
class Program
{
    static void Main()
    {
        // สร้างออบเจ็กต์จาก Dog
        Dog myDog = new Dog("Buddy", 3, "Golden Retriever");

        // ใช้เมธอดจาก Base Class
        myDog.Eat();
        myDog.Sleep();

        // ใช้เมธอดจาก Derived Class
        myDog.Bark();
    }
}
```

5. ผลลัพธ์การรัน

```
Buddy is eating.
```

Buddy is sleeping.

Buddy the Golden Retriever is barking!

---

6. Access Modifiers กับ Inheritance

- public ⟶ เข้าถึงได้จากทุกที่
- protected ⟶ เข้าถึงได้จากในคลาสตัวเองและคลาสที่สืบทอด
- private ⟶ เข้าถึงได้เฉพาะในคลาสตัวเอง
- internal ⟶ เข้าถึงได้ภายใน Assembly เดียวกัน

---

7. ข้อดีของ Inheritance

- ลดการซ้ำซ้อนของโค้ด (Code Reuse)
- ส่งเสริมหลัก OOP เช่น **Polymorphism**
- โค้ดอ่านง่ายและดูเป็นลำดับชั้น

---

8. ข้อควรระวัง

- ใช้ Inheritance เมื่อ **มีความสัมพันธ์ "is-a" จริง ๆ**

  เช่น Dog *is a* Animal
- อย่าใช้สืบทอดถ้าแค่ต้องการใช้บางเมธอด ควรใช้ **Composition** แทน
- การสืบทอดที่ลึกเกินไป (Deep Inheritance) ทำให้โค้ดซับซ้อนและยากต่อการบำรุงรักษา

ตัวอย่างโปรแกรม **Inheritance (การสืบทอดคลาส)** ให้แบบเต็ม ๆ ทั้ง **3 โปรแกรมพื้นฐาน** และ **3 โปรแกรมแนวประยุกต์** พร้อมโครงสร้างโฟลเดอร์, คำอธิบายโค้ด และผลการรัน

โปรแกรมพื้นฐาน Inheritance (3 ตัวอย่าง)

---

1. ตัวอย่างพื้นฐาน: สัตว์ (Animal) กับ สุนัข (Dog)

โครงสร้างโปรเจกต์

/BasicInheritance1/

  Program.cs

Program.cs

```
using System;

// Base class
```

```
public class Animal
{
    public string Name { get; set; }

    public Animal(string name)
    {
        Name = name;
    }

    public void Eat()
    {
        Console.WriteLine($"{Name} is eating.");
    }
}

// Derived class
public class Dog : Animal
{
    public Dog(string name) : base(name)
    {
    }

    public void Bark()
    {
        Console.WriteLine($"{Name} is barking!");
    }
}

class Program
{
    static void Main()
    {
        Dog dog = new Dog("Bingo");
        dog.Eat();    // เรียกเมธอดจากคลาส Animal
```

```
        dog.Bark();  // เรียกเมธอดของ Dog เอง
    }
}
```

คำอธิบาย

- Animal เป็นคลาสฐาน มีคุณสมบัติ Name และเมธอด Eat()
- Dog สืบทอดจาก Animal และเพิ่มเมธอด Bark()
- สร้างอ็อบเจ็กต์ Dog ชื่อ "Bingo" แล้วเรียกเมธอดทั้ง 2

ผลการรัน

```
Bingo is eating.
Bingo is barking!
```

---

2. ตัวอย่างการใช้ Constructor และ Properties ที่สืบทอด

โครงสร้างโปรเจกต์

```
/BasicInheritance2/
  Program.cs
```

Program.cs

```
using System;

public class Vehicle
{
    public string Brand { get; set; }
    public Vehicle(string brand)
    {
        Brand = brand;
    }

    public void StartEngine()
    {
        Console.WriteLine($"{Brand} engine started.");
    }
}

public class Car : Vehicle
{
```

```
    public string Model { get; set; }

    public Car(string brand, string model) : base(brand)
    {
        Model = model;
    }

    public void Honk()
    {
        Console.WriteLine($"{Brand} {Model} honks!");
    }
}

class Program
{
    static void Main()
    {
        Car car = new Car("Toyota", "Corolla");
        car.StartEngine();
        car.Honk();
    }
}
```

คำอธิบาย

- Vehicle เป็นคลาสฐานเก็บแบรนด์ของรถและเมธอดสตาร์ทเครื่องยนต์
- Car สืบทอดจาก Vehicle เพิ่มคุณสมบัติ Model และเมธอด Honk()
- แสดงการเรียก constructor base ด้วย : base(brand)

ผลการรัน

Toyota engine started.

Toyota Corolla honks!

---

3. ตัวอย่างการ Override เมธอด (Polymorphism เบื้องต้น)

โครงสร้างโปรเจกต์

/BasicInheritance3/

Program.cs

Program.cs

```
using System;

public class Person
{
    public string Name { get; set; }

    public Person(string name)
    {
        Name = name;
    }

    public virtual void Introduce()
    {
        Console.WriteLine($"Hello, my name is {Name}.");
    }
}

public class Student : Person
{
    public Student(string name) : base(name)
    {
    }

    public override void Introduce()
    {
        Console.WriteLine($"Hi, I'm student {Name}.");
    }
}

class Program
{
    static void Main()
    {
```

```
        Person person = new Person("Alice");
        person.Introduce();

        Person student = new Student("Bob");
        student.Introduce();  // เรียก override method
    }
}
```

คำอธิบาย

- Person มีเมธอด Introduce() ที่สามารถถูก override ด้วย virtual
- Student สืบทอดจาก Person และ override Introduce()
- แสดง polymorphism เมื่อเรียกผ่านตัวแปร base

ผลการรัน

```
Hello, my name is Alice.
Hi, I'm student Bob.
```

โปรแกรมแนวประยุกต์ Inheritance (3 ตัวอย่าง)

---

1. ระบบจัดการพนักงาน (Employee Management System)

โครงสร้างโปรเจกต์

```
/AppInheritance1/
  Program.cs
```

Program.cs

```
using System;

public class Employee
{
    public string Name { get; set; }
    public decimal Salary { get; set; }

    public Employee(string name, decimal salary)
    {
        Name = name;
        Salary = salary;
    }
```