C#

Programming: ADVANCE

(Integrative-Generative AI Edition)

CONTENTS

- Design Patterns
- Memory Management
- Generics
- Delegate & Events
- Threading and Parallel

Bibliography

Student Price Book Center

ai

คำนำ

การพัฒนาซอฟต์แวร์ในระดับมืออาชีพในปัจจุบันไม่เพียงแต่ต้องเขียนโค้ดที่ทำงานได้ถูกต้องเท่านั้น แต่ยังต้องคำนึงถึงโครงสร้าง การออกแบบ ระบบการจัดการทรัพยากร และแนวทางปฏิบัติที่เหมาะสม เพื่อให้ซอฟต์แวร์มีประสิทธิภาพ ปลอดภัย และง่ายต่อการบำรุงรักษา หนังสือเล่มนี้ถูกออกแบบมาเพื่อพานักพัฒนาสู่การเรียนรู้ **C#** ในระดับ advance โดยเน้นทั้งแนวคิดเชิงลึก เทคนิคเชิงปฏิบัติ และตัวอย่างบูรณาการที่สามารถนำไปประยุกต์ใช้ได้จริง

บทที่ 11 จะพาผู้อ่านเจาะลึก **Design Patterns ที่สำคัญใน C#** ครอบคลุมทั้ง **Singleton, Factory, Strategy, Observer และ Dependency Injection (DI)** การเรียนรู้แพทเทิร์นเหล่านี้ช่วยให้นักพัฒนาสามารถออกแบบระบบที่ยืดหยุ่น ลดความซ้ำซ้อน และรองรับการขยายตัวในอนาคตได้อย่างมั่นคง พร้อมตัวอย่างเชิงลึกและการผสานแพทเทิร์นเข้าด้วยกัน เพื่อให้เข้าใจทั้งแนวคิดและการประยุกต์ใช้งานจริงในโปรเจกต์

บทที่ 12 จะเน้นการ **จัดการหน่วยความจำ (Memory Management)** ใน C# โดยครอบคลุมกลไกของ **Garbage Collector (GC)** การใช้ **IDisposable** ร่วมกับ **using statement** และการเข้าใจ **Boxing/Unboxing** การเรียนรู้หัวข้อเหล่านี้ช่วยให้นักพัฒนาสามารถเขียนโค้ดที่มีประสิทธิภาพ ใช้ทรัพยากรอย่างเหมาะสม และลดปัญหาการรั่วไหลของหน่วยความจำ รวมถึงสามารถวิเคราะห์และปรับปรุง performance ของแอปพลิเคชันได้อย่างมีประสิทธิภาพ

บทที่ 13 เจาะลึก **Generics** ซึ่งเป็นคุณสมบัติสำคัญที่ช่วยให้โค้ดยืดหยุ่น ใช้ซ้ำได้ และปลอดภัยต่อชนิดข้อมูล ผู้อ่านจะได้เรียนรู้การใช้งาน **List และ Dictionary<K,V>**, การสร้าง **generic classes และ methods** ด้วยตัวเอง และการกำหนด **constraints** เพื่อเพิ่มความปลอดภัยและความชัดเจนในการออกแบบ การเข้าใจ generics อย่างลึกซึ้งช่วยให้การพัฒนา API และ library มีความ maintainable และรองรับการขยายตัวได้ง่าย

บทที่ 14 ครอบคลุม **Delegate และ Events** ซึ่งเป็นกลไกสำคัญในการสร้างระบบ **event-driven** ใน C# การเรียนรู้ความหมายของ delegate, การใช้งาน **Multicast Delegate**, และการสร้าง **Event/ EventHandler** จะช่วยให้โค้ดสามารถตอบสนองต่อเหตุการณ์ต่าง ๆ ได้อย่างยืดหยุ่น ลดความซับซ้อน และรองรับการขยายตัวของระบบอย่างปลอดภัย นอกจากนี้ยังมีตัวอย่างบูรณาการและโปรเจกต์เต็มบทเพื่อให้ผู้อ่านเห็นภาพการใช้งานจริง

บทที่ 15 นำเสนอแนวคิด **Threading และ Parallel Programming** ซึ่งเป็นพื้นฐานสำคัญสำหรับการสร้างแอปพลิเคชันที่ตอบสนองเร็วและใช้ทรัพยากรอย่างมีประสิทธิภาพ ผู้อ่านจะได้เรียนรู้การใช้ **Thread และ ThreadPool**, ความแตกต่างระหว่าง **Task vs Thread**, การป้องกัน race condition ด้วย **lock และ Monitor**, และแนวคิด **async vs parallel vs concurrent** พร้อมตัวอย่างโปรเจกต์บูรณาการเพื่อให้เข้าใจทั้งทฤษฎีและการใช้งานจริง

ตลอดเล่ม หนังสือเน้นทั้ง **หลักการเชิงลึกและตัวอย่างเชิงปฏิบัติ** เพื่อให้นักพัฒนาสามารถต่อยอดความรู้และสร้างซอฟต์แวร์ที่มีคุณภาพสูง ทั้งในด้านการออกแบบระบบ การจัดการทรัพยากร การเขียนโค้ดยืดหยุ่น และการพัฒนาแอปพลิเคชันแบบมัลติทาสก์ หนังสือเล่มนี้จึงเป็นคู่มือที่เหมาะสำหรับนักพัฒนาที่ต้องการยกระดับทักษะ **C#** ไปสู่ระดับมืออาชีพ และพร้อมรับมือกับความท้าทายของงานพัฒนาซอฟต์แวร์สมัยใหม่ได้อย่างมั่นใจ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 11 Design Patterns ที่สำคัญใน C# (Design Patterns)..1

- Design Patterns ที่สำคัญใน C#
- Singleton — อินสแตนซ์เดียวทั่วระบบ (Deep dive)
- Factory — แยกการสร้างออกจากการใช้งาน (Deep dive)
- Observer — event-driven, push-based (Deep dive)
- Dependency Injection (DI) — ลึกสุดสำหรับการออกแบบระบบ
- การผสานแพทเทิร์นเข้าด้วยกัน (Integration)
- Singleton Pattern ใน C#
- Design Pattern: Factory Pattern ใน C#
- Design Pattern: Strategy ของภาษา C#
- Observer Pattern ซึ่งเป็น Behavioral Design Pattern
- Dependency Injection (DI) ซึ่งเป็น หลักการสำคัญของการออกแบบซอฟต์แวร์ ใน C#

บทที่ 12 การจัดการหน่วยความจำ (C# Memory Management) 125

- การจัดการหน่วยความจำ
- เชิงลึกของบทที่ 12: การจัดการหน่วยความจำใน C#
- รายละเอียดเชิงลึกเกี่ยวกับ Garbage Collector (GC) ใน C#
- IDisposable และ using statement
- Boxing และ Unboxing
- ตัวอย่างบูรณาการ

บทที่ 13 Generics (Generics)..181

- บทที่ 13: Generics
- การอธิบาย Generics ใน C# อย่างละเอียดเชิงลึก
- การอธิบาย List และ Dictionary<K,V> ใน C# แบบละเอียดเชิงลึก
- การสร้างคลาสและเมธอด Generic เองใน C#
- Constraints สำหรับ Generic ใน C#
- ตัวอย่างบูรณาการ

บทที่ 14 Delegate & Events (Delegate & Events) 247
- บทที่ 14: Delegate & Events
- รายละเอียดเชิงลึกของ Delegate & Events ใน C#
- Delegate คืออะไร ใน C# แบบเชิงลึก
- Multicast Delegate ใน C# แบบละเอียดเชิงลึก
- Event และ EventHandler ใน C# แบบละเอียดเชิงลึก
- ตัวอย่างบูรณาการ
- โปรเจกต์ C# เดียวแบบบูรณาการเต็มบท

บทที่ 15 Threading และ Parallel Programming (Threading and Parallel Programming) 307
- บทที่ 15: Threading และ Parallel Programming
- Threading และ Parallel Programming ใน C#
- การใช้ Thread และ ThreadPool ใน C#
- Task vs Thread ใน C#
- lock และ Monitor ใน C#
- async vs parallel vs concurrent
- โปรเจกต์บูรณาการ

บรรณานุกรม 384

บทที่ 11

Design Patterns ที่สำคัญใน C#

(Design Patterns)

เนื้อหา

- Design Patterns ที่สำคัญใน C#
- Singleton — อินสแตนซ์เดียวทั่วระบบ (Deep dive)
- Factory — แยกการสร้างออกจากการใช้งาน (Deep dive)
- Observer — event-driven, push-based (Deep dive)
- Dependency Injection (DI) — ลึกสุดสำหรับการออกแบบระบบ
- การผสานแพทเทิร์นเข้าด้วยกัน (Integration)
- Singleton Pattern ใน C#
- Design Pattern: Factory Pattern ใน C#
- Design Pattern: Strategy ของภาษา C#
- Observer Pattern ซึ่งเป็น Behavioral Design Pattern
- Dependency Injection (DI) ซึ่งเป็น หลักการสำคัญของการออกแบบซอฟต์แวร์ ใน C#

การออกแบบซอฟต์แวร์ที่มีคุณภาพสูงไม่เพียงแต่ต้องการโค้ดที่ทำงานได้ถูกต้อง แต่ยังต้องมีโครงสร้างและรูปแบบที่ช่วยให้การพัฒนา บำรุงรักษา และขยายระบบเป็นไปอย่างมีประสิทธิภาพ **Design Patterns** จึงถือเป็นเครื่องมือสำคัญสำหรับนักพัฒนาในการแก้ปัญหาซ้ำ ๆ ด้วยวิธีที่ผ่านการพิสูจน์แล้ว ในบทนี้จะเน้นการประยุกต์ใช้งาน Design Patterns ที่สำคัญใน **C#** ซึ่งช่วยให้นักพัฒนาสามารถออกแบบระบบที่ยืดหยุ่นและสามารถปรับตัวได้ตามความต้องการ

เริ่มจาก **Singleton Pattern** ซึ่งช่วยให้การสร้างออบเจ็กต์เกิดขึ้นเพียงครั้งเดียวและสามารถเข้าถึงได้ทั่วทั้งระบบ การใช้ Singleton เหมาะสำหรับการจัดการทรัพยากรที่ต้องใช้ร่วมกัน เช่น การเชื่อมต่อฐานข้อมูล หรือการจัดการ Configuration ของแอปพลิเคชัน การเข้าใจหลักการและข้อควรระวังในการใช้งาน Singleton เป็นสิ่งสำคัญเพื่อหลีกเลี่ยงปัญหาด้าน concurrency และ dependency ที่อาจเกิดขึ้น

ต่อด้วย **Factory Pattern** ซึ่งเป็นแนวทางในการสร้างออบเจ็กต์โดยไม่เปิดเผยตรรกะการสร้างให้กับผู้เรียกใช้งาน ทำให้ออบเจ็กต์มีความยืดหยุ่นและสามารถเปลี่ยนแปลงได้ง่าย ตัวอย่างเช่น การสร้างประเภทของ Product ที่แตกต่างกันตามเงื่อนไข โดยที่โค้ดส่วนที่เรียกใช้งานไม่จำเป็นต้องรู้

รายละเอียดของการสร้างออบเจ็กต์ การใช้ Factory ช่วยให้ระบบรองรับการขยายในอนาคตได้อย่างสะดวก

Strategy Pattern เป็นอีกหนึ่งเครื่องมือที่สำคัญสำหรับการกำหนดพฤติกรรมของออบเจ็กต์แบบไดนามิก โดยอนุญาตให้เปลี่ยนวิธีการทำงานของออบเจ็กต์ใน runtime ได้อย่างยืดหยุ่น การใช้ Strategy เหมาะกับกรณีที่ต้องการสนับสนุนหลายวิธีในการประมวลผลหรือคำนวณ เช่น การคำนวณค่าคอมมิชชั่นที่ต่างกันตามประเภทลูกค้า หรือการเลือกอัลกอริทึมในการจัดเรียงข้อมูล

นอกจากนี้ **Observer Pattern** ช่วยให้ระบบสามารถแจ้งเตือนการเปลี่ยนแปลงของออบเจ็กต์ไปยังหลาย ๆ ตัวที่สนใจอย่างอัตโนมัติ การประยุกต์ใช้ Observer ช่วยให้ส่วนประกอบของระบบไม่ขึ้นต่อกันโดยตรง ลดความซับซ้อน และสนับสนุนการพัฒนาเชิง event-driven ทำให้โค้ดอ่านง่ายและปรับปรุงได้ง่าย

สุดท้าย **Dependency Injection (DI)** เป็นแนวทางที่ช่วยจัดการความสัมพันธ์ระหว่างออบเจ็กต์โดยการฉีด dependency จากภายนอก แทนการสร้างออบเจ็กต์เองในคลาส ทำให้ระบบมีความยืดหยุ่นทดสอบง่าย และรองรับการเปลี่ยนแปลงของ dependency ได้โดยไม่กระทบต่อโค้ดหลัก การใช้ DI เป็นพื้นฐานสำคัญในการออกแบบซอฟต์แวร์ที่สะอาดและสามารถปรับตัวต่อเทคโนโลยีใหม่ ๆ ได้

บทนี้จึงมุ่งเน้นให้ผู้อ่านเข้าใจหลักการทำงานของ **Singleton, Factory, Strategy, Observer และ Dependency Injection** ใน C# พร้อมตัวอย่างเชิงปฏิบัติและแนวทางการประยุกต์ใช้จริง เพื่อให้สามารถออกแบบซอฟต์แวร์ที่มีโครงสร้างมั่นคง ยืดหยุ่น และรองรับการบำรุงรักษาในระยะยาวได้อย่างมีประสิทธิภาพ

การประยุกต์ใช้ Design Patterns อย่างถูกต้องไม่เพียงช่วยแก้ปัญหาเฉพาะหน้า แต่ยังสร้างกรอบการพัฒนาที่ช่วยให้นักพัฒนาทำงานร่วมกันได้อย่างมีระบบและลดความซับซ้อนของโค้ด โดยบทนี้จะเป็นพื้นฐานสำคัญสำหรับนักพัฒนาที่ต้องการยกระดับทักษะการออกแบบซอฟต์แวร์ให้เป็นมืออาชีพ

Design Patterns ที่สำคัญใน C#

- Singleton
- Factory
- Strategy
- Observer
- Dependency Injection (DI)

ต่อไปนี้คือ “บทที่ 11: Design Patterns ที่สำคัญใน C#” อธิบายแบบใช้งานจริง พร้อมตัวอย่างโค้ดสั้นกระชับ และชี้ข้อดี/ข้อควรระวังของแต่ละแพทเทิร์น

Singleton

แนวคิด

มี “อินสแตนซ์เดียวทั่วระบบ” และเข้าถึงได้จากทุกที่ เช่น ตัวตั้งค่าคอนฟิก, ตัวจัดการ cache, ตัวล็อก

ใช้เมื่อไหร่

- ทรัพยากรที่ต้องมีเพียงหนึ่ง (global config, logger, telemetry client)
- ต้องการแชร์ state ร่วมกันและควบคุมการสร้างออบเจ็กต์

โค้ดตัวอย่าง (Thread-safe + Lazy)

```
public sealed class AppConfig
{
    private AppConfig() { /* โหลดค่า config แพง ๆ */ }

    private static readonly Lazy<AppConfig> _instance =
        new Lazy<AppConfig>(() => new AppConfig(), isThreadSafe: true);

    public static AppConfig Instance => _instance.Value;

    public string ConnectionString { get; set; } = "Server=.;Database=App;";
}
```

ใช้ยังไง

```
var cs = AppConfig.Instance.ConnectionString;
```

ข้อดี

- ควบคุมจำนวนอินสแตนซ์, ประหยัดทรัพยากร
- Lazy<T> ปลอดภัยกับ multi-thread โดยไม่ต้องเขียน lock เอง

ข้อควรระวัง

- ซ่อน dependency ⟶ ทดสอบยาก (ทำให้เป็น DI-friendly ดีกว่า)
- ระวัง “hidden global state” และลำดับการ init ใน unit tests

Factory (Simple/Factory Method/Abstract Factory)

แนวคิด

แยก “การตัดสินใจสร้างออบเจ็กต์” ออกไปให้คลาสเฉพาะ ช่วยลด new กระจัดกระจายและ if/else ยาว ๆ

ใช้เมื่อไหร่

- สร้าง “ตระกูล” ออบเจ็กต์ตามคีย์/สภาพแวดล้อม (เช่น ช่องทางแจ้งเตือน: Email, SMS, Push)
- ต้องการสลับชนิด runtime โดยไม่แตะโค้ดผู้ใช้ปลายทาง

Simple Factory (เหมาะกับเริ่มต้น)

```
public interface INotifier { void Send(string msg); }
```

```
public class EmailNotifier : INotifier
{
    public void Send(string msg) => Console.WriteLine($"[EMAIL] {msg}");
}

public class SmsNotifier : INotifier
{
    public void Send(string msg) => Console.WriteLine($"[SMS] {msg}");
}

public static class NotifierFactory
{
    public static INotifier Create(string channel) => channel switch
    {
        "email" => new EmailNotifier(),
        "sms"   => new SmsNotifier(),
        _       => throw new NotSupportedException(channel)
    };
}

// ใช้งาน
var notifier = NotifierFactory.Create("email");
notifier.Send("Order created");
```

Factory Method (ขยายผ่าน subclass)

```
public abstract class Report { public abstract string Render(); }
public class PdfReport : Report { public override string Render() => "PDF"; }
public class HtmlReport : Report { public override string Render() => "HTML"; }

public abstract class ReportCreator
{
    // Factory Method
    protected abstract Report CreateReport();
    public string Export() => CreateReport().Render();
}
```

```
public class PdfReportCreator : ReportCreator
{
    protected override Report CreateReport() => new PdfReport();
}
```

Abstract Factory (เป็น “โรงงานของโรงงาน” ผลิตเป็นชุดเข้ากัน)

เหมาะกับผลิต “families” เช่น DarkTheme/LightTheme ที่มี Button/TextField เข้าชุดเดียวกัน

ข้อดี

- ลดการผูกติด (decouple) ผู้ใช้กับคอนกรีตคลาส
- เปิดทางขยายชนิดใหม่ได้ง่าย (Open/Closed)

ข้อควรระวัง

- Simple Factory โตแล้วจะมี switch ยาว ⟶ พิจารณา Dictionary mapping หรือ DI container
- Abstract Factory มากเกินไปจะซับซ้อนโดยไม่จำเป็น

Strategy

แนวคิด

ห่อ “อัลกอริทึมที่สลับได้” ไว้หลัง interface เดียว เช่น คำนวณค่าขนส่ง/ส่วนลด/การเรียงลำดับ แล้วเลือกกลยุทธ์ runtime

ใช้เมื่อไหร่

- มีหลายอัลกอริทึมเทียบเท่ากัน ต่างกันตามกฎธุรกิจ/คอนฟิก
- ต้องการทดสอบแยกแต่ละกลยุทธ์ได้ง่าย

โค้ดตัวอย่าง

```
public interface IDiscountStrategy
{
    decimal Apply(decimal subtotal);
}

public class NoDiscount : IDiscountStrategy
{
    public decimal Apply(decimal subtotal) => subtotal;
}
public class PercentageDiscount : IDiscountStrategy
{
    private readonly decimal _rate; // 0.10 = 10%
```

```
    public PercentageDiscount(decimal rate) => _rate = rate;
    public decimal Apply(decimal subtotal) => subtotal * (1 - _rate);
}

public class CheckoutService
{
    private readonly IDiscountStrategy _strategy;
    public CheckoutService(IDiscountStrategy strategy) => _strategy = strategy;

    public decimal Finalize(decimal subtotal) => _strategy.Apply(subtotal);
}

// ใช้งาน
var svc = new CheckoutService(new PercentageDiscount(0.15m));
Console.WriteLine(svc.Finalize(100m)); // 85.00
```

ข้อดี

- แยกกฎธุรกิจให้ทดสอบได้ง่าย
- เพิ่มกลยุทธ์ใหม่โดยไม่ต้องแก้โค้ดเดิม

ข้อควรระวัง

- กลยุทธ์จำนวนมาก ⟶ ควบคุมการเลือกด้วย DI/คอนฟิกหรือ factory อีกชั้น

Observer

แนวคิด

ผู้สังเกต (Observers) สมัครรับเหตุการณ์จากผู้ถูกสังเกต (Subject) แล้วถูกแจ้งเมื่อมีการเปลี่ยนแปลง

ใช้เมื่อไหร่

- โมเดล-วิว, ระบบแจ้งเตือน, event stream
- ต้อง decouple ผู้ส่งเหตุการณ์กับผู้รับหลายราย

รูปแบบเหตุการณ์แบบ .NET (Event Pattern)

```
public class PriceChangedEventArgs : EventArgs
{
    public required string Symbol { get; init; }
    public required decimal NewPrice { get; init; }
}
```

```
public class StockTicker
{
    public event EventHandler<PriceChangedEventArgs>? PriceChanged;

    public void Update(string symbol, decimal price)
        => PriceChanged?.Invoke(this, new PriceChangedEventArgs { Symbol = symbol,
NewPrice = price });
}

public class AlertService
{
    public void Subscribe(StockTicker ticker)
        => ticker.PriceChanged += OnPriceChanged;

    private void OnPriceChanged(object? sender, PriceChangedEventArgs e)
    {
        if (e.NewPrice > 100) Console.WriteLine($"ALERT {e.Symbol} > 100");
    }
}

// ใช้งาน
var ticker = new StockTicker();
var alert = new AlertService();
alert.Subscribe(ticker);

ticker.Update("MSFT", 99m);
ticker.Update("MSFT", 120m); // จะพิมพ์ ALERT
```

ตัวเลือกใน .NET

- event/EventHandler<T>: เบา เรียบง่าย
- IObservable<T>/IObserver<T>: ดันข้อมูลแบบ reactive เหมาะกับสตรีม/การผสาน Rx

ข้อควรระวัง

- ลืม -= unsubscribe ⟶ memory leak; ใช้ using var sub = observable.Subscribe(...) (Rx) หรือ weak event pattern เมื่อจำเป็น
- จัดการ concurrency ใน handler ให้ดี (อาจต้อง queue/lock หรือใช้ SynchronizationContext)

Dependency Injection (DI)

แนวคิด

กลับด้านการพึ่งพา (Inversion of Control): คลาส “บอกว่าต้องการอะไร” ผ่าน interface/constructor แต่ไม่ “new” เอง ทำให้ทดสอบ/สลับ implementation ได้

ใช้เมื่อไหร่

- โค้ดต้องทดสอบได้ดี (mock ได้)
- แยกเลเยอร์ (Domain, Infra) และลดการผูกติดคอนกรีต
- ต้องการจัดการ lifecycle (Transient/Scoped/Singleton)

โครงร่างคลาส

```
public interface IEmailSender
{
    Task SendAsync(string to, string subject, string body);
}

public class SmtpEmailSender : IEmailSender
{
    public Task SendAsync(string to, string subject, string body)
    {
        Console.WriteLine($"SMTP → {to}: {subject}");
        return Task.CompletedTask;
    }
}

public class OrderService
{
    private readonly IEmailSender _email;
    public OrderService(IEmailSender email) => _email = email;

    public async Task PlaceAsync()
    {
        // ... บันทึกคำสั่งซื้อ
        await _email.SendAsync("user@example.com", "Order Placed", "Thanks!");
    }
```

```
}
```

ลงทะเบียนด้วย Microsoft.Extensions.DependencyInjection

```
using Microsoft.Extensions.DependencyInjection;

var services = new ServiceCollection();

// เลือก lifecycle ให้เหมาะ
services.AddSingleton<IEmailSender, SmtpEmailSender>();
services.AddTransient<OrderService>();

var provider = services.BuildServiceProvider();

// Resolve และใช้งาน
var order = provider.GetRequiredService<OrderService>();
await order.PlaceAsync();
```

Lifetimes ที่พบบ่อย

- Transient: สร้างใหม่ทุกครั้ง เหมาะกับ service ไม่มี state
- Scoped: หนึ่งครั้งต่อ scope (เช่นต่อ HTTP request ใน ASP.NET Core)
- Singleton: อินสแตนซ์เดียวตลอดอายุแอป (ระวัง state ที่แชร์กัน/ไม่ thread-safe)

ข้อดี

- ทดสอบง่าย (mock/fake)
- ลดการ coupling, สนับสนุน Clean Architecture
- จัดการ lifecycle/คอนฟิกได้จากศูนย์กลาง

ข้อควรระวัง

- หลีกเลี่ยง **Service Locator** (anti-pattern) เพราะซ่อน dependency
- อย่าใส่ dependency มากเกินไปใน constructor (code smell) ⟶ แยก service หรือใช้ Facade
- รอบคอบกับ Singleton + dependency ที่เป็น scoped/transient (captured scope)

เคล็ดลับรวม (ใช้งานในโปรเจ็กต์จริง)

- **Singleton**: ถ้าต้องมีก็ทำให้ “pure” และไม่เก็บ mutable state ที่กว้างเกินจำเป็น
- **Factory**: เริ่มที่ Simple Factory ⟶ โตขึ้นค่อยย้ายไป DI container/Abstract Factory
- **Strategy**: แมปคอนฟิก ⟶ กลยุทธ์ด้วย Dictionary/DI (เช่น “discount=vip” ⟶ VipStrategy)

- **Observer**: บันทึก/กำหนดนโยบาย unsubscribe ให้ชัดเจน (โดยเฉพาะ long-lived subjects)
- **DI**: พิมพ์ผ่าน **interface** เสมอ, จง "ประกาศสิ่งที่ต้องการ" ใน constructor, จัด registration/lifetime ให้ถูก

Singleton — อินสแตนซ์เดียวทั่วระบบ (Deep dive)

แนวคิดสั้น ๆ: มีเพียงอินสแตนซ์เดียวของคลาสนั้น ๆ ตลอดอายุแอป และถูกเข้าถึงจากทุกที่

เมื่อใช้

- Shared configuration, cache, telemetry client, logging (แต่ใน .NET มักใช้ DI สำหรับ logging แทน)
- โค้ดที่ต้องการ global access point แต่ต้องควบคุมการสร้าง

รูปแบบการทำ (C#)

1. **Eager initialization (static readonly)** — ง่าย & thread-safe

```
public sealed class EagerSingleton
{
    public static readonly EagerSingleton Instance = new EagerSingleton();
    private EagerSingleton() { }
}
```

2. **Static constructor (guarantees no beforefieldinit issues)**

```
public sealed class StaticCtorSingleton
{
    public static readonly StaticCtorSingleton Instance;
    static StaticCtorSingleton() { Instance = new StaticCtorSingleton(); }
    private StaticCtorSingleton() { }
}
```

3. **Lazy (แนะนำ)** — ปลอดภัยใน multi-thread และอ่านง่าย

```
public sealed class LazySingleton
{
    private static readonly Lazy<LazySingleton> _inst = new Lazy<LazySingleton>(() => new
LazySingleton());
    public static LazySingleton Instance => _inst.Value;
    private LazySingleton() { }
}
```

4. **Double-checked locking (manually)** — เก่า แต่เข้าใจแนวคิด

```
public sealed class DclSingleton
{
    private static volatile DclSingleton? _instance;
    private static readonly object _lock = new();
    private DclSingleton() { }
    public static DclSingleton Instance
    {
        get
        {
            if (_instance == null)
            {
                lock (_lock)
                {
                    if (_instance == null) _instance = new DclSingleton();
                }
            }
            return _instance;
        }
    }
}
```

DI-friendly Singleton (แนะนำสำหรับแอปจริง)

แทนที่จะใช้ static singletons ให้ลงทะเบียน service เป็น singleton ใน DI container:

```
services.AddSingleton<IMyService, MyService>();
```

ปัญหาสำคัญ: อย่าฉีด (inject) Scoped service ลงใน singleton โดยตรง — จะเกิดการจับอายุของ dependency ผิด lifecycle (capturing scoped in singleton)

วิธีแก้: ให้ singleton รับ IServiceScopeFactory แล้วสร้าง scope เมื่อจำเป็น หรือรับ factory/delegate เพื่อสร้าง scoped instance ตามต้องการ:

```
public class SingletonUsingScoped
{
    private readonly IServiceScopeFactory _scopeFactory;
    public SingletonUsingScoped(IServiceScopeFactory scopeFactory) => _scopeFactory =
scopeFactory;
    public void DoWork()
    {
```

```
        using var scope = _scopeFactory.CreateScope();
        var repo = scope.ServiceProvider.GetRequiredService<IMyScopedRepo>();
        repo.Do();
    }
}
```

ข้อควรระวัง / Pitfalls

- **Hidden global state** ⟶ ยากต่อ unit testing
- **Serialization / Reflection / Multiple AppDomains** ⟶ อาจสร้าง instance เพิ่มได้ (rare)
- **IDisposable**: ถ้า singleton ถือ resource ที่ต้อง dispose — ให้ container เป็นผู้จัดการ lifecycle หรือจัดการ disposal ชัดเจน
- **Testing**: static singletons ยากต่อ mocking — prefer DI singletons

---

Factory — แยกการสร้างออกจากการใช้งาน (Deep dive)

แนวคิด: แยกหน้าที่การสร้างออบเจ็กต์ (creation) ออกจากการใช้งาน (consumption) — ลด new กระจัดกระจายและ switch หรือ if ใน business code

ประเภท

- **Simple Factory:** ฟังก์ชัน/คลาสที่คืนค่าวัตถุตามคีย์
- **Factory Method:** ให้ subclass ตัดสินใจว่าจะสร้างคอนกรีตชนิดไหน (ใช้ inheritance)
- **Abstract Factory:** สร้าง family ของออบเจ็กต์ที่เข้ากัน (theme, provider)
- **Generic Factory / Delegate Factory:** ใช้ generic หรือ Func<T> เพื่อยืดหยุ่น

ตัวอย่าง: Simple Factory ⟶ ปรับให้ DI-friendly

Simple switch ⟶ ปรับเป็น Dictionary + DI เพื่อให้ extensible

```
// Interface
public interface IPaymentProcessor { void Pay(decimal amount); }

// Implementations
public class CardProcessor : IPaymentProcessor { public void Pay(decimal amount) =>
Console.WriteLine("Card"); }
public class PaypalProcessor : IPaymentProcessor { public void Pay(decimal amount) =>
Console.WriteLine("PayPal"); }

// Factory that uses DI to resolve concrete processors by key
public interface IPaymentFactory { IPaymentProcessor Create(string key); }
```

```
public class PaymentFactory : IPaymentFactory
{
    private readonly IServiceProvider _sp;
    private readonly IReadOnlyDictionary<string, Type> _map;
    public PaymentFactory(IServiceProvider sp)
    {
        _sp = sp;
        _map = new Dictionary<string, Type>
        {
            ["card"] = typeof(CardProcessor),
            ["paypal"] = typeof(PaypalProcessor)
        };
    }
    public IPaymentProcessor Create(string key)
    {
        if (!_map.TryGetValue(key, out var type)) throw new KeyNotFoundException(key);
        return (IPaymentProcessor)_sp.GetRequiredService(type);
    }
}

// Registration
services.AddTransient<CardProcessor>();
services.AddTransient<PaypalProcessor>();
services.AddSingleton<IPaymentFactory, PaymentFactory>();
```

ข้อดี: การลงทะเบียนชนิดใหม่ไม่ต้องแก้โค้ดผู้ใช้วิธีการ — แค่ลงทะเบียน type และ map key

Generic Factory

```
public interface IFactory<T> { T Create(); }
public class GenericFactory<T> : IFactory<T> where T : new() { public T Create() => new T(); }
```

Factory Method (ตัวอย่าง)

ใช้เมื่อ subclass ต้องควบคุมการสร้าง object

```
public abstract class Document { }
public class PdfDocument : Document { }
public class WordDocument : Document { }
```

```
public abstract class DocumentCreator
{
    protected abstract Document CreateDocument();
    public void Save() { var doc = CreateDocument(); /* save */ }
}
public class PdfCreator : DocumentCreator { protected override Document CreateDocument()
=> new PdfDocument(); }
```

ข้อควรระวัง

- อย่าใส่ business logic เยอะใน factory — ให้ factory ทำหน้าที่สร้างเท่านั้น
- ถ้าต้องการความยืดหยุ่นสูง ให้พิจารณาผสานกับ DI container หรือ Registry pattern

การทดสอบ

- ทดสอบ mapping ของ key ⟶ type
- ทดสอบ behavior ของคอนกรีตคลาสแยกกัน (mock dependencies)

Strategy — เลือกอัลกอริทึมแบบ runtime (Deep dive)

แนวคิด: แลกเปลี่ยนอัลกอริทึมได้โดยผูกพฤติกรรมผ่าน interface หรือ delegate

ใช้เมื่อ

- หลายอัลกอริทึมที่ทำหน้าที่เดียวกัน แต่กฎแตกต่างกัน (discounts, routing, sorting)
- ต้อง swap logic โดยไม่แก้โค้ด caller

รูปแบบการใช้งาน

1. **Interface-based** (class-per-strategy)

```
public interface IDiscountStrategy { decimal Apply(decimal amount); }

public class NoDiscount : IDiscountStrategy { public decimal Apply(decimal amount) => amount;
}
public class TenPercent : IDiscountStrategy { public decimal Apply(decimal amount) => amount
* 0.9m; }

public class Checkout
{
    private readonly IDiscountStrategy _strategy;
    public Checkout(IDiscountStrategy strategy) => _strategy = strategy;
    public decimal FinalPrice(decimal subtotal) => _strategy.Apply(subtotal);
```

```
}
```

2. **Delegate-based** — ไวกว่า/ไม่ต้องสร้างคลาสเยอะ:

```
Func<decimal, decimal> noDisc = x => x;
Func<decimal, decimal> tenPct = x => x * 0.9m;
```

3. **Using DI + Factory for selection by key/config**

```
// register
services.AddTransient<IDiscountStrategy, NoDiscount>("none"); // (pseudo, .NET DI ไม่มี named by default)
// หรือ ลงทะเบียนหลาย implementation และ resolve IEnumerable<IDiscountStrategy>
```

ข้อดี

- เพิ่ม/เปลี่ยนกลยุทธ์โดยไม่กระทบ caller
- แต่ละกลยุทธ์ทดสอบได้ง่าย

Pitfalls

- ถ้ามีกลยุทธ์จำนวนมาก ควรมี mapping/factory เพื่อเลือก
- อย่าให้กลยุทธ์รู้ผลข้อมูล global มากเกินไป — ทำให้ทดสอบยาก

Advanced: Strategy + Policy

ใช้ Strategy ร่วมกับ policy configuration (IOptions) เพื่อสลับกลยุทธ์จาก config ขณะ runtime

---

Observer — event-driven, push-based (Deep dive)

แนวคิด: Subject แจ้ง Observer เมื่อมีเหตุการณ์ — decouples source and listeners

.NET แบบดั้งเดิม: event + EventHandler<T>

```
public sealed class PriceChangedEventArgs : EventArgs { public string Symbol { get; init; } public decimal NewPrice { get; init; } }

public class StockTicker
{
    public event EventHandler<PriceChangedEventArgs>? PriceChanged;
    public void Update(string symbol, decimal price) =>
        PriceChanged?.Invoke(this, new PriceChangedEventArgs { Symbol = symbol, NewPrice = price });
}
```

การเรียกใช้ให้ปลอดภัย (thread-safe invoke):

```
var handler = PriceChanged;
```

```
handler?.Invoke(this, args);
```

(ใน C# 6+ PriceChanged?.Invoke(this, args); ก็เพียงพอ แต่การคัดลอก local ref ป้องกัน race condition ในบางกรณี)

ปัญหา & แนวทางแก้

- **Memory leak:** ถ้าผู้ฟัง (subscriber) มีอายุสั้นกว่า publisher และไม่ unsubscribe ⟶ เกิด leak

  วิธีแก้:
  - subscriber ต้อง -= เมื่อเลิกใช้งาน (Dispose pattern, finalizer, event aggregator)
  - ใช้ **weak event** (เช่น WPF WeakEventManager) หรือ implement weak-reference subscription
- **Concurrency:** Handler อาจรันบน thread ใดก็ได้ ⟶ ให้ handler เป็น thread-safe หรือ marshal กลับสู่ synchronization context (UI thread)
- **Async handlers:** ปกติ event handler ไม่ await; ถ้าต้องการ async ให้ใช้ Func<object, TArgs, Task> หรือ pattern OnSomethingAsync และ await Task.WhenAll(handlers)

ตัวอย่าง async event raising

```
public event Func<object, PriceChangedEventArgs, Task>? PriceChangedAsync;
public async Task UpdateAsync(string symbol, decimal price)
{
    var handlers = PriceChangedAsync;
    if (handlers == null) return;
    var invocationList = handlers.GetInvocationList().Cast<Func<object, PriceChangedEventArgs,
Task>>();
    var tasks = invocationList.Select(h => h(this, new PriceChangedEventArgs { Symbol =
symbol, NewPrice = price }));
    await Task.WhenAll(tasks);
}
```

Reactive Pattern: IObservable<T> / IObserver<T>

- เหมาะกับ data streams, backpressure, composition (LINQ over streams)
- คุณสามารถ implement simple observable:

```
public class SimpleObservable<T> : IObservable<T>
{
    private readonly List<IObserver<T>> _observers = new();
    public IDisposable Subscribe(IObserver<T> observer)
    {
```

```
        _observers.Add(observer);
        return new Unsubscriber<T>(_observers, observer);
    }
    public void Publish(T item) { foreach (var o in _observers) o.OnNext(item); }
}
```

หรือใช้ **Rx.NET** (System.Reactive) สำหรับการทำงานแบบ complex stream (buffering, throttling, windowing)

Event Aggregator / Mediator

เมื่อมีผู้ส่งและผู้รับที่หลากหลาย ใช้ event aggregator เพื่อ decouple:

```
public interface IEventAggregator { void Publish<T>(T ev); IDisposable Subscribe<T>(Action<T> handler); }
```

(Frameworks เช่น MediatR, Prism มี implementations ที่แข็งแกร่ง)

Testing

- ทดสอบว่า publisher เรียก handler เมื่อเกิดเหตุการณ์
- ทดสอบว่า subscriber ถูก unsubscribe เมื่อ dispose

Dependency Injection (DI) — ลึกสุดสำหรับการออกแบบระบบ

แนวคิด: Inversion of Control — ลูกไม่สร้าง dependency เอง แต่รับผ่าน constructor/property/method ⟶ ลด coupling ⟶ เพิ่ม testability

Injection types

- **Constructor injection (แนะนำ)** — ชัดเจน, immutable dependencies
- **Property injection** — สำหรับ optional dependencies; เปิดโอกาสให้ dependency ไม่ถูก inject (less explicit)
- **Method injection** — สำหรับ dependencies ที่ใช้ชั่วครั้งชั่วคราว

.NET Core DI (Microsoft.Extensions.DependencyInjection)

```
var services = new ServiceCollection();
services.AddTransient<IRepository, SqlRepository>();
services.AddScoped<IUnitOfWork, UnitOfWork>();
services.AddSingleton<ICache, MemoryCache>();
var provider = services.BuildServiceProvider();
```

Lifetimes (สำคัญ)

- **Transient** — new ทุก resolve
- **Scoped** — หนึ่งต่อ scope (ASP.NET Core: per HTTP request)

- **Singleton** — หนึ่งอินสแตนซ์ตลอดแอป

สำคัญ: อย่าให้ Singleton ขึ้นอยู่กับ Scoped/Transient ที่ควรมีอายุสั้นกว่า — จะจับ scope ผิด (lifetime capture bug)

การจัดการ Circular Dependencies

ถ้าคอนสตรัคเตอร์ต้องการกันและกัน:

- **Refactor** to remove circular dependency (best)
- Use **Lazy** or **Func** or IServiceProvider to defer resolution

```
public class A { public A(Lazy<B> b) { var actualB = b.Value; } }
```

Registration patterns & advanced

- **Open generics**

```
services.AddTransient(typeof(IRepository<>), typeof(Repository<>));
```

- **Multiple implementations** — inject IEnumerable<IHandler> และเลือก/aggregate
- **Named registrations** — .NET DI ไม่มี built-in; alternatives:
  - register dictionary: IDictionary<string, Func<IServiceProvider, IMyService>>
  - use keyed factory service
  - use third-party container (Autofac) for native named registrations
- **Factory delegates**

```
services.AddTransient<Func<string, INotifier>>(sp => key =>
{
    return key switch {
        "email" => sp.GetRequiredService<EmailNotifier>(),
        ...
    };
});
```

- **IOptions pattern** สำหรับ configuration-bound services

Composition Root (สำคัญสถาปัตยกรรม)

- เพียงที่เดียวในแอปที่รู้วิธีประกอบ object graph (startup/bootstrap)
- ทุก registration เกิดที่นี่ — โค้ดอื่นไม่ควรเรียก new สำหรับ service ที่ควรถูก inject

Service Locator — anti-pattern

การเรียก serviceProvider.GetService<T>() ทั่วโค้ดเป็น service locator — ซ่อน dependency ทำให้ test ยาก

Performance & memory

- DI resolution มีค่าเล็กน้อย — ไม่เป็นปัญหาใน most apps
- แต่ avoid resolving transient heavy objects in hot loops — create once if expensive

- Dispose: container จะ dispose services ที่ implement IDisposable ถ้าคุณใช้ container เพื่อสร้าง them (scoped/singleton)

Unit testing with DI

- สร้าง ServiceCollection ใน test และ override registrations
- หรือ instantiate class โดยส่ง mock dependencies (Moq)

```
// xUnit + Moq example (concept)
var mockEmail = new Mock<IEmailSender>();
var svc = new OrderService(mockEmail.Object);
await svc.PlaceAsync();
mockEmail.Verify(m => m.SendAsync(It.IsAny<string>(), It.IsAny<string>(), It.IsAny<string>()),
Times.Once);
```

Common real-world problems & solutions

- **Too many constructor parameters** ⟶ แยกเป็น cohesive services หรือใช้ Facade
- **Scoped services in background singleton tasks** ⟶ create scope inside background job before using scoped services
- **Lifetime mismatch** ⟶ validate via analyzer/tools or unit test composition root

---

การผสานแพทเทิร์นเข้าด้วยกัน (Integration)

แอปจริงมักใช้หลาย pattern ร่วมกัน:

- **Factory + DI:** Factory ใช้ IServiceProvider เพื่อ resolve implementations (extensible)
- **Strategy + DI:** register strategies and resolve according to config or runtime key
- **Observer (Event Aggregator) + DI:** event aggregator registered as singleton, subscribers registered as transient/scoped and subscribe at construction or via DI hook
- **Singleton + Scoped:** singleton uses IServiceScopeFactory to get scoped services per operation

ตัวอย่างสั้น ๆ: Payment flow

- **Factory** เลือก PaymentProcessor (Card/PayPal)
- **Strategy** คำนวณ fee (percentage/fixed)
- **Observer** แจ้งระบบอื่นว่า payment สำเร็จ (EventAggregator)
- **DI** จัดการการลงทะเบียนและ lifecycle

Testing & Maintainability — เคล็ดลับเชิงปฏิบัติ

- Prefer **constructor injection** ⟶ makes dependencies explicit

- Mock dependencies in unit tests; avoid static singletons in core logic
- Keep factories simple — ถ้ามี logic มาก ให้ย้ายไปที่ service/class แยกต่างหาก
- For events, ensure subscribers unsubscribe — use IDisposable pattern for subscription lifetime
- Validate DI registrations: run integration tests that build ServiceProvider and resolve root services

---

Cheat-sheet / Quick Decisions

- ต้องการ **single global instance** + control lifecycle ⟶ **Singleton via DI** (not static)
- ต้องการ **swap concrete type runtime** (e.g., provider) ⟶ **Factory**
- ต้องการ **swap algorithm/behavior** ⟶ **Strategy** (or Func delegate)
- ต้องการ **publish/subscribe** (decoupled) ⟶ **Observer** / **EventAggregator** / IObservable
- ต้องการ **testable, decoupled wiring** ⟶ **Dependency Injection** (composition root, constructor injection)

Singleton Pattern ใน C#

1. แนวคิด (Concept)

Singleton เป็น Design Pattern ที่ใช้เพื่อให้ **คลาสหนึ่งมีเพียงแค่ Instance เดียวในระบบตลอดการทำงานของโปรแกรม** และให้มี **จุดเข้าถึง (Global Access Point)** เพียงจุดเดียว

- ใช้ในกรณีที่ต้องการการควบคุม **state ส่วนกลาง (global state)**
- เหมาะสำหรับ resource ที่ต้องใช้ร่วมกัน เช่น
  - Logger (ระบบบันทึก log)
  - Configuration Manager (จัดการค่าคอนฟิก)
  - Database Connection Manager
  - Cache

---

2. โครงสร้าง (Structure)

```
 -----------------
|   Singleton     |
|-----------------|
| - instance      |  (static field)
| + Instance      |  (static property)
```

| - constructor() | (private)

- **private constructor** ⟶ ป้องกันไม่ให้สร้าง object โดยตรงผ่าน new
- **static field** ⟶ เก็บ instance เพียงหนึ่งเดียว
- **static property/method** ⟶ ใช้เข้าถึง instance

---

3. ตัวอย่างโค้ด (Basic Implementation)

```
public sealed class Singleton
{
    // เก็บ instance เดียวของ Singleton
    private static Singleton _instance = null;

    // ตัวล็อกสำหรับ Thread-Safety
    private static readonly object _lock = new object();

    // ซ่อน constructor
    private Singleton()
    {
        Console.WriteLine("Singleton Instance Created!");
    }

    // Property สำหรับเข้าถึง Instance
    public static Singleton Instance
    {
        get
        {
            // Double-Check Locking
            if (_instance == null)
            {
                lock (_lock)
                {
                    if (_instance == null)
                    {
                        _instance = new Singleton();
```

```
                    }
                }
            }
            return _instance;
        }
    }

    // ตัวอย่าง method
    public void DoWork(string task)
    {
        Console.WriteLine($"Doing: {task}");
    }
}
```

---

4. วิธีใช้งาน

```
class Program
{
    static void Main()
    {
        var s1 = Singleton.Instance;
        var s2 = Singleton.Instance;

        s1.DoWork("Task A");
        s2.DoWork("Task B");

        Console.WriteLine(Object.ReferenceEquals(s1, s2)); // True
    }
}
```

☐ **ผลลัพธ์:**

```
Singleton Instance Created!
Doing: Task A
Doing: Task B
True
```

จะเห็นว่า instance ถูกสร้างเพียงครั้งเดียว แม้จะเรียกใช้งานหลายครั้ง

---

5. Thread Safety และ Lazy Initialization

(ก) แบบใช้ Lazy

```
public sealed class Singleton
{
    private static readonly Lazy<Singleton> _instance =
        new Lazy<Singleton>(() => new Singleton());

    private Singleton()
    {
        Console.WriteLine("Lazy Singleton Created!");
    }

    public static Singleton Instance => _instance.Value;
}
```

- Lazy<T> ของ .NET จัดการ **thread-safe** ให้เรียบร้อย
- instance จะถูกสร้างก็ต่อเมื่อมีการเรียกใช้งานครั้งแรก (**Lazy Initialization**)

---

6. การประยุกต์ใช้งานจริง

(ก) Singleton Logger

```
public sealed class Logger
{
    private static readonly Lazy<Logger> _instance =
        new Lazy<Logger>(() => new Logger());

    private Logger() { }

    public static Logger Instance => _instance.Value;

    public void Log(string message)
    {
        Console.WriteLine($"[{DateTime.Now}] {message}");
    }
}
```

การใช้งาน:

```
Logger.Instance.Log("Application Started");
Logger.Instance.Log("User logged in");
```

(ข) Singleton Configuration Manager

```
public sealed class ConfigManager
{
    private static readonly Lazy<ConfigManager> _instance =
        new Lazy<ConfigManager>(() => new ConfigManager());

    private Dictionary<string, string> _settings;

    private ConfigManager()
    {
        _settings = new Dictionary<string, string>
        {
            { "AppName", "MyApp" },
            { "Version", "1.0.0" }
        };
    }

    public static ConfigManager Instance => _instance.Value;

    public string Get(string key) => _settings.ContainsKey(key) ? _settings[key] : null;
}
```

การใช้งาน:

```
string appName = ConfigManager.Instance.Get("AppName");
Console.WriteLine(appName); // MyApp
```

---

7. ข้อดี (Pros)

- มี **Instance เดียว** ควบคุม state ส่วนกลางได้ง่าย
- ประหยัด resource (เช่น Database Connection Pool)
- มี **Global Access Point** เข้าถึงสะดวก

8. ข้อเสีย (Cons)

- ถ้าใช้มากเกินไป ⟶ อาจกลายเป็น **Global Variable** ⟶ ยากต่อการทดสอบ (Unit Test)
- สร้าง **Coupling สูง** เพราะทุกส่วนในระบบเรียกใช้งานผ่าน Singleton
- ในระบบที่ **ต้องการ scalability** เช่น Cloud หรือ Microservices ⟶ การมี instance เดียวอาจเป็น bottleneck

---

9. แนวปฏิบัติที่เหมาะสม (Best Practices)

1. ใช้ Lazy<T> แทนการเขียน Double-Check Lock เอง
2. จำกัดการใช้งาน ⟶ เฉพาะ resource ที่ควรมีเพียงหนึ่งเดียวจริง ๆ (Logger, Config, Cache)
3. ถ้าระบบมีความซับซ้อนมาก ⟶ พิจารณาใช้ **Dependency Injection (DI)** แทน Singleton (จะ flexible และ testable กว่า)
4. หาก Singleton ต้องเก็บ state ⟶ ต้องระวังเรื่อง **Thread Safety** เสมอ

☐ สรุป:

Singleton เป็น Pattern ที่ใช้ควบคุมการสร้าง instance เพียงหนึ่งเดียวของคลาส และให้จุดเข้าถึงเพียงจุดเดียว เหมาะกับ resource ที่ใช้ร่วมกัน แต่ถ้าใช้ผิดที่ผิดทางอาจทำให้ระบบ coupling สูงและยากต่อการทดสอบได้

- **ตัวอย่างพื้นฐาน 3 โปรแกรม** (Basic Singleton ใช้ในสถานการณ์ต่าง ๆ)
- **ตัวอย่างแนวประยุกต์ 3 โปรแกรม** (Integration/Real-world Usage)
- ทุกตัวอย่างจะเป็น **โครงสร้างโฟลเดอร์ + ไฟล์เต็ม + คำอธิบายโค้ด + ผลการรัน**

☐ **พาร์ทที่ 1: โปรแกรมพื้นฐาน 3 ตัวอย่าง**

☐ **ตัวอย่างที่ 1 – Singleton แบบ Thread-Safe พื้นฐาน**

☐ **โครงสร้างโปรเจกต์**

```
SingletonExample1/
├── Program.cs
└── Logger.cs
```

☐ **Logger.cs**

```
using System;

namespace SingletonExample1
{
```

```
    // Logger ที่ใช้ Singleton
    public sealed class Logger
    {
        private static readonly Lazy<Logger> _instance = new Lazy<Logger>(() => new Logger());

        public static Logger Instance => _instance.Value;

        private Logger()
        {
            Console.WriteLine("Logger created!");
        }

        public void Log(string message)
        {
            Console.WriteLine($"[LOG] {message}");
        }
    }
}
```

☐ **Program.cs**

```
using System;

namespace SingletonExample1
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Start logging...");

            Logger logger1 = Logger.Instance;
            logger1.Log("Hello from logger1");

            Logger logger2 = Logger.Instance;
            logger2.Log("Hello from logger2");
```

```
            Console.WriteLine(Object.ReferenceEquals(logger1, logger2)
                ? "logger1 and logger2 are the same instance."
                : "Different instances!");
        }
    }
}
```

☐ **คำอธิบายโค้ด**

- ใช้ Lazy<T> เพื่อให้การสร้าง instance ปลอดภัยต่อ multi-thread
- Logger.Instance จะคืนค่า object ตัวเดียวเสมอ
- ตรวจสอบด้วย Object.ReferenceEquals ว่า instance เดียวกันจริง

☐ **ผลการรัน**

```
Start logging...
Logger created!
[LOG] Hello from logger1
[LOG] Hello from logger2
logger1 and logger2 are the same instance.
```

☐ **ตัวอย่างที่ 2 – Global Configuration Singleton**

☐ **โครงสร้าง**

```
SingletonExample2/
├── Program.cs
└── AppConfig.cs
```

☐ **AppConfig.cs**

```
using System;

namespace SingletonExample2
{
    public sealed class AppConfig
    {
        private static readonly AppConfig _instance = new AppConfig();

        public static AppConfig Instance => _instance;
```

```
        public string AppName { get; private set; }
        public string Version { get; private set; }

        private AppConfig()
        {
            AppName = "MyApplication";
            Version = "1.0.0";
        }
    }
}
```

☐ **Program.cs**

```
using System;

namespace SingletonExample2
{
    class Program
    {
        static void Main(string[] args)
        {
            var config = AppConfig.Instance;
            Console.WriteLine($"App: {config.AppName}, Version: {config.Version}");
        }
    }
}
```

☐ **ผลการรัน**

```
App: MyApplication, Version: 1.0.0
```

☐ **ตัวอย่างที่ 3 – Singleton ใน Multithreading**

☐ **โครงสร้าง**

```
SingletonExample3/
├── Program.cs
└── Counter.cs
```

☐ **Counter.cs**

```
using System;
```

```
namespace SingletonExample3
{
    public sealed class Counter
    {
        private static readonly Lazy<Counter> _instance = new Lazy<Counter>(() => new
Counter());
        public static Counter Instance => _instance.Value;

        private int _count = 0;
        private readonly object _lock = new object();

        private Counter() { }

        public void Increment()
        {
            lock (_lock)
            {
                _count++;
            }
        }

        public int GetCount() => _count;
    }
}
```

☐ **Program.cs**

```
using System;
using System.Threading.Tasks;

namespace SingletonExample3
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            Parallel.For(0, 1000, i =>
            {
                Counter.Instance.Increment();
            });

            Console.WriteLine($"Final Count: {Counter.Instance.GetCount()}");
        }
    }
}
```

☐ **ผลการรัน**

Final Count: 1000

(แสดงว่าทำงานปลอดภัยใน multi-thread)

☐ พาร์ทที่ 2: โปรแกรมแนวประยุกต์ 3 ตัวอย่าง

☐ ตัวอย่างแนวประยุกต์ที่ 1 – Database Connection Manager

☐ **โครงสร้าง**

```
SingletonApp1/
├── Program.cs
└── DatabaseConnection.cs
```

☐ **DatabaseConnection.cs**

```
using System;

namespace SingletonApp1
{
    public sealed class DatabaseConnection
    {
        private static readonly Lazy<DatabaseConnection> _instance =
            new Lazy<DatabaseConnection>(() => new DatabaseConnection());

        public static DatabaseConnection Instance => _instance.Value;

        private DatabaseConnection()
```

```
        {
            Console.WriteLine("Database connection established.");
        }

        public void Query(string sql)
        {
            Console.WriteLine($"Executing SQL: {sql}");
        }
    }
}
```

☐ **Program.cs**

```
using System;

namespace SingletonApp1
{
    class Program
    {
        static void Main(string[] args)
        {
            var db1 = DatabaseConnection.Instance;
            db1.Query("SELECT * FROM Users");

            var db2 = DatabaseConnection.Instance;
            db2.Query("INSERT INTO Users VALUES ('Alice')");

            Console.WriteLine(Object.ReferenceEquals(db1, db2)
                ? "Single DB instance used!"
                : "Multiple DB instances detected!");
        }
    }
}
```

☐ **ผลการรัน**

```
Database connection established.
Executing SQL: SELECT * FROM Users
```

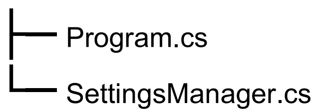
Executing SQL: INSERT INTO Users VALUES ('Alice')

Single DB instance used!

☐ **ตัวอย่างแนวประยุกต์ที่ 2 – App Settings Loader (จำลองอ่านจากไฟล์)**

☐ **โครงสร้าง**

SingletonApp2/



☐ **SettingsManager.cs**

```
using System;

namespace SingletonApp2
{
    public sealed class SettingsManager
    {
        private static readonly Lazy<SettingsManager> _instance =
            new Lazy<SettingsManager>(() => new SettingsManager());

        public static SettingsManager Instance => _instance.Value;

        public string Language { get; private set; }
        public string Theme { get; private set; }

        private SettingsManager()
        {
            Console.WriteLine("Loading settings from config file...");
            Language = "English";
            Theme = "Dark";
        }
    }
}
```

☐ **Program.cs**

```
using System;
```

```
namespace SingletonApp2
{
    class Program
    {
        static void Main(string[] args)
        {
            var settings = SettingsManager.Instance;
            Console.WriteLine($"Language: {settings.Language}, Theme: {settings.Theme}");

            var settings2 = SettingsManager.Instance;
            Console.WriteLine($"Language: {settings2.Language}, Theme: {settings2.Theme}");
        }
    }
}
```

☐ **ผลการรัน**

```
Loading settings from config file...
Language: English, Theme: Dark
Language: English, Theme: Dark
```

☐ **ตัวอย่างแนวประยุกต์ที่ 3 – Logger ที่เก็บ Log ทั้งหมดใน Memory**

☐ **โครงสร้าง**

```
SingletonApp3/
├── Program.cs
└── MemoryLogger.cs
```

☐ **MemoryLogger.cs**

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

namespace SingletonApp3
{
    public sealed class MemoryLogger
    {
        private static readonly Lazy<MemoryLogger> _instance =
            new Lazy<MemoryLogger>(() => new MemoryLogger());
```

```
        public static MemoryLogger Instance => _instance.Value;

        private readonly List<string> _logs = new List<string>();

        private MemoryLogger() { }

        public void Log(string message)
        {
            _logs.Add($"{DateTime.Now}: {message}");
        }

        public void ShowLogs()
        {
            Console.WriteLine("==== Logs ====");
            foreach (var log in _logs)
            {
                Console.WriteLine(log);
            }
        }
    }
}
```

☐ **Program.cs**

```
using System;

namespace SingletonApp3
{
    class Program
    {
        static void Main(string[] args)
        {
            var logger = MemoryLogger.Instance;

            logger.Log("App started");
```