F# Programming: Advance (Integrative-Generative AI Edition)

CONTENTS:

by Student Price Book Center

คำนำ

F# เป็นภาษาที่ถูกออกแบบภายใต้แนวคิด **Functional-first** บนแพลตฟอร์ม .NET ซึ่งมุ่งเน้นการเขียนโปรแกรมแบบฟังก์ชันเชิงบริสุทธิ์ แต่ก็ยังรองรับรูปแบบ **Object-Oriented Programming (OOP)** และ **Imperative Programming** ได้อย่างเต็มรูปแบบ ความโดดเด่นของ F# คือ การรวมความสามารถในการจัดการ **ข้อมูล, ความซับซ้อนของโปรแกรม, และการประมวลผลแบบ concurrent** เข้าไว้ด้วยกันอย่างชาญฉลาด ทำให้เหมาะกับทั้งงานด้านธุรกิจ, วิทยาศาสตร์, และระบบซอฟต์แวร์ขนาดใหญ่

หนังสือเล่มนี้มีจุดมุ่งหมายเพื่อเป็นคู่มือสำหรับผู้ที่มีพื้นฐาน F# ในระดับกลางแล้ว และต้องการยกระดับความเข้าใจไปสู่ **แนวคิดและเทคนิคขั้นสูง** ที่สามารถนำไปใช้ในงานจริงได้อย่างมีประสิทธิภาพ เนื้อหาครอบคลุมทั้ง **Advanced Functional Programming**, **Type Providers**, **Computation Expressions**, **Parallel & Reactive Programming**, และ **Advanced OOP + Interop** โดยแต่ละบทได้ถูกออกแบบให้ผสมผสาน **ทฤษฎี** และ **ตัวอย่างโค้ดเชิงปฏิบัติ** เพื่อให้ผู้อ่านสามารถเรียนรู้จากกรณีศึกษาที่สมจริง

ใน **บทที่ 13 Advanced Functional Programming** ผู้อ่านจะได้เรียนรู้เรื่อง **Currying, Partial Application, Function Composition และ Continuation Passing Style (CPS)** พร้อมตัวอย่างบูรณาการที่สอดแทรกการใช้งานฟังก์ชันขั้นสูงใน F# ซึ่งช่วยให้เข้าใจการสร้างโปรแกรมที่ **ยืดหยุ่น, reusable และ maintainable** ได้ดียิ่งขึ้น

ต่อมาที่ **บทที่ 14 Type Providers** จะพาผู้อ่านเจาะลึก **การใช้ FSharp.Data** สำหรับการเข้าถึงข้อมูลจาก **JSON, CSV, XML, และ SQL Database** อย่าง type-safe โดยไม่ต้องเขียน parser ด้วยตนเอง นอกจากนี้ยังมีตัวอย่างโครงการบูรณาการตั้งแต่บท 7–13 ซึ่งแสดงให้เห็นถึงการประยุกต์ Type Providers ร่วมกับข้อมูลจริงและโมเดลแบบ functional

บทที่ 15 Computation Expressions จะเน้นเทคนิคการเขียนโปรแกรมแบบ **monadic style** ผ่าน computation expressions ทั้งแบบ built-in เช่น **Async และ Seq** และการสร้าง **Custom Computation Expressions** เพื่อจัดการ control flow และ side-effects ได้อย่างชาญฉลาด เทคนิคเหล่านี้มีความสำคัญอย่างยิ่งในงานที่ต้องจัดการกับ **I/O, asynchronous programming, และ sequence processing**

ใน **บทที่ 16 Parallel & Reactive Programming** ผู้อ่านจะได้ทำความรู้จักกับแนวคิด **MailboxProcessor (Actor Model), Async.Parallel, Parallel LINQ และ Reactive Extensions (Rx)** เพื่อสร้างโปรแกรมที่สามารถ **parallel execution และ responsive** ได้อย่างมีประสิทธิภาพ โดยใช้ syntax ของ F# ที่ concise และ type-safe

สุดท้าย **บทที่ 17 Advanced OOP + Interop** จะสอนการผสมผสาน **Generic Classes/Methods, Reflection, และการใช้งานร่วมกับ C# Libraries** ทำให้สามารถพัฒนาแอปพลิเคชัน .NET ที่ซับซ้อนและสามารถทำงานร่วมกับ ecosystem ของ C# ได้อย่างราบรื่น

ตลอดเล่ม ผู้อ่านจะพบว่า **F# สามารถใช้ functional-first paradigm ร่วมกับเทคนิคเชิงวัตถุและ imperative ได้อย่างกลมกลืน** หนังสือเล่มนี้จึงเหมาะสำหรับนักพัฒนาที่ต้องการยกระดับความเชี่ยวชาญ, นักวิจัยด้านซอฟต์แวร์, และผู้ที่สนใจ **การสร้างระบบที่มีประสิทธิภาพสูง, ปลอดภัยจาก runtime errors, และ maintainable**

ด้วยโครงสร้างบทที่เป็นขั้นตอนและตัวอย่างโค้ดที่ครบถ้วน หนังสือเล่มนี้จะช่วยให้ผู้อ่าน **เข้าใจลึกซึ้งทั้งแนวคิดและการใช้งานจริงของ F#** พร้อมนำไปสร้างโครงการซอฟต์แวร์ขั้นสูงได้อย่างมั่นใจ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 13

Advanced Functional Programming

(Advanced Functional Programming)

เนื้อหา

- Advanced Functional Programming
- เจาะลึก Advanced Functional Programming ใน F#
- เจาะลึก Currying ใน F#
- Partial Application ใน F#
- Function Composition ใน F#
- Continuation Passing Style (CPS) ใน F#
- ตัวอย่างบูรณาการ

บทนำ

บทที่ 13 **Advanced Functional Programming** มุ่งเน้นการยกระดับทักษะการเขียนโปรแกรมเชิงฟังก์ชันใน F# ไปสู่ระดับสูงขึ้น ด้วยเทคนิคและแนวคิดที่ช่วยให้โค้ดมีความยืดหยุ่น กระชับ และสามารถจัดการความซับซ้อนของโปรแกรมได้อย่างมีประสิทธิภาพ

หนึ่งในหัวข้อสำคัญคือ **Currying** ซึ่งเป็นการแปลงฟังก์ชันที่มีหลายอาร์กิวเมนต์ให้เป็นลำดับของฟังก์ชันที่รับอาร์กิวเมนต์ทีละตัว เทคนิคนี้ช่วยให้สามารถสร้างฟังก์ชันใหม่จากฟังก์ชันเดิมได้ง่าย และทำให้โค้ดมีความโมดูลาร์สูง

ต่อมาคือ **Partial Application** ซึ่งเป็นการนำฟังก์ชันที่มีหลายอาร์กิวเมนต์มาสร้างฟังก์ชันใหม่โดยกำหนดค่าบางอาร์กิวเมนต์ไว้ล่วงหน้า แนวคิดนี้ช่วยลดความซ้ำซ้อนของโค้ดและเพิ่มความยืดหยุ่นในการนำฟังก์ชันไปใช้ในหลายบริบท

หัวข้อ **Function Composition** เป็นการรวมฟังก์ชันหลายตัวเข้าด้วยกันเพื่อสร้างฟังก์ชันใหม่ เทคนิคเชิงลึกนี้ทำให้โค้ดมีลำดับการทำงานที่ชัดเจนและสามารถอ่านได้ง่ายขึ้น ส่งผลให้โปรแกรมมีความ declarative และเข้าใจง่ายตามหลักการ Functional Programming

นอกจากนี้ **Continuation Passing Style (CPS)** เป็นเทคนิคที่ช่วยจัดการลำดับการประมวลผลของโปรแกรมโดยการส่ง continuation หรือฟังก์ชันต่อไปที่ควรทำงานหลังจากฟังก์ชันปัจจุบัน เทคนิคนี้มีประโยชน์อย่างยิ่งในการควบคุม flow ของโปรแกรมที่ซับซ้อนและสามารถประยุกต์ใช้กับงาน asynchronous ได้

บทนี้ยังให้ตัวอย่างการใช้งานจริงเพื่อให้ผู้อ่านเข้าใจการประยุกต์ใช้ Currying, Partial Application, Function Composition และ CPS ในโปรเจกต์ F# ที่ซับซ้อน ช่วยให้สามารถนำแนวคิดไปใช้ในงานจริงได้อย่างมีประสิทธิภาพ

เป้าหมายของบทนี้คือให้ผู้อ่านสามารถเขียนฟังก์ชันเชิงฟังก์ชันขั้นสูงได้อย่างมั่นใจ เข้าใจแนวคิดเชิงโมดูลาร์ และสามารถจัดการความซับซ้อนของโปรแกรมได้อย่างเป็นระบบ
ด้วยการเรียนรู้หัวข้อเหล่านี้ ผู้พัฒนาจะมีเครื่องมือและแนวคิดที่ช่วยให้สามารถออกแบบโปรแกรม F# ที่มีความยืดหยุ่น กระชับ และรองรับการขยายงานในอนาคตได้อย่างมืออาชีพ

Advanced Functional Programming

- Currying
- Partial Application
- Function Composition ลึก
- Continuation Passing Style (CPS)

บทที่ 13: Advanced Functional Programming ใน F# โดยละเอียด พร้อมตัวอย่างแนวคิดเชิงลึก

☐ 1. Currying

Concept:

- ใน F# ทุกฟังก์ชันที่มีหลายพารามิเตอร์จริง ๆ แล้วเป็น **ฟังก์ชันที่คืนฟังก์ชัน** (curried function)
- ตัวอย่าง: ฟังก์ชัน (x:int) (y:int) -> x + y จริง ๆ คือ x -> (y -> x + y)

```
let add x y = x + y
let addCurried = fun x -> fun y -> x + y

let sum1 = add 3 5          // ใช้งานปกติ
let sum2 = addCurried 3 5   // ใช้งานเหมือนกัน
```

รายละเอียดเชิงลึก

- Currying ช่วยให้เราสามารถ **สร้างฟังก์ชันเฉพาะบางพารามิเตอร์** ได้ง่าย
- ฟังก์ชันหลายพารามิเตอร์ใน F# เป็น **curried โดย default**

☐ 2. Partial Application

Concept:

- การ **ให้พารามิเตอร์บางตัวก่อน** เพื่อสร้างฟังก์ชันใหม่
- เกิดขึ้นได้จากการใช้ **currying**

```
let multiply x y = x * y
let double = multiply 2   // Partial Application
let triple = multiply 3

let result1 = double 5     // 10
let result2 = triple 5     // 15
```

รายละเอียดเชิงลึก

- Partial Application เป็นวิธีสร้าง **higher-order function**
- ใช้ร่วมกับ **pipelines และ composition** ได้สะดวก

☐ 3. Function Composition ลึก

Concept:

- การเชื่อมต่อหลายฟังก์ชันเข้าด้วยกัน
- >> คือ **forward composition**: f >> g = g(f(x))
- << คือ **backward composition**: f << g = f(g(x))

```
let add2 x = x + 2
let square x = x * x

let forward = add2 >> square
let backward = square << add2

let fResult = forward 3    // (3+2)^2 = 25
let bResult = backward 3   // (3+2)^2 = 25
```

รายละเอียดเชิงลึก

- Function composition ทำให้โค้ด **อ่านง่าย, concise, และ immutable**
- สามารถต่อเนื่องได้หลายชั้น เช่น f >> g >> h >> i
- ใช้ใน **data pipelines, async pipelines, หรือ functional transformations**

☐ 4. Continuation Passing Style (CPS)

Concept:

- เทคนิคเขียนฟังก์ชันโดยส่ง **"ต่อ" (continuation function)** แทนการคืนค่า
- ฟังก์ชัน CPS ไม่คืนค่าโดยตรง แต่เรียก continuation เพื่อจัดการผลลัพธ์

```
let addCPS x y cont =
    cont (x + y)
```

```
let printResult result =
    printfn "Result: %d" result

addCPS 3 5 printResult  // Result: 8
```

รายละเอียดเชิงลึก

- CPS ช่วย **ควบคุม flow ของโปรแกรม**
- ใช้ทำ **asynchronous, backtracking, หรือ exception handling แบบ functional**
- สามารถเปลี่ยนฟังก์ชันแบบ synchronous เป็น async ได้ง่าย

```
let divideCPS x y cont errCont =
    if y = 0 then errCont "Division by zero"
    else cont (x / y)

divideCPS 10 2 (fun res -> printfn "OK: %d" res) (fun e -> printfn "Error: %s" e)
divideCPS 10 0 (fun res -> printfn "OK: %d" res) (fun e -> printfn "Error: %s" e)
```

ผลการรัน

```
OK: 5
Error: Division by zero
```

ข้อดีของ CPS

- ทำให้สามารถ **รวม error handling กับ flow control** ได้อย่างยืดหยุ่น
- เป็นพื้นฐานของ **monads, async workflows, และ functional pipelines**

เจาะลึก Advanced Functional Programming ใน F#

---

1 Currying (เจาะลึก)

แนวคิดเชิงลึก

- ทุกฟังก์ชันที่มีหลายพารามิเตอร์ใน F# เป็น **curried function โดย default**
- ฟังก์ชัน let add x y = x + y จริง ๆ แล้วเป็น let add = fun x -> (fun y -> x + y)
- ช่วยให้สามารถ **สร้างฟังก์ชันย่อยโดยให้ค่า parameter บางตัวก่อน** (partial application)

ตัวอย่างเชิงลึก

```
let add x y z = x + y + z

let curriedAdd = add 1       // ผลลัพธ์: fun y z -> 1 + y + z
```

```
let addWith1And2 = curriedAdd 2  // ผลลัพธ์: fun z -> 1 + 2 + z
let total = addWith1And2 3       // ผลลัพธ์: 6
```

ประโยชน์

- ใช้งานร่วมกับ **pipelines** ได้สะดวก
- ทำให้ฟังก์ชัน **reusable & composable**

---

2 Partial Application (เจาะลึก)

แนวคิดเชิงลึก

- การให้ค่าเพียงบาง parameter ของฟังก์ชัน curried เพื่อสร้าง **ฟังก์ชันใหม่**
- ใช้ประหยัดการเขียนซ้ำ และทำให้ code concise

```
let multiply x y = x * y
let double = multiply 2       // Partial application
let triple = multiply 3

let results = [5;10;15] |> List.map double  // [10;20;30]
```

ประโยชน์เชิงลึก

- Partial application + pipelining ⟶ ทำให้โค้ด **อ่านง่าย & maintainable**
- สามารถสร้าง **function factories** หรือ **custom operators** ได้

---

3 Function Composition (เจาะลึก)

แนวคิดเชิงลึก

- การต่อฟังก์ชันเข้าด้วยกัน
- Forward composition (>>) = f >> g ≡ g(f(x))
- Backward composition (<<) = f << g ≡ f(g(x))
- สามารถต่อเนื่องหลายฟังก์ชันได้แบบ **pipeline ที่ immutable**

ตัวอย่างเชิงลึก

```
let add2 x = x + 2
let square x = x * x
let negate x = -x

let complexPipeline = add2 >> square >> negate

let result = complexPipeline 3   // ((3+2)^2) * -1 = -25
```

ประโยชน์เชิงลึก

- Function composition ช่วยสร้าง **domain-specific pipelines**
- ใช้ร่วมกับ **async workflows, error handling, หรือ collection transformations**

---

4 Continuation Passing Style (CPS) (เจาะลึก)

แนวคิดเชิงลึก

- ฟังก์ชัน CPS **ไม่คืนค่าโดยตรง** แต่ส่งผลลัพธ์ให้ฟังก์ชันอื่น (continuation)
- รูปแบบทั่วไป: f arg cont = cont (process arg)
- ใช้ใน **asynchronous programming, backtracking, error handling, functional pipelines**

ตัวอย่างพื้นฐาน

```
let addCPS x y cont = cont (x + y)

addCPS 5 10 (fun result -> printfn "Result: %d" result)
```

ตัวอย่าง Error Handling ด้วย CPS

```
let divideCPS x y success failure =
    if y = 0 then failure "Division by zero"
    else success (x / y)

divideCPS 10 2
    (fun res -> printfn "Success: %d" res)
    (fun err -> printfn "Error: %s" err)

divideCPS 10 0
    (fun res -> printfn "Success: %d" res)
    (fun err -> printfn "Error: %s" err)
```

ผลลัพธ์

```
Success: 5
Error: Division by zero
```

ประโยชน์เชิงลึก

- CPS ช่วยให้ **แยก flow ของโปรแกรมกับผลลัพธ์ออกจากกัน**
- พื้นฐานของ **monads, async workflows, railway-oriented programming**
- ทำให้เราสามารถ **ควบคุม error propagation และ side effects** แบบ functional

☐ **สรุปเชิงลึก**

Concept	ประโยชน์เชิงลึก	การใช้งานหลัก
Currying	ฟังก์ชันหลาย parameter ⟶ ฟังก์ชันซ้อนกัน	สร้าง partial function, pipelines
Partial Application	ให้ค่า parameter บางตัว ⟶ สร้างฟังก์ชันใหม่	Function factories, reusable code
Function Composition	ต่อฟังก์ชันเข้าด้วยกัน	Pipelines, data transformations
CPS	ส่งต่อผลลัพธ์ให้ continuation	Async programming, ROP, error handling

เจาะลึก Currying ใน F#

☐ **แนวคิดของ Currying**

- **Currying** คือการแปลงฟังก์ชันหลายพารามิเตอร์ให้เป็น **ฟังก์ชันซ้อนกัน (nested functions)**
- ฟังก์ชันใน F# ที่มีหลายพารามิเตอร์ **เป็น curried โดย default**

```
let add x y = x + y
```

- จริง ๆ แล้วเทียบเท่ากับ:

```
let addCurried = fun x -> (fun y -> x + y)
```

- การเรียกใช้:

```
let result = add 3 5        // 8
let partial = add 3         // partial function: fun y -> 3 + y
let result2 = partial 10    // 13
```

☐ **ประโยชน์ของ Currying**

1. **สร้าง Partial Functions** ได้ง่าย
2. let multiply x y = x * y
3. let double = multiply 2
4. double 5 // 10
5. **ทำให้ฟังก์ชัน reusable**
    - ใช้ฟังก์ชันเดียว สร้างฟังก์ชันใหม่ด้วย parameter ต่างกัน

6. **ทำงานร่วมกับ Pipeline / Composition**
7. let add2 x = x + 2
8. let square x = x * x
9.
10. let pipeline = add2 >> square
11. pipeline 3 // 25

☐ **ตัวอย่างเชิงลึก: Currying กับหลายพารามิเตอร์**

```
let calculate x y z = x + y * z

let curried = calculate 2     // fun y z -> 2 + y * z
let step2 = curried 3          // fun z -> 2 + 3 * z
let result = step2 4           // 14
```

- **ขั้นตอน**:
    1. curried ให้ค่า x=2
    2. step2 ให้ค่า y=3
    3. result ให้ค่า z=4
- แสดงให้เห็น **การสร้างฟังก์ชันย่อยจากค่า parameter ที่เลือกก่อน**

☐ **ตัวอย่างใช้งานจริง: Currying + List.map**

```
let multiply x y = x * y
let numbers = [1;2;3;4;5]

let double = multiply 2
let doubledNumbers = numbers |> List.map double
printfn "%A" doubledNumbers  // [2;4;6;8;10]
```

- เราสามารถสร้าง **partial function double** จากฟังก์ชัน curried multiply
- ใช้กับ List.map ทำให้โค้ด **สั้นและอ่านง่าย**

☐ **สรุป Currying**

- ฟังก์ชันหลายพารามิเตอร์ใน F# ⟶ **เป็น curried function โดย default**
- ใช้สร้าง **partial application** ได้ง่าย
- ช่วยให้ **code reusable, concise, และเข้ากับ pipelines / composition ได้ดี**

ตัวอย่าง **Currying ใน F#** แบบเต็ม workflow ทั้ง **3 โปรแกรมพื้นฐาน + 3 โปรแกรมแนวประยุกต์** พร้อม **โครงสร้าง, คำอธิบายโค้ด, และผลการรัน**

□ โครงสร้าง Project แนะนำ

```
FSharpCurryingExamples/
│
├── Program1_Basic.fs        # ตัวอย่างพื้นฐาน 1
├── Program2_List.fs         # ตัวอย่างพื้นฐาน 2
├── Program3_Pipeline.fs     # ตัวอย่างพื้นฐาน 3
├── Program4_Calc.fs         # ตัวอย่างประยุกต์ 1
├── Program5_Async.fs        # ตัวอย่างประยุกต์ 2
└── Program6_ERP.fs          # ตัวอย่างประยุกต์ 3
```

==========

□ ตัวอย่างพื้นฐาน 3 โปรแกรม

1. Program1_Basic.fs – Currying พื้นฐาน

```
// Program1_Basic.fs
open System

// ฟังก์ชัน curried
let add x y = x + y

[<EntryPoint>]
let main argv =
    let sum = add 3 5
    printfn "3 + 5 = %d" sum

    let add3 = add 3        // partial function
    printfn "3 + 10 = %d" (add3 10)
    0
```

คำอธิบายโค้ด

- add เป็น curried function
- สร้าง partial function add3
- แสดงการคำนวณด้วยการให้ parameter แยกกัน

ผลการรัน

3 + 5 = 8

3 + 10 = 13

---

2. Program2_List.fs – Currying กับ List.map

```
// Program2_List.fs
open System

let multiply x y = x * y
let numbers = [1;2;3;4;5]

[<EntryPoint>]
let main argv =
    let double = multiply 2       // partial application
    let doubledNumbers = numbers |> List.map double
    printfn "Original: %A" numbers
    printfn "Doubled: %A" doubledNumbers
    0
```

คำอธิบายโค้ด

- multiply เป็น curried function
- ใช้ partial application double
- ใช้ List.map กับฟังก์ชัน partial

ผลการรัน

Original: [1; 2; 3; 4; 5]

Doubled: [2; 4; 6; 8; 10]

---

3. Program3_Pipeline.fs – Currying กับ Pipeline

```
// Program3_Pipeline.fs
open System

let add x y = x + y
let square x = x * x

[<EntryPoint>]
```

```
let main argv =
    let add3 = add 3
    let numbers = [1;2;3;4]
    let results = numbers |> List.map add3 |> List.map square
    printfn "Results: %A" results
    0
```

คำอธิบายโค้ด

- ใช้ currying สร้าง add3
- ใช้ pipeline |> เพื่อ map ฟังก์ชันหลายขั้น
- อ่านง่ายและ functional

ผลการรัน

Results: [16; 25; 36; 49]

==========

☐ ตัวอย่างประยุกต์ 3 โปรแกรม

4. Program4_Calc.fs – Currying + Partial Application + Calculator

```
// Program4_Calc.fs
open System

let calc x y op =
    match op with
    | "add" -> x + y
    | "sub" -> x - y
    | "mul" -> x * y
    | "div" -> x / y
    | _ -> 0

[<EntryPoint>]
let main argv =
    let addOp = calc >> (fun f -> f 10)  // partial + curry
    printfn "10 + 5 = %d" (calc 10 5 "add")
    printfn "10 * 3 = %d" (calc 10 3 "mul")
    0
```

คำอธิบายโค้ด

- calc เป็น curried function
- ใช้ partial + composition
- ทำ calculator แบบ dynamic

ผลการรัน

10 + 5 = 15

10 * 3 = 30

---

5. Program5_Async.fs – Currying + Async

```
// Program5_Async.fs
open System
open System.Threading.Tasks

let asyncAdd x y = async { return x + y }

[<EntryPoint>]
let main argv =
    let add3Async = asyncAdd 3  // partial
    let task = add3Async 7 |> Async.RunSynchronously
    printfn "3 + 7 = %d" task
    0
```

คำอธิบายโค้ด

- Currying + partial + Async workflow
- add3Async เป็นฟังก์ชัน asynchronous
- เรียก Async.RunSynchronously เพื่อรอผล

ผลการรัน

3 + 7 = 10

---

6. Program6_ERP.fs – Currying + Functional Pipeline + Error Handling

```
// Program6_ERP.fs
open System

let safeDivide x y =
    if y = 0 then None else Some(x / y)
```

```
let divideCPS x y cont =
    match safeDivide x y with
    | Some r -> cont r
    | None -> printfn "Division by zero"

[<EntryPoint>]
let main argv =
    let div10 = divideCPS 10
    div10 2 (fun r -> printfn "10 / 2 = %d" r)
    div10 0 (fun r -> printfn "10 / 0 = %d" r)
    0
```

คำอธิบายโค้ด

- ใช้ currying + CPS + option type
- ฟังก์ชันย่อย div10 ถูกสร้างจาก partial application
- การจัดการ division by zero เป็น functional

ผลการรัน

```
10 / 2 = 5
Division by zero
```

Partial Application ใน F#

☐ **แนวคิดของ Partial Application**

- **Partial Application** คือการ **ให้ค่า parameter เพียงบางตัวของฟังก์ชัน curried** เพื่อสร้างฟังก์ชันใหม่
- เกิดขึ้นได้จาก **currying** โดย default ของ F#

```
let multiply x y = x * y
```

- สร้าง partial function:

```
let double = multiply 2
```

- การใช้งาน:

```
double 5  // 10
```

- ฟังก์ชัน double คือ fun y -> 2 * y

☐ **ประโยชน์ของ Partial Application**

1. **สร้างฟังก์ชันใหม่จากฟังก์ชันเดิม** โดยไม่ต้องเขียนซ้ำ

2. **ทำงานร่วมกับ pipelines และ higher-order functions**
3. **ทำให้โค้ด concise และ reusable**

☐ **ตัวอย่างพื้นฐาน**

```
let add x y = x + y
let add5 = add 5     // partial function

printfn "5 + 10 = %d" (add5 10)
```

ผลลัพธ์

```
5 + 10 = 15
```

- เราไม่ต้องเขียนฟังก์ชัน fun y -> 5 + y เอง

☐ **ตัวอย่างเชิงลึก: Partial Application กับ List.map**

```
let multiply x y = x * y
let numbers = [1;2;3;4;5]

let triple = multiply 3
let tripledNumbers = numbers |> List.map triple

printfn "Tripled: %A" tripledNumbers
```

ผลลัพธ์

```
Tripled: [3;6;9;12;15]
```

- ใช้ partial function triple กับ List.map เพื่อแปลงทุก element

---

☐ **ตัวอย่างใช้งานจริง: Calculator**

```
let calc x y op =
    match op with
    | "add" -> x + y
    | "sub" -> x - y
    | "mul" -> x * y
    | "div" -> if y = 0 then 0 else x / y
    | _ -> 0

let addOp = calc >> (fun f -> f 10)
```

```
let mulOp = calc >> (fun f -> f 5)


printfn "10 + 5 = %d" (calc 10 5 "add")
printfn "5 * 3 = %d" (calc 5 3 "mul")
```

- Partial application ใช้ร่วมกับ **composition**
- สร้าง **ฟังก์ชันย่อยสำหรับ operation เฉพาะ**

☐ **สรุป Partial Application**

Feature	ประโยชน์
ให้ค่า parameter บางตัว	สร้างฟังก์ชันใหม่โดยไม่ต้องเขียนซ้ำ
ใช้กับ curried function	เพิ่มความ reusable
ใช้กับ pipelines / List.map	ทำ code concise และ readable
ใช้ร่วมกับ composition	สร้างฟังก์ชัน domain-specific

ตัวอย่าง **Partial Application ใน F#** แบบเต็ม workflow ทั้ง **3 โปรแกรมพื้นฐาน + 3 โปรแกรมแนวประยุกต์** พร้อม **โครงสร้าง, คำอธิบายโค้ด, และผลการรัน**

☐ **โครงสร้าง Project แนะนำ**

```
FSharpPartialApplication/
│
├── Program1_Basic.fs        # ตัวอย่างพื้นฐาน 1
├── Program2_List.fs         # ตัวอย่างพื้นฐาน 2
├── Program3_Pipeline.fs     # ตัวอย่างพื้นฐาน 3
├── Program4_Calc.fs         # ตัวอย่างประยุกต์ 1
├── Program5_Async.fs        # ตัวอย่างประยุกต์ 2
└── Program6_ERP.fs          # ตัวอย่างประยุกต์ 3
```

==========

☐ **ตัวอย่างพื้นฐาน 3 โปรแกรม**

1. Program1_Basic.fs – Partial Application พื้นฐาน

```
// Program1_Basic.fs
open System
```

```
let add x y = x + y

[<EntryPoint>]
let main argv =
    let add5 = add 5        // partial function
    printfn "5 + 10 = %d" (add5 10)
    printfn "5 + 20 = %d" (add5 20)
    0
```

คำอธิบายโค้ด

- add เป็น curried function
- สร้าง partial function add5
- ใช้ partial function คำนวณค่าโดยไม่ต้องเขียนฟังก์ชันซ้ำ

ผลการรัน

```
5 + 10 = 15
5 + 20 = 25
```

---

2. Program2_List.fs – Partial Application กับ List.map

```
// Program2_List.fs
open System

let multiply x y = x * y
let numbers = [1;2;3;4;5]

[<EntryPoint>]
let main argv =
    let triple = multiply 3        // partial function
    let tripledNumbers = numbers |> List.map triple
    printfn "Original: %A" numbers
    printfn "Tripled: %A" tripledNumbers
    0
```

คำอธิบายโค้ด

- multiply เป็น curried function
- Partial application triple ใช้กับ List.map

- ทำให้โค้ด concise และ readable

ผลการรัน

Original: [1; 2; 3; 4; 5]

Tripled: [3; 6; 9; 12; 15]

---

3. Program3_Pipeline.fs – Partial Application กับ Pipeline

```
// Program3_Pipeline.fs
open System

let add x y = x + y
let square x = x * x

[<EntryPoint>]
let main argv =
    let add3 = add 3              // partial function
    let numbers = [1;2;3;4]
    let results = numbers |> List.map add3 |> List.map square
    printfn "Pipeline Results: %A" results
    0
```

คำอธิบายโค้ด

- Partial function add3 ใช้ใน pipeline
- ใช้ List.map สองครั้งกับ pipeline
- Functional และ readable

ผลการรัน

Pipeline Results: [16; 25; 36; 49]

==========

□ ตัวอย่างประยุกต์ 3 โปรแกรม

4. Program4_Calc.fs – Partial Application + Calculator

```
// Program4_Calc.fs
open System

let calc x y op =
    match op with
```

```
    | "add" -> x + y
    | "sub" -> x - y
    | "mul" -> x * y
    | "div" -> if y = 0 then 0 else x / y
    | _ -> 0

[<EntryPoint>]
let main argv =
    let add10 = calc 10 >> (fun f -> f 5 "add")  // partial application + composition
    let mul2 = calc 2 >> (fun f -> f 8 "mul")
    printfn "10 + 5 = %d" (add10 ())
    printfn "2 * 8 = %d" (mul2 ())
    0
```

คำอธิบายโค้ด

- ใช้ partial application + composition
- สร้างฟังก์ชันย่อยสำหรับ operation เฉพาะ
- ใช้งาน dynamic calculator

ผลการรัน

```
10 + 5 = 15
2 * 8 = 16
```

---

5. Program5_Async.fs – Partial Application + Async

```
// Program5_Async.fs
open System
open System.Threading.Tasks

let asyncAdd x y = async { return x + y }

[<EntryPoint>]
let main argv =
    let add3Async = asyncAdd 3        // partial application
    let result = add3Async 7 |> Async.RunSynchronously
    printfn "3 + 7 = %d" result
    0
```

คำอธิบายโค้ด

- Partial function add3Async สำหรับ async workflow
- ใช้ Async.RunSynchronously เพื่อรอผล
- สามารถขยายไปทำ async pipelines ได้

ผลการรัน

3 + 7 = 10

---

6. Program6_ERP.fs – Partial Application + Error Handling (CPS)

```
// Program6_ERP.fs
open System

let safeDivide x y =
    if y = 0 then None else Some(x / y)

let divideCPS x y cont =
    match safeDivide x y with
    | Some r -> cont r
    | None -> printfn "Division by zero"

[<EntryPoint>]
let main argv =
    let div10 = divideCPS 10      // partial application
    div10 2 (fun r -> printfn "10 / 2 = %d" r)
    div10 0 (fun r -> printfn "10 / 0 = %d" r)
    0
```

คำอธิบายโค้ด

- ใช้ partial application + CPS + option type
- ฟังก์ชัน div10 ถูกสร้างจาก partial application
- การจัดการ division by zero เป็น functional

ผลการรัน

10 / 2 = 5

Division by zero

Function Composition ใน F#