F# Programming: Intermediate

(Integrative-Generative AI Edition)

Student Price Book Center

F# Programming:

(Integrative-Generative AI Edition)

.NET

Student Price Book Center

คำนำ

ภาษา **F#** เป็นหนึ่งในภาษาของตระกูล .NET ที่เน้นการเขียนโปรแกรมเชิงฟังก์ชัน (Functional Programming) แต่ในขณะเดียวกันก็รองรับการเขียนโปรแกรมเชิงวัตถุ (Object-Oriented Programming) และการจัดการข้อมูลเชิงโครงสร้างอย่างครบถ้วน ความยืดหยุ่นเหล่านี้ทำให้ F# เหมาะกับการพัฒนาแอปพลิเคชันทั้งขนาดเล็กและระบบซอฟต์แวร์ขนาดใหญ่ที่ซับซ้อน หนังสือเล่มนี้ **"F# Programming: Intermediate"** ถูกออกแบบขึ้นเพื่อเสริมทักษะและความเข้าใจของผู้ที่มีพื้นฐาน F# มาแล้วให้สามารถพัฒนาโปรแกรมในระดับกลางถึงสูงได้อย่างมั่นใจ

เนื้อหาของหนังสือเล่มนี้เริ่มต้นที่ **บทที่ 7 Modules & Namespaces** ซึ่งเป็นหัวใจสำคัญในการจัดระเบียบโค้ดอย่างเป็นระบบ การสร้างโมดูล การใช้คำสั่ง open System และการจัดการ namespace ช่วยให้โค้ดของผู้พัฒนามีความชัดเจน สามารถนำกลับมาใช้ซ้ำ และลดความซับซ้อนในการพัฒนาระบบขนาดใหญ่ บทนี้ยังนำเสนอ **ตัวอย่างโปรเจกต์บูรณาการ** ที่ผู้อ่านสามารถนำแนวคิดไปประยุกต์ใช้งานจริงได้

ต่อมาที่ **บทที่ 8 Object-Oriented in F#** หนังสือจะพาผู้อ่านทำความเข้าใจการใช้แนวคิด OOP ใน F# ตั้งแต่การสร้างคลาส การกำหนด Properties และ Methods ไปจนถึงการสืบทอด (Inheritance) และการออกแบบด้วย Interfaces การเรียนรู้หัวข้อเหล่านี้จะช่วยให้ผู้พัฒนาสามารถผสมผสานแนวคิดเชิงวัตถุกับแนวคิดเชิงฟังก์ชันได้อย่างมีประสิทธิภาพ พร้อมตัวอย่างโปรเจกต์บูรณาการที่เน้นการประยุกต์ใช้งานจริง

บทที่ 9 Collections & Functional Programming จะนำเสนอการจัดการข้อมูลด้วยโครงสร้างพื้นฐาน เช่น **List, Array, Seq, Map และ Set** พร้อมเทคนิคการประมวลผลข้อมูลแบบฟังก์ชัน เช่น map, filter, fold และ reduce นอกจากนี้ยังเจาะลึกการใช้ **Pipeline และ Composition** เพื่อจัดลำดับการทำงานของฟังก์ชันอย่างชัดเจน แนวทางนี้ช่วยให้โค้ดกระชับ อ่านง่าย และสอดคล้องกับแนวคิด Functional Programming Style

ในส่วนของ **บทที่ 10 Exception Handling** หนังสือเน้นการจัดการข้อผิดพลาดที่มีประสิทธิภาพ ตั้งแต่การใช้ try/with/finally ไปจนถึงการใช้ **Result<'T,'Error> type** และแนวคิด **Railway Oriented Programming (ROP)** เพื่อออกแบบการประมวลผลที่รองรับข้อผิดพลาดได้อย่างต่อเนื่องและปลอดภัย แนวทางเชิงฟังก์ชันนี้ช่วยให้โค้ดมีความชัดเจน ลดความซับซ้อน และเพิ่มความทนทานของซอฟต์แวร์

บทที่ 11 Asynchronous Programming จะพาผู้อ่านเข้าใจการประมวลผลแบบไม่บล็อกโปรแกรมหลักผ่าน **Async Workflow (async { ... })** เปรียบเทียบการใช้งาน **Async กับ Task** ใช้งาน **Async.Parallel** สำหรับรันหลายงานพร้อมกัน และจัดการการยกเลิกงานด้วย **CancellationToken** การเรียนรู้หัวข้อเหล่านี้ช่วยให้สามารถพัฒนาโปรแกรมที่ตอบสนองรวดเร็ว ประมวลผลพร้อมกันหลายงาน และสามารถควบคุมการทำงานได้อย่างยืดหยุ่น

สุดท้าย **บทที่ 12 File & IO** ครอบคลุมการอ่านและเขียนไฟล์ด้วย **System.IO** การใช้ **Async IO** เพื่อประมวลผลไฟล์แบบไม่บล็อกโปรแกรมหลัก และการจัดการข้อมูลเชิงโครงสร้างอย่าง **JSON และ XML** บทนี้ช่วยให้ผู้พัฒนาสามารถจัดเก็บ ดึงข้อมูล และประมวลผลไฟล์ได้อย่างมีประสิทธิภาพ ปลอดภัย และเหมาะกับการพัฒนาแอปพลิเคชันจริง

หนังสือเล่มนี้ออกแบบเนื้อหาให้ผู้อ่านสามารถเรียนรู้จากพื้นฐานไปสู่การประยุกต์ใช้งานจริง ผ่าน **ตัวอย่างโปรเจกต์บูรณาการ** ในแต่ละบท เพื่อให้ผู้อ่านสามารถเชื่อมโยงแนวคิดเชิงทฤษฎีกับการปฏิบัติได้อย่างครบถ้วน เหมาะสำหรับผู้ที่ต้องการพัฒนาทักษะ F# ในระดับกลางและก้าวไปสู่การพัฒนาโปรแกรมเชิงมืออาชีพ

ด้วยรักและปรารถนาดี

ศูนย์หนังสือราคานักเรียน

สารบัญ

หน้า

บทที่ 10

Exception Handling

(Exception Handling)

เนื้อหา

- Exception Handling
- บทที่ 10: Exception Handling ใน F# แบบเชิงลึก
- Try / With / Finally
- การใช้ Result<'T,'Error> type ใน F#
- Error-handling แบบ functional ใน F# โดยใช้ Railway Oriented Programming (ROP)
- ตัวอย่างบูรณาการ

บทนำ

การจัดการข้อผิดพลาด (**Exception Handling**) เป็นหัวใจสำคัญของการพัฒนาโปรแกรมที่มีความเสถียรและน่าเชื่อถือ เนื่องจากซอฟต์แวร์ทุกระบบล้วนเผชิญกับความไม่แน่นอน ไม่ว่าจะเป็นข้อมูลที่ไม่ถูกต้อง การเชื่อมต่อเครือข่ายที่ล้มเหลว หรือการทำงานของทรัพยากรที่ไม่คาดคิด หากไม่มีการวางกลไกจัดการข้อผิดพลาดอย่างเป็นระบบ โปรแกรมอาจหยุดทำงานหรือให้ผลลัพธ์ที่ไม่ถูกต้องได้

ในภาษา **F#** แนวทางดั้งเดิมในการจัดการข้อผิดพลาดสามารถทำได้ด้วยคำสั่ง **try / with / finally** ซึ่งช่วยให้โปรแกรมสามารถจับข้อผิดพลาดที่เกิดขึ้น จัดการตามเงื่อนไขที่กำหนด และยังมั่นใจได้ว่าการทำงานบางอย่าง เช่น การคืนค่าทรัพยากร จะถูกดำเนินการในส่วน finally เสมอ วิธีนี้ทำให้โปรแกรมมีความทนทานต่อเหตุการณ์ที่ไม่คาดคิด

อย่างไรก็ตาม F# ยังมีรูปแบบที่ก้าวหน้ากว่า โดยการใช้ **Result<'T,'Error> type** เพื่อแทนค่าผลลัพธ์ที่อาจสำเร็จหรือเกิดข้อผิดพลาดขึ้น การใช้ Result type ช่วยให้นักพัฒนาสามารถจัดการข้อผิดพลาดในรูปแบบของค่าข้อมูลแทนการใช้กลไกการขว้าง exception เพียงอย่างเดียว ส่งเสริมให้โค้ดมีความชัดเจนและคาดการณ์ได้ง่าย

นอกจากนี้ F# ยังสนับสนุนแนวคิด **Error-handling แบบ Functional** โดยเฉพาะ **Railway Oriented Programming (ROP)** ซึ่งใช้การไหลของค่าผ่านฟังก์ชันที่มีทั้งเส้นทาง “สำเร็จ” และ “ผิดพลาด” คล้ายรางรถไฟสองเส้น การออกแบบเช่นนี้ทำให้สามารถเขียนโค้ดที่รองรับข้อผิดพลาดได้อย่างต่อเนื่องและไม่ซับซ้อน

บทนี้จึงมุ่งเน้นให้ผู้อ่านเข้าใจทั้งวิธีการดั้งเดิมและวิธีการเชิงฟังก์ชันในการจัดการข้อผิดพลาด โดยจะครอบคลุมการใช้ try/with/finally, การประยุกต์ใช้ Result<'T,'Error> และการออกแบบด้วยแนวคิด Railway Oriented Programming เพื่อสร้างโปรแกรมที่มีความทนทานและมีคุณภาพสูง

Exception Handling

- Try / With / Finally
- การใช้ Result<'T,'Error> type
- Error-handling แบบ functional (Railway Oriented Programming)

บทที่ 10 **Exception Handling** ใน F# เป็นหัวข้อสำคัญในการจัดการข้อผิดพลาดทั้งแบบ **imperative** และ **functional** ให้โค้ดมีความปลอดภัยและอ่านง่าย

☐ 1. Try / With / Finally

ใน F# การจัดการ exception แบบ **imperative style** ใช้ try ... with ... finally

โครงสร้าง

```
try
    // โค้ดที่อาจ throw exception
with
| :? System.DivideByZeroException ->
    // จัดการ exception ประเภท DivideByZero
| ex ->
    // จัดการ exception อื่น ๆ
finally
    // โค้ดที่ต้องรันเสมอ ไม่ว่ามี exception หรือไม่
```

ตัวอย่าง

```
open System

[<EntryPoint>]
let main argv =
    try
        printf "Enter a number: "
        let x = Console.ReadLine() |> int
        let result = 100 / x
        printfn "100 / %d = %d" x result
    with
```

```
    | :? System.DivideByZeroException -> printfn "Cannot divide by zero!"
    | :? System.FormatException -> printfn "Invalid input, please enter a number!"
    finally
        printfn "Program finished."

    0
```

คำอธิบาย:

- try ⟶ โค้ดที่อาจเกิด exception
- with ⟶ pattern matching เพื่อจัดการ exception ตามประเภท
- finally ⟶ ใช้สำหรับ cleanup เช่น ปิดไฟล์, ปลด resource

ผลการรันตัวอย่าง:

```
Enter a number: 0
Cannot divide by zero!
Program finished.
```

□ 2. การใช้ Result<'T,'Error> type

F# มี type Result<'T,'Error> เพื่อจัดการ error แบบ **functional**

- Ok(value) ⟶ การทำงานสำเร็จ
- Error(err) ⟶ เกิดข้อผิดพลาด

ตัวอย่าง

```
let safeDivide x y =
    if y = 0 then Error "Cannot divide by zero"
    else Ok (x / y)

[<EntryPoint>]
let main argv =
    match safeDivide 100 0 with
    | Ok result -> printfn "Result: %d" result
    | Error msg -> printfn "Error: %s" msg

    match safeDivide 100 5 with
    | Ok result -> printfn "Result: %d" result
    | Error msg -> printfn "Error: %s" msg
```

```
0
```

ผลการรัน:

```
Error: Cannot divide by zero
Result: 20
```

ข้อดี:

- ปลอดภัยกว่า exception
- โค้ดอ่านง่ายและ functional
- ใช้ร่วมกับ pipeline ได้ดี

---

3. Error-handling แบบ functional (Railway Oriented Programming)

Railway Oriented Programming (ROP) ⟶ แนวคิดจัดการ error แบบ **functional pipeline**

- ข้อมูลไหลผ่าน **สองเส้นทาง**: success (Ok) กับ failure (Error)
- ใช้ Result + function composition (>>) หรือ pipeline (|>)

ตัวอย่าง

```
let parseInt str =
    try Ok (int str)
    with | :? System.FormatException -> Error "Not a valid number"

let divide100 x =
    if x = 0 then Error "Cannot divide by zero"
    else Ok (100 / x)

// Pipeline แบบ ROP
let process str =
    str
    |> parseInt
    |> Result.bind divide100

[<EntryPoint>]
let main argv =
    ["10"; "0"; "abc"]
    |> List.iter (fun s ->
        match process s with
        | Ok result -> printfn "100 / %s = %d" s result
```

```
    | Error msg -> printfn "Input %s Error: %s" s msg
)
0
```

คำอธิบาย:

- Result.bind ⟶ ใช้ต่อ function ถ้า Ok, ถ้า Error จะหยุด pipeline
- Input "10" ⟶ 100/10 = 10
- Input "0" ⟶ Error: Cannot divide by zero
- Input "abc" ⟶ Error: Not a valid number

ผลการรัน:

100 / 10 = 10

Input 0 Error: Cannot divide by zero

Input abc Error: Not a valid number

---

4. สรุปความแตกต่าง

วิธี	Style	ข้อดี	ข้อเสีย
try/with/finally	Imperative	เหมาะกับโค้ดที่ต้อง handle exceptions ทันที	โค้ด less functional, side effects
Result<'T,'Error>	Functional	Functional, readable, pipeline friendly	ต้องเปลี่ยน logic ให้ return Result
Railway Oriented Programming	Functional pipeline	สร้าง flow ของ success/error ชัดเจน, composable	ต้องเรียนรู้ pattern, อาจ verbose เล็กน้อย

สรุป:

- F# สนับสนุน **ทั้ง exception แบบดั้งเดิม** และ **functional error handling**
- ROP เป็นแนวคิดที่ใช้ **Result type + pipeline** ทำให้โค้ดอ่านง่ายและปลอดภัย

บทที่ 10: Exception Handling ใน F# แบบเชิงลึก

---

1 Try / With / Finally

นี่คือวิธีจัดการ **imperative exception** แบบดั้งเดิมใน F#

Syntax

```
try
```

```
    // โค้ดที่อาจเกิด exception
with
| :? ExceptionType1 as ex -> // จัดการ ExceptionType1
| :? ExceptionType2 ->        // จัดการ ExceptionType2
| ex ->                       // จัดการ exception อื่น ๆ
finally
    // cleanup code ที่ต้องรันเสมอ
```

Key Points:

1. **Pattern Matching**: ใช้ :? เพื่อตรวจสอบ type ของ exception
2. **Finally Block**: ใช้ปิด resource เช่น ไฟล์, database connection
3. **สามารถจับหลายประเภท exception พร้อมกัน**

ตัวอย่างเชิงลึก

```
open System
open System.IO

[<EntryPoint>]
let main argv =
    try
        let path = "test.txt"
        // เขียนไฟล์ (อาจเกิด IO exception)
        File.WriteAllText(path, "Hello F#")
        // อ่านไฟล์
        let content = File.ReadAllText(path)
        printfn "File Content: %s" content
    with
    | :? IOException as ioEx -> printfn "IO Error: %s" ioEx.Message
    | :? UnauthorizedAccessException -> printfn "No permission to access the file"
    | ex -> printfn "Unexpected Error: %s" ex.Message
    finally
        printfn "Cleaning up resources..."
    0
```

ข้อดี/ข้อเสีย

- □ เหมาะกับ resource management
- □ จับ exception ได้หลาย type

- Less functional, side effects

---

2 การใช้ Result<'T,'Error> type

Result<'T,'Error> เป็น **type-safe approach** แทนการ throw exception

Syntax

```
type Result<'T,'Error> =
    | Ok of 'T
    | Error of 'Error
```

การใช้ Result

- Ok(value) ⟶ การทำงานสำเร็จ
- Error(errorMsg) ⟶ เกิดข้อผิดพลาด

ตัวอย่างเชิงลึก

```
let parseNumber str =
    match System.Int32.TryParse(str) with
    | true, num -> Ok num
    | false, _ -> Error "Not a valid number"

let safeDivide x y =
    if y = 0 then Error "Cannot divide by zero"
    else Ok (x / y)

[<EntryPoint>]
let main argv =
    let result =
        parseNumber "10"
        |> Result.bind (fun x -> safeDivide 100 x)

    match result with
    | Ok v -> printfn "Result: %d" v
    | Error e -> printfn "Error: %s" e
    0
```

Key Points

1. Result.bind ⟶ ต่อ pipeline ของ function ที่ return Result
2. Functional, type-safe, ป้องกันการ throw exception

3. สามารถใช้ร่วมกับ pipeline (|>) หรือ composition (>>) ได้ดี

---

3 Error-handling แบบ functional (Railway Oriented Programming – ROP)

แนวคิดหลัก:

- ข้อมูลไหลผ่าน **สอง track**:
  1. Success (Ok)
  2. Failure (Error)
- แต่ละ function **ไม่ throw exception** แต่ return Result
- สามารถ **compose pipeline** ของหลาย function ได้

ตัวอย่างเชิงลึก

```
let parseInt str =
    try Ok (int str)
    with | :? System.FormatException -> Error "Invalid number"

let divide100 x =
    if x = 0 then Error "Division by zero"
    else Ok (100 / x)

let addTen x = Ok (x + 10)

let process str =
    str
    |> parseInt
    |> Result.bind divide100
    |> Result.bind addTen

[<EntryPoint>]
let main argv =
    let inputs = ["10"; "0"; "abc"]
    inputs |> List.iter (fun s ->
        match process s with
        | Ok value -> printfn "Input %s Result: %d" s value
        | Error msg -> printfn "Input %s Error: %s" s msg
    )
```

0

ผลการรัน

Input 10 Result: 20

Input 0 Error: Division by zero

Input abc Error: Invalid number

ข้อดีของ ROP

1. **Composable** ⟶ สร้าง pipeline ของ function ที่ return Result
2. **อ่านง่าย** ⟶ flow ชัดเจน ไม่ต้อง nested try/catch
3. **Functional style** ⟶ ไม่มี side-effects

☐ **สรุปเชิงลึก**

ประเด็น	Style	Key Concept	ข้อดี	ข้อเสีย
Try/With/Finally	Imperative	Pattern match exception types	เหมาะกับ resource, จับหลาย exception	Less functional, side effects
Result<'T,'Error>	Functional	Return type แทน throw	Type-safe, pipeline friendly	ต้องเปลี่ยน logic return value
Railway Oriented Programming	Functional pipeline	Compose function track (Ok/Error)	Composable, readable, functional	ต้องเข้าใจ pattern, verbose

เคล็ดลับ

- ถ้าต้องการ **readability + safety** ⟶ ใช้ Result หรือ ROP
- ถ้าใช้ **resource management / legacy code** ⟶ ใช้ try/with/finally

---

Try / With / Finally

☐ **1. Syntax และโครงสร้าง**

```
try
    // โค้ดที่อาจเกิด exception
with
| :? ExceptionType1 as ex -> // จัดการ exception ประเภท ExceptionType1
| :? ExceptionType2 ->        // จัดการ exception ประเภท ExceptionType2
| ex ->                      // จัดการ exception อื่น ๆ
finally
```

```
// โค้ด cleanup ที่ต้องรันเสมอ
```

คำอธิบายเชิงลึก:

1. **try block** – โค้ดที่อาจเกิด exception
2. **with block** – ใช้ pattern matching เพื่อจัดการ exception ตามประเภท
    - :? ExceptionType ⟶ ตรวจสอบประเภทของ exception
    - as ex ⟶ เก็บ object exception ไว้ใช้งาน
3. **finally block** – โค้ดที่จะรันเสมอ ไม่ว่า exception จะเกิดหรือไม่
    - ใช้สำหรับ resource cleanup เช่น ปิดไฟล์, database connection

---

□ 2. ตัวอย่างเชิงลึก

```
open System
open System.IO

[<EntryPoint>]
let main argv =
    let filePath = "test.txt"

    try
        // เขียนไฟล์
        File.WriteAllText(filePath, "Hello F#")
        // อ่านไฟล์
        let content = File.ReadAllText(filePath)
        printfn "File Content: %s" content

        // ทำการแบ่งด้วยเลข 0 เพื่อทดสอบ exception
        let x = 0
        let result = 10 / x
        printfn "10 / %d = %d" x result

    with
    | :? IOException as ioEx ->
        printfn "IO Error: %s" ioEx.Message
    | :? DivideByZeroException ->
        printfn "Cannot divide by zero!"
```

```
| ex ->
    printfn "Unexpected Error: %s" ex.Message

finally
    printfn "Cleaning up resources..."
    // ลบไฟล์ทดสอบ
    if File.Exists(filePath) then File.Delete(filePath)

0
```

☐ **3. คำอธิบายโค้ด**

1. File.WriteAllText และ File.ReadAllText ⟶ อาจเกิด IOException
2. 10 / x ⟶ อาจเกิด DivideByZeroException
3. with ⟶ จับ exception ตาม type
4. finally ⟶ ทำ cleanup เสมอ เช่น ลบไฟล์

☐ **4. ผลการรัน**

File Content: Hello F#

Cannot divide by zero!

Cleaning up resources...

☐ **5. ข้อดี/ข้อเสีย**

ประเด็น	ข้อดี	ข้อเสีย
Try/With/Finally	เหมาะกับ resource management, จับหลาย exception	Less functional, side effects, nested try อาจทำให้โค้ดอ่านยาก

ตัวอย่าง Try / With / Finally ใน F# แบบ **เต็มไฟล์** จำนวน **6 โปรแกรม**

- **พื้นฐาน 3 ตัว** ⟶ สาธิตหลักการ Try / With / Finally
- **แนวประยุกต์ 3 ตัว** ⟶ ใช้ใน workflow จริง เช่น File I/O, Division, Resource Management

---

☐ **โปรแกรมพื้นฐาน 3 ตัว**

โปรแกรมที่ 1: Division Safe

โครงสร้างไฟล์

BasicTryCatch1.fs

โค้ดเต็ม

```
open System

[<EntryPoint>]
let main argv =
    try
        printf "Enter a number to divide 100 by: "
        let input = Console.ReadLine()
        let x = int input
        let result = 100 / x
        printfn "100 / %d = %d" x result
    with
    | :? System.DivideByZeroException ->
        printfn "Error: Cannot divide by zero!"
    | :? System.FormatException ->
        printfn "Error: Invalid input, not a number."
    finally
        printfn "Program finished."
    0
```

ผลการรัน

```
Enter a number to divide 100 by: 0
Error: Cannot divide by zero!
Program finished.
```

โปรแกรมที่ 2: File Reading

โครงสร้างไฟล์

BasicTryCatch2.fs

โค้ดเต็ม

```
open System
open System.IO
```

```
[<EntryPoint>]
let main argv =
    let path = "example.txt"
    try
        let content = File.ReadAllText(path)
        printfn "File Content: %s" content
    with
    | :? FileNotFoundException -> printfn "Error: File not found!"
    | :? UnauthorizedAccessException -> printfn "Error: No permission to read the file."
    finally
        printfn "Attempted to read file: %s" path
    0
```

ผลการรัน

```
Error: File not found!
Attempted to read file: example.txt
```

โปรแกรมที่ 3: Array Access Safe

โครงสร้างไฟล์

BasicTryCatch3.fs

โค้ดเต็ม

```
open System

[<EntryPoint>]
let main argv =
    let arr = [|1; 2; 3|]
    try
        printfn "Accessing index 5: %d" arr.[5]
    with
    | :? IndexOutOfRangeException -> printfn "Error: Index out of range!"
    finally
        printfn "Array access attempt completed."
    0
```

ผลการรัน

```
Error: Index out of range!
```

Array access attempt completed.

☐ โปรแกรมแนวประยุกต์ 3 ตัว

โปรแกรมที่ 4: File Write and Read

โครงสร้างไฟล์

AppliedTryCatch1.fs

โค้ดเต็ม

```
open System
open System.IO

[<EntryPoint>]
let main argv =
    let path = "data.txt"
    try
        // เขียนไฟล์
        File.WriteAllText(path, "Hello F# Exception Handling")
        // อ่านไฟล์
        let content = File.ReadAllText(path)
        printfn "File Content: %s" content
    with
    | :? IOException as ioEx -> printfn "IO Error: %s" ioEx.Message
    finally
        printfn "Cleaning up..."
        if File.Exists(path) then File.Delete(path)
    0
```

ผลการรัน

```
File Content: Hello F# Exception Handling
Cleaning up...
```

โปรแกรมที่ 5: Multiple Operations with Try / With / Finally

โครงสร้างไฟล์

AppliedTryCatch2.fs

โค้ดเต็ม

```
open System

[<EntryPoint>]
let main argv =
    try
        printf "Enter dividend: "
        let dividend = int(Console.ReadLine())
        printf "Enter divisor: "
        let divisor = int(Console.ReadLine())
        let result = dividend / divisor
        printfn "%d / %d = %d" dividend divisor result
    with
    | :? System.DivideByZeroException -> printfn "Cannot divide by zero!"
    | :? System.FormatException -> printfn "Invalid input, please enter numbers only!"
    finally
        printfn "Operation attempted."
    0
```

ผลการรัน

```
Enter dividend: 50
Enter divisor: 0
Cannot divide by zero!
Operation attempted.
```

โปรแกรมที่ 6: Resource Management Simulation

โครงสร้างไฟล์

AppliedTryCatch3.fs

โค้ดเต็ม

```
open System

type DummyResource() =
    member this.Open() = printfn "Resource opened."
    member this.Close() = printfn "Resource closed."
    member this.Use() =
        printfn "Using resource..."
```

```
        // จำลอง exception
        failwith "Unexpected error during resource usage"

[<EntryPoint>]
let main argv =
    let resource = DummyResource()
    try
        resource.Open()
        resource.Use()
    with
    | ex -> printfn "Error: %s" ex.Message
    finally
        resource.Close()
    0
```

ผลการรัน

```
Resource opened.
Using resource...
Error: Unexpected error during resource usage
Resource closed.
```

☐ **สรุป**

- **พื้นฐาน 3 ตัว** ⟶ Division, File I/O, Array access
- **ประยุกต์ 3 ตัว** ⟶ File write/read, Multiple user input, Resource management simulation
- **ข้อดีของ Try/With/Finally** ⟶ จับ exception หลาย type, ทำ cleanup, ป้องกัน crash

---

การใช้ Result<'T,'Error> type ใน F#

☐ **1. Concept ของ Result<'T,'Error>**

Result<'T,'Error> เป็น **type-safe alternative** ของ exception handling ใน F#

```
type Result<'T,'Error> =
    | Ok of 'T       // สำเร็จ
    | Error of 'Error // เกิดข้อผิดพลาด
```

Key Points:

1. **Ok(value)** ⟶ แสดงการทำงานสำเร็จ และส่งต่อค่า
2. **Error(err)** ⟶ แสดงข้อผิดพลาด และสามารถจับ pipeline ต่อได้
3. ใช้ร่วมกับ **pipeline (|>)** และ **function composition (>>)** ได้

☐ **2. Basic Usage**

ตัวอย่างพื้นฐาน

```
let safeDivide x y =
    if y = 0 then Error "Cannot divide by zero"
    else Ok (x / y)

[<EntryPoint>]
let main argv =
    match safeDivide 100 0 with
    | Ok result -> printfn "Result: %d" result
    | Error msg -> printfn "Error: %s" msg

    match safeDivide 100 5 with
    | Ok result -> printfn "Result: %d" result
    | Error msg -> printfn "Error: %s" msg
    0
```

ผลการรัน:

```
Error: Cannot divide by zero
Result: 20
```

ข้อดี:

- ปลอดภัยกว่า exception
- อ่านง่าย
- Functional-friendly, pipeline friendly

☐ **3. Using Result.bind for Function Chaining**

Result.bind ช่วยให้คุณ **ต่อ pipeline ของหลาย function ที่ return Result**

ตัวอย่าง

```
let parseInt str =
    match System.Int32.TryParse(str) with
```

```
    | true, n -> Ok n
    | false, _ -> Error "Invalid number"

let divide100 x =
    if x = 0 then Error "Cannot divide by zero"
    else Ok (100 / x)

// Bind example: chain functions safely
let process str =
    parseInt str |> Result.bind divide100

[<EntryPoint>]
let main argv =
    ["10"; "0"; "abc"]
    |> List.iter (fun s ->
        match process s with
        | Ok value -> printfn "Input %s Result: %d" s value
        | Error msg -> printfn "Input %s Error: %s" s msg
    )
    0
```

ผลการรัน:

```
Input 10 Result: 10
Input 0 Error: Cannot divide by zero
Input abc Error: Invalid number
```

คำอธิบายเชิงลึก:

- Result.bind divide100 ⟶ ถ้า input เป็น Ok จะส่งค่าไป function ต่อ
- ถ้า Error จะหยุด pipeline และ return Error ทันที

---

4. Using Result.map for Transformation

Result.map ใช้ **แปลงค่าของ Ok** โดยไม่กระทบ Error

```
let addTen x = x + 10

let process2 str =
    parseInt str
```

```
    |> Result.bind divide100
    |> Result.map addTen

[<EntryPoint>]
let main argv =
    ["10"; "0"; "abc"]
    |> List.iter (fun s ->
        match process2 s with
        | Ok value -> printfn "Input %s Result +10: %d" s value
        | Error msg -> printfn "Input %s Error: %s" s msg
    )
    0
```

ผลการรัน:

Input 10 Result +10: 20

Input 0 Error: Cannot divide by zero

Input abc Error: Invalid number

☐ **5. Advantages of Result<'T,'Error>**

Feature	Advantage
Type-safe	ป้องกัน exception ที่ไม่คาดคิด
Pipeline-friendly	ใช้ `
Composable	chain function ด้วย bind
Readable	flow ของ success/error ชัดเจน
Functional style	ปลอด side-effects, maintain immutability

สรุปเชิงลึก:

- Result<'T,'Error> เป็น **foundation ของ functional error handling**
- เหมาะสำหรับ **pipeline processing**, **input validation**, และ **composable workflows**
- ถ้ารวมกับ **Railway Oriented Programming** จะสามารถสร้าง **error-safe pipelines** ได้

ตัวอย่าง **Result<'T,'Error>** แบบเต็มใน F# จำนวน **6 โปรแกรม**

- **พื้นฐาน 3 ตัว** ⟶ เรียนรู้การใช้ Result พื้นฐาน

- **แนวประยุกต์ 3 ตัว** ⟶ ใช้ใน workflow จริง เช่น pipeline, validation, calculation

---

□ โปรแกรมพื้นฐาน 3 ตัว

โปรแกรมที่ 1: Safe Division

โครงสร้างไฟล์

BasicResult1.fs

โค้ดเต็ม

```
open System

let safeDivide x y =
    if y = 0 then Error "Cannot divide by zero"
    else Ok (x / y)

[<EntryPoint>]
let main argv =
    let result1 = safeDivide 10 2
    let result2 = safeDivide 10 0

    match result1 with
    | Ok v -> printfn "10 / 2 = %d" v
    | Error e -> printfn "Error: %s" e

    match result2 with
    | Ok v -> printfn "10 / 0 = %d" v
    | Error e -> printfn "Error: %s" e

    0
```

ผลการรัน

```
10 / 2 = 5
Error: Cannot divide by zero
```

โปรแกรมที่ 2: Parse Integer

โครงสร้างไฟล์

BasicResult2.fs

โค้ดเต็ม

```
let parseInt str =
    match System.Int32.TryParse(str) with
    | true, num -> Ok num
    | false, _ -> Error "Invalid number"

[<EntryPoint>]
let main argv =
    let inputs = ["42"; "abc"]
    inputs
    |> List.iter (fun s ->
        match parseInt s with
        | Ok n -> printfn "Parsed: %d" n
        | Error e -> printfn "Error: %s" e
    )
    0
```

ผลการรัน

```
Parsed: 42
Error: Invalid number
```

โปรแกรมที่ 3: Chained Safe Operations

โครงสร้างไฟล์

BasicResult3.fs

โค้ดเต็ม

```
let parseInt str =
    match System.Int32.TryParse(str) with
    | true, n -> Ok n
    | false, _ -> Error "Invalid number"

let safeDivide100 x =
    if x = 0 then Error "Cannot divide by zero"
    else Ok (100 / x)
```

```
let process str =
    parseInt str |> Result.bind safeDivide100

[<EntryPoint>]
let main argv =
    ["10"; "0"; "abc"]
    |> List.iter (fun s ->
        match process s with
        | Ok v -> printfn "Input %s Result: %d" s v
        | Error e -> printfn "Input %s Error: %s" s e
    )
    0
```

ผลการรัน

```
Input 10 Result: 10
Input 0 Error: Cannot divide by zero
Input abc Error: Invalid number
```

☐ **โปรแกรมแนวประยุกต์ 3 ตัว**

โปรแกรมที่ 4: Safe User Input & Calculation

โครงสร้างไฟล์

AppliedResult1.fs

โค้ดเต็ม

```
let parseInt str =
    match System.Int32.TryParse(str) with
    | true, n -> Ok n
    | false, _ -> Error "Invalid input"

let safeDivide x y =
    if y = 0 then Error "Division by zero"
    else Ok (x / y)

let addTen x = Ok (x + 10)
```