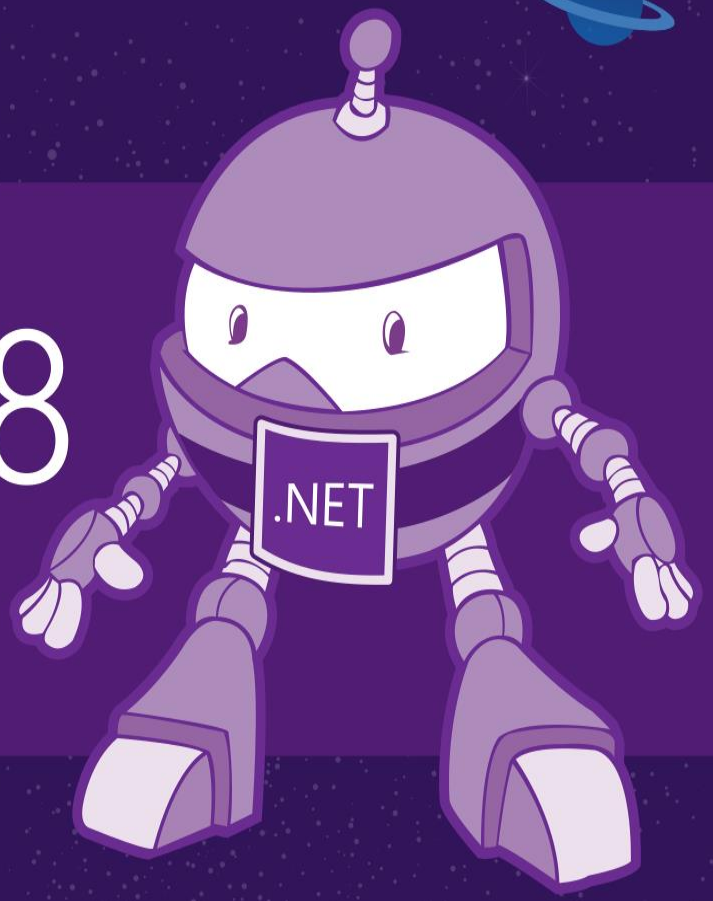
.NET Conf 2018

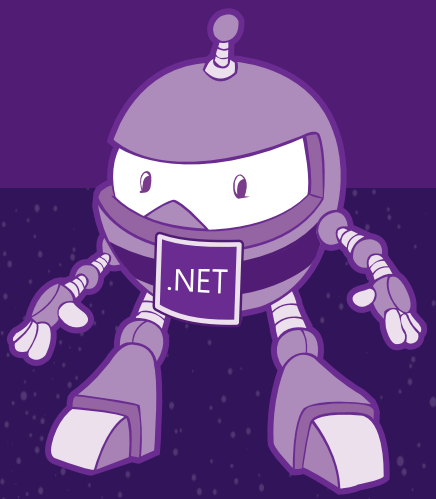
Discover the world of .NET



www.dotnetconf.net

Introducing Functional Programming in C#

Kevin Yang, Sam Xiao



STUDY4.TW
為學習而生

.NET Conf 2018

About Kevin

- Angular Taiwan、Angular Girls Taiwan 、TypeScript Taiwan 社群管理者
- Angular 線上讀書會主持人
- Angular/Web GDE (Google Developer Expert) 開發專家
- 微軟最有價值專家(Visual Studio and Development Technologies)。
- 部落格：https://blog.kevinyang.net/
- FB粉絲頁：https://www.facebook.com/CKNotepad/

About Sam

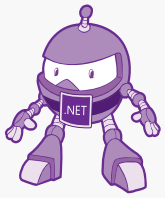
- Laravel 台中社群小聚 2016 講師
- Community Open Camp 2016 講師
- PHPConf 2016 講師
- COSCUP 2017 講師
- Angular 台北社群小聚 2017 講師
- 微軟最有價值專家(Visual Studio and Development Technologies)
- 部落格：點燈坊
- FB 粉絲頁：點燈坊

Outline

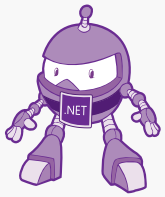
- Introduction
- Definition
- FP 文藝復興
- OOP vs. FP
- C# 對 FP 支援
- 如何學習與導入 FP ?
- Reference

Introduction

- FP 是 Paradigm， 不是 Pattern 也不是 Framework, 更不是 Language
- 是一種以 Function 為中心的「思考方式」與「程式風格」
- 有別於目前主流 OOP，比較冷門，但不代表你不認識他



其實我們天天在用的 LINQ，就是 FP

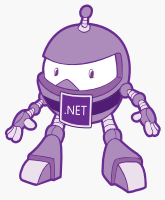


LINQ

```
Range(1, 3)
 .Select(x => x + 1)
 .OrderBy(x => -x)
 .ToList()
 .ForEach(WriteLine);
```

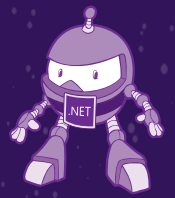
```
4
3
2
```

- LINQ 的 IEnumerable 提供了大量的 Operator
- 我們只需為客製化部分提供 Lambda 即可

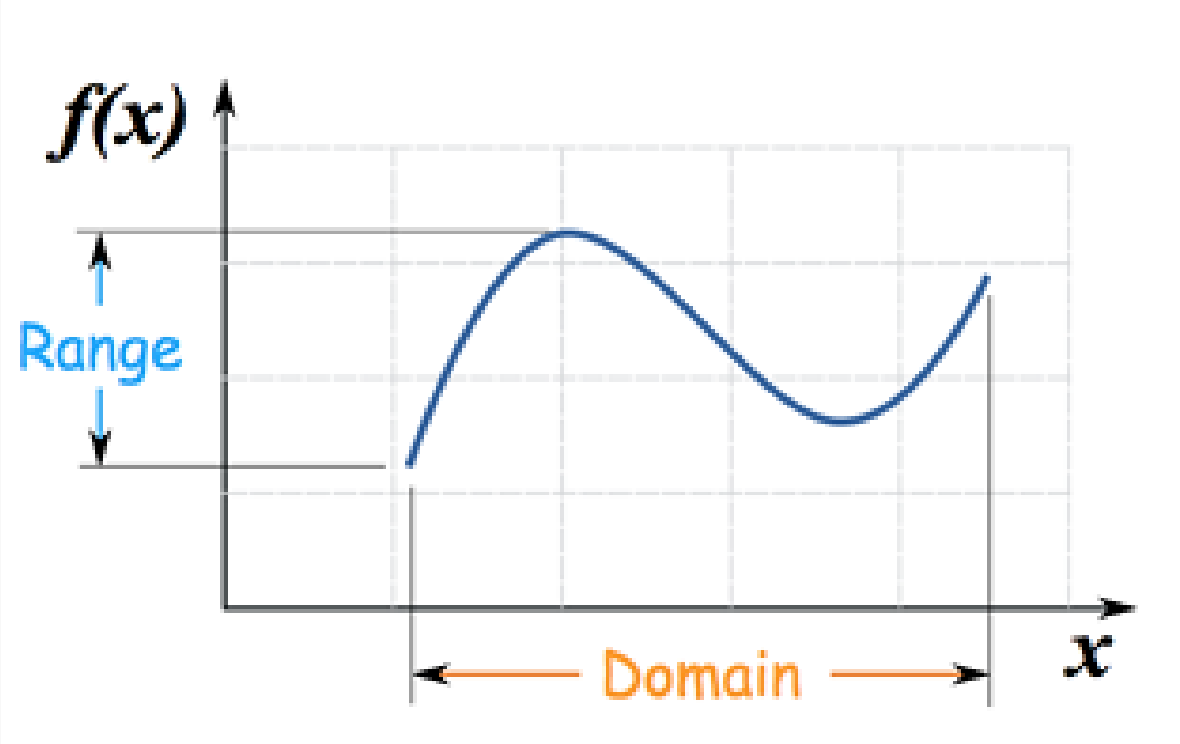


Definition

.NET

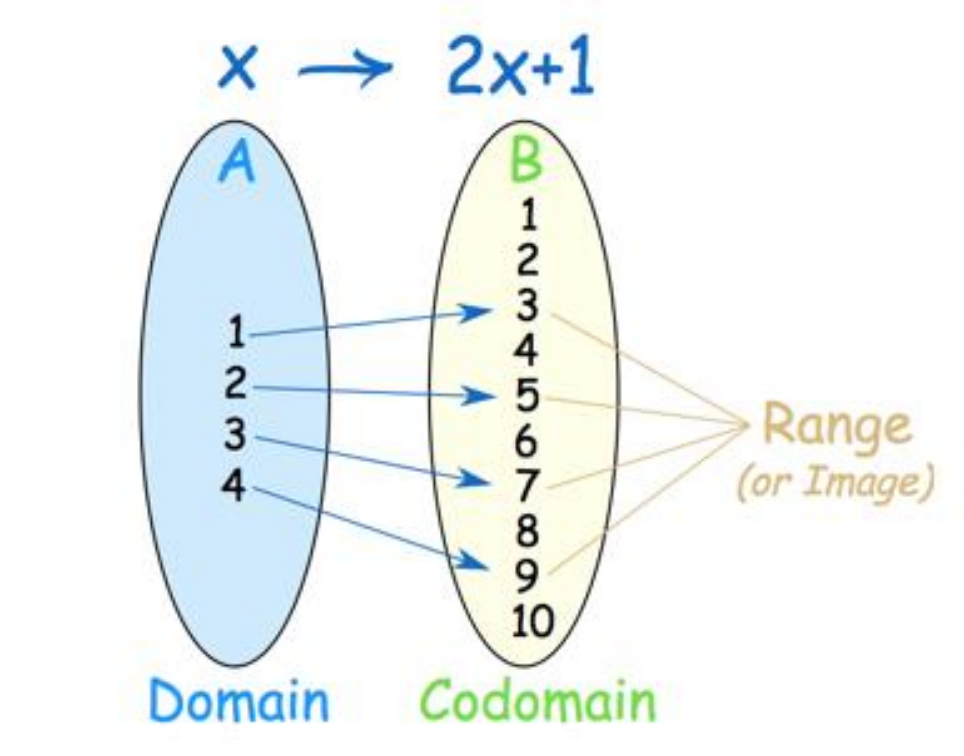


Mathematical Function



- 在數學裡，function 就是 x 與 y = f(x) 的對應關係
- f(x) 結果只與 x 的輸入有關，不會其他任何東西相關

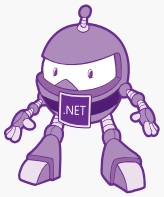
Mathematical Function



- Domain : 所有 x 稱為 **定義域**
- Codomain : 所有**可能**的 f(x) 稱為 **對應域**
- Range : 所有**實際**的 f(x) 稱為 **值域**

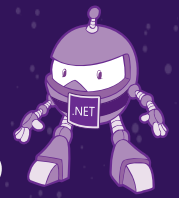
代數

國中就學過了



FP

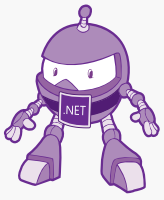
.NET



Pure Function

```
int Foo(int x)
{
   return x + 1;
}
```

- 將 Mathematical Function 以 code 呈現，稱為 Pure Function
- Pure Function : function 的結果只與參數有關，不會與其他任何東西相關

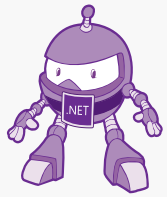


Programming Function

```
int z = 1;
int w = 0;

int Foo(int x)
{
   w = 3;
   return 2 * x + z + 3;
}
```

- Func() 的結果會與參數以外的資料相關 => 非 Pure Function
- z 的改變可能影響到其他 function (Side Effect)
- w 的改變可能影響 Func() (Side Effect)



Debug 大都在釐清 Side Effect

Side Effect

- 修改 Function 外部的變數
- 修改 Function 的 input 參數
- 拋出 Exception
- 處理 I/O

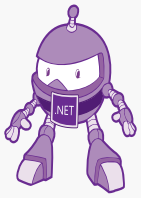
Functional Programming

- Function as Data / Higher Order Functions
- No State Mutation

Function as Data

- First-Class Function
- 能將 Function 指定給變數 (Delegate)
- 能將 Function 傳進 Function (Higher Order Function)
- 能使 Function 回傳 Function (Higher Order Function)
- 能將 Function 存進 Collection

C# 是以 Delegate 為基礎的 FP

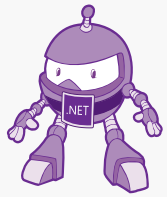


能將 Function 指定給變數 (Delegate)

```
Func<int, int> Triple = x => x * 3;
Range(1, 3)
   .Select(Triple)
   .OrderBy(x => -x)
   .ToList()
   .ForEach(WriteLine);
```

```
9
6
3
```

- 將 Lambda 指定給 Triple Delegate
- Func<int, string> : 輸入為 int，回傳為 string

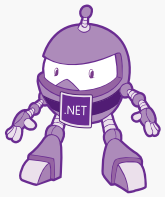


能將 Function 傳進 Function

```
int Triple = x => x * 3;
Range(1, 3)
    .Select(Triple)
    .OrderBy(x => -x)
    .ToList()
    .ForEach(WriteLine);
```

```
9
6
3
```

- 定義 Triple() Local Function
- Triple() 會自動轉成 Delegate 傳入 Select()
- Select() 為 Higher Order Function

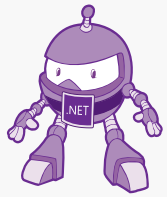


能使 Function 回傳 Function

```
Func<int, Func<int,int>> Triple = x => y => x * y;
Range(1, 3)
   .Select(Triple(3))
   .OrderBy(x => -x)
   .ToList()
   .ForEach(WriteLine);
```

```
9
6
3
```

- 將 3 也變成參數
- Triple(3) 回傳的是 Function (Delegate)
- Tripe() 為 Higher Order Function

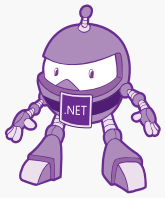


能將 Function 存進 Collection

```
var lut = new Dictionary<char, Func<int, int>> {
                {'+', x => x + x},
                {'*', x => x * x}
            };
Range(1, 3).
    .Select(lut[ '*' ])
    .ToList()
    .ForEach(WriteLine);
```

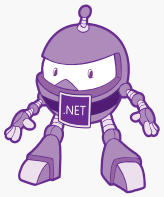
```
1
4
9
```

- 使用 Dictionary 實現 Function Factory



No State Mutation

資料不能修改，只能建立新的
(Immutable)



Mutable

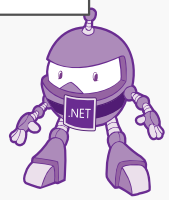
```
public class Rectangle
{
   public int Length { get; set; }
   public int Height { get; set; }

  public void Grow(int length, int height)
  {
        Length += length;
        Height += height;
  }
}
```

```
Rectangle r1 = new Rectangle {
      Length = 5, Height = 10
};

r1.Grow(10, 10);
```

15; 20



Immutable

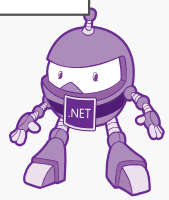
```
public class Rectangle
{
    public int Length { get; }
    public int Height { get; }

     public Rectangle(int length, int height)  {
          Length = length;
          Height = height;
     }

    public Rectangle Grow(int length, int
height)  => new Rectangle(Length + length,
Height + height);
}
```

```
Rectangle r1 = new Rectangle(5,
10);
var r2 = r1.Grow(10, 10);
```

15; 20

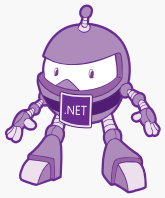


Mutable

```
var data = new List<int> { 3, 2, 1 };
data.Sort();
data.ForEach(WriteLine);
```

```
1
2
3
```

- Sort() 直接對 data 做排序，修改原本資料
- Sort() 為 void，無法繼續串下去

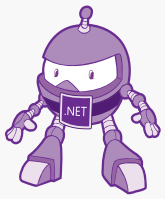


Immutable

```
var data = new List<int> { 3, 2, 1 };
data
    .OrderBy(item => item)
    .ToList()
    .ForEach(WriteLine);
```

```
1
2
3
```

- OrderBy() 沒有宣改原本 data，而是建立新的 data

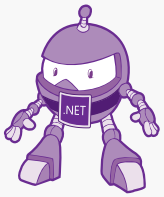


Parallel Computation (Mutable)

```
var data = Range(-10000, 20001).Reverse().ToList();

Action task1 = () => WriteLine(data.Sum());
Action task2 = () => { data.Sort(); WriteLine(data.Sum()); };
Parallel.Invoke(task1, task2);
```

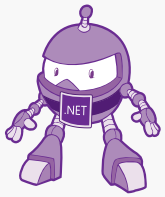
- 每次執行結果都不一樣 (Race Condition)



Parallel Computation (Immutable)

```
var data = Range(-10000, 20001).Reverse().ToList();
Action task1 = () => WriteLine(data.Sum());
Action task2 = () => WriteLine(data.OrderBy(x => x).Sum());
Parallel.Invoke(task1, task2);
```

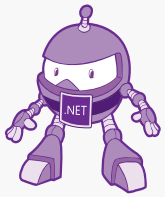
- 每次執行結果都一樣
- 沒有 Race Condition



Parallel Computation (Immutable)

```
int Triple(int x) => x *3 ;
Range(1, 100)
    .AsParallel()
    .Select(Triple)
    .OrderBy(x => x)
    .ToList()
    .ForEach(WriteLine);
```

- 只要加上 AsParrale()，完全無痛升級多核心運算



FP 不斷地建立新資料，執行速度 OK 嗎？

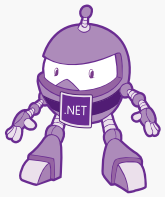
Lazy Evaluation 與 Yield

- Lazy Evaluation : 程式碼不會立即執行，等到有需求時才執行
- Yield : 避免在 Function 傳遞之間不斷重新建立 Data

LINQ Select 非 FP 作法

```
public static IEnumerable<U> Select<T, U>(this
IEnumerable<T> data, Func<T, U> fn)
{
    var result = new List<U>();
    foreach (var item in list)
    {
       result.Add(fn(item));
    }
    return result;
}
```

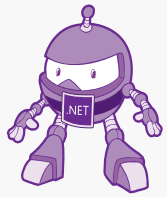
- 需要不斷建立 result，然後回傳，執行效率差



LINQ Select FP 作法

```
public static IEnumerable<U> Select<T, U>(this
IEnumerable<T> data, Func<T, U> fn)
{
    foreach (var item in list)
    {
       yield return fn(item));
    }
}
```

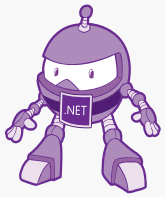
- 直到 WriteLine 執行時，才會執行 fn(item) 回傳



FP 文藝復興

FP 文藝復興

- 單核心無法超越 5GHz，改成多核心設計
- Side Effect 難寫 Unit Test
- OOP 造成 Interface 爆炸
- Reactive Programming 崛起
- 雲端 Stateless (Azure Function / AWS Lambda)



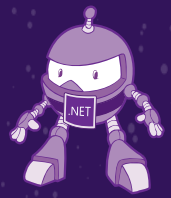
使用 FP 的公司

OOP vs. FP

.NET

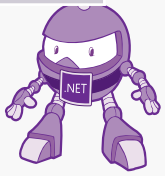
STUDY4.TW
為學習而生

.NET Conf 2018



OOP 與 FP 比較表

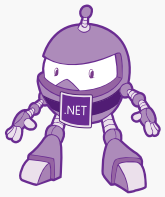
	OOP	FP
Encapsulation	Data 與 Logic 包在 class 內	Data 與 Logic 分家
State Mutation	Mutable	Immutable
Modularity	Class	Function
Dependency	IoC Container	Higher Order Function
Loose Coupling	Interface	Signature



C# 對 FP 支援

C# 版本進化 - FP 支援度

- 1.0
  - Delegate
- 3.0
  - Func、Action
  - Lambda
  - LINQ
  - Extension Method
- 6.0
  - Using Static
  - Expression Body
  - Getter Only Property
- 7.0 ~ 7.3
  - Local Function
  - Tuple
  - Pattern Matching
  - Expression Body

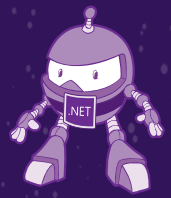


C# 8.0

- Record Type
- 更完整 Pattern Matching
- 更完整 Tuple

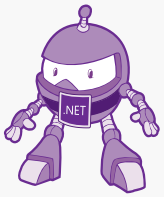
如何學習與導入 FP？

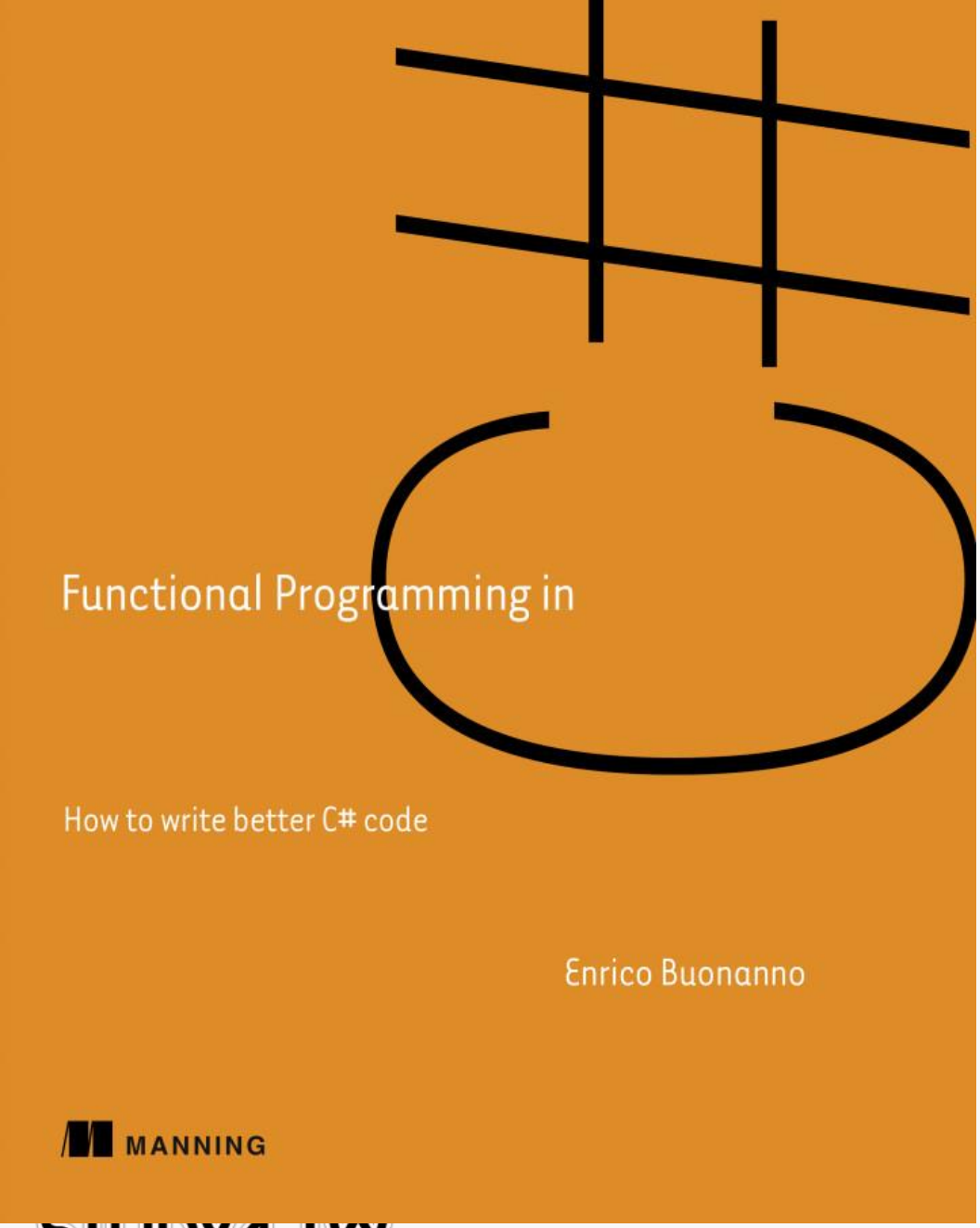
.NET



學習 FP

- Step 1 : 多使用 LINQ 處理 Data，少自己硬幹
- Step 2 : 將 Data 與 Function 分離，為 Data 型別寫 Extension Method
- Step 3 : 將 Side Effect 部分隔離在 Data Flow 前後





Functinal Programming in C#

Enrico Buonanno
August, 2017
ISBN : 9781617293955
Manning

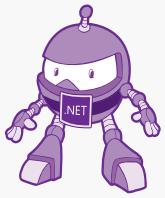
- 全書以 C# 6/7 為範例
- 將抽象的概念使用易懂的文字與圖片介紹
- 手把手建立一個 FP Library，彌補目前 C# 還沒支援的功能 (Option、Either...)
- 以後端的角度介紹 FP，範例貼近實務

導入 FP

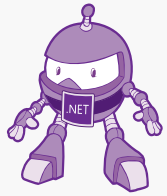
- OOP x FP
- Pure FP

Reference

- Channel 9, Functional Programming in C#
- NDC, Functional Techniques for C#
- NDC, C# 8
- NDC, Logic vs. Side Effects : Functional Godness you don't hear about



特別感謝



.NET Conf 2018