Delegates In Functional Programming

إيه هي الـ Delegates؟

لو بتشتغل في عالم البرمجة، خاصة بلغات زي C# في .NET، أكيد سمعت عن الـ Delegates. الـ Delegates هي زي references أو pointers للـ functions، يعني نوع من الأنواع اللي بيسمحلك تمرر دالة زي ما بتمرر متغير عادي. ده بيفتح الباب للبرمجة الوظيفية (Functional Paradigm)، اللي بتركز على معاملة الدوال كـ Objects من الدرجة الأولى (First-Class Functions). قبل الـ Delegates، كان الوضع معقد شوية في لغات زي C++ أو حتى الإصدارات القديمة من C#، بس دلوقتي هي أداة قوية لكتابة كود أكثر مرونة وإعادة استخدام.

في البرمجة الوظيفية، الفكرة الأساسية إن الـ Functions مش بس أوامر، لكنها بيانات يمكن تمريرها، إرجاعها، أو حتى تخزينها. الـ Delegates بتساعد في ده عن طريق إنها تقدر تشير لأي function تطابق الـ Signature بتاع الـ Delegates، زي عدد الـ parameters و الـ return type.

1. دور الـ Delegates في البرمجة الوظيفية

البرمجة الوظيفية بتركز على مفاهيم زي الـ Pure Functions (دوال بدون آثار جانبية)، Higher-Order Functions (دوال تقبل دوال كـ parameters)، وImmutabilityg (البيانات مش بتتغير). الـ Delegates بتكون الجسر ده في لغات زي C#، اللي أساسًا موجهة للكائنات (Object-Oriented).

- **تمرير الدوال كـ parameters**

بدل ما تكتب كود مكرر، تقدر تمرر دالة لدالة تانية. ده يشبه الـ Callbacks في JavaScript.

مثال بسيط:

```
// تعريف Delegate
public delegate int MathOperation(int a, int b);

// دالة Higher-Order
public int PerformOperation(int x, int y, MathOperation
operation) {
    return operation(x, y);
}

// استخدام
int sum = PerformOperation(5, 3, (a, b) => a + b); //
Lambda Expression
Console.WriteLine(sum); // 8 :الناتج
```

هنا الـ Delegate MathOperation بيسمحلك تمرر عملية حسابية زي الجمع أو الطرح كـ parameter، وده مثال كلاسيكي للبرمجة الوظيفية.

- **الـ Events والـ Observers**

في البرمجة الوظيفية، بنحتاج نسمع للتغييرات بدون تغيير الحالة. الـ Delegates بتستخدم في الـ Events، زي في UI أو asynchronous programming.

مثال:

```
public delegate void NotificationHandler(string message);

public class Publisher {
    public event NotificationHandler OnNotify;

    public void SendNotification(string msg) {
        OnNotify?.Invoke(msg);
    }
}
```

```
// استخدام
Publisher pub = new Publisher();
pub.OnNotify += (msg) => Console.WriteLine("تلقيت: " + msg);
pub.SendNotification("مرحبا!"); // الناتج: تلقيت: مرحبا!
```

ده يشبه الـ Observables في البرمجة الوظيفية، حيث الدوال بتتفاعل بدون حالة مشتركة.

- **الـ Lambdas والـ Anonymous Methods**

من C# 3.0، الـ Delegates بتدعم الـ Lambda Expressions، اللي بتخلي الكود أقصر وأكثر وظيفية.

بدل كتابة دالة كاملة، تقدر تكتب inline function.

2. إزاي كانت البرمجة الوظيفية قبل الـ Delegates؟

قبل ظهور الـ Delegates في .NET (حوالي 2002 مع C# 1.0)، كان الوضع مختلف في لغات زي C أو C++:

استخدام Function Pointers: زي في C، كنت تشير لدالة بـ void (*func)(int)، بس ده كان غير آمن (No Type Safe) ومعرض لأخطاء زي Null Pointers.

عدم دعم للحالات المتعددة: مش كان سهل إنك تضيف أكتر من دالة واحدة للإشارة (Multicast Delegates)، فكنت لازم تكتب كود إضافي للـ Callbacks.

قلة المرونة: في لغات قديمة زي Visual Basic 6، كان صعب تمرير دوال، فكنت تعتمد على Subroutines بس بدون Higher-Order Functions.

العيوب:

الكود كان أكثر تعقيدًا وأقل أمانًا.

صعب تنفيذ مفاهيم وظيفية زي Map أو Filter بدون مكتبات خارجية.

في C++، كنت تستخدم Templates أو Functors، بس ده كان أثقل من الـ Delegates.

3. التحسينات اللي جابتها الـ Delegates في البرمجة الوظيفية

الـ Delegates غيرت اللعبة في .NET، وجابت تحسينات كبيرة مقارنة بالطرق القديمة:

- **الأمان والتحقق من الأنواع (Type Safety)**

الـ Delegate بيتأكد إن الدالة اللي بتشيرلها تطابق التوقيع، فقلل الأخطاء زي اللي في Function Pointers.

- **دعم الـ Multicast**

تقدر تضيف أكتر من دالة واحدة لـ Delegate واحد باستخدام +=، وده يسمح بتنفيذ Observer Pattern بسهولة، زي في Events.

- **التكامل مع LINQ والـ Lambdas**

في C# الحديث، الـ Delegates أساس LINQ، اللي بيطبق مفاهيم وظيفية زي Where أو Select.

مثال:

```
List<int> numbers = new List<int> { 1, 2, 3, 4 };
var even = numbers.Where(n => n % 2 == 0); // Delegate داخليًا
```

ده يجعل الكود أكثر إيجازًا ووظيفية.

- **المرونة في Asynchronous Programming**

مع async/await، الـ Delegates بتساعد في تمرير Callbacks للعمليات الغير متزامنة، زي Task.Run.

- **قلة الآثار الجانبية**

بتشجع على كتابة Pure Functions، لأنك بتمرر الدوال بدون تغيير حالة خارجية.

- **دعم الـ Generic Delegates**

زي Func<T> و Action<T>، اللي جاهزين في .NET، فمبقيتش محتاج تعرف Delegates جديدة كل مرة.

مثال:

```
static void Main()
{
        // استخدام Func لدالة تأخذ عددين وترجع ناتج
        Func<int, int, int> operation;

        // تعيين دالة الجمع باستخدام Lambda
        operation = (a, b) => a + b;
        Console.WriteLine($"الجمع: {operation(5, 3)}"); // الناتج: 8

// تعيين دالة الطرح //
        operation = (a, b) => a - b;
        Console.WriteLine($"الطرح: {operation(5, 3)}"); // الناتج: 2

// تعيين دالة الضرب //
        operation = (a, b) => a * b;
        Console.WriteLine($"الضرب: {operation(5, 3)}"); // الناتج: 15
}
```

- هنا استخدمنا Func<int, int, int> اللي معناها: دالة تأخذ عددين (int) وترجع int.

- بدل ما نعرف Delegate جديد زي delegate int MathOperation(int a, int b);، استخدمنا Func جاهزة.

4. أخطاء شائعة ونصائح

🧠 الناس بتتلخبط في إيه؟

🔸 **خلط بين Delegate و Event:** الـ Event هو Delegate مع حماية إضافية (متقدرش تشغله إلا من داخل الكلاس).

🔸 **نسيان التحقق من Null:** قبل Invoke()، استخدم ?. عشان تتجنب NullReferenceException.

🔸 **استخدام مفرط:** لو الدالة بسيطة، استخدم Lambda مباشرة بدل Delegate كامل.

⚙️ Tips مهمة:

✔️ ابدأ بـ Func و Action الجاهزين: Func<int, int, int> للجمع مثلًا.

✔️ استخدم الـ Delegates في الـ Dependency Injection لتمرير سلوكيات ديناميكية.

✔️ في البرمجة الوظيفية، ركز على Pure Functions عشان تتجنب المشاكل.

✔️ **جرب مع LINQ:** هي أفضل طريقة تطبق فيها الـ Delegates عمليًا.