1:

```
using System;

class Employee
{
   public string Name { get; set; }
   public int Salary { get; set; }

   public override string ToString()
   {
      return $"{Name} - {Salary}";
   }
}

class SortingAlgorithm<T> where T : IComparable<T>
{
   public static void Sort(T[] array)
   {
      for (int i = 0; i < array.Length - 1; i++)
      {
         for (int j = i + 1; j < array.Length; j++)
         {
            if (array[i].CompareTo(array[j]) > 0)
            {
               T temp = array[i];
               array[i] = array[j];
               array[j] = temp;
            }
         }
```

```
        }
    }
}

class EmployeeComparer : IComparable<Employee>
{
    private Employee _employee;

    public EmployeeComparer(Employee employee)
    {
        _employee = employee;
    }

    public int CompareTo(Employee other)
    {
        return _employee.Salary.CompareTo(other.Salary);
    }

    public static implicit operator EmployeeComparer(Employee e) => new EmployeeComparer(e);

    public static implicit operator Employee(EmployeeComparer ec) => ec._employee;

    public override string ToString() => _employee.ToString();
}

class Program
{
    static void Main()
    {
```

```
    Employee[] employees =
    {
        new Employee { Name = "Ali", Salary = 4000 },
        new Employee { Name = "Sara", Salary = 2500 },
        new Employee { Name = "Omar", Salary = 5000 }
    };

    EmployeeComparer[] employeeComparers = Array.ConvertAll(employees, e => new
EmployeeComparer(e));

    SortingAlgorithm<EmployeeComparer>.Sort(employeeComparers);

    foreach (var e in employeeComparers)
        Console.WriteLine(e);
  }
}
```

- **Generic sorting algorithm** (زي `SortingAlgorithm<T>`بيدينا):
  1. **إعادة الاستخدام**: نفس الخوارزمية تشتغل مع أي نوع.(int, string, Employee...)
  2. **أداء أفضل**: بيقلل الحاجة لعمليات boxing/unboxing مقارنة بـ non-generic collections.
  3. **Type safety**: بيضمن إن الأنواع صح وقت الكومبايل، وبالتالي يمنع أخطاء runtime.
  4. **مرونة**: تقدر تحدد طريقة المقارنة باستخدام `IComparable<T>` أو `IComparer<T>`.

2:

```
using System;

class SortingTwo<T>
{
```

```
    public static void Sort(T[] array, Func<T, T, int> compare)
    {
        for (int i = 0; i < array.Length - 1; i++)
        {
            for (int j = i + 1; j < array.Length; j++)
            {
                if (compare(array[i], array[j]) > 0)
                {
                    T temp = array[i];
                    array[i] = array[j];
                    array[j] = temp;
                }
            }
        }
    }
}

class Program
{
    static void Main()
    {
        int[] numbers = { 5, 2, 9, 1, 6 };
```

```
    SortingTwo<int>.Sort(numbers, (a, b) => b.CompareTo(a)); // ترتيب تنازلي


    Console.WriteLine(string.Join(", ", numbers));
  }
}
```

- **Lambda expressions**بتخلي الكود أقصر وأسهل في القراءة بدل ما نكتب كلاس كامل أو ميثود منفصلة للمقارنة.
- بتخلي **الفرز مرن جدًا** :تقدر تغير ترتيب (تصاعدي/تنازلي) أو شرط المقارنة بسهولة في سطر واحد.
- بتحسن **إعادة الاستخدام** :نفس خوارزمية الفرز تشتغل مع أنواع كتيرة باستخدام Lambda مختلفة.

مثال:

- `(a, b) => a.CompareTo(b)` → ترتيب تصاعدي.
- `(a, b) => b.CompareTo(a)` → ترتيب تنازلي.

3:

```
using System;


class SortingTwo<T>
{
  public static void Sort(T[] array, Func<T, T, int> compare)
  {
    for (int i = 0; i < array.Length - 1; i++)
    {
      for (int j = i + 1; j < array.Length; j++)
```

```
        {
            if (compare(array[i], array[j]) > 0)
            {
                T temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
    }
}

class Program
{
    static void Main()
    {
        string[] words = { "apple", "kiwi", "banana", "pear", "fig" };

        SortingTwo<string>.Sort(words, (a, b) => a.Length.CompareTo(b.Length));

        Console.WriteLine(string.Join(", ", words));
    }
}
```

- استخدام **comparer function ديناميكي** بيسمح بتخصيص أسلوب الفرز لأي نوع بيانات.
- لو ما استخدمناش comparer ، هنكون محصورين في مقارنة افتراضية) زي الترتيب الأبجدي في الـ string أو القيم العددية في الأرقام.(
- الفوايد:
  1. **مرونة** :نقدر نحدد طريقة فرز مختلفة حسب الحاجة (طول النص، السعر، التاريخ...).
  2. **إعادة الاستخدام** :نفس خوارزمية الفرز تشتغل مع أنواع متعددة بدون إعادة كتابة كود جديد.
  3. **قابلية التوسع** :نقدر نضيف قواعد مقارنة جديدة بسهولة في المشروع

4:

```
using System;


class Employee

{

   public string Name { get; set; }

   public int Salary { get; set; }


   public override string ToString()

   {

      return $"{Name} - {Salary}";

   }

}


class Manager : Employee, IComparable<Manager>

{

   public string Department { get; set; }
```

```
    public int CompareTo(Manager other)
    {
        if (other == null) return 1;
        return this.Salary.CompareTo(other.Salary);
    }

    public override string ToString()
    {
        return $"{Name} ({Department}) - {Salary}";
    }
}

class SortingAlgorithm<T> where T : IComparable<T>
{
    public static void Sort(T[] array)
    {
        for (int i = 0; i < array.Length - 1; i++)
        {
            for (int j = i + 1; j < array.Length; j++)
            {
                if (array[i].CompareTo(array[j]) > 0)
                {
```

```
                T temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
    }
}

class Program
{
    static void Main()
    {
        Manager[] managers =
        {
            new Manager { Name = "Ali", Department = "IT", Salary = 4000 },
            new Manager { Name = "Sara", Department = "HR", Salary = 2500 },
            new Manager { Name = "Omar", Department = "Finance", Salary = 5000 }
        };

        SortingAlgorithm<Manager>.Sort(managers);

        foreach (var m in managers)
```

```
        Console.WriteLine(m);
    }
}
```

- لما الـ **derived class** زي (`Manager`) يطبق `IComparable<Manager>`:

1. بيحدد **قواعد المقارنة** الخاصة بيه) هنا حسب الـ(Salary.
2. أي خوارزمية فرز Generic تقدر تستخدم `CompareTo` مباشرة من غير ما تحتاج تعرف منطق المقارنة.
3. بيدينا **تحكم كامل** في طريقة الفرز (ممكن نخليها بالمرتب، الاسم، القسم... إلخ).

- ده بيخلي الكائنات الموروثة زي `Manager` تتفرز بسهولة مع أي sorting method بتتعامل مع Generics.

5:

```
using System;

class Employee
{
    public string Name { get; set; }
    public int Salary { get; set; }

    public override string ToString()
    {
        return $"{Name} - {Salary}";
    }
}
```

```
class SortingTwo<T>
{
    public static void Sort(T[] array, Func<T, T, bool> compare)
    {
        for (int i = 0; i < array.Length - 1; i++)
        {
            for (int j = i + 1; j < array.Length; j++)
            {
                if (compare(array[i], array[j]))
                {
                    T temp = array[i];
                    array[i] = array[j];
                    array[j] = temp;
                }
            }
        }
    }
}

class Program
{
    static void Main()
    {
```

```
        Employee[] employees =
        {
            new Employee { Name = "Ali", Salary = 4000 },
            new Employee { Name = "Mohammed", Salary = 5000 },
            new Employee { Name = "Sara", Salary = 3000 }
        };

        SortingTwo<Employee>.Sort(employees, (a, b) => a.Name.Length > b.Name.Length);

        foreach (var e in employees)
            Console.WriteLine(e);
    }
}
```

استخدام **built-in delegates** زي `Func<T, T, TResult>` بيوفر:

1. **اختصار للكود** :مش محتاج نعرف Delegate جديد يدويًا لكل حالة.
2. **مرونة** :نقدر نمرر أي لامبدا أو ميثود مناسبة على طول.
3. **قابلية إعادة الاستخدام** :نفس الميثود تشتغل مع أنواع مختلفة من غير ما نعيد كتابة الـdelegates.
4. **تكامل مع LINQ**: معظم دوال LINQ زي (`Where`, `Select`, `OrderBy`) بتعتمد علىFunc

6:

```
using System;

class SortingTwo<T>
{
    public static void Sort(T[] array, Func<T, T, int> compare)
```

```
    {
        for (int i = 0; i < array.Length - 1; i++)
        {
            for (int j = i + 1; j < array.Length; j++)
            {
                if (compare(array[i], array[j]) > 0)
                {
                    T temp = array[i];
                    array[i] = array[j];
                    array[j] = temp;
                }
            }
        }
    }
}

class Program
{
    static void Main()
    {
        int[] numbers1 = { 5, 3, 8, 1, 2 };

        // باستخدام Anonymous Function
```

```
        SortingTwo<int>.Sort(numbers1, delegate (int a, int b) { return a.CompareTo(b); });
        Console.WriteLine("Anonymous Function: " + string.Join(", ", numbers1));

        int[] numbers2 = { 5, 3, 8, 1, 2 };

        // باستخدام Lambda Expression
        SortingTwo<int>.Sort(numbers2, (a, b) => a.CompareTo(b));
        Console.WriteLine("Lambda Expression: " + string.Join(", ", numbers2));
    }
}
```

- **Anonymous Functions**:
  - صياغتها أطول شوية.(`delegate (int a, int b) { ... }`)
  - بتديك نفس المرونة زي اللـامبدا، لكن أقل وضوحًا للقراءة.
  - مفيدة في الإصدارات القديمة من C# قبل ما يظهر الـLambda.

- **Lambda Expressions**:
  - صياغتها أبسط وأوضح.(`(a, b) => a.CompareTo(b)`)
  - أسهل في الكتابة والقراءة خصوصًا مع الـLINQ.
  - مفيش فرق جوهري في الأداء، لكن الـ compiler بيحولها بشكل أكتر كفاءة في معظم الحالات

7:

```
using System;

class SortingAlgorithm<T> where T : IComparable<T>
{
```

```
    public static void Sort(T[] array)
    {
        for (int i = 0; i < array.Length - 1; i++)
        {
            for (int j = i + 1; j < array.Length; j++)
            {
                if (array[i].CompareTo(array[j]) > 0)
                {
                    Swap(ref array[i], ref array[j]);
                }
            }
        }
    }

    public static void Swap<U>(ref U a, ref U b)
    {
        U temp = a;
        a = b;
        b = temp;
    }
}

class Program
```

```
{
   static void Main()
   {
      int[] numbers = { 1, 2, 3, 4, 5 };

      Console.WriteLine("قبل Swap: " + string.Join(", ", numbers));

      SortingAlgorithm<int>.Swap(ref numbers[0], ref numbers[4]);

      Console.WriteLine("بعد Swap: " + string.Join(", ", numbers));
   }
}
```

- **Generic methods** زي `Swap<T>` مفيدة لأنها:
  1. **إعادة استخدام**: نفس الميثود تشتغل مع أي نوع بيانات.(int, string, Employee…)
  2. **Type safety**: الكومبايلر بيتأكد إن الأنواع صح، فمفيش أخطاء وقت التشغيل.
  3. **مرونة**: بدل ما نكتب دوال Swap متعددة لكل نوع، بنكتب واحدة بس Generic.
  4. **أداء أفضل**: مفيش حاجة لعمليات Boxing/Unboxing مع الـ value types.

8:

```
using System;

class Employee
{
   public string Name { get; set; }
   public int Salary { get; set; }
```

```
    public override string ToString()
    {
        return $"{Name} - {Salary}";
    }
}

class SortingTwo<T>
{
    public static void Sort(T[] array, Func<T, T, int> compare)
    {
        for (int i = 0; i < array.Length - 1; i++)
        {
            for (int j = i + 1; j < array.Length; j++)
            {
                if (compare(array[i], array[j]) > 0)
                {
                    T temp = array[i];
                    array[i] = array[j];
                    array[j] = temp;
                }
            }
        }
```

```
    }
}

class Program
{
    static void Main()
    {
        Employee[] employees =
        {
            new Employee { Name = "Ali", Salary = 4000 },
            new Employee { Name = "Sara", Salary = 4000 },
            new Employee { Name = "Omar", Salary = 5000 },
            new Employee { Name = "Mona", Salary = 2500 }
        };

        SortingTwo<Employee>.Sort(employees, (a, b) =>
        {
            int salaryCompare = a.Salary.CompareTo(b.Salary);
            if (salaryCompare == 0)
                return a.Name.CompareTo(b.Name);
            return salaryCompare;
        });
```

```
        foreach (var e in employees)
            Console.WriteLine(e);
    }
}
```

التحديات:

1. زيادة **تعقيد الكود** لما يكون فيه أكتر من معيار (مرتب، اسم، تاريخ...).
2. صعوبة **الصيانة والتوسع** لو المنطق بيكبر جدًا.
3. محتاج **مقارنة دقيقة** علشان تتجنب أخطاء زي التعادل غير المتوقع.

الفوائد:

1. بتدي **مرونة عالية** في التحكم في ترتيب البيانات.
2. تشتغل مع أي نوع وأي معايير generic method إعادة الاستخدام: نفس الـ.
3. بتحسن **دقة النتائج** لما يكون معيار واحد مش كفاية.

9:

```
using System;

class Helper
{
    public static T GetDefault<T>()
    {
        return default(T);
    }
}
```

```
class Program
{
   static void Main()
   {
      int defaultInt = Helper.GetDefault<int>();
      string defaultString = Helper.GetDefault<string>();
      bool defaultBool = Helper.GetDefault<bool>();

      Console.WriteLine($"Default int: {defaultInt}");
      Console.WriteLine($"Default string: {defaultString ?? "null"}");
      Console.WriteLine($"Default bool: {defaultBool}");
   }
}
```

- الكلمة المفتاحية **`default(T)`** مهمة جدًا في **generic programming** لأنها:
  1. بترجع قيمة افتراضية مناسبة لنوع البيانات **من غير ما نعرف النوع مسبقًا**.
  2. بتحل مشكلة "إيه القيمة اللي أديها لو مش قادر أرجع نتيجة؟" في الميثودات الـGeneric.
- الفرق في التعامل:
  - **Value Types** (زي `int`, `bool`, `struct`): بيرجع القيمة الابتدائية الخاصة بيها (`0`، `false`، أو struct فاضي).
  - **Reference Types** (زي `string`, `class`, `object`): بيرجع `null`.

10:

```
using System;

class Employee : ICloneable
```

```
{
    public string Name { get; set; }
    public int Salary { get; set; }

    public object Clone()
    {
        return new Employee { Name = this.Name, Salary = this.Salary };
    }

    public override string ToString()
    {
        return $"{Name} - {Salary}";
    }
}

class SortingAlgorithm<T> where T : IComparable<T>, ICloneable
{
    public static void Sort(T[] array)
    {
        for (int i = 0; i < array.Length - 1; i++)
        {
            for (int j = i + 1; j < array.Length; j++)
            {
```

```
            if (array[i].CompareTo(array[j]) > 0)
            {
                T temp = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
    }
  }

  public static T[] CloneArray(T[] array)
  {
    T[] clonedArray = new T[array.Length];
    for (int i = 0; i < array.Length; i++)
    {
      clonedArray[i] = (T)array[i].Clone();
    }
    return clonedArray;
  }
}

class Program
{
```

```
    static void Main()
    {
        Employee[] employees =
        {
            new Employee { Name = "Ali", Salary = 4000 },
            new Employee { Name = "Sara", Salary = 2500 },
            new Employee { Name = "Omar", Salary = 5000 }
        };

        Employee[] clonedEmployees = SortingAlgorithm<Employee>.CloneArray(employees);

        SortingAlgorithm<Employee>.Sort(clonedEmployees);

        Console.WriteLine("Original:");
        foreach (var e in employees)
            Console.WriteLine(e);

        Console.WriteLine("\nCloned & Sorted:");
        foreach (var e in clonedEmployees)
            Console.WriteLine(e);
    }
}
```

- **Constraints** (زي `where T : ICloneable`) بتخلي الكومبايلر يضمن إن الأنواع اللي بنمررها عندها الإمكانيات المطلوبة.

- الفوايد:
  1. **Type Safety**: يمنع تمرير نوع ما ينفعش يتعمله Clone أو Compare.
  2. **Reliability**: الكود بيبقى مضمون إنه هيشتغل صح مع الأنواع اللي تحقق الشرط.
  3. **مرونة + أمان**: بدل ما نستنى خطأ وقت التشغيل، الكومبايلر نفسه يوقف الخطأ من البداية.

11:

```
using System;

using System.Collections.Generic;


delegate string StringTransformer(string input);


class StringHelper

{

   public static List<string> TransformList(List<string> items, StringTransformer transformer)

   {

      List<string> result = new List<string>();

      foreach (var item in items)

      {

         result.Add(transformer(item));

      }

      return result;

   }

}


class Program
```

```
{
    static void Main()
    {
        List<string> words = new List<string> { "hello", "world", "csharp" };

        var upperCase = StringHelper.TransformList(words, s => s.ToUpper());
        Console.WriteLine("UpperCase: " + string.Join(", ", upperCase));

        var reversed = StringHelper.TransformList(words, s => new string(s.Reverse().ToArray()));
        Console.WriteLine("Reversed: " + string.Join(", ", reversed));

        var withStars = StringHelper.TransformList(words, s => $"***{s}***");
        Console.WriteLine("With Stars: " + string.Join(", ", withStars));
    }
}
```

- **فوائد استخدام delegates** مع string transformations في أسلوب:functional programming
  1. **مرونة عالية** :نقدر نمرر أي transformation مختلفة (UpperCase, Reverse, Trim…) من غير ما نغير الميثود الأصلية.
  2. **إعادة الاستخدام** :نفس الميثود `TransformList` تشتغل مع أي دالة تحول.string
  3. **قابلية التوسع** :لو عايزين نضيف transformation جديدة، بنكتب لامبدا أو ميثود ونمررها فقط.
  4. **فصل المنطق** :الميثود المسؤولة عن التحويل منفصلة عن طريقة التطبيق، وده بيخلي الكود أنظف وأسهل للصيانة.

12:

```
using System;
```

```
class MathHelper
{
    public static double Calculate(double a, double b, Func<double, double, double> operation)
    {
        return operation(a, b);
    }
}

class Program
{
    static void Main()
    {
        double div = MathHelper.Calculate(10, 2, (x, y) => x / y);
        Console.WriteLine($"Division: {div}");

        double pow = MathHelper.Calculate(2, 3, (x, y) => Math.Pow(x, y));
        Console.WriteLine($"Exponentiation: {pow}");

        double multiply = MathHelper.Calculate(4, 5, (x, y) => x * y);
        Console.WriteLine($"Multiplication: {multiply}");
    }
}
```

- **Lambda expressions**بتزود **التعبيرية (expressiveness)** في الحسابات الرياضية لأنها:
  1. بتخلي الكود أقصر وأوضح بدل ما نكتب ميثود منفصلة لكل عملية.

2. بتسمح بتمرير العمليات كـ **Functions أول-درجة(first-class functions)** ، يعني العملية نفسها تبقى متغير أو باراميتر.
3. مرنة جدًا: نقدر نغير العملية (جمع، قسمة، أس، جذر...) بسطر واحد من غير ما نغير في الكلاس الأساسي.
4. بتخلي الكود أقرب للرياضيات، وده بيحسن القابلية للقراءة خصوصًا في السيناريوهات العلمية والهندسية.

Wafaa Mohammed · أنت

الآن ·

كتير منا أول ما يسمع كلمة Functional Programming يحس إن الموضوع كبير أو معقد، بس الحقيقة إن C# عندها أدوات بتسهل علينا ندخل عالم الـ Functional Style من غير ما نسيب الـ OOP. واحدة من الأدوات دي هي Delegates.

🎯 يعني إيه Delegate؟

الـ Delegate ببساطة هو مرجع لدالة (Reference to a Method).
زي ما المتغير بيشيل قيمة (int أو string)، الـ Delegate بيشيل عنوان دالة وتقدر تستدعيها عن طريقه.

🤷‍♂️ ليه مهم في الـ Functional Paradigm؟

الـ Functional Style بيعتمد إنك تمرر الدوال كأنها بيانات (Functions as Parameters).
وده اللي بيخلينا نكتب كود مرن – قابل لإعادة الاستخدام – وقليل التكرار.
والـ Delegates بيدونا الإمكانية دي.

مثال

```
public delegate void Printer(string msg);

class Program
                                                    }
   static void PrintToConsole(string text) =>
Console.WriteLine(text);

   static void Main()
                                                    }
      Printer p = PrintToConsole;
      p("Hello from delegate!");
                                                    {
                                                     {
```

إيه اللي حصل هنا؟

عرفنا Delegate اسمه Printer يستقبل نص (string).

كتبنا دالة PrintToConsole بتطبع النص.

خزنا الدالة جوه المتغير p.

نادينا على p("Hello from delegate!") ← فاشتغلت الدالة وطبعته.

📌 الـ Delegate بيخلينا نتعامل مع الدوال كأنها بيانات: نخزنها، نمررها، ونغيرها بسهولة.
وده جزء من مفهوم Functional Programming اللي بيخلي الكود أبسط وأكتر مرونة.

#CSharp #Delegates #FunctionalProgramming

1. Parallel Programming وConcurrency

- **Concurrency (التزامن)** يعني إن البرنامج يقدر يتعامل مع أكتر من مهمة في نفس الوقت، زي السيرفر اللي بيتعامل مع آلاف المستخدمين مع بعض. مش شرط يعملهم فعلًا مع بعض، ممكن ينقل بينهم بسرعة.
- **Parallelism (التوازي)** هو فعليًا تشغيل مهام في نفس اللحظة باستخدام أكتر من نواة في البروسيسور. زي ما يكون عندك 4 عمال كل واحد بيشتغل على حاجة مختلفة في نفس اللحظة.

---

2. Unit Testing وTDD

- **Unit Testing**: ده إنك تكتب اختبارات صغيرة تختبر دالة أو كود معين وتتأكد إنه بيطلع النتيجة اللي متوقعة. لو الكود اتغير بعدين، الاختبارات دي هتقولك لو حصل خطأ.
- **TDD (Test-Driven Development)**: هنا الموضوع بيكون بالعكس، بتكتب الاختبار الأول، وبعدها تكتب الكود اللي يخلّي الاختبار ينجح. بعدين تنظف الكود وتحسنه. يعني الاختبارات بتقود عملية كتابة الكود.

---

3. Asynchronous Programming (async/await)

- ساعات في عمليات بتاخد وقت زي تحميل بيانات من الإنترنت. لو البرنامج استنى لحد ما العملية دي تخلص، كل حاجة هتقف.
- عشان كده في لغات زي C# أو JavaScript نستخدم `async` و`await`. الكود بيبان كأنه ماشي خطوة بخطوة، لكن في الحقيقة العملية اللي بتاخد وقت بتشتغل في الخلفية، والبرنامج يكمل شغله عادي لحد ما ترجع النتيجة.

مثال صغير بالـJavaScript:

```
async function getData() {
let data = await fetch("https://api.example.com");
console.log("البيانات وصلت");
}
```

هنا البرنامج هيستنى نتيجة `fetch` من غير ما يوقف باقي الكود.
