بِسۡمِ ٱللَّهِ ٱلرَّحۡمَٰنِ ٱلرَّحِيمِ

دا كل ما يخص Linq

ايه هو Linq : Language integrated query

بمعنى آخر تسمح لنا تقنية LINQ بالقيام بعمليات الاستعلام، وإجراء عمليات الإضافة والحذف والتعديل والبحث والترتيب والتجميع وغير ذلك.

مش فاهم طيب يعنى بكل بساطه كده بكتب بيه كود سى شارب ولكن لما بيتنفذ عند الداتا بيز بيتنفذ ككود سكوال عادى كأنى بعمل كويرى عاديه علشان اجيب الداتا بتاعتى علشان اعرضها أو اشتغل عليها هما عباره عن Extension Methods عددهم 40 واحده وليهم Overloads موجودين ف الكلاص إلى اسمه Enumerable .

## مميزات LINQ تشمل:

1) سهولة الاستخدام : تتيح لك LINQ كتابة استعلامات بيانات بطريقة بسيطة وواضحة داخل الكود، مما يجعلها أسهل من استخدام استعلامات SQL التقليدية.
2) دعم اللغات المتعددة :يمكن استخدام LINQ مع أنواع بيانات مختلفة مثل الكائنات(Objects) ، XML، SQL، وغيرها
3) يقلل من حدوث أخطاء في وقت التشغيل.

يمكن استخدام نوعين مختلفين من الصيغ لكتابة الاستعلامات على البيانات
الفرق بين **Method Syntax** و **Query Syntax** في **LINQ**

### 1) Query Syntax

**Query Syntax**هي الصيغة التي تشبه استعلامات SQL التقليدية.

```
var result = from student in context.Students
where student.Age > 18
select student ;
```

**مميزات:Query Syntax**

- يشبه بشكل كبير استعلاماتSQL ، مما يسهل فهمه للمطورين الذين لديهم خلفية في SQL
- يسهل قراءة الاستعلامات المعقدة مع العديد من الشروط والتصفية والترتيب.

## 2) Method Syntax

Method Syntaxهي الصيغة التي تعتمد على استخدام دوال مثل ( Where, Select, OrderByإلخ) لتنفيذ الاستعلامات. هذه الصيغة تستخدم بشكل أساسي مع مجموعات البيانات في الذاكرة مثل List أو Array

```
var result = context.Students
Where(student => student.Age > 18).
Select(student => student) ;
```

**مميزات: Method Syntax**

- يوفر مرونة أكبر لأنه يعتمد على استخدام دوال يمكن تركيبها معًا.
- يمكن استخدامه مع عمليات معقدة وسلسلة من الدوال.
- هو الأنسب عندما يكون لديك استعلامات تتطلب العديد من العمليات المختلفة مثل ( GroupBy, Join إلخ.

اي الصيغتين أفضل؟

- Query Syntax مناسب عندما يكون الاستعلام بسيطًا أو عند العمل مع مصادر بيانات مثل قواعد البيانات أوXML
- Method Syntax أفضل عندما تكون بحاجة إلى مزيد من المرونة في استخدام عدة دوال في استعلام واحد، أو عندما تحتاج إلى التعامل مع مجموعات بيانات في الذاكرة.

الخلاصة:

- Query Syntax : يشبهSQL ، سهل القراءة عند كتابة استعلامات بسيطة.
- Method Syntax : أكثر مرونة ويمكن استخدامها في استعلامات معقدة باستخدام دوال متعددة.

---

االفرق م بين First && FirstOrDefault ف الشكل ده

Int [] number = [] ;

First : لو راحت ملكتش ولا Element موجود هترمى Exception

FirstOrDefault :لو راحت ملكتش ولا Element موجود هترجع default value اللى هو الصفر

| Method | Returns | Empty Collection | No Matches | More than one match | Stop after first match |
|---|---|---|---|---|---|
| **First** | The first element of the sequence that satisfies a condition | Exception | Exception | The first element of a sequence | Yes |
| **FirstOrDefault** | The first element of the sequence that satisfies a condition, or a specified default value if no such element is found | Default Value | Default Value | The first element of a sequence | Yes |
| ..... | ..... | ..... | ..... | ..... | ..... |
| ..... | ..... | ..... | ..... | ..... | ..... |
| ..... | ..... | ..... | ..... | ..... | ..... |
| ..... | ..... | ..... | ..... | ..... | ..... |

اى الفرق بين المثال الاول والتانى

1) Var result = Cars

.Where (C=>C.Make)

.Last();

2) Var result = Cars

.Last(C=>C.Make);

Where بيمشى من الاول للاخر لحد ما يلاقى اللى ينطبق ع الشرط

Last بيمشى من الاخر للاول لحد ما يلاقى اللى ينطبق ع الشرط

فهنا Last افضل من where علشان performance

Single : يتم استخدام طريقة Single() لإرجاع العنصر الوحيد من المجموعة، والذي يلبي الشرط .في حالة العثور على أكثر من عنصر واحد في المجموعة أو عدم العثور على أي عنصر في المجموعة، فسوف يتم طرح خطأInvalidOperationException"

ما هو الفرق بين طرق First() و Single()

First() يبحث في مجموعة البيانات ويُرجع أول عنصر يتوافق مع الشرط المحدد.

إذا كانت المجموعة فارغة (أي لا تحتوي على أي عناصر)، فإن First() سيرمي خطا من نوع InvalidOperationException.

يمكن استخدامه عندما تتوقع وجود عنصر واحد على الأقل أو تريد العمل مع أول عنصر موجود في المجموعة.

Single() يقوم بالبحث في مجموعة البيانات ويُرجع عنصر واحد فقط.

إذا كانت المجموعة تحتوي على أكثر من عنصر واحد يتوافق مع الشرط المحدد، سيرمي خطا من نوع InvalidOperationException أيضًا.

إذا كانت المجموعة فارغة، أيضًا سيرمي استثناء.

يستخدم Single() عندما تتوقع أن يكون هناك عنصر واحد فقط في المجموعة يتوافق مع الشرط.

**متى تستخدم كل منهما؟**

استخدم First() إذا كنت لا تهتم بعدد العناصر المتوافقة مع الشرط وتريد أول عنصر فقط.

استخدم Single() عندما تتأكد من وجود عنصر واحد فقط يتوافق مع الشرط في المجموعة.

طب لخص الفرق بين SingleOrDefault و FirstOrDefault و LastOrDefaul

**1. SingleOrDefault():**

- المهمة : يبحث عن عنصر واحد فقط يطابق الشرط.
- إذا كانت المجموعة فارغة، ستعيد القيمة الافتراضية (default) لنوع البيانات المعين
- إذا كان العنصر من نوع int، سيتم إرجاع 0.
- إذا كان العنصر من نوع Reference (مثل كائنات)، سيتم إرجاع null. إذا كان هناك أكثر من عنصر يطابق الشرط :يرمى استثناء من نوع InvalidOperationException.
- متى تستخدم؟ : عندما تتوقع أن يكون هناك عنصر واحد فقط يتوافق مع الشرط.

**. LastOrDefault():**

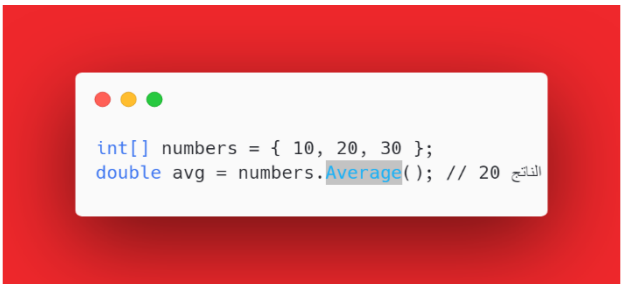
- **المهمة** :يبحث عن آخر عنصر يطابق الشرط.
- **إذا كانت المجموعة فارغة** :يُرجع القيمة الافتراضية.
- **إذا كان هناك أكثر من عنصر يطابق الشرط** :يُرجع آخر عنصر فقط يطابق الشرط.

**. FirstOrDefault():**

- **المهمة** :يبحث عن أول عنصر يطابق الشرط.
- **إذا كانت المجموعة فارغة** :يُرجع القيمة الافتراضية.
- **إذا كان هناك أكثر من عنصر يطابق الشرط** :يُرجع أول عنصر فقط يطابق الشرط.
- **متى تستخدم؟** :عندما تحتاج إلى أول عنصر يتوافق مع الشرط أو إذا كنت تريد قيمة افتراضية إذا لم يوجد أي عنصر.

## ✅ استخدام `Average()` في LINQ

`Average()` تُستخدم لحساب **متوسط العناصر الرقمية** في مجموعة (مثل `List<int>` أو `int[]`).




## ❌ حالات تؤدي إلى InvalidOperationException :

في هذه الحالة لن يظهر InvalidOperationException، بل NullReferenceException لأنك تحاول استدعاء دالة على null < obgect


## ✅ لتجنب هذه الأخطاء:

1. تحقق أن المصفوفة ليست null
2. تحقق أن بها عناصر (Count > 0)

✅ استخدام Max() في LINQ :

Max() تُستخدم للعثور على أكبر قيمة داخل مجموعة أرقام ( زي List<int> أو int[] )

كل اللى بيحصل مع Avrage هو هو Max الاختلاف ف الاستخدام بس كل واحده مختلفة

زى مثلا حالات ظهور InvalidOperationException وازاى اتجنبه

✅ MaxBy() هي تختلف عن Max() في إنها بتسمحلك تحدد أي خاصية (Property) تستخدمها للمقارنة، وبدل ما ترجع أكبر قيمة، هي بترجع العنصر نفسه اللي فيه أكبر قيمة في الخاصية اللي انت اخترتها.

| الدالة | ترجع إيه؟ | تستخدم في إيه؟ |
|---|---|---|
| Max() | ترجع أكبر قيمة | لو عندك أرقام وعايز تعرف أكبر رقم مباشرةً |
| MaxBy() | ترجع العنصر اللي فيه أكبر قيمة | لو عندك (Objects) وعايز ترجع الكائن نفسه |

هنا MaxBy( s => s.Grade ) معناها: هاتلي الطالب اللي عنده أعلى درجة، وارجعلي **الطالب** **نفسه** مش الدرجة.

```
public class Student
{
    public string Name { get; set; }
    public int Grade { get; set;
}

var students = new List<Student>
{
    new Student { Name = "Ahmed", Grade = 85 },
    new Student { Name = "Salma", Grade = 92 },
    new Student { Name = "Youssef", Grade = 78 }
};

// باستخدام MaxBy:
var topStudent = students.MaxBy(s => s.Grade);

Console.WriteLine(topStudent.Name);  // الناتج: Salma
```

**❗ملاحظات مهمة:**

- لو students فاضي MaxBy() → هترجع null

- لو → students = null يحصل NullReferenceException

خلى بالك هنا ف اختلاف بينها وبين max وهو هنا لو شغال ع Array ولقيتها فاضية هترجع Null اما Max هترجع InvalidOperationException

✅ Min & MinBy ===== <<< زيهم زى Max & MaxBy

### ✅ أولًا Split() :

Split() بتُستخدم لتقسيم نص معين (string) إلى مجموعة كلمات أو أجزاء، بناءً على فاصل (زي المسافة أو فاصلة أو أي رمز).

```
string sentence = "I love programming";

string[] words = sentence.Split(' ');
// النتيجة: ["I", "love", "programming"]
```

### ✅ ثانيًا Reverse() :

Reverse() بتُستخدم لعكس ترتيب العناصر في مجموعة زي مصفوفة أو List أو String بعد تحويله لمصفوفة.

```
int[] numbers = { 1, 2, 3 };
Array.Reverse(numbers);
// النتيجة: [1 ,2 ,3]
```

```
                                Select
ApplicationDbcontext _context = new ApplicationDbcontext();
var Blogs = _context.posts.Select(m=> new {PostId=m.PostId,Tital=m.Tital}).ToList();
foreach (var item in Blogs)
   Console.WriteLine($"ID: {item.PostId}:{item.Tital}");


                        Distinct
ApplicationDbcontext _context = new ApplicationDbcontext();
var Blogs = _context.posts.Select(m => new { cont = m.Content, Tital = m.Tital }).Distinct().ToList();
foreach (var item in Blogs)
    Console.WriteLine($"ID: {item.cont}:{item.Tital}");


                    Skip && Take  20 الى 11 هنا بقوله ابدء من
ApplicationDbcontext _context = new ApplicationDbcontext();
var Blogs = _context.posts.Skip(11).Take(20).ToList();
foreach (var item in Blogs)
    Console.WriteLine(item);
```

## ✅ ثانيًا Chunk() :

Chunk() تُستخدم لتقسيم List إلى مجموعات صغيرة **(Chunks)** كل مجموعة لها حجم معين.

```
int[] numbers = { 1, 2, 3, 4, 5, 6, 7 };

// تقسيم الأرقام إلى مجموعات تحتوي على 3 عناصر
var chunks = numbers.Chunk(3);

foreach (var chunk in chunks)
{
    Console.WriteLine($"Chunk: {string.Join(", ", chunk)}");
}

Chunk: 1, 2, 3
Chunk: 4, 5, 6
Chunk: 7
```

**✅ الخلاصة:**

- Chunk() تُقسّم البيانات إلى أجزاء صغيرة.
- تُستخدم مثلاً لعرض البيانات في صفحات، أو تجزئة قائمة طويلة.

✅ أولاً: ما هي `Take()`؟

`Take()` تعيد اول n عنصر من المجموعه

📌 الهدف:

لو عندك قائمة طويلة، وتبغى تأخذ أول X عناصر منها فقط.

✅ Skip(n) يتجاهل أول n عنصر ويعيد العناصر المتبقية بعدهم .

هى هى Take

```
List<int> numbers = new List<int> { 10, 20, 30, 40, 50, 60 };

// نأخذ أول 3 عناصر فقط
var firstThree = numbers.Take(3);

foreach (var num in firstThree)
{
    Console.WriteLine(num);  // 10,20,30
}
```

الفرق بين Where و TakeWhile

Where : بتمشي على كل العناصر وأي عنصر يحقق الشرط بتاخده والباقي بتسيبه.

بتفلتر العناصر بناءً على شرط معين على كل العناصر في المجموعة، وتُرجع العناصر اللي بتحقق الشرط بغض

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 1, 5, 0, 6 };

var result = numbers.Where(n => n > 2);

foreach (var num in result)
{
    Console.WriteLine(num); // 3 4 5 6
}
```

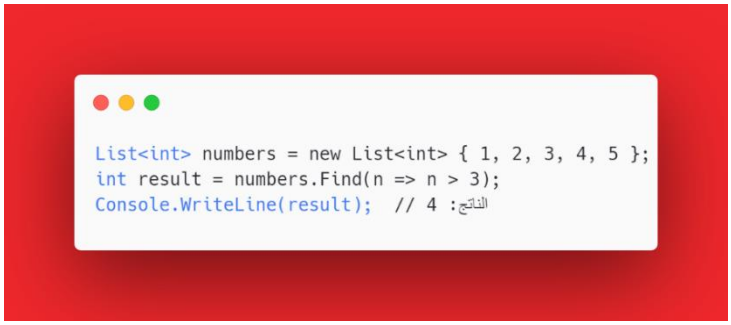
TakeWhile : بتاخد العناصر اللي بتحقق الشرط وأول ما تلاقي عنصر مش محقق الشرط بتقف ومتكملش حتى لو فيه عناصر بعده بتحقق الشرط.

```
List<int> numbers = new List<int> { 1, 2, 3, 0, 4, 5, 6 };

var result = numbers.TakeWhile(n => n > 0);

foreach (var num in result)
{
    Console.WriteLine(num);  // 1 2 3
}
```

الفرق بين Find و :: FirstOrDefault

Find<T>DbSet لاحظ قصدى ايه

Find بيدور في الميموري لو لقى في الميموري بيرجع العنصر ولو ملقاش في الميموري بيروح علي الداتا بيز لو لقي بيرجعة ولو ملقاش بيرجع null



```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
int result = numbers.Find(n => n > 3);
Console.WriteLine(result);  // 4 :الناتج
```

FirstOrDefault : بيروح على الداتا بيز علي طول حتي لو العناصر موجوده علي مستوي الميموري ولو ملقاش بيرجع null .

✅ كيف يمكن تحسين استعلامات LINQ من حيث الأداء؟

a) يمكنك استخدام IQueryable بدلاً من IEnumerable لتحسين الأداء مع قواعد البيانات.
b) تجنب الاستعلامات الزائدة أو غير الضرورية.
c) استخدم Select بدلاً من جلب كل الحقول لتقليل البيانات المنقولة.
d) انتبه إلى عمليات "Deferred Execution" و "Immediate Execution" لأن الأخيرة قد تؤدي إلى استعلامات إضافية غير مرغوب فيها.

🎯يعني إيهDeferred Execution ؟

ببساطة جدًا:

LINQ مش بيجيب البيانات على طول لما تكتب الاستعلام، هو بيستنّى لحد ما تطلب البيانات فعلاً.

يعني تقدر تقول كده:

الاستعلام مكتوب، بس مش شغال دلوقتي... هيشتغل لما تحتاج البيانات فعلاً.

🧠طيب نفهمها بمثال واقعي:

تخيل إنك كتبت استعلام في:LINQ

```
var query = _context.Brands.Where(b => b.BrandId > 4);
```

هل في اللحظة دي بيروح يجيب البيانات من قاعدة البيانات؟
❌ لأ خالص!

هو بس حفظلك شكل الاستعلام ...لكن ما نفّذوش.

🔵**إمتى الاستعلام يتنفذ فعليًا؟**

لما تعمل حاجة زي:

- foreach (var item in query) ← يعني بدأت تستخدم البيانات
- query.ToList() ← يعني قلت له "هاتلي البيانات كلها في ليست"
- query.First() ← يعني عايز أول عنصر
- query.Count() ← عايز تعرف عدد النتائج

في اللحظة دي، الاستعلام بيتنفذ فعلًا، وبيروح لقاعدة البيانات يجيب البيانات


🤔طيب إيه فايدة التأجيل (Deferred Execution) ده؟

✅ تقدر تعدّل الاستعلام قبل ما يتنفذ.


يعني نقدر نحط الشروط على مراحل، وكلها تتجمع وتتعمل في استعلام واحد لما نطلب البيانات فعلاً.

✅ **يحسّن الأداء**

لإنك مش بتنفذ الاستعلام 10 مرات... بتنفذه مرة واحدة بس وقت الحاجة.

**ما هو "Deferred Execution" في LINQ ؟**

- **إجابة نموذجية:**

- **Deferred Execution** يعني أن الاستعلام لا يتم تنفيذه عند تعريفه، بل فقط عندما تطلب البيانات فعلاً(إحضار البيانات من قاعدة البيانات) (مثل استخدام foreach أو ToList() أو First() أو Count(). أما إذا كنت تستخدم دوال **ترشيح** مثل OrderBy(), Select(), Where()، فإن الاستعلام يعتمد على **Deferred Execution** لأنه لم يتم تنفيذه حتى الآن.
- يسمح لك هذا بتعديل الاستعلام قبل أن يتم تنفيذه فعلاً، مما يتيح لك تحسين الأداء وتوفير مرونة أكبر عند التعامل مع البيانات.

- لماذا Deferred Execution مفيد؟

- & تحسين الأداء :بما أن الاستعلام لا يتم تنفيذه فورًا، فإنك تستطيع تعديل الاستعلام بسهولة قبل أن تُنفذ. يمكنك إضافة شروط أو تعديلات أخرى على الاستعلام قبل أن يتم جلب البيانات.
- & التأخير في جلب البيانات :يساعدك ذلك على تجنب جلب بيانات غير ضرورية إلا إذا كنت في حاجة إليها بالفعل.

**ما هو "Immediate Execution" في LINQ ؟**

- **إجابة نموذجية:**

  - "Immediate Execution" تعني أن الاستعلام يتم تنفيذه فورًا عند استدعائه. يتم استخدام ToList(), ToArray(), Count(), First(), إلخ، لإجبار LINQ على تنفيذ الاستعلام فورًا
  - الاستعلام بيتنفذ فورًا في لحظة ما تكتبه أو تناديه، وبيتم جلب البيانات من قاعدة البيانات أو من الذاكرة في نفس اللحظة.

✅ مثال عملي:

```
var query = _context.Products.Where(p => p.Price > 100).ToList() ;
```

✅ متى يحصل Immediate Execution ؟

لما تستخدم دوال بتنفّذ الاستعلام فورًا زي:

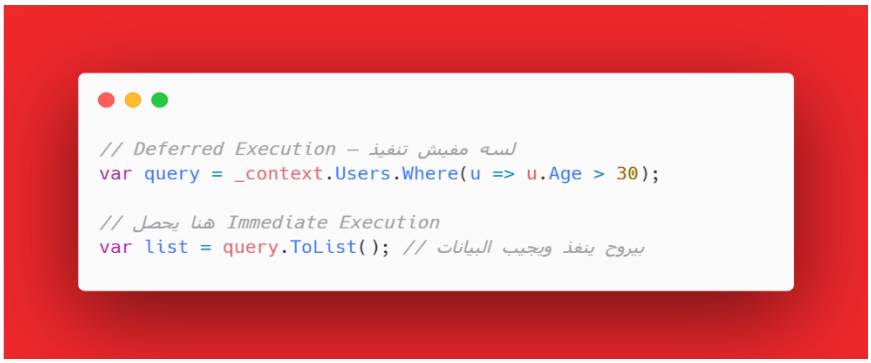
| الدالة | الوظيفة |
|---|---|
| ToList() | يحوّل النتيجة إلى List ، وينفذ الاستعلام فورًا |
| ToArray() | يحوّل النتيجة إلى Array |
| Count() | يحسب عدد العناصر وينفذ الاستعلام |
| First() | يجيب أول عنصر فورًا |
| FirstOrDefault() | يجيب أول عنصر أو null |
| Single() | يجيب عنصر واحد ويتأكد إنه الوحيد |
| Any() | يتحقق هل فيه عناصر ولا لأ |

✅ طيب إيه الفرق بين Immediate و Deferred Execution ؟

| المقارنة | Deferred Execution | Immediate Execution |
|---|---|---|
| وقت التنفيذ | لاحقًا – عند طلب البيانات فعليًا | فورًا – بمجرد كتابة الاستعلام |
| الاستخدام | Where(),Select() فقط | ToList(),First(),Count()... إلخ. |

| المقارنة | Deferred Execution | Immediate Execution |
| --- | --- | --- |
| الفائدة | يسمح بتعديل الاستعلام قبل التنفيذ | مفيد لو عايز النتيجة فورًا |
| التنفيذ علىDB ؟ | لأ، إلا لما تطلب النتيجة | أيوه، ينفذ على طول |

✅ مثال توضيحي سريع:




1) Take(2) :هات أول 2 عنصر || 2) TakeLast(2) : هات آخر 2.

3) Skip(2) : سيب أول 2 وهات الباقي || 4) SkipLast(2) : سيب آخر 2.

* TakeWhile(Condation ) : هات لحد ما الشرط يبقى False

هنا النتيجة 10 فقط

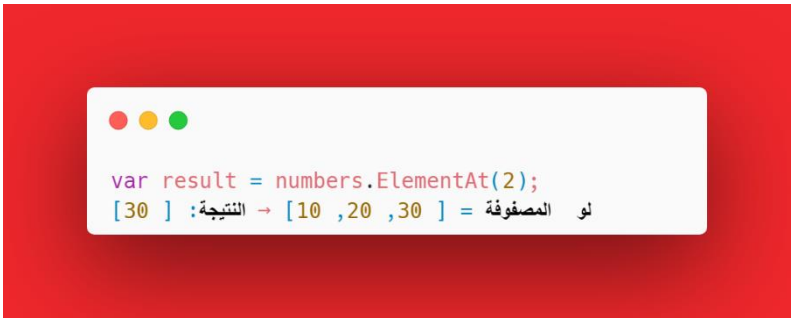
```
class Program
{
    static void Main()
    {
       var numbar = new[] { 10, 30, 20, 10 };
       var xx = numbar.TakeWhile(m => m < 30);
       foreach (var item in xx)
         {
           Console.WriteLine(item);  // output 10
         }
    }
}
```

* SkipWhile(Condation) : سيب لحد ما الشرط يبقى False وبعدها هات الباقي

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

       var num2 = new[] { 10,20,30,40,10 };
       var xx = num2.SkipWhile(m => m < 30);
       foreach (var item in xx)
       {
           Console.WriteLine(item);  // output 30,40,10
       }
    }
}
```

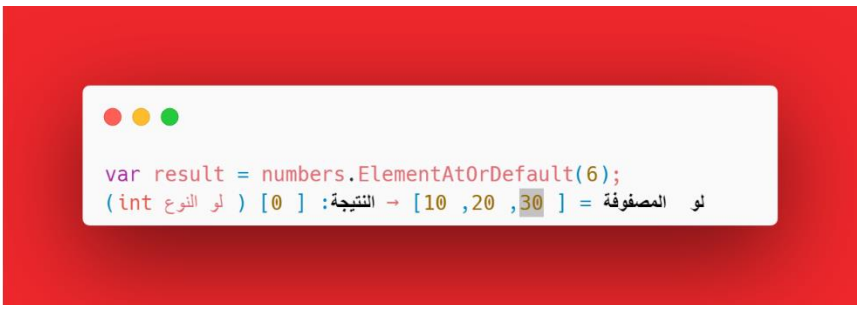
* ElementAt(index) : هات العنصر في مكان معين ولو مش موجود هيديك Error

هات العنصر اللي في المكان رقم index لو مش موجود ← يرمي استثناء.(Error)




ElementAtOrDefault(index) : هات العنصر ولو مش موجود هيرجع Null

نفس ElementAt بس لو العنصر مش موجود ← يرجّع القيمة الافتراضية null أو 0 حسب النوع.




**ما الفرق بين IEnumerable<T> و IQueryable<T> ؟**

- **إجابة نموذجية:**
  - IEnumerable<T> هو interface تستخدم للبيانات التي يتم تحميلها بالكامل في الذاكرة وتسمح بعمليات التكرار عبر هذه البيانات.

◈تخيل معايا:

عندك كتاب مطبوع فيه كل المنتجات (Products)
يعني كل البيانات موجودة قدامك في الكتاب، وانت بتقلب صفحة صفحة يعني(in memory –)

### 🔷يعني إيه عمليًا؟

`IEnumerable<T>` = بيشتغل على البيانات اللي موجودة فعلًا في memory
يعني لازم كل البيانات تكون اتحملت من قبل كده، وانت بس بتتصفحها أو بتفلترها.

- الأداء ضعيف مع Large Data Sets لأنه بيستهلك Memory أكتر وبيسبب بطء.

- مش بيحول LINQ Queries لـ SQL، وده يخليه مش مناسب للتعامل مع Databases

```
List<int> numbers = new List<int> { 10, 20, 30, 40, 50 };

var result = numbers.Where(n => n > 20); // هنا الفلترة بتحصل في الذاكرة

foreach (var n in result)
{
    Console.WriteLine(n);
}
```

البيانات موجودة بالكامل، وبتفلتر عليها في جهازك .

### ثانيًا: يعني إيه IQueryable<T>؟

IQueryable<T> مش بيحمّل البيانات كلها، هو بيحوّل الاستعلام بتاعك إلى SQL وبيبعته للسيرفر مثلاً(SQL Server )، وهناك يحصل الفلترة، وبعدين يجيب لك فقط البيانات اللي انت محتاجها.

- أفضل في الأداء مع Large Data Sets لأنه بيقلل Memory Usage
- بيستخدم Lazy Loading ، يعني الداتا مش بتتحمل غير لما تحتاجها.

```
var query = _context.Products.Where(p => p.Price > 1000);

// لسه مفيش حاجة حصلت، الاستعلام لسه مكتوب

var list = query.ToList(); // هنا بيتحول الاستعلام لـ SQL وبتنفذ على قاعدة البيانات
```


**الخلاصة البسيطة جدًا:**

IEnumerable<T> = اشتغل على بيانات موجودة عندك في الجهاز

IQueryable<T> = اشتغل على بيانات لسه هتجيلك من قاعدة البيانات

الفرق بينهم في مين اللي بينفذ الفلترة؟

- IEnumerable → الفلترة بتحصل عندك
- IQueryable → الفلترة بتحصل في قاعدة البيانات

IEnumerable و IQueryable هما interfaces بتستخدمهم LINQ للتعامل مع البيانات، بس الفرق الأساسي في أين و كيف بيتم تنفيذ الاستعلام.

- IEnumerable بيستخدم مع البيانات اللي موجودة بالفعل في ال Ram، زي List, Array. أول ما أعمل ToList() مثلًا، كل البيانات بتتحمل في ال Ram، وأي فلترة أو ترتيب بعدها بيتم على مستوى التطبيق.(client-side)
- لكن IQueryable بيشتغل بشكل مختلف، هو مصمم للتعامل مع مصادر بيانات خارجية زي قواعد البيانات. الاستعلام بيتحول لـ SQL وبيتنفذ على السيرفر نفسه، فبالتالي الأداء بيكون أحسن، خصوصًا مع البيانات الكبيرة.

كمان IQueryable بيدعم query composition، يعني ممكن أبني استعلامات معقدة (فلترة، ترتيب، pagination ) كلها تتحول وتتنفذ في قاعدة البيانات مباشرة.

## 🎯يعني إيه IQueryable بيدعم Query Composition؟

الـ Query Composition معناها إنك تقدر تبني استعلامك خطوة خطوة — كل خطوة بتضيف فلترة أو ترتيب أو تحديد عدد النتائج — وفي الآخر كل الخطوات دي بتتحول مع بعض لاستعلام SQL واحد ويتنفذ في قاعدة البيانات.

🧠اللي بيحصل هنا:

أنت كتبت الكود خطوة بخطوة، لكن الـ IQueryable ما نفذش حاجة لسه هو بس بيجهز الاستعلام، ولما توصل لـ ToList() في الآخر، ساعتها:

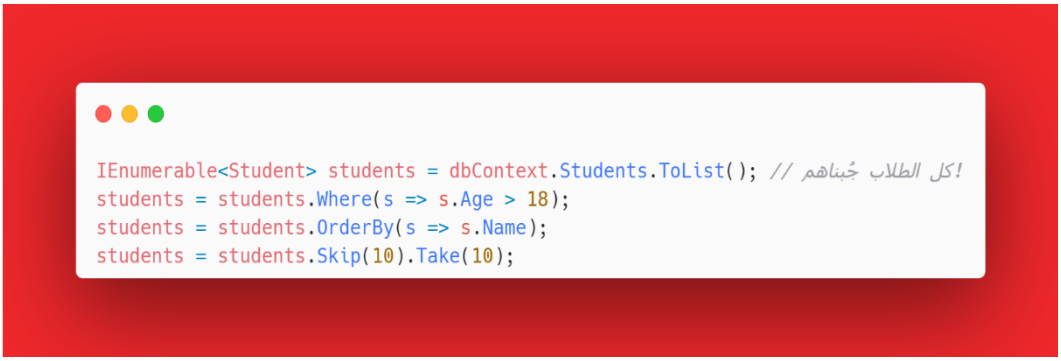
```
IQueryable<Student> students = dbContext.Students;

// فلترة
students = students.Where(s => s.Age > 18);

// ترتيب
students = students.OrderBy(s => s.Name);

// pagination (التخطي والتحديد)
students = students.Skip(10).Take(10);

// تنفيذ الاستعلام
var result = students.ToList();
```

📜كل الفلاتر والـ OrderBy والـ Skip/Take بتتحول لاستعلام SQL واحد زي:

```
SELECT * FROM Students
WHERE Age > 18
ORDER BY Name
OFFSET 10 ROWS FETCH NEXT 10 ROWS ONLY
```

⚠️طب إيه لو كنت استخدمت IEnumerable؟

لو كنت كتبت:




هنا، كل الطلاب اتحملوا الأول في الذاكرة باستخدام ()ToList.، وبعدين بقى التصفية والترتيب والـ pagination بتحصل عندك في الكود — وده سيئ جدًا لو البيانات كتير.

Pagination == ترقيم

✅الخلاصة اللي تقولها في الانترفيو:

لما أستخدم IQueryable، أقدر أبني الاستعلام على مراحل — فلترة، ترتيب، pagination — من غير ما يتنفذ فورًا. وكل الخطوات دي بتتحول في الآخر لاستعلام SQL واحد يتنفذ في قاعدة البيانات، وده بيحسن الأداء جدًا وبيقلل التحميل على الذاكرة.

IEnumerable ==> فيها نوعين

1- IEnumerable (القديمة – non-generic)

2- IEnumerable<T> (الحديثة – generic) → دي الأفضل لأنها strongly typed.

◈ الفايدة الأساسية

بتخليك تمر على مجموعة عناصر (Collection) من غير ما تهتم هي متخزنة إزاي.

بتشتغل مع: Array,List<T>,Dictionary,Queue, Stack,LINQ queries

Collections اللي انت تعملها بنفسك وتطبق IEnumerable.

📌 هنا foreach بيشتغل لأن List<T> أصلاً بتطبق IEnumerable<T>

```
using System;
using System.Collections.Generic;

class Program
{
    static void Main()
    {
        List<string> names = new List<string>() { "Ali", "Sara", "Omar" };

        // لأن List<T> بتطبق IEnumerable<string>
        foreach (var name in names)
        {
            Console.WriteLine(name);
        }
    }
}
```

**GroupBy** هي دالة في LINQ بتمكنني من تجميع العناصر في مجموعات حسب Proprety معينة زى City) بتكون مفيدة لما أحتاج أعمل إحصائيات أو أنظّم الداتا حسب فئة معينة.

```
class Person
{
    public string Name { get; set; }
    public string City { get; set; }
}

 static void Main()
    {
        var people = new List<Person>
        {
            new Person { Name = "Ali", City = "Cairo" },
            new Person { Name = "Sara", City = "Alex" },
            new Person { Name = "Omar", City = "Cairo" },
            new Person { Name = "Laila", City = "Alex" }
        };

        var grouped = people.GroupBy(p => p.City);

        foreach (var group in grouped)
        {
            Console.WriteLine($"المدينة: {group.Key}");
            foreach (var person in group)
            {
                Console.WriteLine($"- {person.Name}");
            }
        }
المدينة: Cairo
- Ali
- Omar
المدينة: Alex
- Sara
- Laila
```

7 إزاي تعمل GroupBy في LINQ ؟ وامتى تحتاجها؟

GroupBy تُستخدم لتجميع العناصر حسب

تحتاجها :لتجميع بيانات مثل "كل المنتجات حسب الفئة."

📌**ملاحظات:**

- تقدر تستخدم Select بعد GroupBy لو عايز تتحكم في شكل النتائج.
- لو محتاج تعدّ كل مجموعة، يعنى كدا

Cairo = 2 Alex = 2

```
var result = people.GroupBy(p => p.City)
        .Select(g => new { City = g.Key, Count = g.Count() });

foreach (var item in result)
{
    Console.WriteLine($"{item.City}: {item.Count}");
}
```

الفرق بين Union و UnionBy في LINQ

* Union بتستخدم لدمج مجموعتين من العناصر بدون تكرار يعني بتشيل الـ(Duplicates)

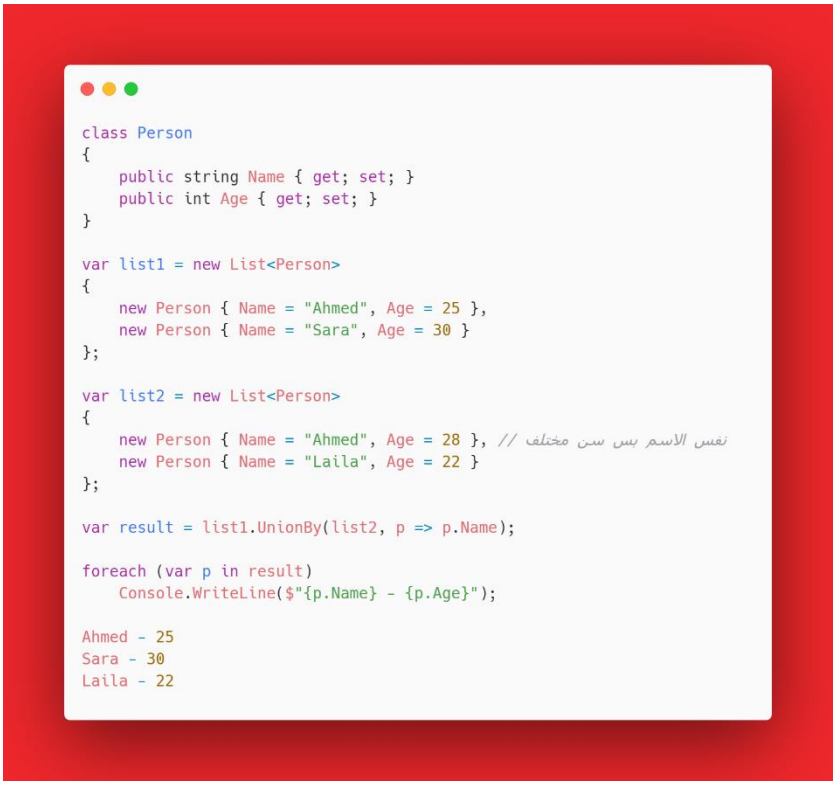
```
var list1 = new List<int> { 1, 2, 3 };
var list2 = new List<int> { 3, 4, 5 };

var result = list1.Union(list2);

foreach (var num in result)
{
    Console.WriteLine(num);  // 1 2 3 4 5    // حذف الرقم 3 المكرر ✅
}
```

UnionBy → لما تكون عايز تتجنب التكرار حسب Proparty معينة من Obgect

🎯المشكلة اللي UnionBy بتحلها:

تخيل إن عندك List 2 من الأشخاص، وكل شخص عنده : Name , Age دلوقتي، أنت عايز تدمج List 2 مع بعض، **بس** : لو فيه شخص مكرر بنفس الاسم، تحسبه مرة واحدة، حتى لو سنه مختلف. وده اللي UnionBy بتعمله!

✅خد "Ahmed" مرة واحدة بس، من أول قائمة (25 سنة)، وتجاهل Ahmedاللي عنده 28 سنة.



```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }
}

var list1 = new List<Person>
{
    new Person { Name = "Ahmed", Age = 25 },
    new Person { Name = "Sara", Age = 30 }
};

var list2 = new List<Person>
{
    new Person { Name = "Ahmed", Age = 28 }, // نفس الاسم بس سن مختلف
    new Person { Name = "Laila", Age = 22 }
};

var result = list1.UnionBy(list2, p => p.Name);

foreach (var p in result)
    Console.WriteLine($"{p.Name} - {p.Age}");

Ahmed - 25
Sara - 30
Laila - 22
```

📦طيب ليه مانستخدمش Union العادي؟

لأن Union العادي يقارن العنصر كامل (يعني الاسم والسن)، ولو فيه اختلاف بسيط يعتبره مختلف!

أما UnionBy بتقول:

"قارن حسب اللي أنا هقولك عليه فقط (الاسم مثلًا)"

✅أولًا: يعني إيه Concat في LINQ ؟

🔷التعريف البسيط:

Concat معناها ضم List 2 مع بعض كما هما، بدون إزالة التكرار.

🧠يعني:

"هات العناصر من أول List ، وبعدين زوّد عليهم عناصر List التانية زي ما هي."

✅ لاحظ: الرقم 3 ظهر مرتين، لأنه موجود في الليستتين Concat .ما بيشيلش التكرار.

```
var list1 = new List<int> { 1, 2, 3 };
var list2 = new List<int> { 3, 4, 5 };

var result = list1.Concat(list2);

foreach (var num in result)
{
    Console.WriteLine(num);  // 1 2 3 4 5
}
```

✅ أولاً: يعني إيه Zip في LINQ ؟

🔷 التعريف البسيط Zip :يعني الدمج عنصر بعنصر من قائمتين، يعني:

خد أول عنصر من List الأولى، وركّبه مع أول عنصر من List التانية بعدين التاني مع التاني، التالت مع التالت... وهكذا.

```
var names = new List<string> { "Ahmed", "Sara", "Laila" };
var ages = new List<int> { 25, 30, 22 };

var result = names.Zip(ages, (name, age) => $"{name} - {age}");

foreach (var item in result)
{
    Console.WriteLine(item);
}
Ahmed - 25
Sara - 30
Laila - 22
```

✅ كل اسم اتربط بالعمر اللي في نفس الترتيب.

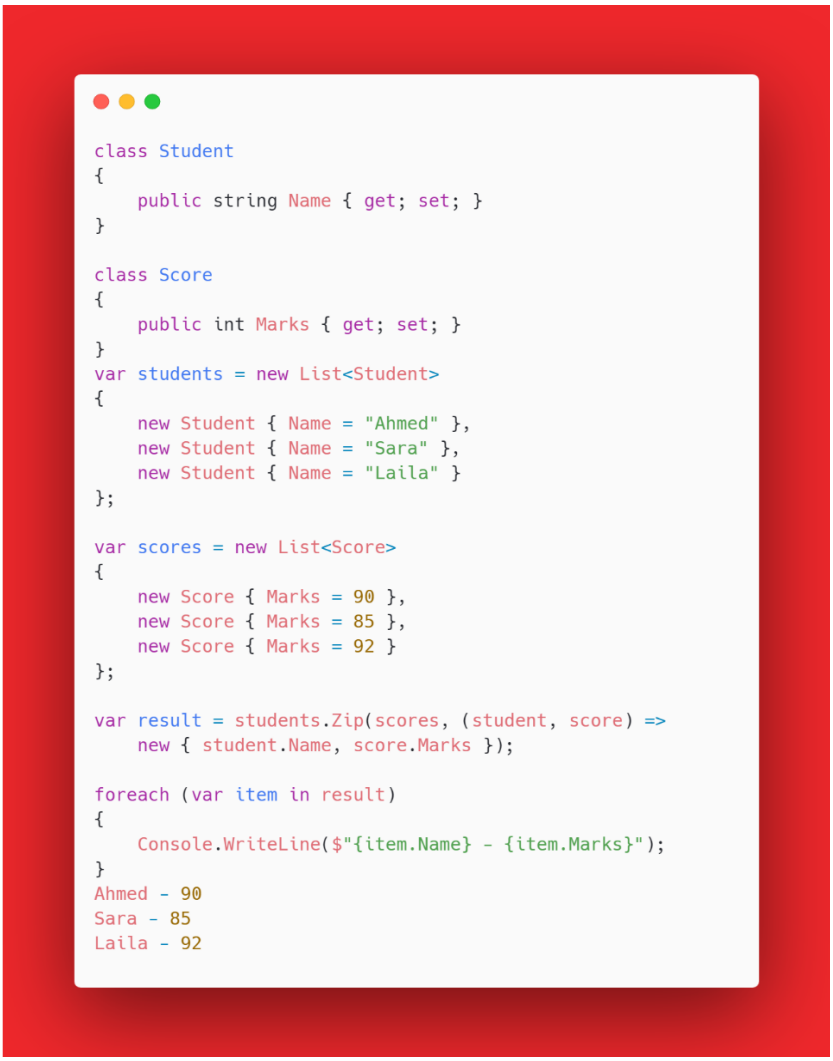
🔥 طيب لو List2 مش نفس الطول؟

Zip بياخد أصغر عدد عناصر في القائمتين .يعني بيوقف أول ما وحدة منهم تخلص.

```
var names = new List<string> { "Ahmed", "Sara" };
var ages = new List<int> { 25, 30, 22 };

var result = names.Zip(ages, (name, age) => $"{name} - {age}");

foreach (var item in result)
{
    Console.WriteLine(item);
}
Ahmed - 25
Sara - 30
```

✅ تجاهل الرقم 22 لأن معهوش اسم يقابله

✅مثال عملي باستخدام كائنات:



```
class Student
{
    public string Name { get; set; }
}

class Score
{
    public int Marks { get; set; }
}
var students = new List<Student>
{
    new Student { Name = "Ahmed" },
    new Student { Name = "Sara" },
    new Student { Name = "Laila" }
};

var scores = new List<Score>
{
    new Score { Marks = 90 },
    new Score { Marks = 85 },
    new Score { Marks = 92 }
};

var result = students.Zip(scores, (student, score) =>
    new { student.Name, score.Marks });

foreach (var item in result)
{
    Console.WriteLine($"{item.Name} - {item.Marks}");
}
Ahmed - 90
Sara - 85
Laila - 92
```

✅يعني إيه ToArray() ببساطة جدًا؟ يعنى بتحول ال Collaction بتاعك الى Array

لما يكون عندك مجموعة عناصر زي List أو نتيجة استعلام(LINQ )، وعايز تحولها لمصفوفة — (Array) يعني شكلها يكون بالشكل ده :

بتستخدم ToArray() عشان تعمل كده int[] arr = {3 ,2 ,1} ;

var numbers = new List <int> { 5 ,4 ,3 ,2 ,1 } ;

// خدت الأرقام الزوجية وحولتهم لمصفوفة

var evenNumbers = numbers.Where(n => n % 2 == 0).ToArray;()

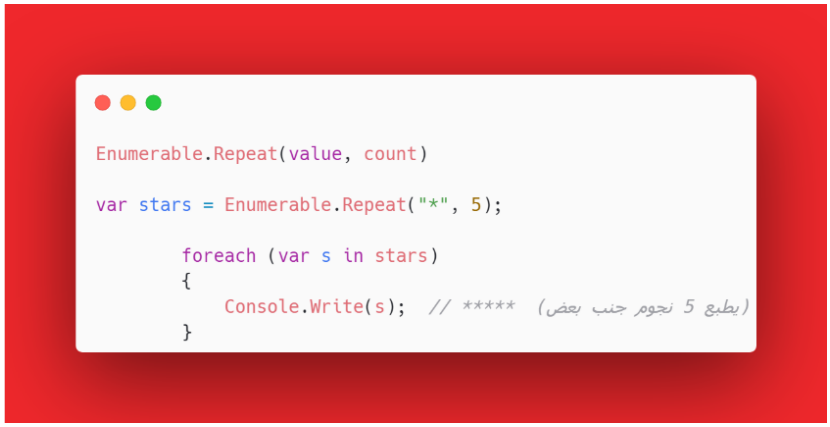
ToDictionery بتحول ال Collaction بتاعك الى Dictionery

```
var numbers = new List<int> { 1, 2, 3, 4, 5 };

var evenNumbers = numbers.ToDictionary(c=>c.id);
foreach (var item in evenNumbers)
{
    Console.WriteLine($"Key { item.key}");
    Console.WriteLine($"Value { item.value}" );

}
```

* foreach ما يشتغلش على null، لكن يشتغل على مجموعة فاضية

✅ يعني إيه Enumerable.Repeat()؟

Repeat هي دالة في LINQ بتكرر قيمة معينة عدد معين من المرات



```
Enumerable.Repeat(value, count)

var stars = Enumerable.Repeat("*", 5);

        foreach (var s in stars)
        {
            Console.Write(s);  // *****  (يطبع 5 نجوم جنب بعض)
        }
```

🎯 أولًا: نفهم الفكرة قبل الكود

✅ Select : بتستخدمها لما كل عنصر بيرجع قيمة واحدة يعني: عنصر ⇨ قيمة وحدة

✅ SelectMany : بتستخدمها لما كل عنصر بيرجع مجموعة List أو Array ، وانت عايز تفرد كل Lists دي في List واحدة كبيرة. يعني: عناصر ⇨ مجموعة من العناصر ⇨ فردها كلها

```
public class Student
{
    public string Name { get; set; }
    public List<string> Courses { get; set; }
}
var students = new List<Student>
{
    new Student { Name = "Ahmed", Courses = new List<string> { "Math", "Physics" } },
    new Student { Name = "Sara", Courses = new List<string> { "Biology" } },
    new Student { Name = "Ali", Courses = new List<string> { "Math", "Chemistry" } }
};

                                : Select أولًا ✅

var result = students.Select(s => s.Courses);
// Result
List of Lists:
[
  ["Math", "Physics"],
  ["Biology"],
  ["Math", "Chemistry"]
]
```

كل عنصر هنا هو قائمة كورسات لطالب واحد.

```
public class Student
{
    public string Name { get; set; }
    public List<string> Courses { get; set; }
}
var students = new List<Student>
{
    new Student { Name = "Ahmed", Courses = new List<string> { "Math", "Physics" } },
    new Student { Name = "Sara", Courses = new List<string> { "Biology" } },
    new Student { Name = "Ali", Courses = new List<string> { "Math", "Chemistry" } }
};

                                : Select أولًا ✅

var result = students.SelectMany(s => s.Courses);
// Result
List of all courses:
[ "Math", "Physics", "Biology", "Math", "Chemistry" ]
```

يعني دمج كل الكورسات في قائمة واحدة.

💬الفرق تقولّه ببساطة في الانترفيو:

Select بيستخدم لما كل عنصر بيرجع قيمة وحدة (أو حتى قائمة، لكن بتفضل داخلها)، أما SelectMany بيفرد القوائم دي كلها في قائمة واحدة كبيرة.

في حالات زي طالب عنده كورسات، لو عايز كورسات كل طالب لوحده أستخدم Select، لكن لو عايز كل الكورسات لكل الطلاب في قائمة وحدة أستخدم SelectMany.

## Filter و Projection

- **Filter** عبر Where() لتحديد العناصر.
- **Projection** عبر Select() لاختيار Propertys

```
List.Where(x => x.Age > 20)
  .Select(x => new { x.Name, x.Age });
```

## Count() vs LongCount()

- Count(): يستخدم – int لـ عدد أقل من 2,147,483,647
- LongCount(): يستخدم – long لـ بيانات ضخمة جدًا

## ToList vs AsEnumerable

- ToList(): ينفذ الاستعلام ويحوله إلىList
- AsEnumerable(): لا ينفذ الاستعلام، لكنه يحوله إلى نوع IEnumerable

## ما هو الفرق بين Take و Skip في LINQ ؟

- **إجابة نموذجية:**
  - Take(n) يعيد أول n عنصر من المجموعة.
  - Skip(n) يتجاهل أول n عنصر ويعيد العناصر المتبقية بعدهم .

## كيف يمكنك استخدام Any و All في LINQ ؟

- إجابة نموذجية:
  - Any تستخدم للتحقق من وجود عنصر واحد أو أكثر يطابق الشرط. وممكن اكتب جوها Condation
  - All: تستخدم للتحقق من أن جميع العناصر تطابق الشرط.

Var خلى بالك هى مش Datatype هى keyword

هى طريقة غير مباشرة لتحديد نوع البيانات

فمن خلال النظر إلى البيانات الموجودة على الجانب الأيمن من بيان المهمة، سيتم تحديد نوع البيانات الموجوده ف اليسار

```
var s1 = "ajf " ;   var f = 3f;   var d = 4d      ;
```

```
Q1:     var total = 100;      var vip = 80;          var x = total >= vip ;
Console.WriteLine(x);
```

```
Q2:                    var s1 = "mohamed" ;          s1+= "reda" ;
Console.WriteLine(s1);
```

* عند استخدام var في C# ، يجب أن يتم تعيين قيمة للمتغير في نفس السطر الذي يتم فيه تعريفه، لأن المترجم يعتمد على هذه القيمة لتحديد نوع المتغير.

```
var x ;       //  خطأ: لا يمكن تعريف المتغير بدون قيمة
```

* المترجم سيحدد نوع المتغير بناءً على القيمة المعينة له في وقت الترجمة

```
var name = "John"; //  المترجم يعرف أن name هو نوع string
```

* لا يمكن تغيير نوع المتغير بعد تعريفه:

```
var x = 10;     x = "Hello"; //  خطأ: لا يمكن تحويل string إلى int
```

var مفيدًا بشكل خاص عندما يكون نوع المتغير معقدًا أو طويلًا،

يمكن استخدام var بدلاً من استخدام List<int> //

```
var list = new List<int>();
```

هي طريقة بتدمج بين لغة البرمجة (زي **#C**) والاستعلامات اللي بتستخدمها عادة في قواعد البيانات (زي **SQL**).
الهدف الأساسي منها هو إنها تبسط عملية التعامل مع الداتا في البرنامج، سواء كانت الداتا دي جاية من **قاعدة بيانات**، أو **XML**، أو حتى **كائنات داخل الكود** زي الـ **List**

LINQ عبارة عن مجموعة من ال **methods** اللي بنسميها **"Operators"** زي Where, Select, OrderBy.
وغيرهم. وعددهم اكتر من 40 حوالى **49** متقسمين ل **13** مجموعة مختلفة كل مجموعة ليها استخدامها
دول بنستخدمهم عشان نقدر نعمل عمليات فلترة، ترتيب، أو تحويل للداتا اللي عندنا.

## 🧠ما هو Except في LINQ ؟

Except هي دالة في LINQ تُستخدم لمقارنة List 2، وترجع العناصر الموجودة في المجموعة الأولى فقط والتي لا توجد في المجموعة الثانية.

يعني ببساطة:
"هاتلي كل العناصر اللي موجودة في List الأولى، ومش موجودة في التانية".

```
var list1 = new List<int> { 1, 2, 3, 4, 5 };
var list2 = new List<int> { 3, 4, 6 };

var result = list1.Except(list2);

foreach (var num in result)
{
    Console.WriteLine(num); // 1 2 5
}
```

العناصر 3 و 4 مشتركة، فتم استبعادها.

## ما الفرق بين Except و SequenceEqual في LINQ ؟

نموذج إجابة:

Except بتستخدم عشان ترجع العناصر الموجودة في مجموعة ومش موجودة في مجموعة تانية. أما SequenceEqual فبتستخدم للمقارنة بين مجموعتين بالكامل والتأكد إنهم متطابقين في القيم والترتيب.

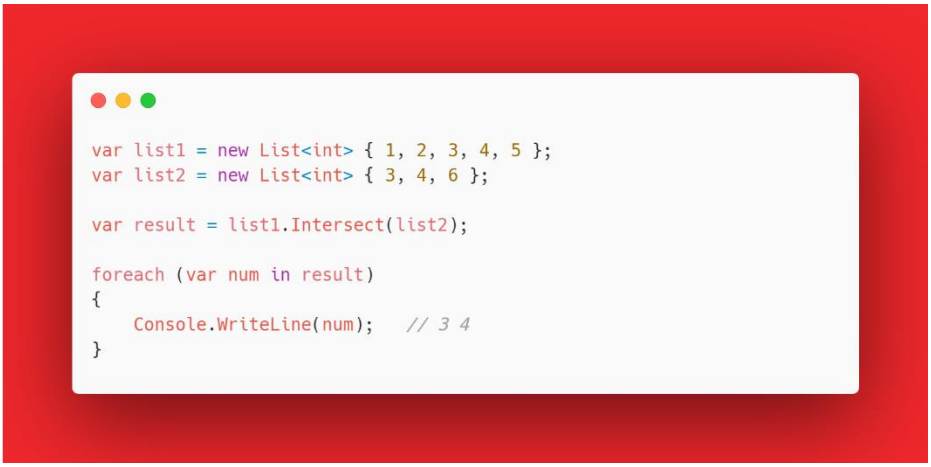
Except بترجع List، و SequenceEqual بترجع bool

```
var list1 = new List<int> { 1, 2, 3 };
var list2 = new List<int> { 1, 2, 3 };
var list3 = new List<int> { 3, 2, 1 };

Console.WriteLine(list1.SequenceEqual(list2)); // ✅ True
Console.WriteLine(list1.SequenceEqual(list3)); // ❌ False (الترتيب مختلف)
```

### 🧠ما هو Intersect؟

Intersectهي دالة في LINQ تُستخدم لاستخراج **العناصر المشتركة** بين مجموعتين.

يعني ببساطة : هاتلي العناصر اللي موجودة **في القايمتين**".



```
var list1 = new List<int> { 1, 2, 3, 4, 5 };
var list2 = new List<int> { 3, 4, 6 };

var result = list1.Intersect(list2);

foreach (var num in result)
{
    Console.WriteLine(num);   // 3 4
}
```

### 🧠ما هي Contains؟

Contains هي دالة تُستخدم علشان تعرف : هل العنصر ده موجود في المجموعة ولا لأ؟

بترجع true أو false


( تلخيص سريع بقاا كدا )

- جاهز يمعلم
- يلا بينا
- هنا يمعلم كل الدوال اللى انت هتحتاجها جمعتهم ليك(مع العلم الحاجات ده تلخيص لباشمهندس زميلنا(محمود عبدالعزيز)

# Master LINQ Methods in C#

دى يمعلم الدوال الخاصة بال Filtering

دى يمعلم الدوال الخاصة بال Ordering


## OrderBy

```
var names = new[] { "Charlie", "Alice", "Bob" };
var sorted = names.OrderBy(n => n);
// Output: { "Alice", "Bob", "Charlie" }
```

## Reverse

```
var numbers = new[] { 10, 20, 30 };
var reversed = numbers.Reverse();
// Output: { 30, 20, 10 }
```

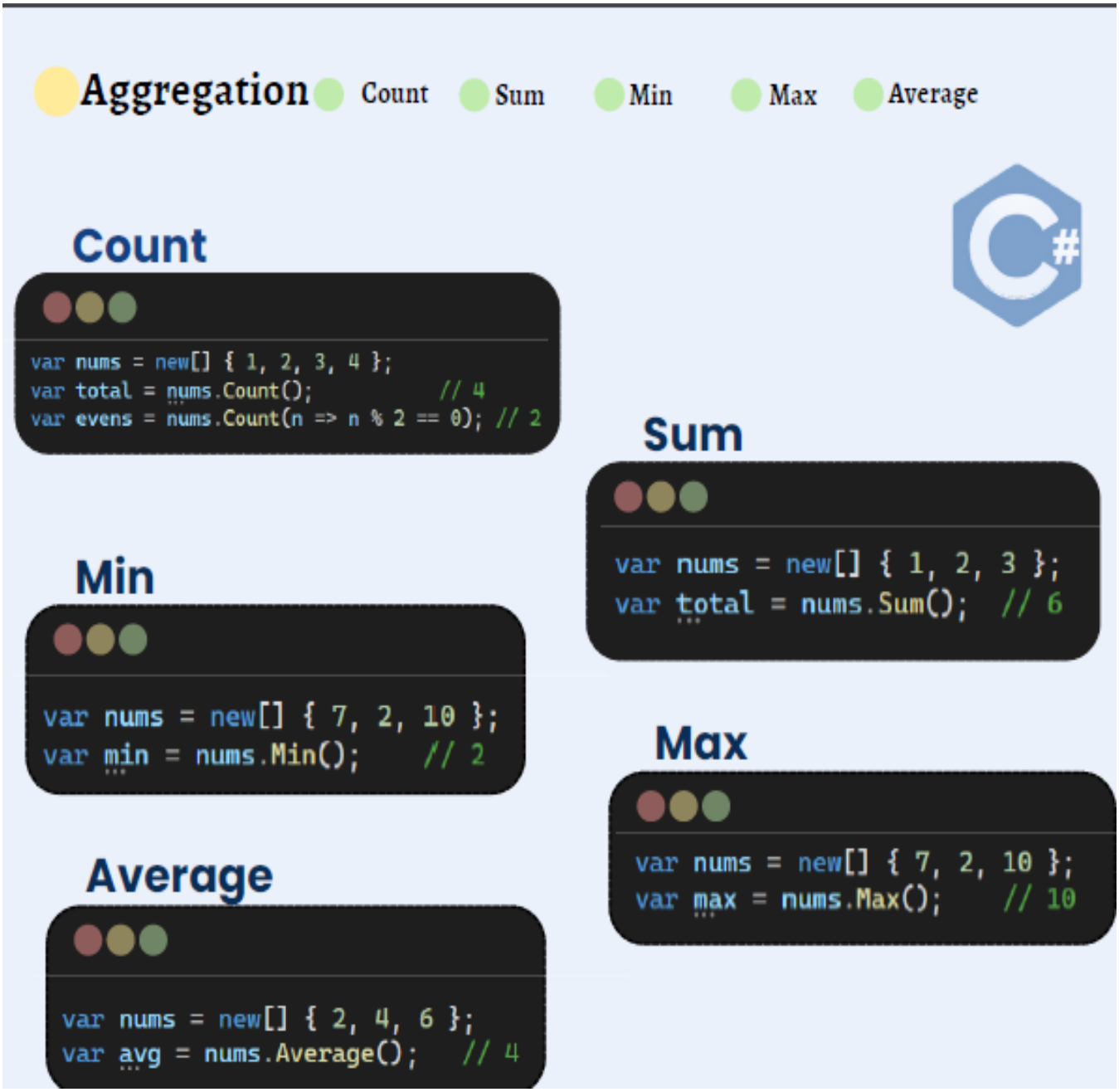
## ThenBy

```
var people = new[]
        { new { Name = "Ali", Age = 30 },
          new { Name = "Ali", Age = 25 },
          new { Name = "Ziad", Age = 20 }};

var sorted = people
    .OrderBy(p => p.Name).ThenBy(p => p.Age);
// Output:
// { Name = "Ali", Age = 25 }
// { Name = "Ali", Age = 30 }
// { Name = "Ziad", Age = 20 }
```

## OrderByDesc

```
var numbers = new[] { 5, 1, 8, 3 };
var sorted = numbers.OrderByDescending(n => n);
// Output: { 8, 5, 3, 1 }
```

دى يمعلم الدوال الخاصة بال Aggregation



دى يمعلم الدوال الخاصة بال Projecting

Projecting Select SelectMany

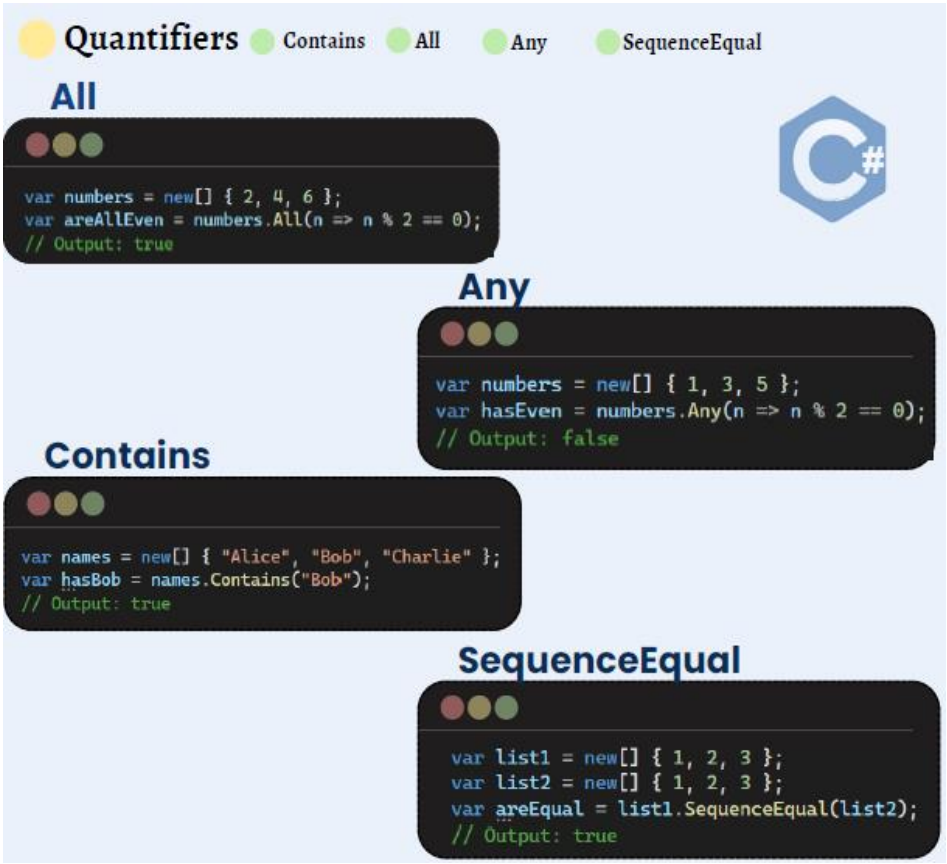
## Select

```
var numbers = new[] { 1, 2, 3 };
var squares = numbers.Select(n => n * n);
// Output: { 1, 4, 9 }
```

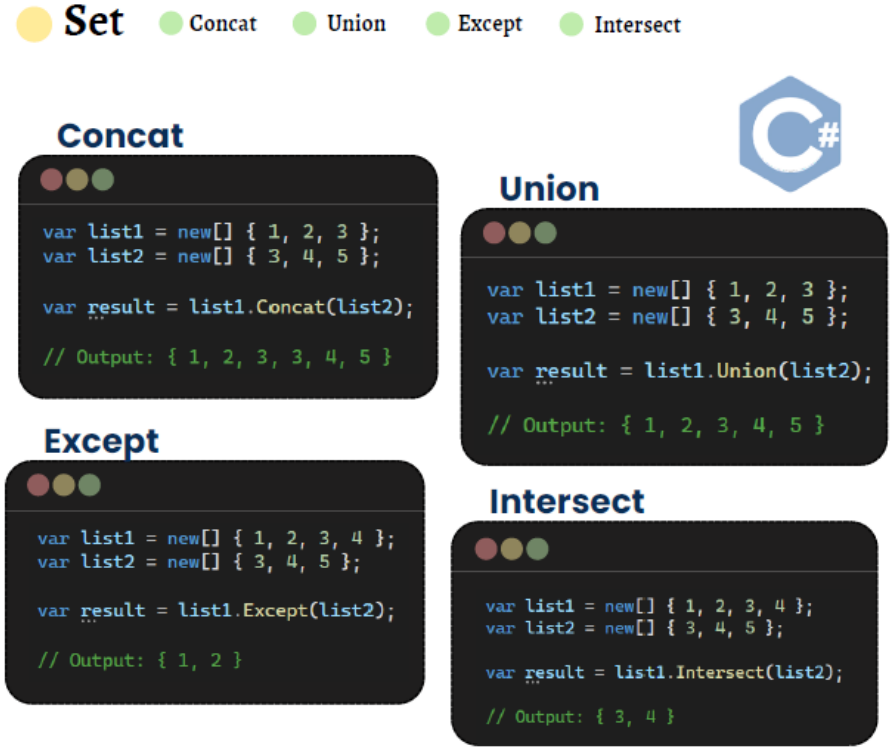
## SelectMany

```
var nested = new[] {  new[] { 1, 2 },
                      new[] { 3 },
                      new[] { 4, 5 }};

var flattened = nested.SelectMany(x => x);

// Output: { 1, 2, 3, 4, 5 }
```

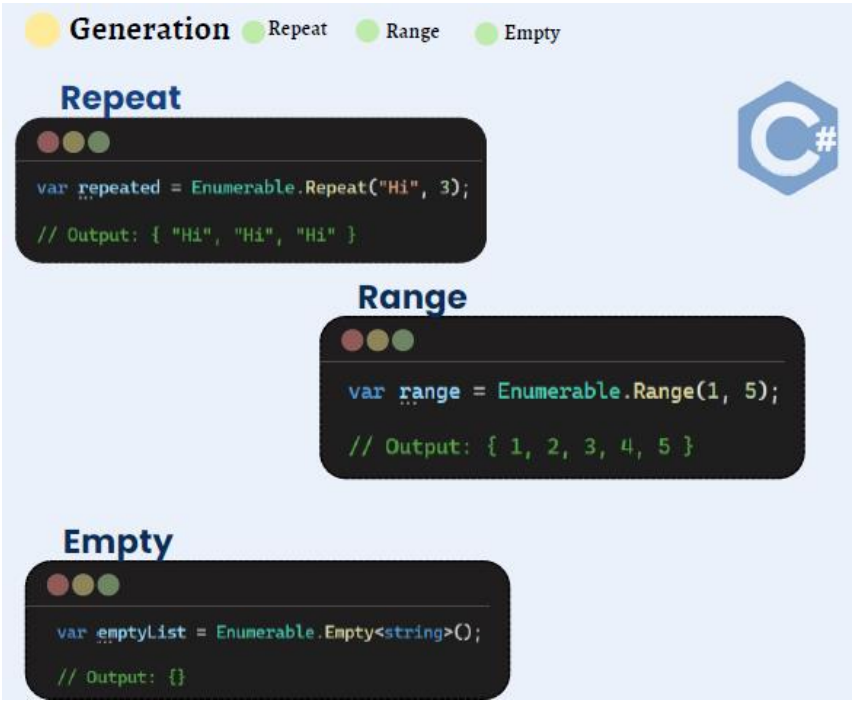
دى يمعلم الدوال الخاصة بال Projecting



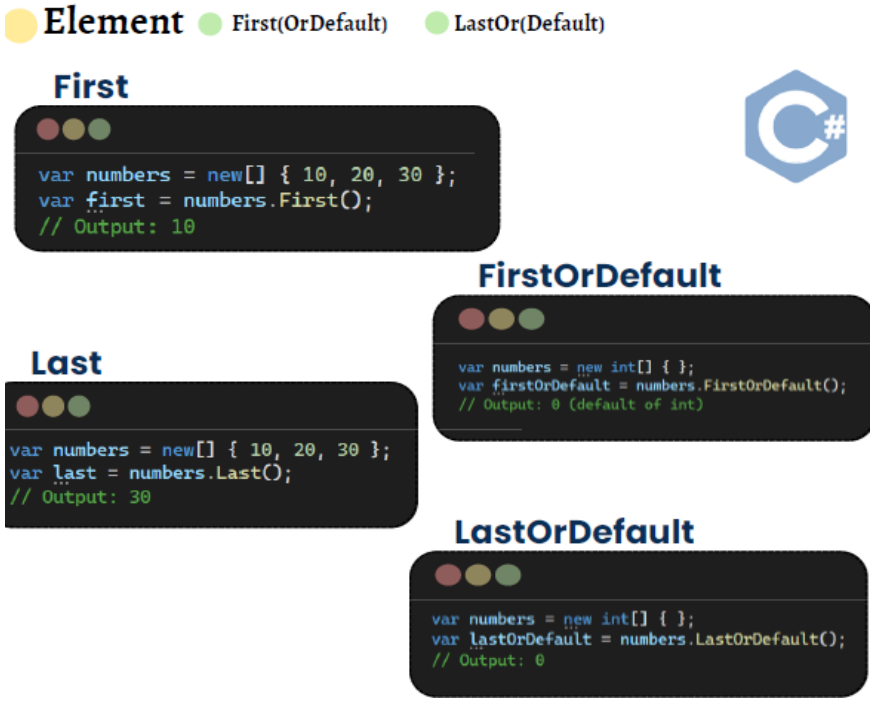
- دى يمعلم الدوال الخاصة بال Set



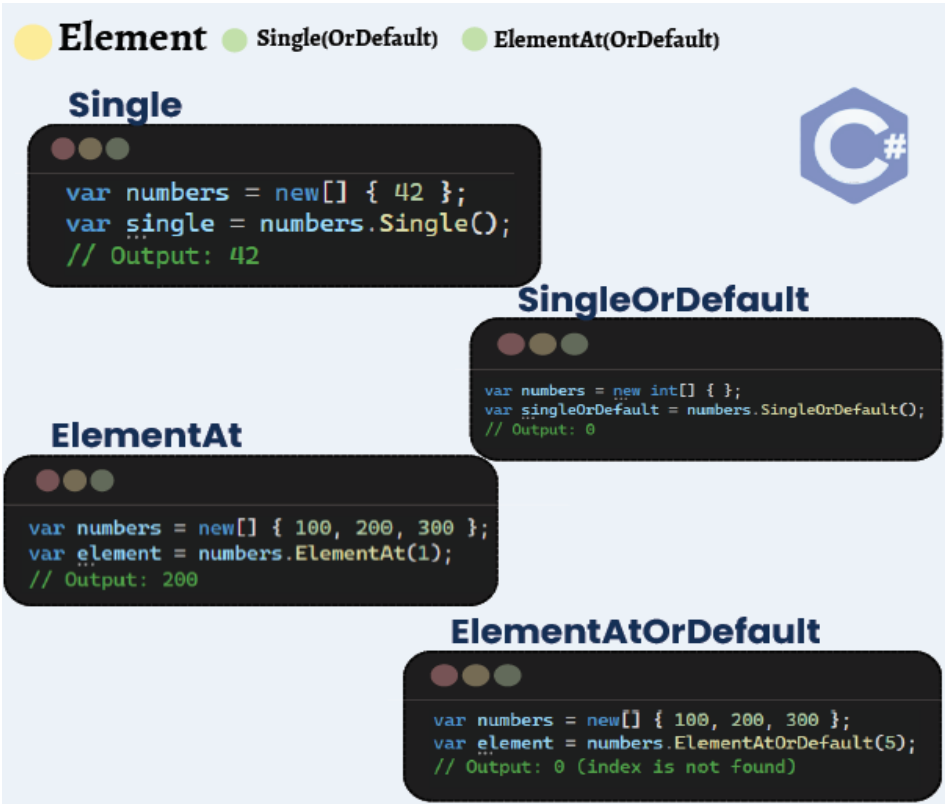

- دى يمعلم الدوال الخاصة بال Generation



- دى يمعلم الدوال الخاصة بال Element




- دى يمعلم الدوال الخاصة بال Element



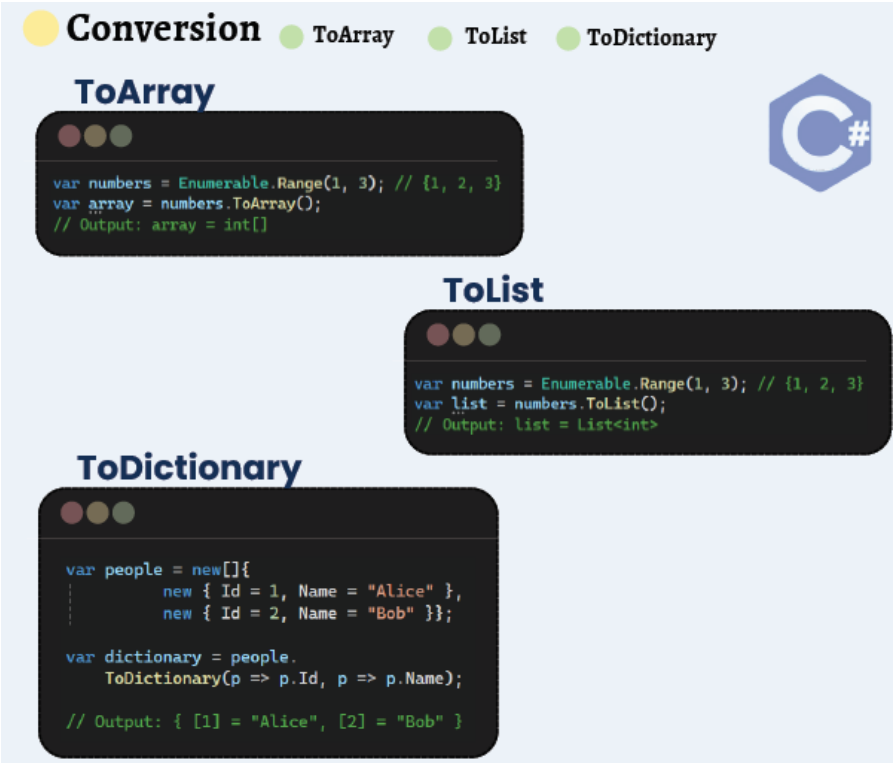
- دى يمعلم الدوال الخاصة بال Grouping

Grouping GroupBy

## GroupBy

```
var students = new[] {
        new { Name = "Alice", Class = "Math" },
        new { Name = "Bob", Class = "Science" },
        new { Name = "Charlie", Class = "Math" },
        new { Name = "David", Class = "Science" },
        new { Name = "Eva", Class = "Math" }};

var grouped = students.GroupBy(s => s.Class);

foreach (var group in grouped)
{
    Console.WriteLine($"Class: {group.Key}");
    foreach (var student in group)
    {
        Console.WriteLine($" - {student.Name}");
    }
}

/* Output
  Class: Math
   - Alice
   - Charlie
   - Eva
  Class: Science
   - Bob
   - David */
```

- دى يمعلم الدوال الخاصة بال Conversion




- دى يمعلم الدوال الخاصة بال Conversion

# Conversion

ToLookup AsEnumerable AsQueryable

## ToLookup

```
var items = new[]{
      new { Category = "Fruit", Name = "Apple" },
      new { Category = "Fruit", Name = "Banana" },
      new { Category = "Vegetable", Name = "Carrot" }};

var lookup = items.ToLookup(i => i.Category);

// Output:
// Fruit => Apple, Banana
// Vegetable => Carrot
```

## AsEnumerable

```
DataTable table = GetDataTable();
var rows = table.AsEnumerable();

// Output: IEnumerable<DataRow>
```

## AsQueryable

```
List<int> list = new List<int> { 1, 2, 3 };
var queryable = list.AsQueryable();

// Output: IQueryable<int>
```