HO
GENT

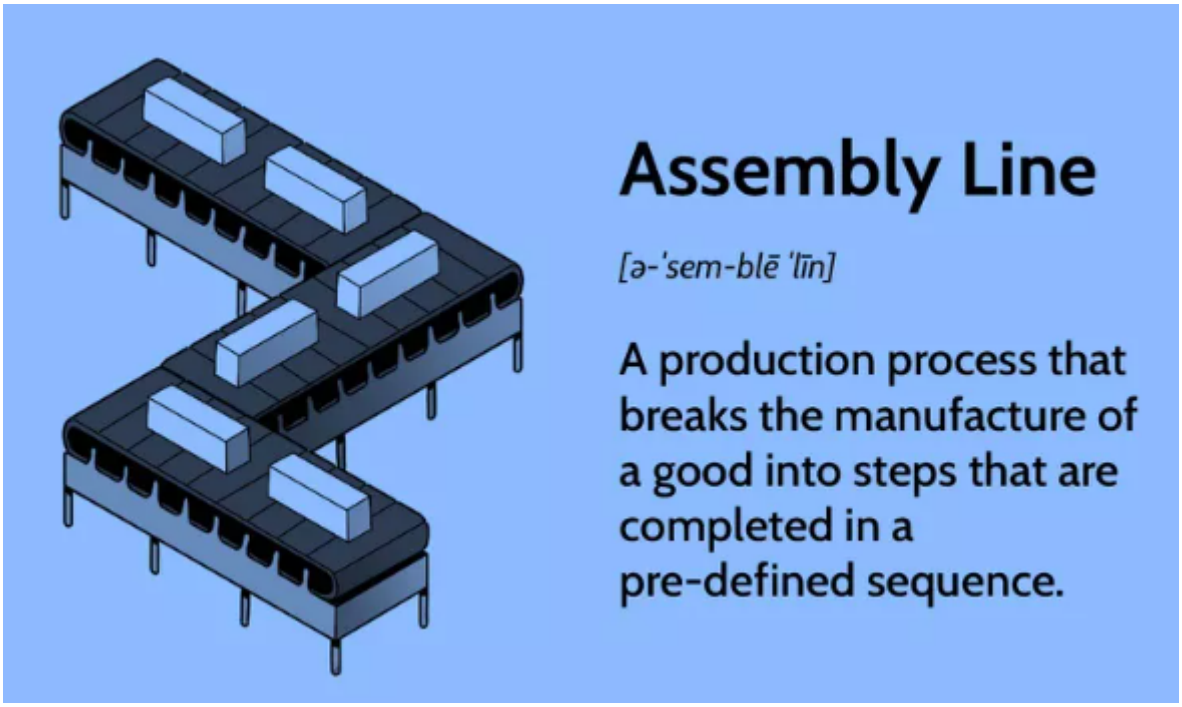
LINQ

Ludwig Stroobant

Table of Contents

1. Het Gebruik van LINQ (Language Integrated Query) - Van Lopende Band naar Code

In dit hoofdstuk zullen we Language Integrated Query (LINQ) introduceren door het te vergelijken met een lopende band in een productiebedrijf. We zullen de concepten en operaties van LINQ illustreren met gedetailleerde voorbeelden, waaronder ook 'Group By', 'Join' en 'Any en All'.



1.1. Inleiding tot LINQ: De Lopende Band

Stel je voor dat je een productiebedrijf hebt en een efficiënte manier nodig hebt om grondstoffen te verwerken, te transformeren en afgewerkte producten te produceren. De lopende band is het essentiële onderdeel van dit proces. Net zoals een lopende band gegevens van de ene stap naar de andere transporteert, biedt LINQ een gestructureerde en efficiënte manier om gegevens te verwerken binnen je C#-programma's.

1.2. De Lopende Band in de Context van LINQ

In ons productiebedrijf is de lopende band als volgt opgebouwd:

- **De Bron (Collectie)**: De lopende band begint met de grondstoffen die moeten worden verwerkt. Dit is onze "collectie" in LINQ, de oorspronkelijke gegevensbron waarop we onze bewerkingen willen uitvoeren.
- **Tussenstations op de Lopende Band (LINQ-methodes)**: De lopende band heeft verschillende tussenstations die specifieke taken uitvoeren, zoals snijden, assembleren en schilderen. In LINQ zijn deze tussenstations de LINQ-methodes, zoals `Where`, `Select`, `OrderBy`, en vele anderen. Elk van deze methodes voert specifieke bewerkingen uit op de gegevens terwijl ze langs de lopende

band bewegen.

- **Het Eindstation (Resultaat ophalen)**: Aan het einde van de lopende band hebben we afgewerkte producten die klaar zijn voor distributie. In LINQ is ons "eindstation" vergelijkbaar met methodes zoals `ToList`, `ToArray`, of andere methodes om het resultaat van onze query-operatie op te halen.

1.3. De Lopende Band in Actie

Laten we eens kijken naar hoe onze lopende band werkt in de context van LINQ:

De Bron (Collectie)

```
List<int> grondstoffen = new List<int> { 5, 12, 8, 15, 20 };
```

Tussenstations op de Lopende Band (LINQ-methodes)

```
var result = grondstoffen
                .Where(item => item > 10) // Alleen items groter dan 10 behouden
                .Select(item => item * 2) // Elk item verdubbelen
                .OrderBy(item => item);  // Sorteer de items in oplopende volgorde
```

Het Eindstation (Resultaat ophalen)

```
List<int> afgewerkteProducten = result.ToList();
```

Op onze lopende band hebben we de gegevensstroom gefilterd, getransformeerd en gesorteerd. Het resultaat is een lijst van afgewerkte producten die voldoen aan onze specificaties.

2. LINQ in Detail: Voorbeelden en Uitleg van tussenstations

2.1. Filteren met `Where`

Laten we beginnen met een gedetailleerd voorbeeld van het filteren van gegevens met de `Where` -methode. We hebben een lijst met cijfers en willen alleen de cijfers groter dan 10 behouden.

```
List<int> cijfers = new List<int> { 5, 12, 8, 15, 20 };
var result = cijfers.Where(cijfer => cijfer > 10);
```

In dit voorbeeld fungeert de `Where`-methode als ons tussenstation op de lopende band. Het lambda-expressiegedeelte (`cijfer ⇒ cijfer > 10`) definieert de voorwaarde waaraan een cijfer moet voldoen om te worden behouden. In dit geval zijn dat de cijfers groter dan 10.

2.2. Transformeren met `Select`

Laten we nu kijken naar een voorbeeld waarbij we gegevens transformeren met de `Select` -methode. We hebben een lijst met namen en willen een nieuwe lijst maken met de lengte van elke naam.

```
List<string> namen = new List<string> { "Alice", "Bob", "Charlie" };
var result = namen.Select(naam => naam.Length);
```

Hier fungeert `Select` als ons tussenstation op de lopende band. De lambda-expressie (`naam ⇒ naam.Length`) definieert de transformatie die wordt toegepast op elke naam in de lijst. In dit geval berekenen we de lengte van elke naam.

2.3. Sorteren met `OrderBy`

Een voorbeeld van sorteren met de `OrderBy`-methode. We hebben een lijst met producten en willen ze oplopend sorteren op basis van hun prijs.

```
List<Product> producten = new List<Product>
{
    new Product { Naam = "Laptop", Prijs = 800 },
    new Product { Naam = "Telefoon", Prijs = 500 },
    new Product { Naam = "Tablet", Prijs = 300 }
};
var result = producten.OrderBy(product => product.Prijs);
```

Hier fungeert `OrderBy` als ons tussenstation op de lopende band. We gebruiken een lambda-expressie (`product ⇒ product.Prijs`) om aan te geven op welk attribuut (in dit geval, `Prijs`) we de lijst willen sorteren.

2.3.1. Sorteren met `ThenBy`

Laat ons het 'OrderBy' voorbeeld uitbreiden met het gebruik van 'ThenBy' en eventueel 'ThenByDescending', wat handig is om secundaire sortering toe te passen in een LINQ-query. Hier is het aangepaste voorbeeld:

```
List<Product> producten = new List<Product>
{
    new Product { Naam = "Laptop", Categorie = "Elektronica", Prijs = 800 },
    new Product { Naam = "Telefoon", Categorie = "Elektronica", Prijs = 500 },
    new Product { Naam = "Tablet", Categorie = "Elektronica", Prijs = 300 },
    new Product { Naam = "Boek", Categorie = "Boeken", Prijs = 20 }
};

var result = producten
    .OrderBy(product => product.Categorie)        // Eerst op categorie sorteren
```

```
    .ThenBy(product => product.Prijs);              // Dan op prijs sorteren

foreach (var product in result)
{
    Console.WriteLine($"Naam: {product.Naam}, Categorie: {product.Categorie}, Prijs:
{product.Prijs}");
}
```

In dit voorbeeld gebruiken we eerst `OrderBy` om de producten te sorteren op categorie (oplopende volgorde). Vervolgens passen we `ThenBy` toe om de producten binnen dezelfde categorie op prijs te sorteren (ook oplopende volgorde).

De uitvoer van deze code zal de producten weergeven in gesorteerde volgorde op categorie, en binnen elke categorie worden de producten gesorteerd op prijs. Hierdoor krijgen we een geordende lijst van producten met een logische hiërarchie in de sortering.

Dit is handig in situaties waarin je gegevens op meerdere criteria wilt sorteren om een geordende lijst te krijgen die voldoet aan complexere sorteerbehoeften.

2.4. Filteren met `Distinct`

Hier is een voorbeeld van het gebruik van de LINQ-operator `Distinct` om dubbele elementen uit een lijst te verwijderen:

```
List<int> getallen = new List<int> { 1, 2, 2, 3, 4, 4, 5, 6, 6, 7 };

// Gebruik 'Distinct' om dubbele getallen te verwijderen
var uniekeGetallen = getallen.Distinct();

Console.WriteLine("Originele lijst van getallen:");
foreach (var getal in getallen)
{
    Console.Write(getal + " ");
}

Console.WriteLine("\n\nLijst van unieke getallen:");
foreach (var getal in uniekeGetallen)
{
    Console.Write(getal + " ");
}
```

In dit voorbeeld hebben we een lijst van getallen waarin sommige getallen dubbel voorkomen. We willen de dubbele getallen verwijderen en een lijst maken met alleen de unieke getallen.

1. We gebruiken de `Distinct`-operator op de lijst van getallen om de duplicaten te verwijderen. Dit resulteert in een nieuwe reeks `uniekeGetallen`.
2. We lopen door de originele lijst van getallen en drukken deze af, zodat je kunt zien welke getallen er oorspronkelijk waren.

3. Vervolgens lopen we door de lijst van unieke getallen en drukken deze af. Hiermee krijg je een lijst met alleen de unieke getallen.

De uitvoer zal aantonen hoe de `Distinct`-operator dubbele elementen verwijdert en een lijst met unieke elementen genereert.

2.5. Groeperen met `Group By`

Laten we nu een voorbeeld bekijken van het groeperen van gegevens met de `Group By`-methode. Stel dat we een lijst van werknemers hebben en we ze willen groeperen op basis van hun afdeling.

```
List<Werknemer> werknemers = new List<Werknemer>
{
    new Werknemer { Naam = "Anna", Afdeling = "Verkoop" },
    new Werknemer { Naam = "Bob", Afdeling = "Verkoop" },
    new Werknemer { Naam = "Carla", Afdeling = "Techniek" }
};

// Gebruik GroupBy om werknemers te groeperen op afdeling
var groepen = werknemers.GroupBy(werknemer => werknemer.Afdeling);

// Itereer over de resulterende groepen en toon de resultaten
foreach (var groep in groepen)
{
    Console.WriteLine($"Afdeling: {groep.Key}");

    foreach (var werknemer in groep)
    {
        Console.WriteLine($" - Naam: {werknemer.Naam}");
    }
}
```

In dit voorbeeld gebruiken we `GroupBy` om de werknemers te groeperen op basis van hun afdeling. Het resultaat van de `GroupBy`-bewerking is een reeks groepen van werknemers, waar elke groep wordt geïdentificeerd door een zogenaamde "key" (in dit geval de `Afdeling`).

Als we de resulterende groepen itereren, kunnen we toegang krijgen tot de werknemers in elke afdelingsgroep. Hierdoor kunnen we de werknemers binnen elke afdeling opvragen en weergeven.

De uitvoer van de code zal er als volgt uitzien:

```
Afdeling: Verkoop
 - Naam: Anna
 - Naam: Bob
Afdeling: Techniek
 - Naam: Carla
```

Dit toont aan hoe `GroupBy` werknemers opsplitst in groepen op basis van hun afdeling, en

vervolgens kunnen we de werknemers binnen elke groep afzonderlijk verwerken. Het is een krachtige bewerking om gegevens te groeperen en inzicht te krijgen in de structuur van je gegevens.

2.5.1. GroupBy vs Dictionary

De uitvoer van een `GroupBy`-bewerking lijkt op een `Dictionary` in die zin dat beide gegevensstructuren een associatief verband tot stand brengen tussen een sleutel en een verzameling waarden. Echter, er zijn enkele belangrijke verschillen tussen beide:

1. **Resultaat van `GroupBy`**: De uitvoer van `GroupBy` is een reeks groepen, waarbij elke groep een sleutel heeft en een reeks elementen bevat die voldoen aan de groeperingscriteria. In C# wordt dit vaak weergegeven als een `IEnumerable<IGrouping<TKey, TElement>>`, waarbij `TKey` de sleutel en `TElement` de elementen in elke groep zijn.
2. **`Dictionary`**: Een `Dictionary<TKey, TValue>` is een gegevensstructuur waarbij je een unieke sleutel aan elke waarde koppelt. In een dictionary kun je waarden efficiënt opzoeken op basis van hun sleutel.
3. **Gebruiksscenario's**
    a. `GroupBy` wordt typisch gebruikt wanneer je gegevens wilt groeperen op basis van een bepaald kenmerk, zoals het groeperen van werknemers per afdeling, maar je wilt mogelijk meerdere elementen met dezelfde sleutel (afdeling).
    b. Een `Dictionary` wordt gebruikt wanneer je een enkele unieke waarde (de sleutel) wilt koppelen aan een enkele waarde (de waarde), bijvoorbeeld om snel gegevens op te halen of te indexeren.
4. **Sleuteluniekheid**
    a. In een `Dictionary` moeten de sleutels uniek zijn; elke sleutel kan slechts één bijbehorende waarde hebben. Bij `GroupBy` kunnen meerdere elementen dezelfde sleutel delen, wat nuttig is voor groeperingsdoeleinden.

Kortom, terwijl zowel `GroupBy` als een `Dictionary` een associatief verband tussen sleutels en waarden tot stand brengen, zijn ze geschikt voor verschillende gebruiksscenario's. `GroupBy` is handig wanneer je gegevens wilt groeperen op basis van gemeenschappelijke kenmerken en meerdere waarden met dezelfde sleutel wilt verwerken, terwijl een `Dictionary` wordt gebruikt om unieke sleutel-waardeparen te beheren voor efficiënte toegang tot individuele waarden.

2.6. Samenvoegen met `Join`

Laten we nu een voorbeeld bekijken van het samenvoegen van twee gegevensbronnen met de `Join` -methode.

Stel dat we twee lijsten hebben: een lijst van werknemers en een lijst van afdelingen. We willen de werknemers samenvoegen met hun respectievelijke afdelingen op basis van een gemeenschappelijk kenmerk, in dit geval een afdelings-ID.

Hier is het voorbeeld:

```
List<Werknemer> werknemers = new List<Werknemer>
{
    new Werknemer { Id = 1, Naam = "Anna", AfdelingId = 1 },
    new Werknemer { Id = 2, Naam = "Bob", AfdelingId = 2 },
    new Werknemer { Id = 3, Naam = "Carla", AfdelingId = 1 }
};

List<Afdeling> afdelingen = new List<Afdeling>
{
    new Afdeling { Id = 1, Naam = "Verkoop" },
    new Afdeling { Id = 2, Naam = "Techniek" }
};

var result = werknemers
                .Join(afdelingen,                              // Binnenste lijst
                      werknemer => werknemer.AfdelingId,       // Sleutel van de
binnenste lijst
                      afdeling => afdeling.Id,                 // Sleutel van de
buitenste lijst
                      (werknemer, afdeling) => new             // Resultaatselector
                      {
                          Naam = werknemer.Naam,
                          Afdeling = afdeling.Naam
                      });

foreach (var item in result)
{
    Console.WriteLine($"Naam: {item.Naam}, Afdeling: {item.Afdeling}");
}
```

Dit voorbeeld maakt gebruik van de `Join`-bewerking om werknemers en afdelingen samen te voegen. Hierbij de belangrijkste elementen:

- We hebben twee lijsten: `werknemers` en `afdelingen`, die we willen samenvoegen op basis van de `AfdelingId` van werknemers en de `Id` van afdelingen.
- In de `Join`-methode geven we eerst de binnenste lijst op (in dit geval `werknemers`), daarna de sleutelselector voor de binnenste lijst (hoe we de afdelings-ID van elke werknemer krijgen), de buitenste lijst (`afdelingen`), en de sleutelselector voor de buitenste lijst (hoe we de ID van elke afdeling krijgen).
- Ten slotte, in de resultaatselector, bepalen we welke gegevens we willen ophalen en tonen voor elke overeenkomst tussen een werknemer en een afdeling. Hiermee creëren we een nieuw samengevoegd resultaat.

De uitvoer van deze code zal de namen van de werknemers en hun respectievelijke afdelingen weergeven, zoals:

```
Naam: Anna, Afdeling: Verkoop
Naam: Bob, Afdeling: Techniek
```

```
Naam: Carla, Afdeling: Verkoop
```

Dit toont hoe de `Join`-bewerking wordt gebruikt om gegevens uit twee lijsten op basis van een gemeenschappelijke sleutel samen te voegen en resultaten te genereren op basis van de gespecificeerde selectiecriteria.

3. LINQ in Detail: Voorbeelden en Uitleg van eindstations

De eindstations vormen het laatste station op de LINQ-productielijn en helpen je de resultaten te verkrijgen op een manier die het beste past bij jouw verdere verwerkingsbehoeften. Of je nu een lijst, een array, een dictionary of andere gegevensstructuren nodig hebt, LINQ biedt diverse eindstations om aan je wensen te voldoen.

Laten we enkele voorbeelden bekijken van eindstations op de LINQ-productielijn.

3.1. `ToList`: Het Transformeren naar een Lijst

De `ToList`-operator zet de verwerkte gegevens om in een lijst, waardoor je gemakkelijk verdere bewerkingen kunt uitvoeren of de resultaten kunt weergeven.

```
var getallen = new List<int> { 1, 2, 3, 4, 5 };
var resultaatLijst = getallen.Where(getal => getal % 2 == 0).ToList();
```

In dit voorbeeld worden de even getallen uit de oorspronkelijke lijst gehaald en omgezet in een nieuwe lijst (`resultaatLijst`) met alleen even getallen.

3.2. `ToDictionary`: Het Omzetten naar een Dictionary

De `ToDictionary`-operator zet de verwerkte gegevens om in een dictionary waarbij je een sleutel kunt specificeren op basis van een eigenschap van de gegevens.

```
var studenten = new List<Student>
{
    new Student { StudentId = 1, Naam = "Alice" },
    new Student { StudentId = 2, Naam = "Bob" },
    new Student { StudentId = 3, Naam = "Charlie" }
};

var studentDictionary = studenten.ToDictionary(student => student.StudentId);
```

In dit voorbeeld worden de studenten omgezet in een dictionary waarbij het `StudentId` de sleutel is. Hiermee kun je eenvoudig studenten opzoeken op basis van hun ID.

3.3. Het aantal elementen met `Count`

Hier is een voorbeeld van het gebruik van de LINQ-operator `Count`:

```
List<int> getallen = new List<int> { 1, 2, 2, 3, 4, 4, 5, 6, 6, 7 };

// Gebruik 'Count' om het aantal elementen in de lijst te tellen
int aantalGetallen = getallen.Count();

Console.WriteLine("Originele lijst van getallen:");
foreach (var getal in getallen)
{
    Console.Write(getal + " ");
}

Console.WriteLine($"\n\nAantal getallen in de lijst: {aantalGetallen}");
```

In dit voorbeeld hebben we een lijst van getallen en willen we het aantal elementen in de lijst tellen met behulp van de `Count`-operator.

1. We gebruiken de `Count`-operator op de lijst van getallen om het aantal elementen in de lijst te tellen. Dit aantal wordt opgeslagen in de variabele `aantalGetallen`.
2. We lopen door de originele lijst van getallen en drukken deze af, zodat je kunt zien welke getallen er in de lijst staan.
3. Vervolgens drukken we het aantal getallen in de lijst af, dat we hebben verkregen met de `Count` -operator.

De uitvoer zal aantonen hoe de `Count`-operator wordt gebruikt om het aantal elementen in de lijst te bepalen. In dit geval zal het aantal getallen in de lijst gelijk zijn aan 10.

3.4. `Sum`, `Average`, `Min`, en `Max`

De LINQ-operatoren `Sum`, `Average`, `Min`, en `Max` zijn bedoeld voor numerieke bewerkingen op een reeks elementen in LINQ.

- `Sum`: De `Sum`-operator berekent de som van alle numerieke waarden in de reeks.
    - **Vereisten voor Elementen**:
        - De elementen in de reeks moeten numerieke gegevens bevatten, zoals `int`, `double`, `decimal`, enz.
        - De reeks mag geen `null`-waarden bevatten, omdat dit een `NullReferenceException` kan veroorzaken.

```
List<int> getallen = new List<int> { 1, 2, 3, 4, 5 };
int som = getallen.Sum(); // De som is 15
```

- `Average`: De `Average`-operator berekent het gemiddelde (het rekenkundig gemiddelde) van alle numerieke waarden in de reeks.
  - **Vereisten voor Elementen**:
    - Net als bij `Sum` moeten de elementen in de reeks numerieke gegevens bevatten, zoals `int`, `double`, `decimal`, enz.
    - De reeks mag geen `null`-waarden bevatten.

```
List<int> getallen = new List<int> { 1, 2, 3, 4, 5 };
double gemiddelde = getallen.Average(); // Het gemiddelde is 3.0
```

- `Min`: De `Min`-operator vindt het minimumelement in de reeks van numerieke waarden.
  - **Vereisten voor Elementen**:
    - De elementen in de reeks moeten de IComparable-interface implementeren, zodat ze vergeleken kunnen worden.
    - De reeks mag geen `null`-waarden bevatten.

```
List<int> getallen = new List<int> { 1, 2, 3, 4, 5 };
int minimum = getallen.Min(); // Het minimum is 1
```

- `Max`: De `Max`-operator vindt het maximumelement in de reeks van numerieke waarden.
  - **Vereisten voor Elementen**:
    - De elementen in de reeks moeten de IComparable-interface implementeren, zodat ze vergeleken kunnen worden.
    - De reeks mag geen `null`-waarden bevatten.

```
List<int> getallen = new List<int> { 1, 2, 3, 4, 5 };
int maximum = getallen.Max(); // Het maximum is 5
```

Deze operatoren zijn handig bij het uitvoeren van wiskundige bewerkingen op numerieke gegevens in een LINQ-query. Zorg ervoor dat de elementen in de reeks voldoen aan de vereisten om ongewenste uitzonderingen te voorkomen.

3.5. Controleren met `Any` en `All`

Laten we tot slot kijken naar een voorbeeld waarbij we controleren of bepaalde voorwaarden in de gegevensbronnen worden vervuld met de `Any`- en `All`-methodes. Stel dat we willen controleren of er volwassenen in een lijst van personen zijn.

```
List<Persoon> personen = new List<Persoon>
{
    new Persoon { Naam = "Alice", Leeftijd = 25 },
```

```
    new Persoon { Naam = "Bob", Leeftijd = 30 },
    new Persoon { Naam = "Carla", Leeftijd = 17 }
};

bool bevatVolwassene = personen.Any(persoon => persoon.Leeftijd >= 18);
bool zijnAllemaalVolwassenen = personen.All(persoon => persoon.Leeftijd >= 18);
```

In deze voorbeelden fungeren `Any` en `All` als tussenstations op de lopende band. Ze controleren of aan de opgegeven voorwaarde wordt voldaan voor ten minste één element (`Any`) en voor alle elementen (`All`) in de lijst.

4. Aangepaste LINQ Query's

Je kunt ook je eigen LINQ-query-operatoren en -methoden definiëren door extensiemethoden te maken die de `IEnumerable<T>`-interface uitbreiden. Dit stelt je in staat om aangepaste query's te schrijven die specifiek zijn voor je eigen gegevenstypen.

Voorbeeld:

```
public static class MyExtensions
{
    public static IEnumerable<T> MyWhere<T>(this IEnumerable<T> source, Func<T, bool>
predicate)
    {
        foreach (T item in source)
        {
            if (predicate(item))
            {
                yield return item;
            }
        }
    }
}
```

5. Keyword `yield`

Het `yield`-keyword in C# wordt gebruikt in combinatie met methoden die een `IEnumerable<T>`- of `IEnumerator<T>`-reeks genereren. Het stelt je in staat om lazy evaluation toe te passen, wat betekent dat de elementen van de reeks pas worden berekend op het moment dat ze worden opgevraagd. Dit kan voordelig zijn bij het verwerken van grote hoeveelheden gegevens, omdat je niet alle gegevens in het geheugen hoeft te houden.

Hier is een uitgebreid voorbeeld dat het gebruik van het `yield`-keyword illustreert. We zullen een reeks van Fibonacci-getallen genereren zonder een volledige lijst in het geheugen op te slaan. In plaats daarvan genereren we elk Fibonacci-getal op aanvraag.

```
class Program
{
    public static void Main()
    {
        // Roep de methode aan om Fibonacci-getallen te genereren en deze af te
drukken
        foreach (var getal in Fibonacci(10))
        {
            Console.Write(getal + " ");
        }
    }

    // Methode om een reeks van Fibonacci-getallen te genereren
    public static IEnumerable<int> Fibonacci(int count)
    {
        int a = 0, b = 1;

        for (int i = 0; i < count; i++)
        {
            // Gebruik 'yield' om het huidige Fibonacci-getal te retourneren
            yield return a;

            // Bereken het volgende Fibonacci-getal
            int temp = a;
            a = b;
            b = temp + b;
        }
    }
}
```

Hier is wat er gebeurt in dit voorbeeld:

1. We definiëren een methode genaamd `Fibonacci` die een reeks van Fibonacci-getallen genereert met een opgegeven aantal elementen (`count`).
2. Binnen de `Fibonacci`-methode gebruiken we de `yield`-instructie om de huidige waarde van `a` (het huidige Fibonacci-getal) terug te geven. Op dit punt wordt de methode niet volledig uitgevoerd; in plaats daarvan wordt de controle teruggegeven aan de beller.
3. Na de `yield`-instructie wordt het volgende Fibonacci-getal berekend door `a` en `b` bij te werken.
4. De loop in de `Fibonacci`-methode blijft doorgaan totdat het gewenste aantal Fibonacci-getallen is gegenereerd.
5. In de `Main`-methode roepen we `Fibonacci(10)` aan om de eerste 10 Fibonacci-getallen te genereren en deze af te drukken.

Het belangrijkste voordeel van het gebruik van `yield` is dat het geheugenefficiënt is, omdat het alleen het huidige element in het geheugen houdt en de rest van de reeks on-demand genereert. Dit is vooral handig bij het verwerken van grote of oneindige reeksen.

6. LINQ's uitgestelde uitvoering en streaming

6.1. Slim Wachten: Uitgestelde Uitvoering

Laten we beginnen met het idee van uitgestelde uitvoering. Stel je voor dat je een lijst hebt met getallen, en je wilt alleen de even getallen kiezen.

```
var getallen = new List<int> { 1, 2, 3, 4, 5 };
var resultaatQuery = getallen.Where(getal => getal % 2 == 0);
```

LINQ voert deze operatie niet onmiddellijk uit. Het wacht eigenlijk tot je het resultaat echt nodig hebt.

```
foreach (var getal in resultaatQuery)
{
    Console.Write(getal + " "); // Nu wordt de filter uitgevoerd!
}
```

6.2. Gegevens verwerken in delen: Streamen in LINQ

Stel je voor dat je niet alleen de even getallen wilt, maar je wilt ze ook verdubbelen.

```
var getallen = new List<int> { 1, 2, 3, 4, 5 };
var resultaatQuery = getallen.Where(getal => getal % 2 == 0).Select(getal => getal *
2);
```

Hier streamt LINQ eigenlijk door de getallen. Het selecteert elk getal, verdubbelt het en geeft het door. Dit is handig bij grote hoeveelheden gegevens omdat het geheugen bespaart.

6.3. Tips

- **Slim Gebruiken**: Uitgestelde uitvoering en streamen zijn als slimme tools. Gebruik ze verstandig voor een efficiënte ervaring.
- **Voorzichtig met Databases**: Als je LINQ met databases gebruikt, let dan op wanneer de query's worden uitgevoerd. Te veel uitstel kan onbedoelde toegang tot externe bronnen veroorzaken.
- **Evaluatiemoment Controleren**: Sommige operatoren, zoals `ToList`, forceren onmiddellijke uitvoering. Houd dit in gedachten, want het kan het uitstellen van een uitvoering stoppen.

7. Conclusie

Net zoals een lopende band efficiëntie en consistentie in een productieproces brengt, biedt LINQ een gestructureerde en consistente manier om gegevens te verwerken in je C#-programma's. Het stelt je in staat om gegevensbronnen te filteren, transformeren, groeperen, samenvoegen en controleren met behulp van duidelijke en leesbare syntaxis, vergelijkbaar met hoe een lopende band grondstoffen verwerkt tot afgewerkte producten.

Dit hoofdstuk heeft LINQ geïntroduceerd met behulp van de lopende band-vergelijking en gedetailleerde voorbeelden om de concepten en operaties te illustreren. LINQ is een waardevolle tool voor ontwikkelaars die met gegevens werken en kan de efficiëntie en leesbaarheid van je code aanzienlijk verbeteren. In de komende hoofdstukken zullen we dieper ingaan op de verschillende LINQ-operatoren en hun toepassingen.