## **Enumeration and Iterators**

In C#, dealing with sequences of data (like items in a list, characters in a string, or results from a database query) is a common task. **Enumeration** refers to the process of stepping through these sequences, and **iterators** are a specialized language feature that makes creating such sequences remarkably easy. Dalam C#, berurusan dengan urutan data (seperti item dalam daftar, karakter dalam string, atau hasil dari kueri basis data) adalah tugas yang umum. Enumerasi mengacu pada proses melangkah melalui urutan-urutan ini, dan iterator adalah fitur bahasa khusus yang membuat pembuatan urutan tersebut menjadi sangat mudah.

### **Enumeration: Stepping Through a Sequence**

At its core, enumeration involves two main concepts: Pada intinya, pencacahan melibatkan dua konsep utama:

1. Enumerator (the Cursor): An enumerator is an object that acts as a read-only, forward-only cursor over a sequence. It knows where it currently is in the sequence and how to move to the next item.  
   C# treats a type as an enumerator if it provides: Pencacah (Kursor): Pencacah adalah objek yang bertindak sebagai kursor hanya-baca dan hanya-maju pada urutan. Ia tahu di mana ia saat ini berada dalam urutan dan bagaimana untuk pindah ke item berikutnya.

C# memperlakukan sebuah tipe sebagai pencacah jika tipe tersebut menyediakan:

* + A public parameterless method named MoveNext(): This method advances the cursor to the next element and returns true if successful, or false if the end of the sequence is reached. Metode publik tanpa parameter bernama MoveNext(): Metode ini memajukan kursor ke elemen berikutnya dan mengembalikan nilai true jika berhasil, atau false jika akhir urutan tercapai.
  + A public property named Current: This property returns the element at the current position of the cursor. Properti publik bernama Current: Properti ini mengembalikan elemen pada posisi kursor saat ini.
  + (Alternatively) It implements System.Collections.Generic.IEnumerator<T> or System.Collections.IEnumerator.

1. Enumerable Object (the Sequence): An enumerable object is the logical representation of a sequence itself. It's not the cursor, but rather an object that produces enumerators (cursors) for iterating over its data. Objek yang dapat dihitung (Urutan): Objek yang dapat dihitung adalah representasi logis dari urutan itu sendiri. Ini bukan kursor, melainkan objek yang menghasilkan pencacah (kursor) untuk mengulang datanya.  
   C# treats a type as enumerable if it provides: C# memperlakukan sebuah tipe sebagai dapat dihitung jika tipe tersebut menyediakan:  
   * A public parameterless method named GetEnumerator(): This method returns an enumerator object. Metode publik tanpa parameter bernama GetEnumerator(): Metode ini mengembalikan sebuah objek pencacah.
   * (Alternatively) It implements System.Collections.Generic.IEnumerable<T> or System.Collections.IEnumerable.
   * (From C# 9) It can bind to an extension method named GetEnumerator. (Dari C# 9) Dapat diikat ke metode ekstensi bernama GetEnumerator.

The common pattern for enumeration looks like this: Pola umum untuk pencacahan terlihat seperti ini:

|  |
| --- |
| // Enumerator (the cursor) class MyEnumerator<T> : System.Collections.Generic.IEnumerator<T> {  public T Current { get { /\* ... returns current element ... \*/ } }  public bool MoveNext() { /\* ... moves to next, returns true/false ... \*/ }  // Other IEnumerator members like Reset(), Dispose() }  // Enumerable (the sequence) class MyEnumerable<T> : System.Collections.Generic.IEnumerable<T> {  public IEnumerator<T> GetEnumerator() { /\* ... returns new MyEnumerator<T>() ... \*/ }  // Non-generic GetEnumerator() for IEnumerable } |

### **The foreach Statement**

The foreach statement is the high-level, convenient way to iterate over an enumerable object. It abstracts away the complexity of managing the enumerator explicitly. Pernyataan foreach adalah cara tingkat tinggi dan nyaman untuk mengulang objek yang dapat dicacah. Pernyataan ini mengabstraksikan kerumitan dalam mengelola pencacah secara eksplisit.

|  |
| --- |
| // High-level: Using foreach foreach (char c in "beer") // "beer" is an enumerable object (string implements IEnumerable<char>) {  Console.WriteLine(c); } // Output: // b // e // e // r |

Underneath, the foreach statement is syntactic sugar for the following low-level operations: Di bawahnya, pernyataan foreach adalah gula sintaksis untuk operasi tingkat rendah berikut:

|  |
| --- |
| // Low-level: Manual enumeration using (var enumerator = "beer".GetEnumerator()) // Get an enumerator from the enumerable object {  while (enumerator.MoveNext()) // Move to the next element  {  var element = enumerator.Current; // Get the current element  Console.WriteLine(element);  } } |

Notice the using statement. If the enumerator returned by GetEnumerator() implements System.IDisposable (which IEnumerator<T> does), the foreach statement implicitly ensures that the Dispose() method on the enumerator is called when the iteration completes or is exited early (e.g., via break). This is crucial for releasing resources. Perhatikan pernyataan penggunaan. Jika pencacah yang dikembalikan oleh GetEnumerator() mengimplementasikan System.IDisposable (yang mana IEnumerator<T> juga), pernyataan foreach secara implisit memastikan bahwa metode Dispose() pada pencacah dipanggil ketika iterasi selesai atau keluar lebih awal (misal, melalui break). Hal ini sangat penting untuk melepaskan sumber daya.

## **Collection Initializers and Collection Expressions** **Inisialisasi Koleksi dan Ekspresi Koleksi**

These features simplify the creation and population of collection objects. Fitur-fitur ini menyederhanakan pembuatan dan populasi objek koleksi.

### **Collection Initializers**

You can instantiate and populate an enumerable object (like a List<T>) in a single step using a **collection initializer**: Anda dapat menginstansiasi dan mengisi objek yang dapat dihitung (seperti List<T>) dalam satu langkah menggunakan inisialisasi koleksi:

|  |
| --- |
| using System.Collections.Generic;  var list = new List<int> { 1, 2, 3 }; // Creates a List<int> and adds 1, 2, 3 |

The compiler translates this into calls to the collection's Add method:

|  |
| --- |
| List<int> list = new List<int>(); list.Add(1); list.Add(2); list.Add(3); |

This requires the collection type to implement IEnumerable and have an accessible Add method with the appropriate parameters. Hal ini mengharuskan tipe koleksi mengimplementasikan IEnumerable dan memiliki metode Add yang dapat diakses dengan parameter yang sesuai.

Collection initializers also work for dictionaries: Inisialisasi koleksi juga dapat digunakan untuk kamus:

|  |
| --- |
| using System.Collections.Generic;  var dict = new Dictionary<int, string>() {  { 5, "five" }, // Key-value pair syntax  { 10, "ten" } }; |

Or, more concisely, using indexer syntax for dictionaries or any type with an indexer: Atau, lebih ringkasnya, menggunakan sintaks pengindeks untuk kamus atau jenis apa pun yang memiliki pengindeks

|  |
| --- |
| var dict = new Dictionary<int, string>() {  [3] = "three", // Indexer syntax  [10] = "ten" }; |

### **Collection Expressions (C# 12+)**

C# 12 introduced **collection expressions**, a further simplification that uses square brackets [] to create and initialize collections. C# 12 memperkenalkan ekspresi koleksi, penyederhanaan lebih lanjut yang menggunakan tanda kurung siku [] untuk membuat dan menginisialisasi koleksi.

|  |
| --- |
| using System.Collections.Generic;  List<int> list = [1, 2, 3]; // More concise way to create and populate a List<int> |

Collection expressions are **target-typed**, meaning the compiler infers the exact collection type from the context where the expression is used. Ekspresi koleksi diketik dengan tipe target, yang berarti kompiler menyimpulkan tipe koleksi yang tepat dari konteks di mana ekspresi tersebut digunakan.

|  |
| --- |
| int[] array = [1, 2, 3]; // Target type is int[] Span<int> span = [1, 2, 3]; // Target type is Span<int> |

This target typing also extends to method calls: Pengetikan target ini juga berlaku untuk pemanggilan metode:

|  |
| --- |
| Foo([1, 2, 3]); // Compiler infers List<int> based on Foo's parameter type  void Foo(List<int> numbers) { /\* ... \*/ } |

## **Iterators: Producing Sequences with yield**

While a foreach statement *consumes* an enumerator, an **iterator** is a method, property, or indexer that *produces* an enumerator. Iterators enable you to define how a sequence is generated on demand, element by element, without having to manually implement the IEnumerator and IEnumerable interfaces. Sementara pernyataan foreach menggunakan pencacah, iterator adalah metode, properti, atau pengindeks yang menghasilkan pencacah. Iterator memungkinkan Anda untuk menentukan bagaimana urutan dihasilkan sesuai permintaan, elemen demi elemen, tanpa harus mengimplementasikan antarmuka IEnumerator dan IEnumerable secara manual.

The key to iterators is the yield keyword:

* **yield return <element>:** This statement indicates that the method should return the specified <element> as the next item in the sequence. Control is then temporarily returned to the caller. However, the *state* of the iterator method (local variables, loop position, etc.) is preserved. When the caller requests the next element, execution resumes from where it left off. menghasilkan pengembalian <elemen>: Pernyataan ini mengindikasikan bahwa metode harus mengembalikan <elemen> yang ditentukan sebagai item berikutnya dalam urutan. Kontrol kemudian dikembalikan untuk sementara ke pemanggil. Namun, kondisi metode iterator (variabel lokal, posisi perulangan, dll.) dipertahankan. Ketika pemanggil meminta elemen berikutnya, eksekusi dilanjutkan dari titik terakhir.
* **yield break:** This statement indicates that the end of the sequence has been reached, and no more elements will be yielded. It functions like a return statement for an iterator. jeda hasil: Pernyataan ini mengindikasikan bahwa akhir dari urutan telah tercapai, dan tidak ada lagi elemen yang akan dihasilkan. Pernyataan ini berfungsi seperti pernyataan pengembalian untuk iterator.

**Example: Fibonacci Sequence Iterator**

|  |
| --- |
| using System; using System.Collections.Generic;  IEnumerable<int> Fibs(int fibCount) // Returns IEnumerable<int> {  for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)  {  yield return prevFib; // Yields the current Fibonacci number  int newFib = prevFib + curFib;  prevFib = curFib;  curFib = newFib;  } }  // ... in your main code: foreach (int fib in Fibs(6)) // Consumes the sequence produced by Fibs {  Console.Write(fib + " "); } // Output: 1 1 2 3 5 8 |

When you call Fibs(6), the code inside Fibs does *not* immediately execute. Instead, the C# compiler generates a hidden, private class that implements IEnumerable<int> and IEnumerator<int>. The Fibs method call simply instantiates this compiler-generated class. Your Fibs code (with the yield return statements) runs *only when you start enumerating* over the result, typically with a foreach loop, which calls MoveNext() and Current on the hidden enumerator. Ketika Anda memanggil Fibs(6), kode di dalam Fibs tidak langsung dieksekusi. Sebagai gantinya, kompiler C# menghasilkan kelas privat yang tersembunyi yang mengimplementasikan IEnumerable<int> dan IEnumerator<int>. Pemanggilan metode Fibs hanya menginstansiasi kelas yang dibuat oleh kompiler ini. Kode Fibs Anda (dengan pernyataan pengembalian hasil) hanya berjalan ketika Anda mulai mencacah hasil, biasanya dengan perulangan foreach, yang memanggil MoveNext() dan Current pada pencacah tersembunyi.

### **Iterator Semantics and Restrictions**

* **Return Types:** An iterator method, property, or indexer must return one of the four enumeration interfaces: System.Collections.IEnumerable, System.Collections.Generic.IEnumerable<T>, System.Collections.IEnumerator, or System.Collections.Generic.IEnumerator<T>. Tipe Pengembalian: Metode iterator, properti, atau pengindeks harus mengembalikan salah satu dari empat antarmuka pencacahan: System.Collections.IEnumerable, System.Collections.Generic.IEnumerable<T>, System.Collections.IEnumerator, atau System.Collections.Generic.IEnumerator<T>.
* **Multiple yield return:** You can have multiple yield return statements. Each one produces an element. Pengembalian imbal hasil berganda: Anda dapat memiliki beberapa laporan pengembalian hasil. Masing-masing menghasilkan sebuah elemen.
* **yield break:** Use yield break to stop producing elements early from an iterator block. return statements are illegal in iterator blocks. yield break (penghentian hasil): Gunakan yield break untuk berhenti memproduksi elemen lebih awal dari blok iterator. pernyataan return tidak diperbolehkan dalam blok iterator.
* **try-catch-finally Restrictions:**
  + A yield return statement cannot appear in a try block that has a catch clause. Pernyataan pengembalian hasil tidak dapat muncul dalam blok percobaan yang memiliki klausa tangkapan.
  + A yield return statement cannot appear in a catch block. Pernyataan pengembalian hasil tidak dapat muncul di blok tangkapan.
  + A yield return statement cannot appear in a finally block. These restrictions exist because translating complex exception handling within the "state machine" generated by the compiler for iterators would be overly complex. Pernyataan pengembalian hasil tidak dapat muncul dalam blok akhirnya. Pembatasan ini ada karena menerjemahkan penanganan pengecualian yang kompleks di dalam “state machine” yang dibuat oleh kompiler untuk iterator akan menjadi terlalu rumit.
  + However, yield return *can* appear in a try block that has *only* a finally block. The code in the finally block will execute when the consuming enumerator finishes the sequence or is disposed (e.g., when the foreach loop ends or is exited early). This ensures proper cleanup. Namun, pengembalian hasil dapat muncul dalam blok coba yang hanya memiliki blok akhirnya. Kode dalam blok finally akan dieksekusi ketika pencacah yang mengkonsumsi menyelesaikan urutan atau dibuang (misalnya, ketika foreach loop berakhir atau keluar lebih awal). Hal ini memastikan pembersihan yang tepat.

### **Composing Sequences**

Iterators are highly **composable**. You can chain them together to create complex data processing pipelines, where each step yields elements on demand. Iterator sangat mudah disusun. Anda dapat merangkainya bersama untuk membuat jalur pemrosesan data yang kompleks, di mana setiap langkah menghasilkan elemen sesuai permintaan.

|  |
| --- |
| using System; using System.Collections.Generic;  // Fibs as defined previously IEnumerable<int> Fibs(int fibCount) { /\* ... \*/ }  // New iterator that filters for even numbers IEnumerable<int> EvenNumbersOnly(IEnumerable<int> sequence) {  foreach (int x in sequence) // Iterates over the input sequence  {  if ((x % 2) == 0)  {  yield return x; // Yields only even numbers  }  } }  // ... in your main code: foreach (int fib in EvenNumbersOnly(Fibs(10))) // Chains Fibs with EvenNumbersOnly {  Console.Write(fib + " "); } // Output (for Fibs(10)): 2 8 34 |

The significant benefit here is **lazy evaluation**. Each element is calculated and processed only when it is actually requested by the consumer (MoveNext()). This makes iterators extremely efficient for working with large or infinite sequences, as you don't generate the entire sequence in memory at once. This composability is a cornerstone of LINQ. Manfaat yang signifikan di sini adalah evaluasi malas. Setiap elemen dihitung dan diproses hanya ketika elemen tersebut benar-benar diminta oleh konsumen (MoveNext()). Hal ini membuat iterator menjadi sangat efisien untuk bekerja dengan urutan yang besar atau tak terbatas, karena Anda tidak menghasilkan seluruh urutan dalam memori sekaligus. Komposabilitas ini adalah landasan dari LINQ.