## **Immutable Collections**

The core idea behind immutable collections is that once they are initialized, they **cannot be modified at all**. If you need to "change" an immutable collection—for instance, by adding an item—you do not modify the original collection. Instead, you must instantiate a *new* collection that incorporates the desired change, leaving the original collection completely untouched.

This principle of immutability, where an object's state cannot change after it's created, is a hallmark of functional programming paradigms and offers compelling benefits:

### **Benefits of Immutability**

1. **Reduced Bugs:** It eliminates a significant class of bugs related to unexpected state changes. When you know an object will not change, you don't need to worry about it being modified by another part of the code, simplifying reasoning.
2. **Simplified Concurrency and Multithreading:** This is perhaps the most profound benefit. Because immutable objects never change, they are inherently thread-safe. You can share them freely across multiple threads without needing locks or other synchronization mechanisms, which drastically simplifies parallel programming and avoids common thread-safety pitfalls.
3. **Easier Code Reasoning and Debugging:** Knowing that an object's state is fixed makes your code easier to understand and predict. Debugging becomes simpler as you don't need to trace where an object's state might have been altered.
4. **Defensive Copying Avoidance:** With mutable collections, you often need to create "defensive copies" to ensure that an external party doesn't modify a collection you're working with. Immutability removes this overhead.

### **Disadvantages and Mitigating Strategies**

The primary disadvantage of immutability is performance overhead. When you need to "change" an immutable collection (e.g., add an item), you are effectively creating a whole new object. This incurs allocation costs and garbage collection pressure.

However, the .NET immutable collections are designed with clever strategies to mitigate this performance hit, often by reusing portions of the original structure.

**Types of Immutable Collections**

The immutable collections are available within .NET (in .NET Framework, they require the System.Collections.Immutable NuGet package). They reside in the System.Collections.Immutable namespace. Here's a brief overview of common types and their internal structures:

| **Type** | **Internal Structure** |
| --- | --- |
| ImmutableArray<T> | Array |
| ImmutableList<T> | AVL Tree |
| ImmutableDictionary<TKey, TValue> | AVL Tree |
| ImmutableHashSet<T> | AVL Tree |
| ImmutableSortedDictionary<TKey, TValue> | AVL Tree |
| ImmutableSortedSet<T> | AVL Tree |
| ImmutableStack<T> | Linked List |
| ImmutableQueue<T> | Linked List |

It's crucial to note: Immutable collections prevent the adding and removing of items themselves; they do *not* prevent the items *within* the collection from being mutated if those items are mutable reference types. For full immutability benefits, ensure that the elements stored in an immutable collection are also immutable.

### **Creating Immutable Collections**

Immutable collection types provide static Create() methods for initialization:

| ImmutableArray<int> array = ImmutableArray.Create(1, 2, 3); |
| --- |

They also offer CreateRange() methods, which accept an IEnumerable<T> for initializing from existing sequences.

Additionally, you can convert any IEnumerable<T> to an immutable collection using extension methods available in the System.Collections.Immutable namespace (e.g., ToImmutableArray(), ToImmutableList(), ToImmutableDictionary()):

| var list = new[] { 1, 2, 3 }.ToImmutableList(); |
| --- |

### **Manipulating Immutable Collections (Non-Destructive Mutation)**

The key distinction in manipulating immutable collections is that methods that appear to alter the collection (such as Add or Remove) do *not* change the original collection. Instead, they return a *new* collection with the requested modification. This is known as **non-destructive mutation**.

| var oldList = ImmutableList.Create(1, 2, 3); ImmutableList<int> newList = oldList.Add(4); // Creates a new list  Console.WriteLine(oldList.Count); // Output: 3 (oldList is unaltered) Console.WriteLine(newList.Count); // Output: 4 |
| --- |

Similarly, the Remove method returns a new collection with the item removed.

Repeatedly calling Add or Remove in a loop can be inefficient due to the continuous creation of new collection objects. For batch operations, it's more efficient to use methods like AddRange() or RemoveRange(), which accept an IEnumerable<T> of items to add or remove in a single operation:

| var anotherList = oldList.AddRange([4, 5, 6]); // Efficiently adds multiple items |
| --- |

Immutable lists and arrays also provide Insert, InsertRange, RemoveAt, and RemoveAll (based on a predicate), all operating in a non-destructive manner.

**Builders: Efficient Batch Operations**

For scenarios involving multiple modifications or complex initialization, each immutable collection class offers a corresponding **builder** class. Builders are mutable counterparts that behave similarly to standard mutable collections (e.g., List<T>) in terms of performance.

You can populate a builder with data, perform multiple additions or removals efficiently, and then call its .ToImmutable() method to obtain a final immutable collection.

| ImmutableArray<int>.Builder builder = ImmutableArray.CreateBuilder<int>(); builder.Add(1); builder.Add(2); builder.Add(3); builder.RemoveAt(0); // Efficient mutable operations ImmutableArray<int> myImmutable = builder.ToImmutable(); // Final immutable snapshot |
| --- |

Builders are also excellent for batch updates to an *existing* immutable collection:

| var myImmutable2 = myImmutable.ToBuilder(); // Create a builder from an existing immutable collection myImmutable2.Add(4); // Efficient myImmutable2.Remove(2); // Efficient // ... more changes to builder... ImmutableArray<int> myFinalImmutable = myImmutable2.ToImmutable(); // Get a new immutable collection with all changes |
| --- |

Using builders significantly mitigates the performance concerns for scenarios involving frequent write operations.

**Immutable Collections and Performance**

While immutability offers significant architectural benefits, it's essential to understand its performance implications compared to mutable collections:

* **Internal Structure:**
  + Most immutable collections (ImmutableList<T>, ImmutableDictionary<TKey, TValue>, ImmutableHashSet<T>, etc.) utilize **AVL trees** internally. AVL trees are self-balancing binary search trees that allow for efficient (logarithmic time) add/remove operations while also enabling **structural sharing**. Structural sharing means that when a "change" occurs, only the affected parts of the tree are recreated, and unaffected subtrees are reused, which reduces the overhead. However, this structure still makes read operations slower than a flat array.
  + ImmutableArray<T>, as its name suggests, uses an internal array. This design provides **very fast read performance** (comparable to a regular mutable array, O(1)) because it's a contiguous block of memory. However, for write operations (add/remove), it's much slower than ImmutableList<T> because an entirely new array must be created and copied, without the benefit of structural sharing.
* **Performance Comparison (Read vs. Add):**

| **Type** | **Read Performance** | **Add Performance** |
| --- | --- | --- |
| ImmutableList<T> | Slow | Slow |
| ImmutableArray<T> | Very fast | Very slow |

* **ImmutableList<T> vs. List<T>:** ImmutableList<T> can be 10 to 200 times slower than List<T> for both read and add operations, depending on list size. This is due to the AVL tree overhead.
* **ImmutableArray<T> vs. Array:** ImmutableArray<T> has comparable read performance to a mutable array, but its add/remove operations are significantly more expensive than ImmutableList<T> because no structural sharing is possible.
* **When to choose ImmutableArray<T>:** It is desirable when your primary need is unimpeded read performance, and you anticipate very few subsequent additions or removals (without using a builder for batch operations).

### **Mitigating Performance Costs**

Despite the overhead, several factors mitigate the practical performance concerns:

1. **Concurrency and Parallelization:** The inherent thread-safety of immutable collections allows for easy concurrency and parallelization. This means you can leverage multiple CPU cores without complex locking, which can often lead to *net performance gains* in multi-threaded scenarios compared to managing mutable state with locks.
2. **No Defensive Copying:** As mentioned, immutability eliminates the need for defensive copies, which saves memory and CPU cycles that would otherwise be spent duplicating data.
3. **Magnitude of Cost:** For most typical applications, collections do not grow to a size where the performance difference becomes a critical bottleneck. An Add operation on an ImmutableList<T> with a million elements might still complete in microseconds, and a read in nanoseconds.
4. **Builders for Batch Operations:** When numerous modifications are required in a loop, builders provide an efficient way to batch these operations, yielding a single new immutable collection at the end, thus avoiding the cost of creating many intermediate immutable collections.