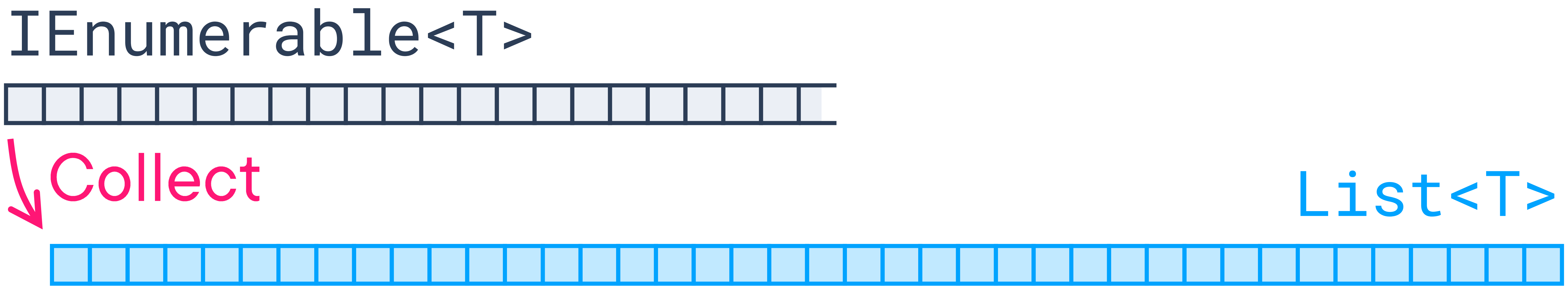
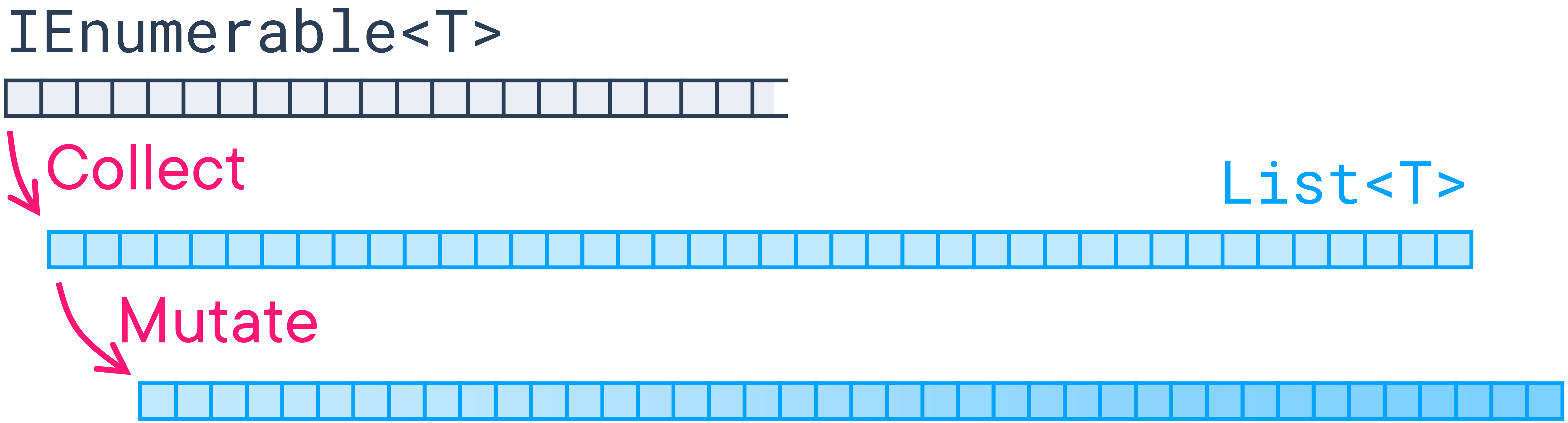
Collections and Generics in C#

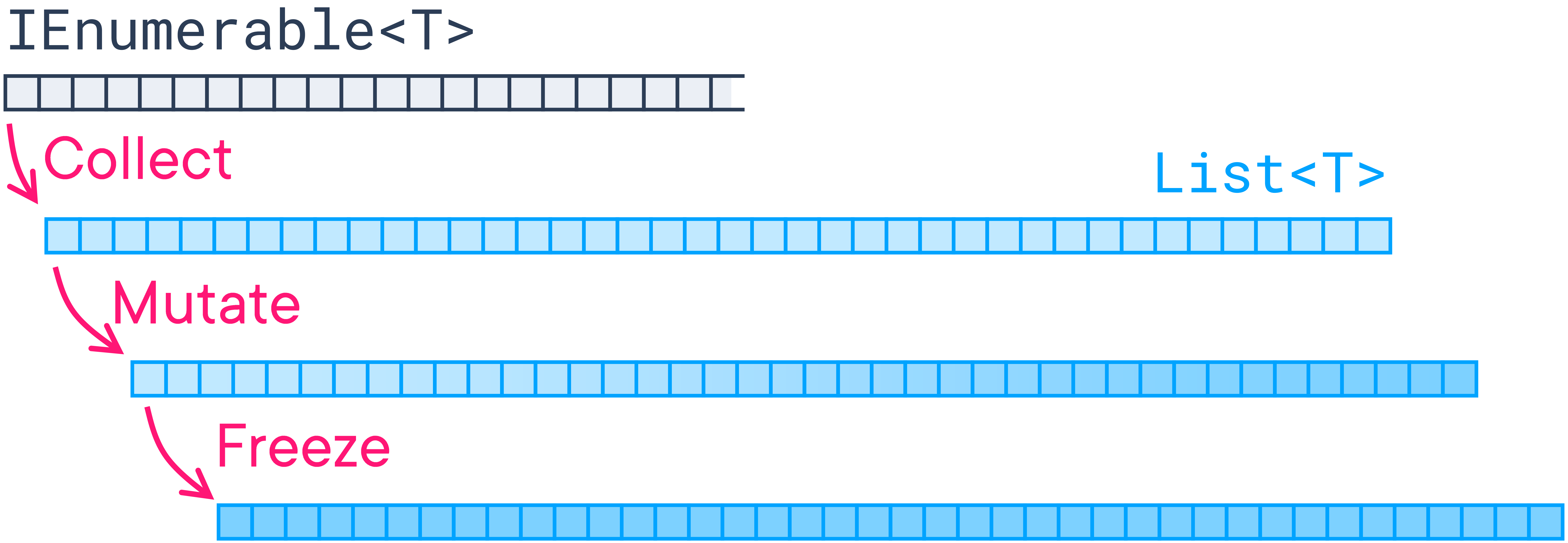
Using Linear Collections: Lists and Arrays

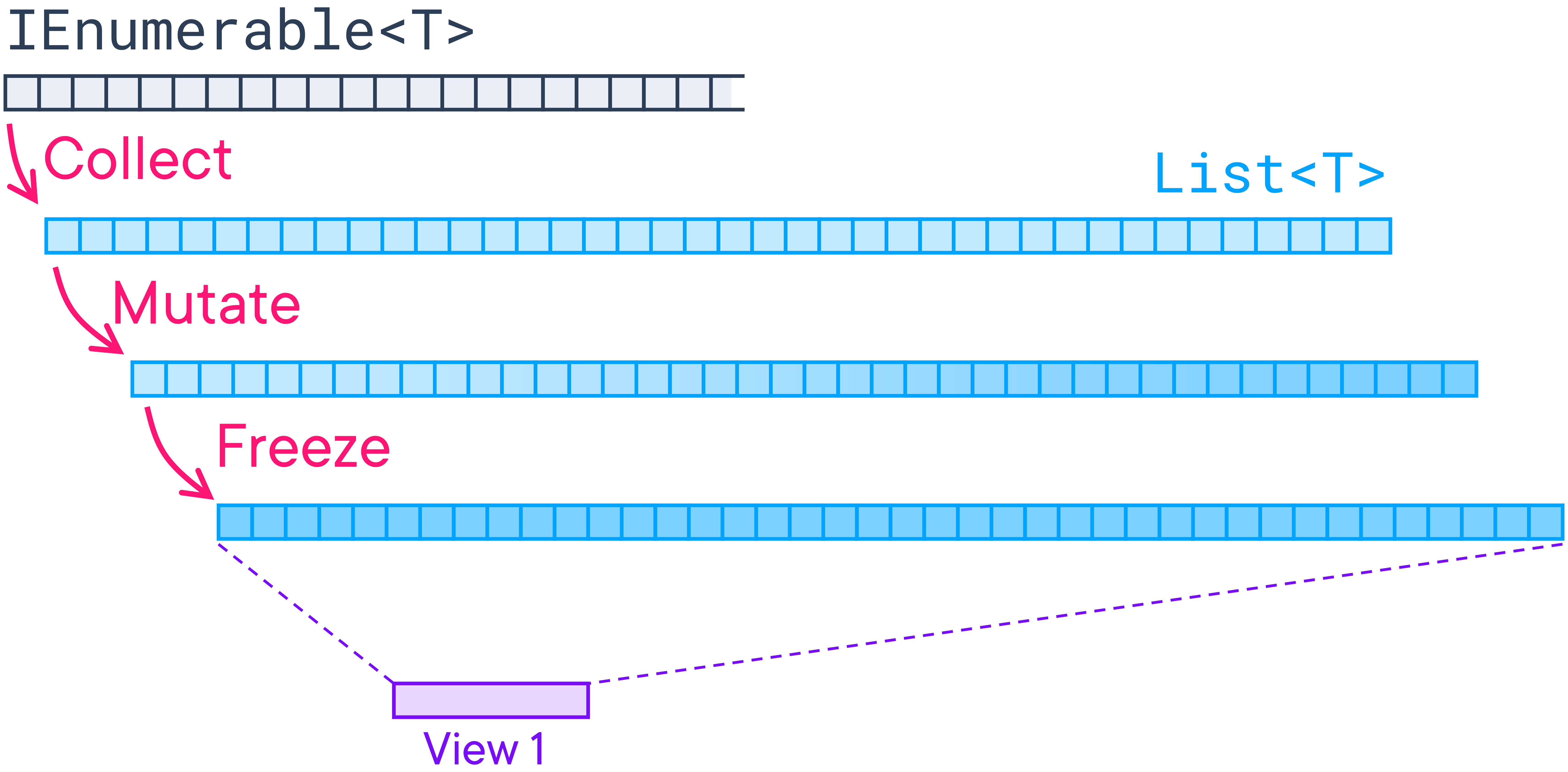
Zoran Horvat
CEO at Coding Helmet
@zoranh75 | https://codinghelmet.com

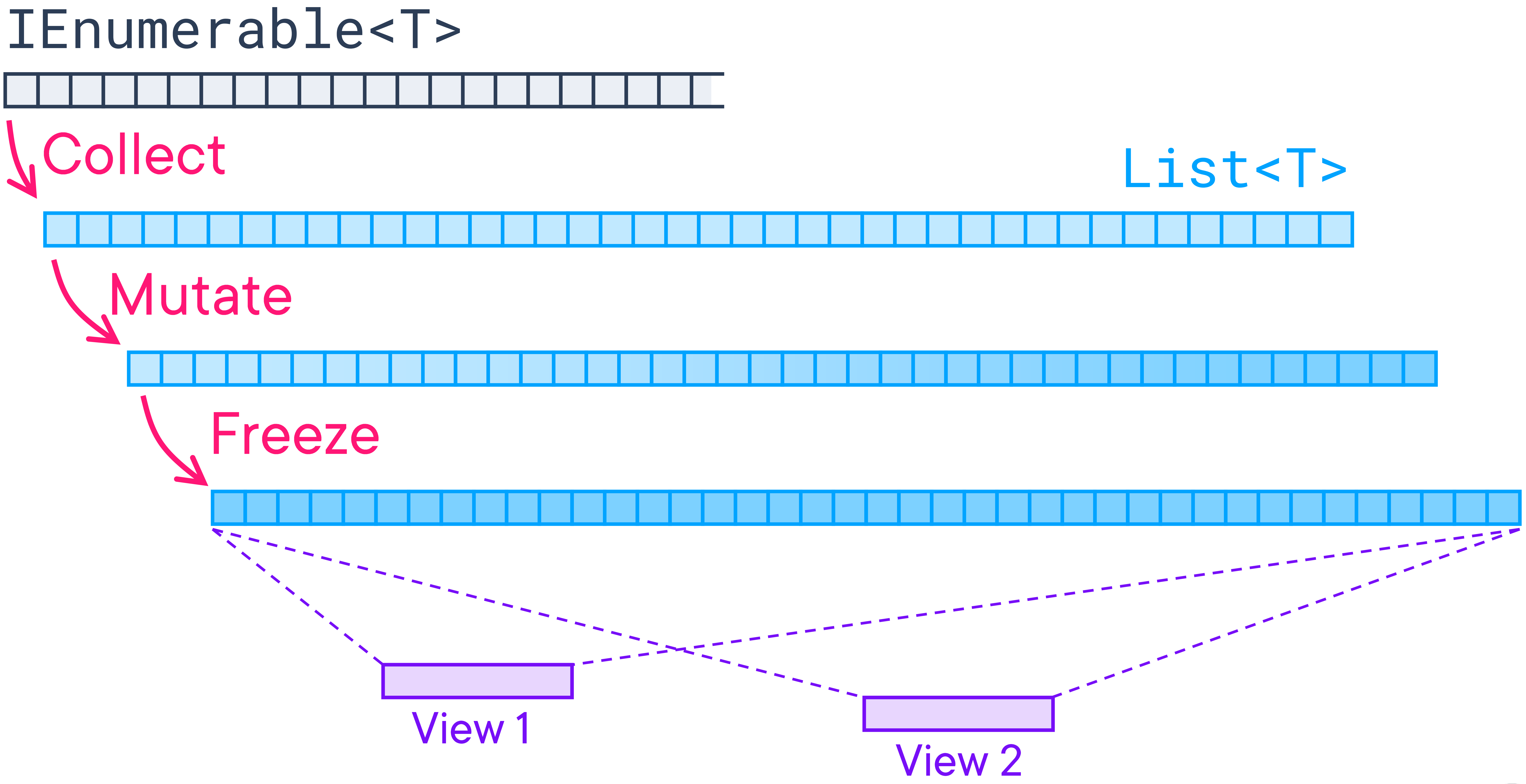
IEnumerable<T>


IEnumerable<T>
Collect
List<T>

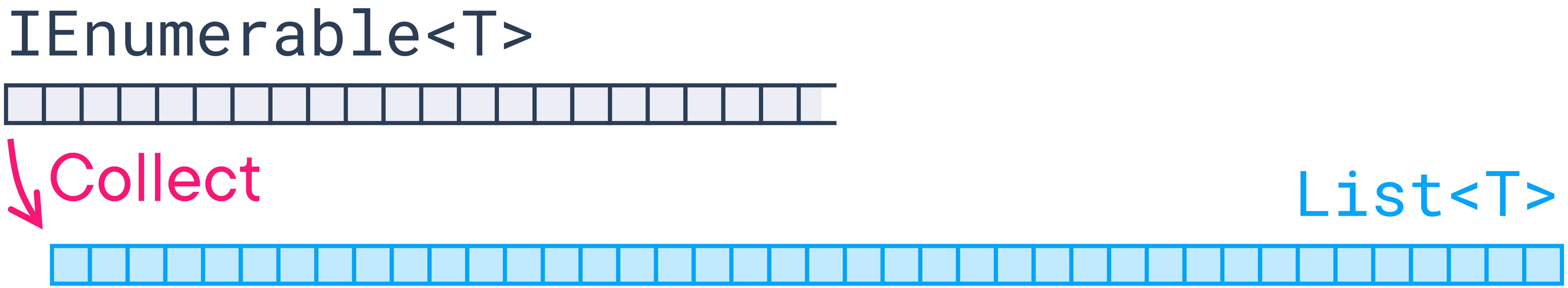

IEnumerable<T>
Collect
List<T>
Mutate

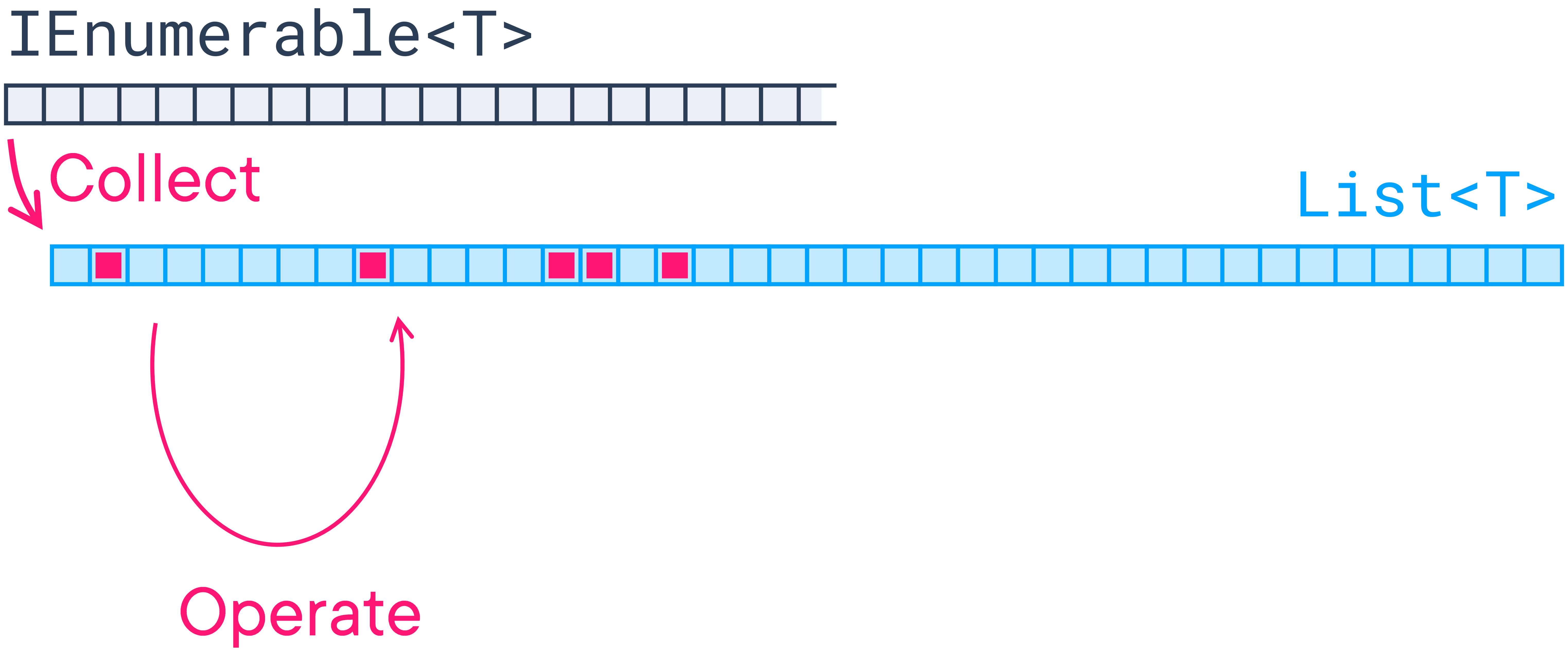

IEnumerable<T>
Collect
List<T>
Mutate
Freeze

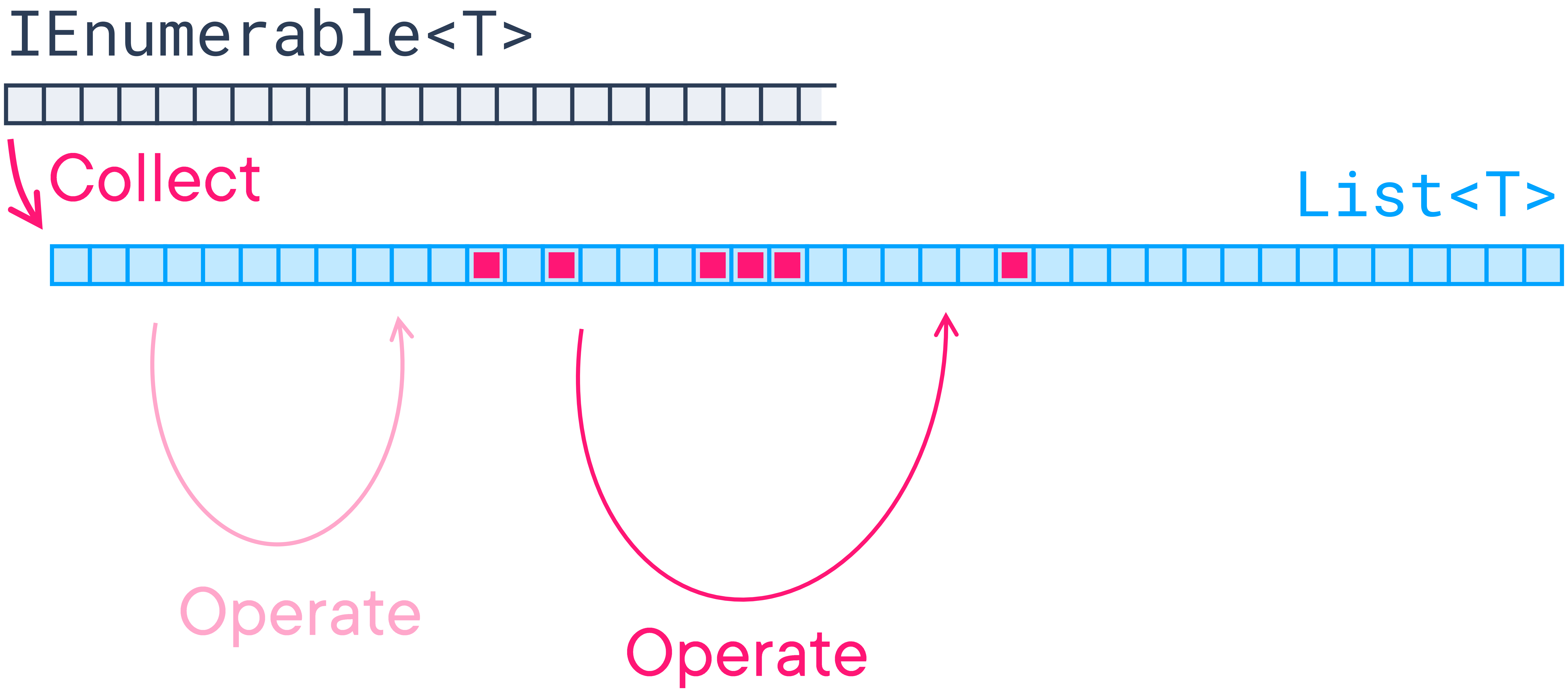

IEnumerable<T>
Collect
List<T>
Mutate
Freeze
View 1

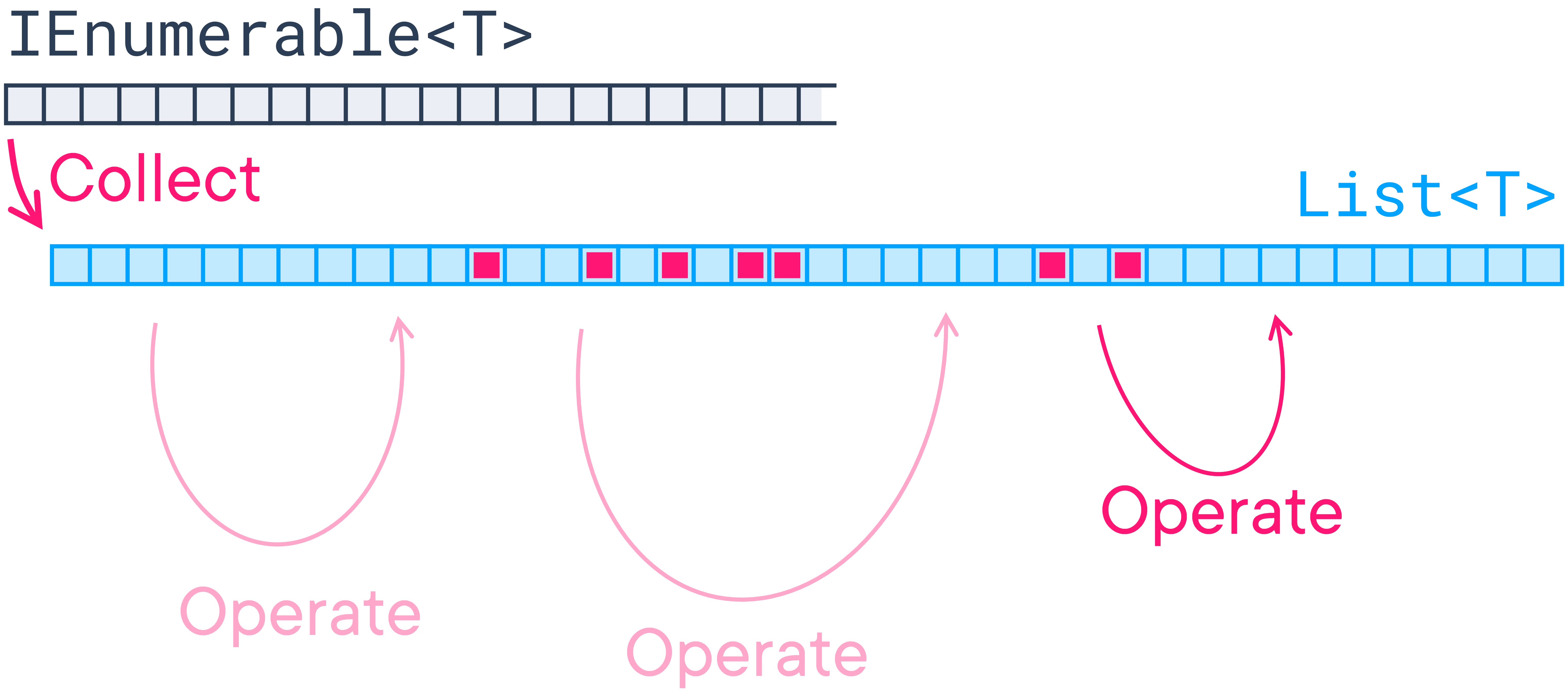


IEnumerable<T>
Collect
List<T>
Mutate
Freeze
View 1
View 2


IEnumerable<T>
Collect
List<T>


IEnumerable<T>
Collect
List<T>
Operate


IEnumerable<T>
Collect
List<T>
Operate
Operate


IEnumerable<T>
Collect
List<T>
Operate
Operate
Operate

The Constructor Principle*

Avoid costly work inside
a constructor without justification

*Opinionated view

The Augmented Constructor Principle*

Avoid work in a constructor that significantly exceeds the dimension of its arguments

*Opinionated view

```csharp
namespace Models.Common;

public class GridFormatter<T>
{
    public GridFormatter(IEnumerable<T> data)
    {
        this.Data = new List<T>(data);
    }

    private IList<T> Data { get; }

    public IEnumerable<string> Format() => Enumerable.Empty<string>();
}
```

Why list?

```csharp
namespace Models.Common;

public class GridFormatter<T>
{
    public GridFormatter(IEnumerable<T> data)
    {
        this.Data = new List<T>(data);
    }

    private IList<T> Data { get; }

    public IEnumerable<string> Format() => Enumerable.Empty<string>();
}
```

Sequence of unknown length

```
namespace Models.Common;

public class GridFormatter<T>
{
    public GridFormatter(IEnumerable<T> data)
    {
        this.Data = new List<T>(data);
    }

    private IList<T> Data { get; }

    public IEnumerable<string> Format() => Enumerable.Empty<string>();
}
```

Sequence of unknown length

List expands as needed

```csharp
namespace Models.Common;

public class GridFormatter<T>
{
    public GridFormatter(IEnumerable<T> data)
    {
        this.Data = new List<T>(data);
    }

    private IList<T> Data { get; }

    public IEnumerable<string> Format() => Enumerable.Empty<string>();
}
```

Sequence of unknown length

List expands as needed

Supports column- and row-wise traversal in a simulated matrix

```csharp
namespace Models.Common;

public class GridFormatter<T>
{
    public GridFormatter(IEnumerable<T> data)
    {
        this.Data = new List<T>(data);
    }

    private IList<T> Data { get; }

    public IEnumerable<string> Format() => Enumerable.Empty<string>();
}
```

Sequence of unknown length

List expands as needed

Supports column- and row-wise traversal in a simulated matrix

Indexer takes $O(1)$ time

```csharp
namespace Models.Common;

public class GridFormatter<T>
{
    public GridFormatter(IEnumerable<T> data)
    {
        this.Data = new List<T>(data);
    }

    private IList<T> Data { get; }

    public IEnumerable<string> Format() => Enumerable.Empty<string>();
}
```

Sequence of unknown length

List expands as needed

Supports column- and row-wise traversal in a simulated matrix

Indexer takes $O(1)$ time

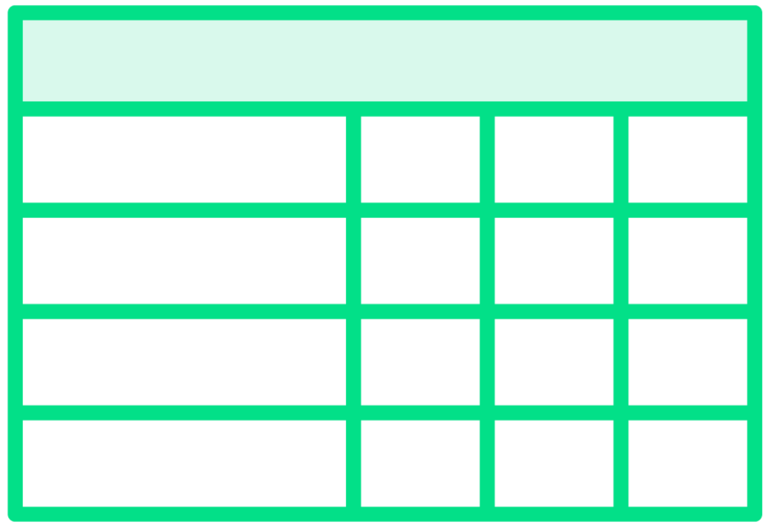
Count property takes $O(1)$ time

Comparing Lists and Arrays

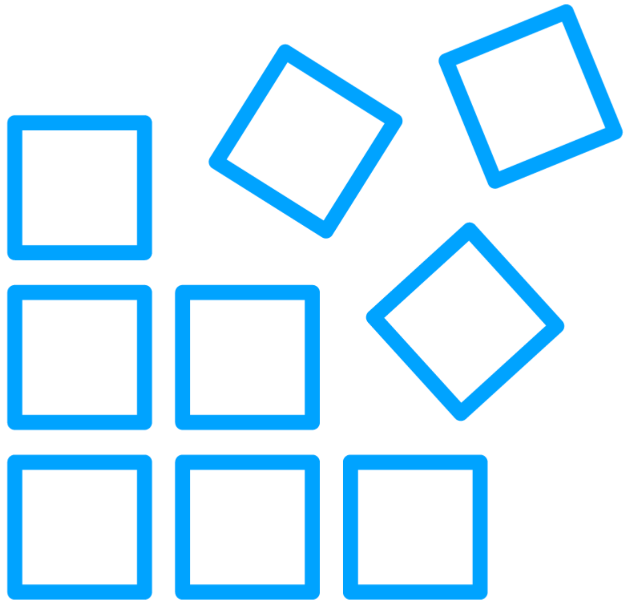
`List<T>` vs `T[]`

`List<T>`	`T[]`
Exposes indexer with range checks	**Exposes indexer with range checks**
	Efficient iteration in some corner cases
Collected using ToList() **operator**	**Collected using** ToArray() **operator**
ToList() **collects straight into the list**	ToArray() **uses intermediate storage**
Completes collecting data in one go	**Requires one more copy operation**
Half of underlying array not used	**All array locations are used**
Not trimmed list wastes memory	**Array uses memory optimally**

The New Problem Domain

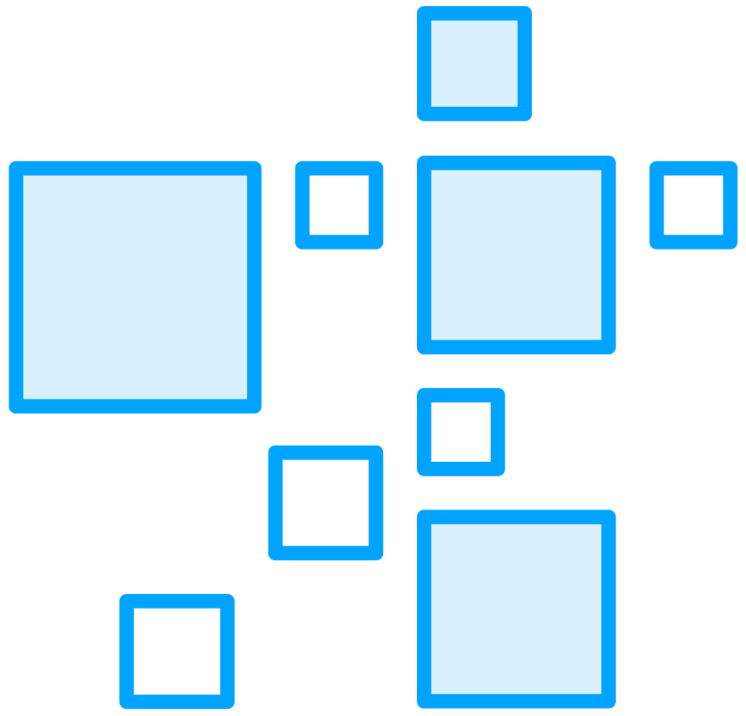


Done:
The grid formatter

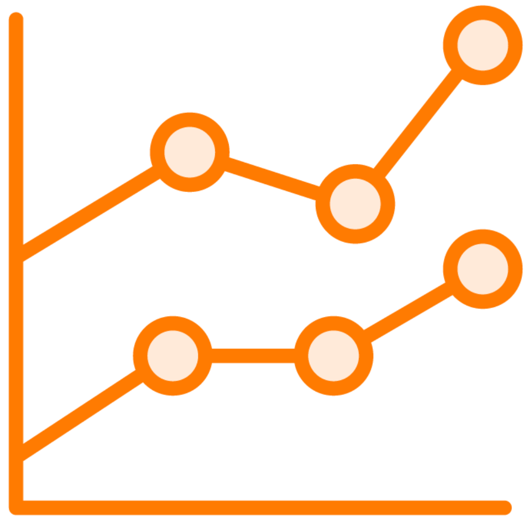


Next task:
The list randomizer

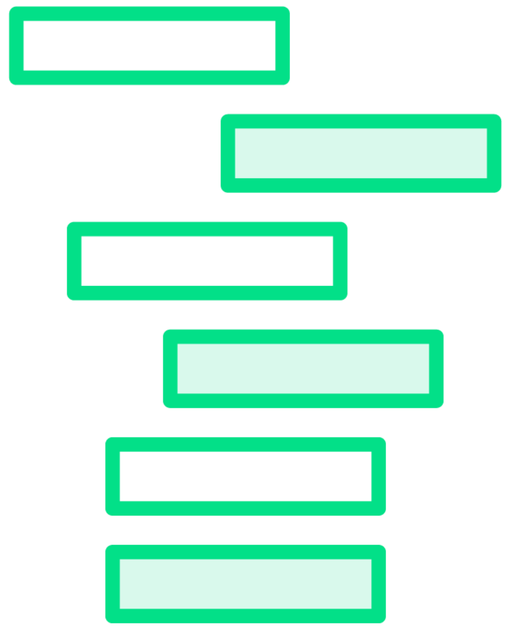
Introducing Randomized Algorithms



Randomized algorithms used in business applications



"What if" analysis simulates future events



We shall implement collection shuffling

Defining Requirements

Given a sequence,
reproduce it in shuffled order

Every permutation is equally
probable and independent

```
IEnumerable<Worker> workers;
```

Repeated reading will yield
a different order of objects

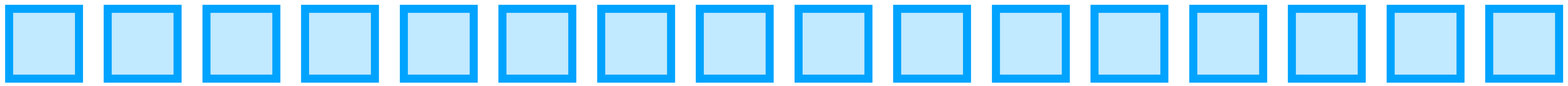
```
var a = shuffle(workers);
var b = shuffle(workers);
```

Inventing the Shuffling Algorithm

Theorem:

Given equally probable, independent permutations, each of the N items has uniform probability distribution of possible positions

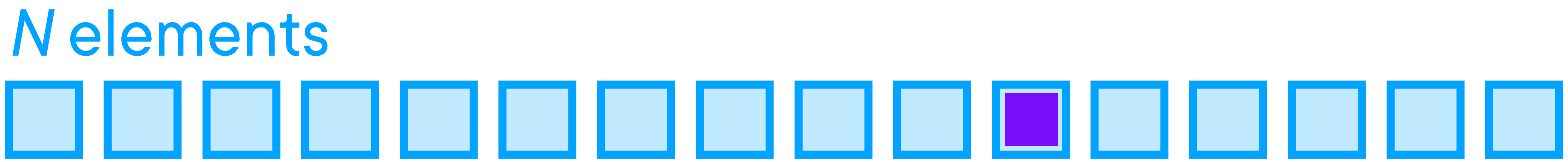
N elements



Inventing the Shuffling Algorithm

Theorem:

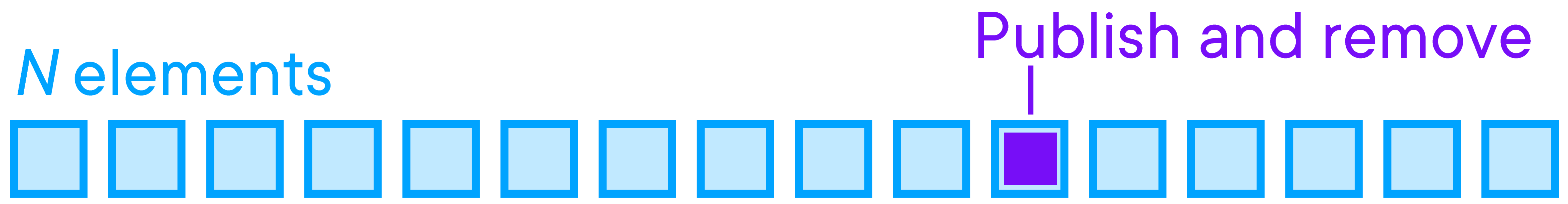
Given equally probable, independent permutations, each of the N items has uniform probability distribution of possible positions



Inventing the Shuffling Algorithm

Theorem:

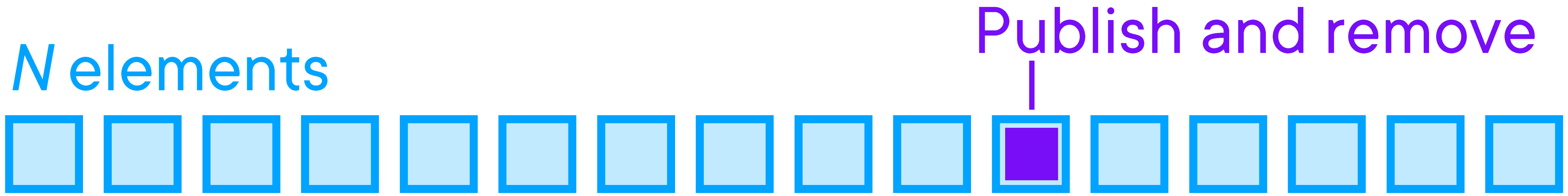
Given equally probable, independent permutations, each of the N items has uniform probability distribution of possible positions



Inventing the Shuffling Algorithm

Theorem:

Given equally probable, independent permutations, each of the N items has uniform probability distribution of possible positions



$$P_1 = \frac{1}{N} \qquad P_2 = \frac{N-1}{N} \cdot \frac{1}{N-1} = \frac{1}{N}$$

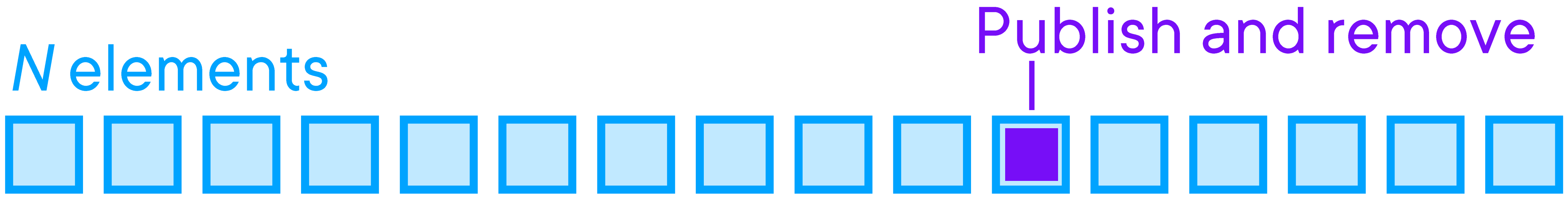
Fisher-Yates Shuffle*

$$P_k = \frac{N-1}{N} \cdot \frac{N-2}{N-1} \cdot \ldots \cdot \frac{N-k+1}{N-k+2} \cdot \frac{1}{N-k+1} = \frac{1}{N}$$

Inventing the Shuffling Algorithm

Theorem:

Given equally probable, independent permutations, each of the N items has uniform probability distribution of possible positions



$$P_1 = \frac{1}{N} \qquad P_2 = \frac{N-1}{N} \cdot \frac{1}{N-1} = \frac{1}{N}$$

Fisher-Yates Shuffle*

$$P_k = \frac{N-1}{N} \cdot \frac{N-2}{N-1} \cdot \ldots \cdot \frac{N-k+1}{N-k+2} \cdot \frac{1}{N-k+1} = \frac{1}{N}$$

*https://en.wikipedia.org/wiki/Fisher-Yates_shuffle

Implementing the Fisher-Yates Shuffle

We need to shuffle an input sequence	Sequence cannot tell the number of elements
Can we use a list/array?	No efficient item removal
Can we use a dictionary?	What would be the key?

Summary

We have used lists and arrays to implement complex algorithms

Sequence (`IEnumerable<T>`) is what we are processing

Collections are required to satisfy (often nonfunctional) requirements

Summary

Comparing a list to an array

- List expands as we add objects to it
- Up to a half of the (untruncated) list's memory remains unused
- Array leaves no unused locations
- Collecting into an array requires one additional reallocation and copying
- Both offer efficient random access

Summary

Demo collecting sequence into a list

- Collected the sequence to ensure there will be a single iteration
- Implementation formed a hierarchy of views/queries into the collection

Demo with a mutating collection

- Successive iterations must be isolated
- Implemented `IEnumerator<T>` to ensure isolation
- Caller must `Reset` the enumerator before reuse

Up Next:

Building on Ordered and Partially Ordered Lists