

INTERFACES

LINQ, Extension Methods, Exceptions, and Using Statements

Interfaces

A set of methods and properties that a class or struct must implement

Like a contract

It's a kind of type that can refer to a category of types

Similar to a base class

Interfaces can be used

As the return
type of a
function

As the type of
a parameter

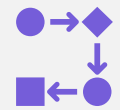
As the type of
a field or
property

As the type of
a variable

You can't create an interface using new



You can only create classes or structs



Therefore values never have a run-time type of an interface



The run-time type is based on the type a value was created with



In other words the type passed to the "new" operator

Interface Segregation Principle

- https://en.wikipedia.org/wiki/Interface_segregation_principle
- Don't require service to implement unnecessary methods
- Don't require clients to depend on non-critical methods
- In summary: keep interfaces as small as possible

Querying Interfaces at Run-time

- They have the run-time type that is associated with the “new” operation
- You can query the run-time type and see if I can assign it to the interface
- <https://learn.microsoft.com/en-us/dotnet/api/system.type.isassignablefrom>

Abstract Base Class vs Interfaces

You can only have one base class but multiple interfaces

Abstract base class can have fields and constructor

Abstract base class can specify static members

Structs can not inherit from classes, but can implement interfaces

What's
wrong with
abstract
base class?

Nothing really, they are
just very permissive

Can increase dependency
on implementation details

Extension Methods

Allow you to add functions to classes or interfaces without modifying code



Can be called as if they were defined in the class or interface

Example of Extension Method

```
public static IEnumerable<T> Repeat<T>(this T self, int count)
{
    for (var i = 0; i < count; i++)
        yield return self;
}

public static void TestRepeat()
{
    foreach (var x in "Hello world".Repeat(10))
        Console.WriteLine(x);
}
```

Extension Method Syntax and Rules

Extension methods can only be placed in a static class

They are applied to any type: class, struct, interface

The first argument must be prefixed with the “this” keyword

It can be invoked as a regular static method, or using object syntax

LINQ

- Language Integrated Natural Query (pronounced “Link”)
- It is a key part of the systems library
- Functions work on all collections and sequences
- Including iterator methods
- Fun and easy

LINQ Implementation

- It is implemented as a library of extension methods
- Works with any class that implements `IEnumerable<T>`
- Including your own
- Easy to write using iterator methods

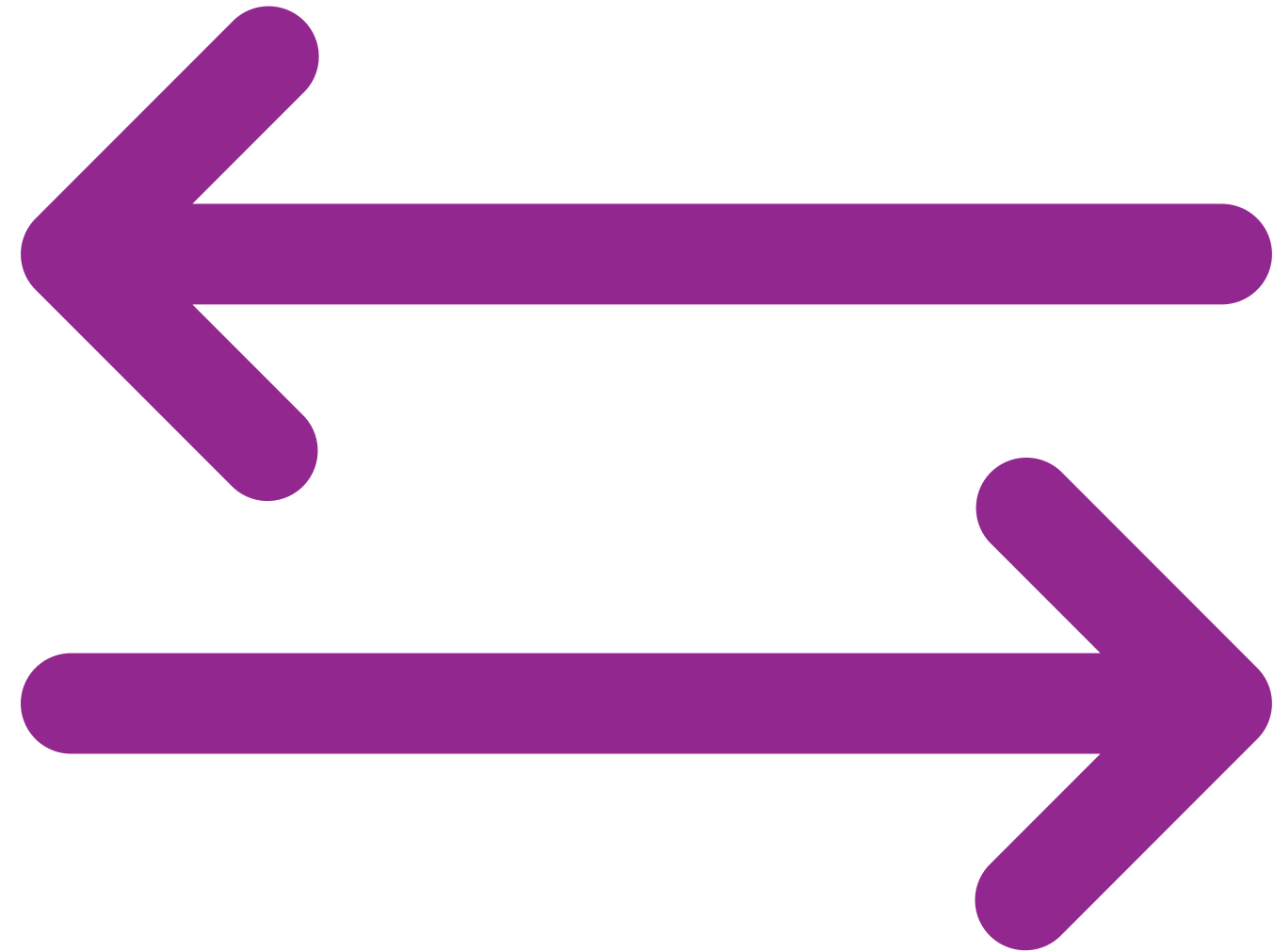
Alternative Query Syntax (not recommended)

//Query syntax:

```
var numQuery1 =  
    from num in numbers  
    where num % 2 == 0  
    orderby num  
    select num;
```

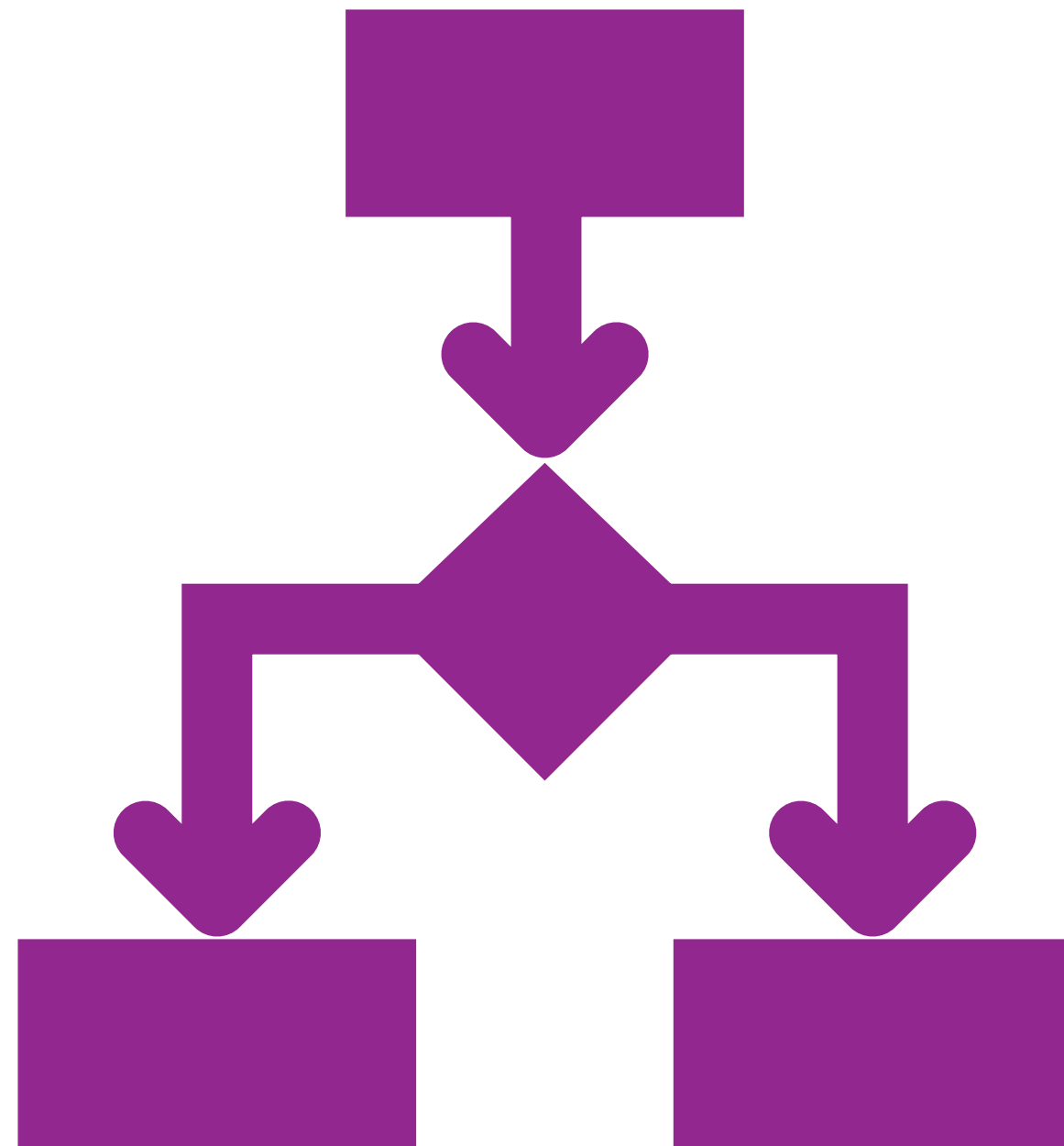
//Method syntax:

```
var numQuery2 = numbers  
    .Where(num => num % 2 == 0)  
    .OrderBy(n => n);
```



Throw Statements

- Throw statements say ... get me out of this function now.
- More than just a return
- Also exits the function that called the current function
- Does this recursively until it reaches a "catch" block



When to use Throw Statements

- Something happened that makes continuing a process complicated or pointless
- For example:
 - A file no longer exists or is inaccessible
 - A value passed as an argument to a function is invalid
 - Something unexpected happened
 - Detected a corrupted intermediate result
- Sometimes people use it to represent “canceling” a user operation



Exception

- Throw statements may be associated with an exception
- An exception is an object that has extra associated information
- Including where the exception was thrown (i.e., stack trace)

Call Stack

- A list of functions that have been called, but not returned yet
- Technically a stack of “frames”
- We step in and out of stack frames using the debugger

Stack Frame

- Records state of a function when invoking another function
- Stores value of local variables
- Pointer to next instruction to be executed when function returned

Garbage Collection

- Non-deterministic
- Sometime object refer to unmanaged resources
- Examples might include a file stream
- When it is no longer required we must “release” it
- In C++ this would be done in the destructor
- C# has destructors, but they might not be called for a long time

IDisposable.Dispose

```
graph LR; A((Provides a mechanism for releasing unmanaged resources)) --> B((For example: closing a file)); B --> C((Or deleting an object in a C++ library));
```

Provides a mechanism for releasing unmanaged resources

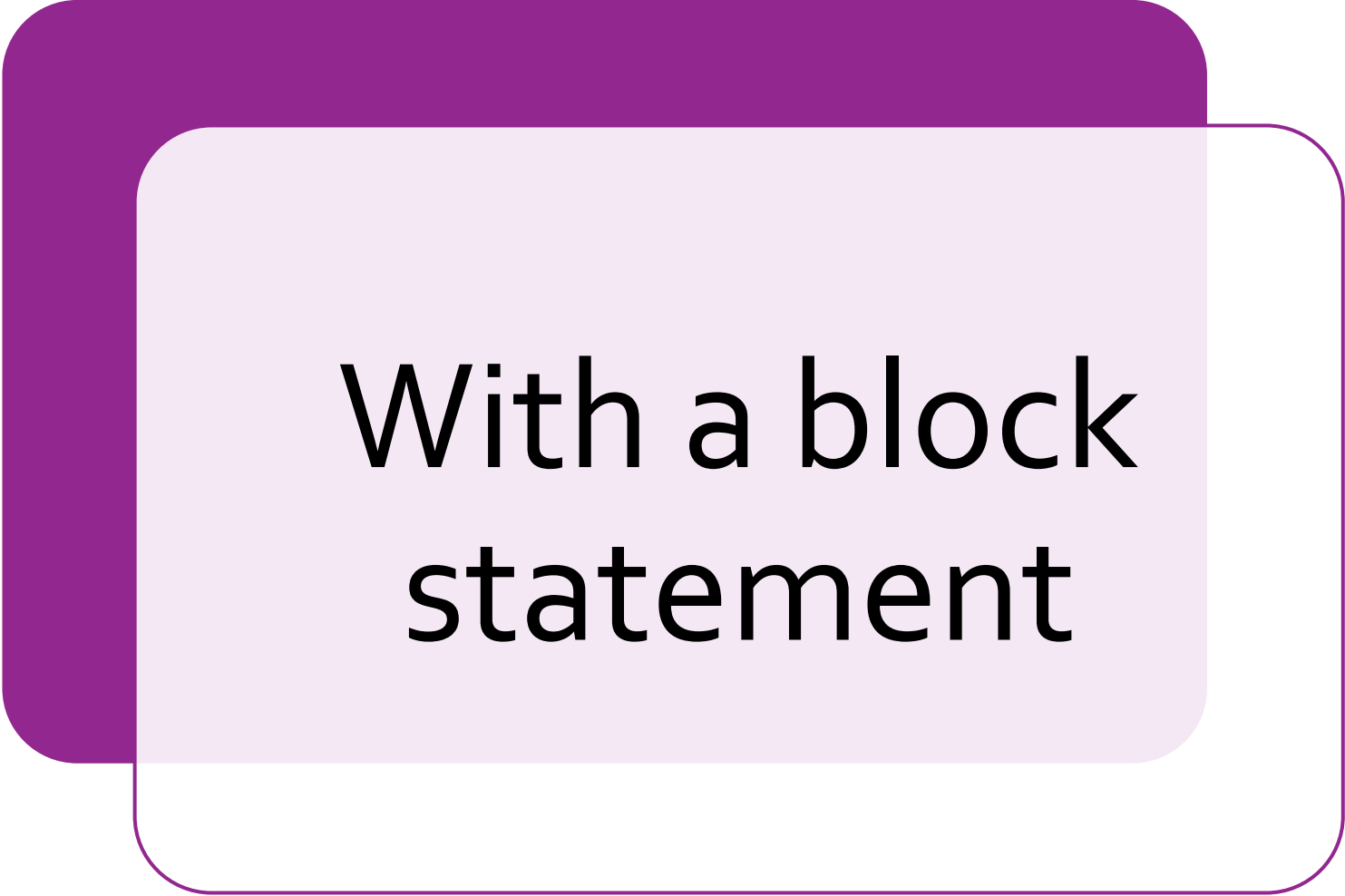
For example: closing a file

Or deleting an object in a C++ library

Using Statements

- Simplifies deterministic clean-up of unmanaged resource
- Usually this is things created by the operating system
- Or maybe something created in an external library
- Don't confuse them with using directives: not related to namespaces

Two Forms of the Using Statement



With a block
statement



Without a block
statement

How using statement works

- Assign an expression to a variable
- That expression implements "IDisposable"
- When the variable goes out of scope the Dispose function is called
- Called the "RAII"

Evaluation on Demand



Also called “lazy evaluation”.



Values in an IEnumerable are retrieved only when requested



This is why an IEnumerable can represent infinite sequences

Implementing Foreach: Attempt #1

```
public static void ForEach1<T>(this IEnumerable<T> self, Action<T> action)
{
    // Warning: instance of enumerator never disposed
    for (var enumerator = self.GetEnumerator(); enumerator.MoveNext(); )
    {
        action(enumerator.Current);
    }
}
```

Implementing Foreach: Attempt #2

```
public static void ForEach2<T>(this IEnumerable<T> self, Action<T> action)
{
    var enumerator = self.GetEnumerator();
    while (enumerator.MoveNext())
    {
        // Note: if an exception is thrown, the enumerator is not disposed
        action(enumerator.Current);
    }
    enumerator.Dispose();
}
```

Implementing Foreach: Attempt #3

```
public static void ForEach3<T>(this IEnumerable<T> self, Action<T> action)
{
    var enumerator = self.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            action(enumerator.Current);
        }
    }
    finally
    {
        // Always executed before leaving the function
        enumerator.Dispose();
    }
}
```

Implementing Foreach: Attempt #4

```
public static void Foreach4<T>(this IEnumerable<T> self, Action<T> action)
{
    // Also assures that Dispose is called before leaving the function
    using (var enumerator = self.GetEnumerator())
    {
        while (enumerator.MoveNext())
        {
            action(enumerator.Current);
        }
    }
}
```

Implementing Foreach: Attempt #5

```
public static void ForEach5<T>(this IEnumerable<T> self, Action<T> action)
{
    // A newer simplified version of the using statement
    using var enumerator = self.GetEnumerator();
    while (enumerator.MoveNext())
    {
        action(enumerator.Current);
    }
}
```

Select

Creates a sequence from a sequence by applying a transform function to each element while respecting the order. Also called a “map”.

```
public static double SumOfSquares(this IEnumerable<double> self)
{
    return self.Select(Math.Sqrt).Sum();
}
```

```
public static IEnumerable<T2> Select<T1, T2>(this IEnumerable<T1> self,  
    Func<T1, T2> transform)  
{  
    foreach (var item in self)  
        yield return transform(item);  
}
```

Select implementation

Where

Creates a new sequence of values from another but keeping only elements for which a predicate returns true. Also called a “filter”.

```
public static IEnumerable<int> Primes(this IEnumerable<int> values)
{
    return values.Where(IsPrime);
}
```

```
public static IEnumerable<T> Where<T>(this IEnumerable<T> self,  
    Func<T, bool> predicate)  
{  
    foreach (var item in self)  
        ....  
        if (predicate(item))  
            yield return item;  
}
```

Where implementation

Aggregate

Applies an accumulator function combining each elements with the accumulated value. A generalization of the “sum” function. Also known as “fold”.

```
public static int Count<T>(this IEnumerable<T> self)
{
    return self.Aggregate(0, (acc, x) => acc + 1);
}
```

```
public static TAcc Aggregate<T, TAcc>(this IEnumerable<T> self,  
    TAcc acc, Func<TAcc, T, TAcc> f)  
{  
    foreach (var item in self)  
        acc = f(acc, item);  
    return acc;  
}
```

Aggregate Implementation

Concat, Append, Prepend

Take a sequence and an element, or another sequence and create a new sequence.

No container is created and the original sequence is not iterated.

Very efficient when used once or twice.

Using them repeatedly to create a sequence from scratch would be very slow

Each function call creates a complex object

Concat, Append, Prepend

```
public static IEnumerable<T> Concat<T>(this IEnumerable<T> self, IEnumerable<T> other)
{
    foreach (var x in self)
        yield return x;
    foreach (var y in other)
        yield return y;
}
```

```
public static IEnumerable<T> Append<T>(this IEnumerable<T> self, T value)
{
    foreach (var x in self)
        yield return x;
    yield return value;
}
```

```
public static IEnumerable<T> Prepend<T>(this IEnumerable<T> self, T value)
{
    yield return value;
    foreach (var x in self)
        yield return x;
}
```

SelectMany

- Given a sequence of sequences, converts it into a single sequence.
- Also called “flat map”.

```
public static IEnumerable<T> SelectMany<T>(this IEnumerable<IEnumerable<T>> self)
{
    foreach (var x in self)
    {
        foreach (var y in x)
        {
            yield return y;
        }
    }
}
```

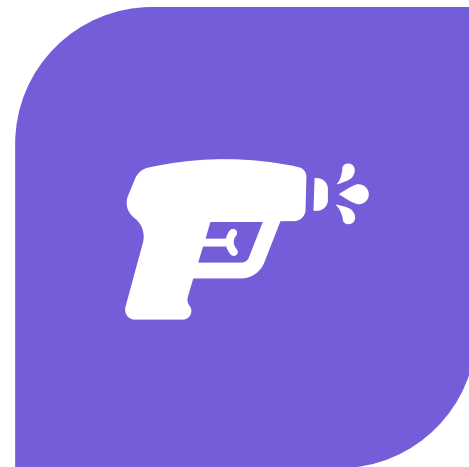
SelectMany using Aggregate

```
public static IEnumerable<T> SelectMany2<T>(this IEnumerable<IEnumerable<T>> self)
{
    return self.Aggregate(Empty<T>(), Concat);
}
```

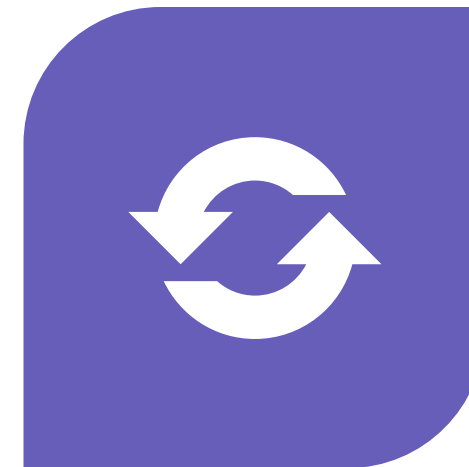

Creating Sequences



EMPTY



RANGE



REPEAT

What is an Iterator Method

Any method that has a yield statement

Creates an object under the hood

This object stores all local variables

Also stores the next instruction

Just like a stack frame

Yield Break and Empty Sequence

```
public static IEnumerable<T> Empty<T>()  
{  
    yield break;  
}
```

Ranges of Integers (Custom Functions)

```
public static IEnumerable<int> Range(this int count)
{
    return 0.UpTo(count);
}
```

```
public static IEnumerable<int> UpTo(this int from, int upTo)
{
    for (var i = from; i < upTo; ++i)
        yield return i;
}
```

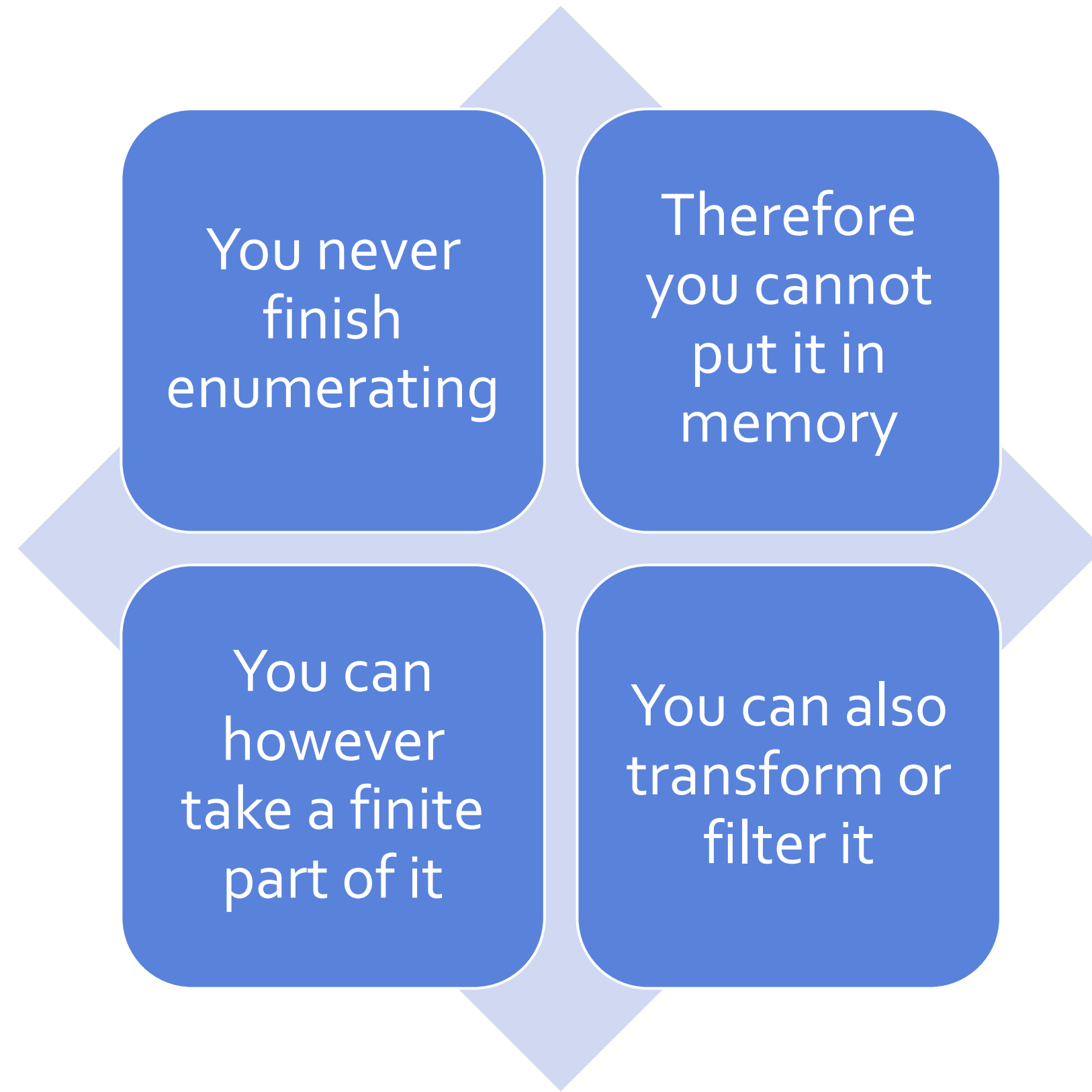
Repeat

```
public static IEnumerable<T> Repeat<T>(this T self, int count)
{
    for (var i = 0; i < count; i++)
        yield return self;
}
```

Fibonacci Series as an Iterator Method

```
public static IEnumerable<int> FibonacciSeries()
{
    yield return 0;

    var a = 0;
    var b = 1;
    while (true)
    {
        var c = a + b;
        yield return c;
        a = b;
        b = c;
    }
}
```



Infinite Sequences

I can call the
following

Select

Where

Prepend

Take

Skip

First

Zip

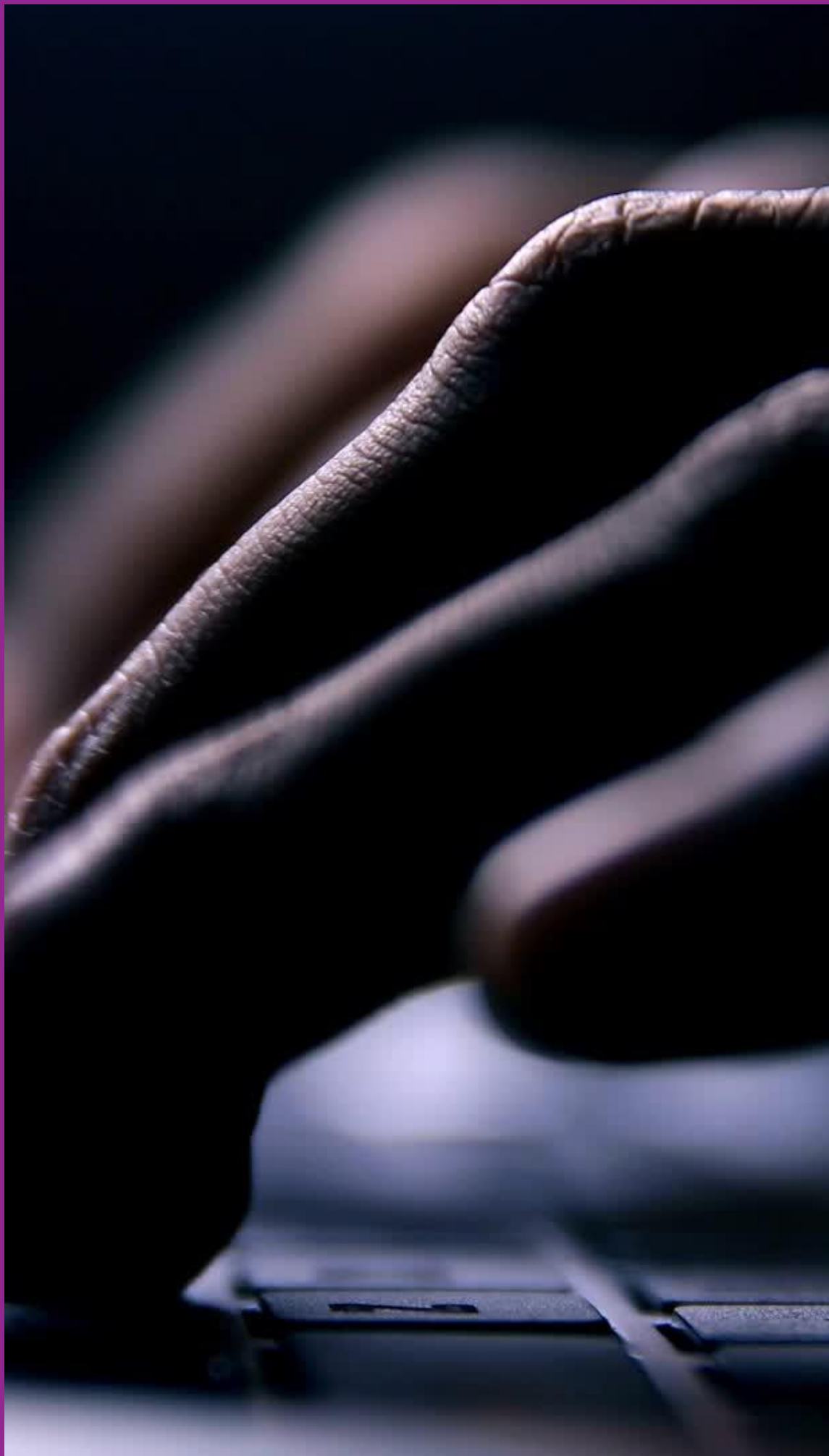
The following will
loop forever

- Count
- Aggregate
- ToArray
- ToList
- Any
- All
- OrderBy
- Last

```
public interface IEnumerable<T>
{
    IEnumerator<T> GetEnumerator<T>();
}
```

```
public interface IEnumerator<T>
{
    public bool MoveNext();
    public T Current { get; }
}
```

IEnumerable AND IEnumerator



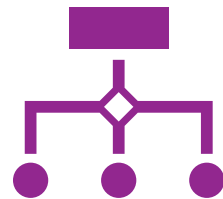
Why you shouldn't enumerate twice

- A sequence might be infinite
- A sequence might be transient – e.g. values read from keyboard
- Inefficient, but this is a minor concern

What is an Array?



A mapping from
integers to values



Can retrieve
count, and
elements, in
constant time



It may or may not
be mutable (i.e.,
allow updates)



If not mutable,
things can get
interesting!

```
public interface IArray<T>
{
    int Count { get; }
    T this[int index] { get; }
}
```

IARRAY

A Concrete Implementation of IArray

```
public class Array<T> : IArray<T>
{
    public int Count { get; }

    public Func<int, T> Mapping { get; }

    public Array(int count, Func<int, T> mapping)
        => (Count, Mapping) = (count, mapping);

    public T this[int index]
        => Mapping(index);
}
```

Why is this relevant?

- LINQ style operations on this class take $O(1)$ memory and $O(1)$ time
- In other words you don't have to iterate over the entire list
- And you don't have to allocate a block of memory

A Helper Function to Create IArrays

```
public static IArray<T> Create<T>(this int count, Func<int, T> func)  
    => new Array<T>(count, func);
```


Skip and Take

```
public static IArray<T> Skip<T>(this IArray<T> self, int count)  
    => Create(self.Count - count, i => self[i + count]);
```

```
public static IArray<T> Take<T>(this IArray<T> self, int count)  
    => Create(count, i => self[i]);
```

Select

```
public static IArray<T2> Select<T1, T2>(this IArray<T1> self, Func<T1, T2> func)  
    => Create(self.Count, i => func(self[i]));
```

Reverse

```
public static IArray<T> Reverse<T>(this IArray<T> self)  
    => Create(self.Count, i => self[self.Count - 1 - i]);
```

Concatenation

```
public static IArray<T> Concat<T>(this IArray<T> self, IArray<T> other)
    => Create(self.Count + other.Count,
        i => i < self.Count
            ? self[i]
            : other[i - self.Count]);
```

Where is where?

Must enumerate a filtered array before you can get count



We lose constant time indexing and constant time count



Either: create a new collection



Or: treat the result as an IEnumerable