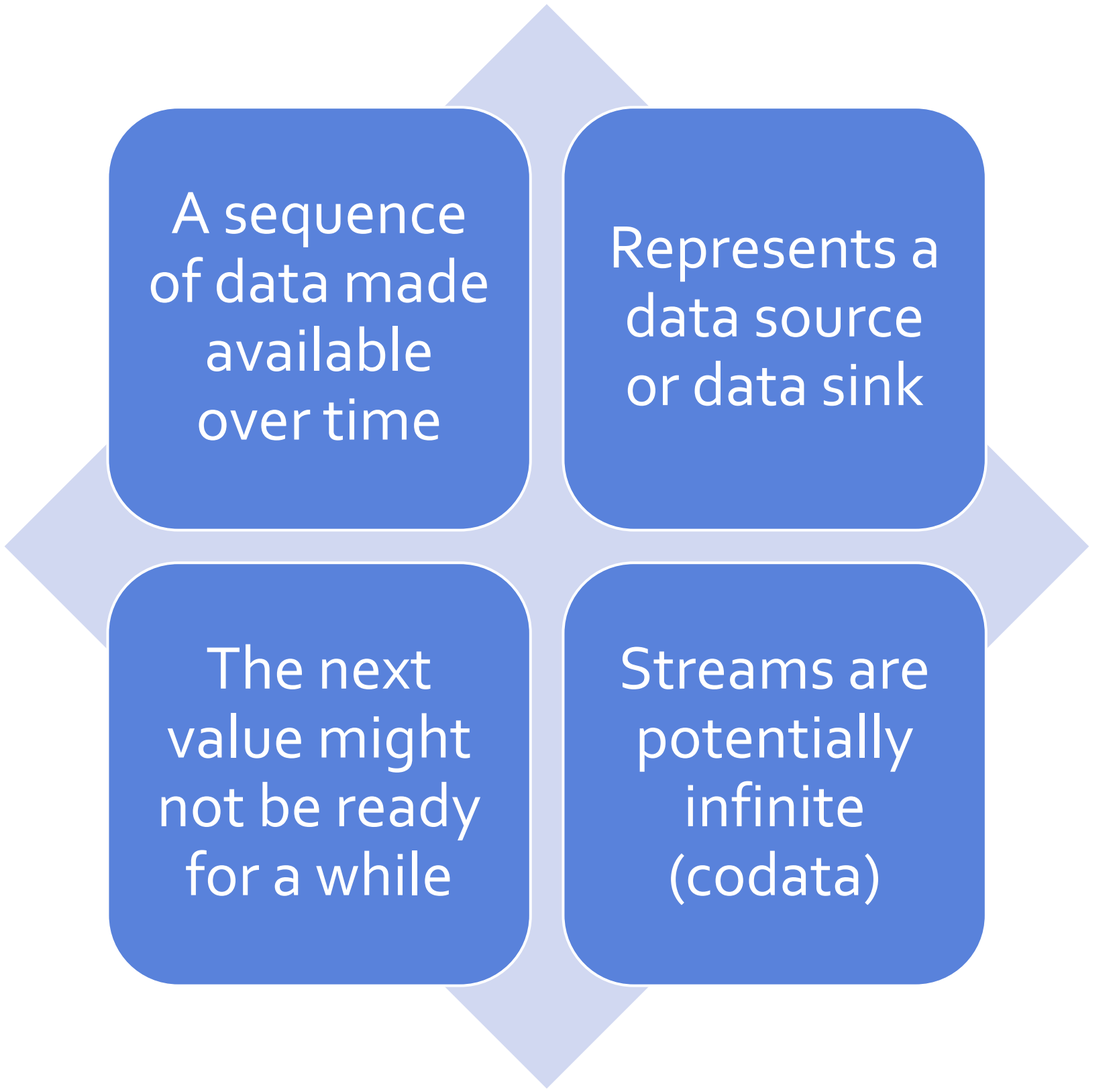
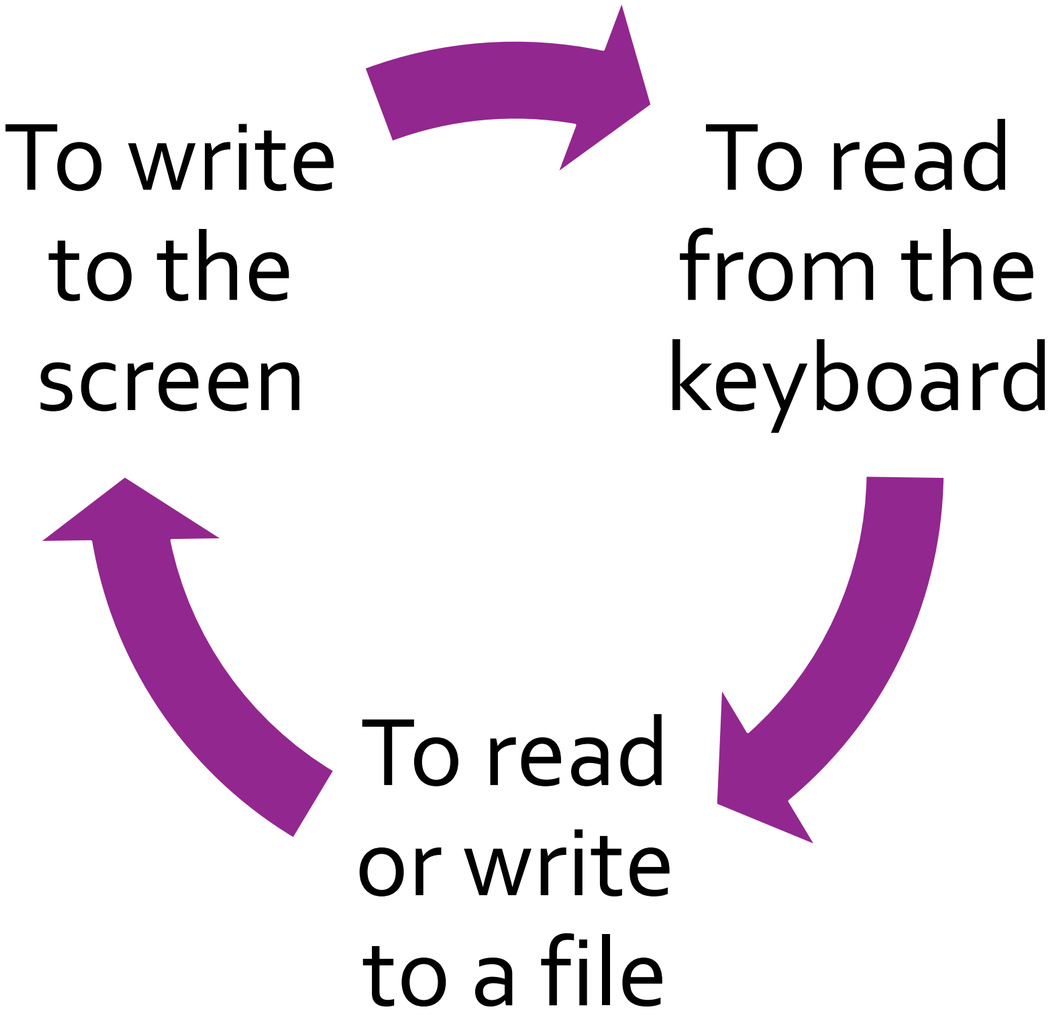
STREAMS AND ITERATORS



Streams are sequential data



When do we usually use streams?

WHAT KIND OF DATA IS IN A STREAM?

Could be any type, but is often characters, bytes, and strings (lines of text)

You already used streams

Console.WriteLine() outputs data to output stream

- That stream by default is connected to the standard output (defined by OS)
- The standard output is connected to screen by the terminal (a program)

Console.ReadLine() reads data from input stream

- That stream by default is connected to the standard input (defined by OS)
- The standard input is connected to keyboard by the terminal (a program)

Why do we use streams?

- When data can’t fit into memory all at once
- Avoids the overhead of copying data into an intermediate collection
- When we aren’t sure we will use the data
- When we want to read or write values on-demand
- When accessing elements is potentially expensive

Containers (collections) are finite

Data is always available

Data can be queried multiple times

Containers/ Collections are not Streams

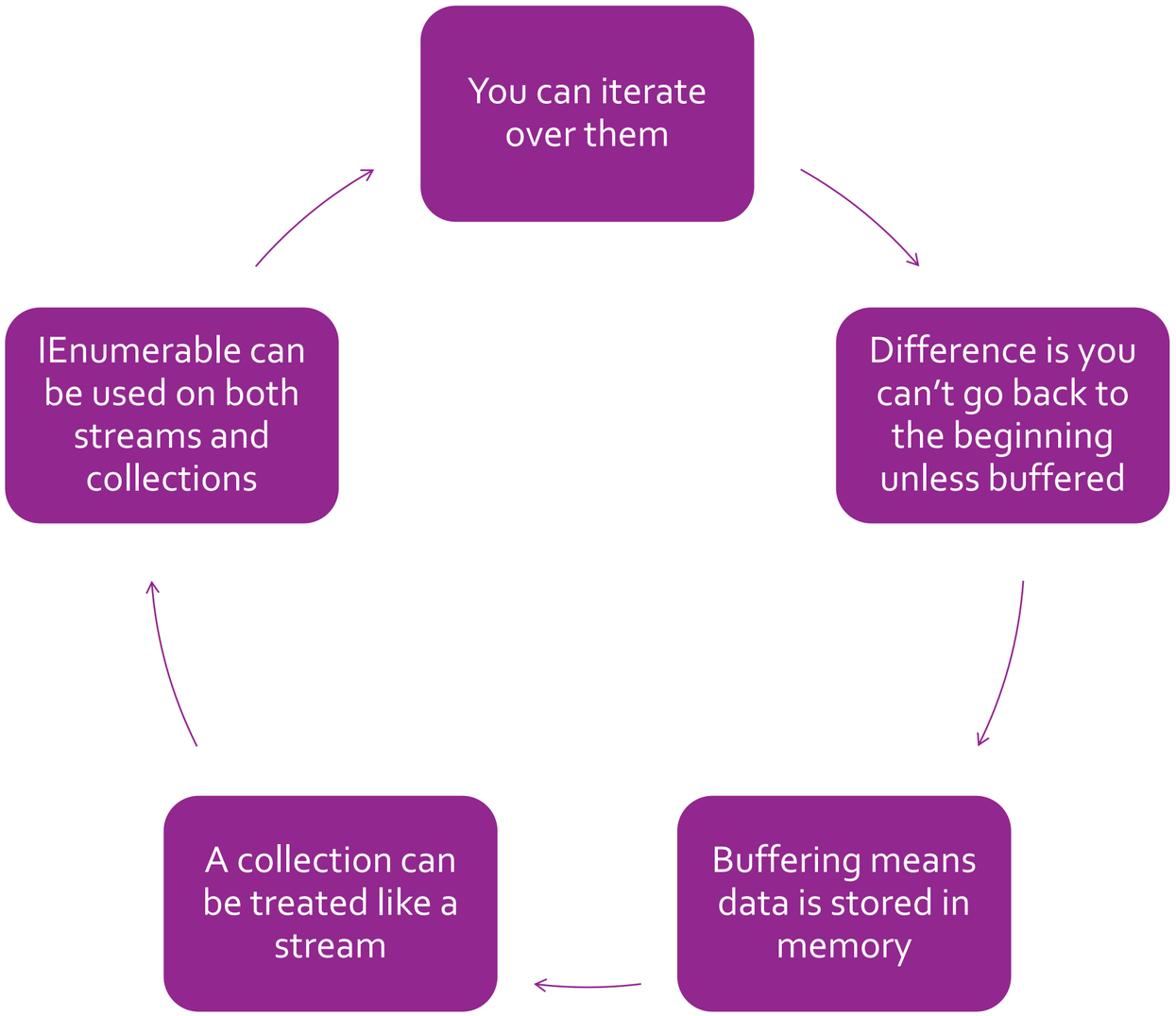
Wikipedia is contradicting itself

- https://en.wikipedia.org/wiki/Stream_(computing)
  - I/O devices can be interpreted as streams, as they produce or consume potentially unlimited data over time
  - In object-oriented programming, input streams are generally implemented as iterators
- https://en.wikipedia.org/wiki/Iterator:
  - In computer programming, an iterator is an object that enables a programmer to traverse a container, particularly lists
- So: iterators can be used for streams and containers / collections
- Don't even read the bit about containers / collections. It is a mess.

Let’s call this a sequence

- Less awkward than saying IEnumerable
- We are including standard containers and streams
- If I was rewriting the library from scratch I’d call it ISequence

Streams are Like Collections



Refresher: IEnumerable<T>

- Exposes an enumerator, which supports iteration over collections (and codata)
- The base interface for all collections that can be enumerated.
- Contains a single method, GetEnumerator(), which returns an Ienumerator<T>
- IEnumerator<T> exposes a Current property and MoveNext() and Reset() methods

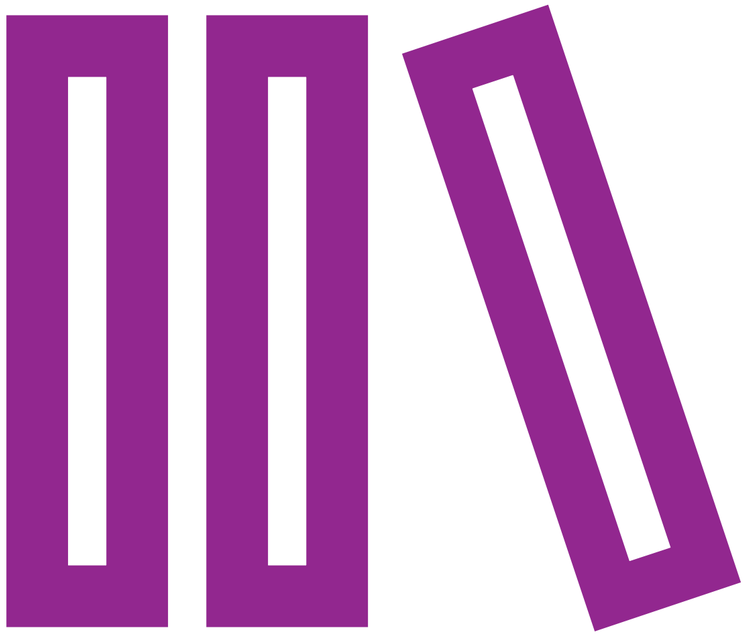
StreamReader as Iterator

- Has a “ReadLine()” method
- Returns a string or null
- If null then no more data
- This is effectively an enumerator
- We will learn how to create this later

```
public class StreamEnumerator : IEnumerator<string>
{
    public StreamEnumerator(StreamReader reader)
        => Reader = reader;

    public StreamReader Reader { get; }

    public string Current { get; private set; }

    object IEnumerator.Current
        => Current;

    public bool MoveNext()
        => (Current = Reader.ReadLine()) != null;

    public void Reset()
        => throw new NotImplementedException();

    public void Dispose()
        => Reader.Dispose();
}
```

The File.ReadLines() function does this

```
public static IEnumerable<string> ReadLines(string path, Encoding encoding)
{
    if (path == null)
        throw new ArgumentNullException(nameof(path));
    if (encoding == null)
        throw new ArgumentNullException(nameof(encoding));
    if (path.Length == 0)
        throw new ArgumentException(SR.Argument_EmptyPath, nameof(path));

    return ReadLinesIterator.CreateIterator(path, encoding);
}
```



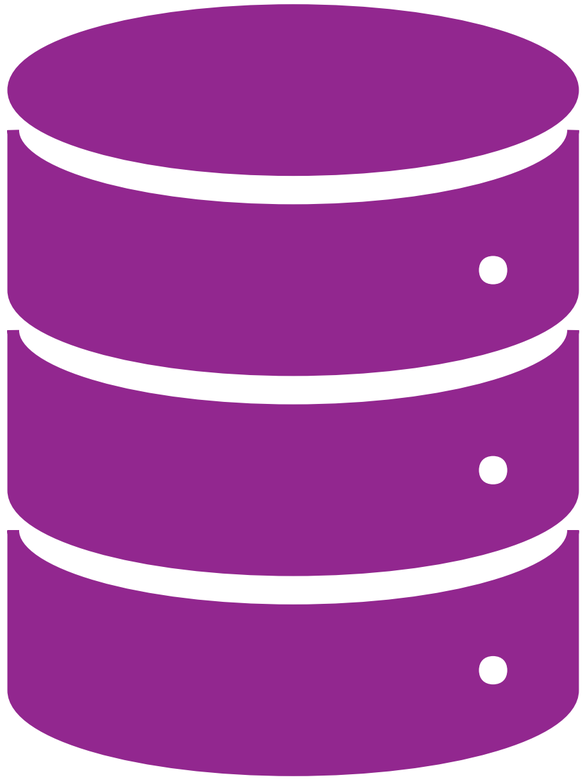
This implies

- Algorithms and functions written for streams can work on collections
- We can write a single library that accepts IEnumerable and save a lot of code
- This is what the “LINQ” library does

Here is an example of reading a file

(From reference source: notice they are using a stream reader)

```
private static string[] InternalReadAllLines(string path, Encoding encoding)
{
    Debug.Assert(path != null);
    Debug.Assert(encoding != null);
    Debug.Assert(path.Length != 0);

    string? line;
    List<string> lines = new List<string>();

    using (StreamReader sr = new StreamReader(path, encoding))
        while ((line = sr.ReadLine()) != null)
            lines.Add(line);

    return lines.ToArray();
}
```



This function returns an IEnumerable

- You can “foreach” over it
- Data is only loaded if/when requested
- Can read only part of data if requested
- No 2GB limit

They could also have written

```
public static IReadOnlyList<string> InternalReadAllLines(string fileName, Encoding encoding)
{
    return File.ReadLines(fileName, encoding).ToList();
}
```

ToList() versus ToArray()

- If I return a “IReadOnlyList()” my function can return a list or array
- A ToList() function is usually better than a “ToArray()”
- Why? Exercise for the reader.
- Hint: think about how you would implement both functions
- Note: a restriction of IEnumerable is that you can only iterate over it once

An example of a streaming into a buffer

```
public static IEnumerable<string> GetAllTextFromStandardInput_v1()
{
    var list = new List<string>();
    var line = Console.ReadLine();
    while (line != null)
    {
        list.Add(line);
        line = Console.ReadLine();
    }
    return list;
}
```

Why return IEnumerable and not List?

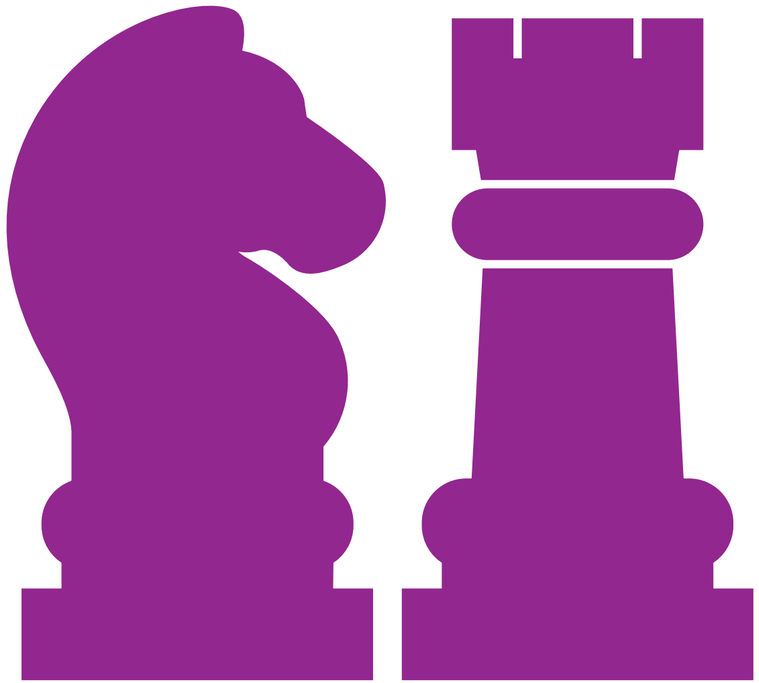
- Remember IEnumerable is an interface and List is a class
- We want flexibility to change the implementation
- We should prefer to return an interface over classes
- The reduces “coupling” and hides implementation details
- We should choose the interface that best describes the result

Could IList<T> makes sense?

- Yes it could, however it adds requirements
- Forces the data to be buffered
- In other words copied into an intermediate collection
- Requires that the intermediate collection is mutable (can be changed)
- We can only handle up to 2 billion items (e.g. 2GB binary data)

So is IReadOnlyList<T> better?

- Yes it is a bit however it still requires “indexing”
- Indexing a stream means it must be in memory
- Or if it is a file we must perform a seek operation
- Seeking in a file is slow, hogs resources, and is best avoided



Choosing the Return Type is Important

- It’s like developing your pieces early in chess
- Decisions you make early on affect your options later on

Guiding Principle

Return interfaces to minimize dependency on implementation details

Only return interfaces that the client needs or expects

This is related to the “Interface Segregation Principle”

Interface segregation principle

Article Talk

From Wikipedia, the free encyclopedia

In the field of software engineering, the **interface segregation principle** (**ISP**) states that no code should be forced to depend on methods it does not use.[1] ISP splits interfaces that are very large into smaller and more specific ones so that clients will only have to know about the methods that are of interest to them. Such shrunken interfaces are also called *role interfaces*.[2] ISP is intended to keep a system decoupled and thus easier to refactor, change, and redeploy. ISP is one of the five SOLID principles of object-oriented design, similar to the High Cohesion Principle of GRASP.[3] Beyond object-oriented design, ISP is also a key principle in the design of distributed systems in general and microservices in particular. ISP is one of the six IDEALS principles for microservice design.[4]

```
public static IEnumerable<string> GetAllTextFromStandardInput()
{
    var line = Console.ReadLine();
    while (line != null)
    {
        yield return line;
        line = Console.ReadLine();
    }
}
```

AN ALTERNATIVE IMPLEMENTATION

Iterator Methods (Coroutines)

- An iterator method is a function that generates a stream
- See: https://learn.microsoft.com/en-us/dotnet/csharp/iterators
- Understanding them fully requires understanding how to implement interfaces
- They act like functions which can be paused and resumed
- In other words they are a form of coroutine