

Reading and Writing Data Incrementally Using Streams



Jason Roberts

.NET Developer

@robertsjason | dontcodetired.com



Module Overview

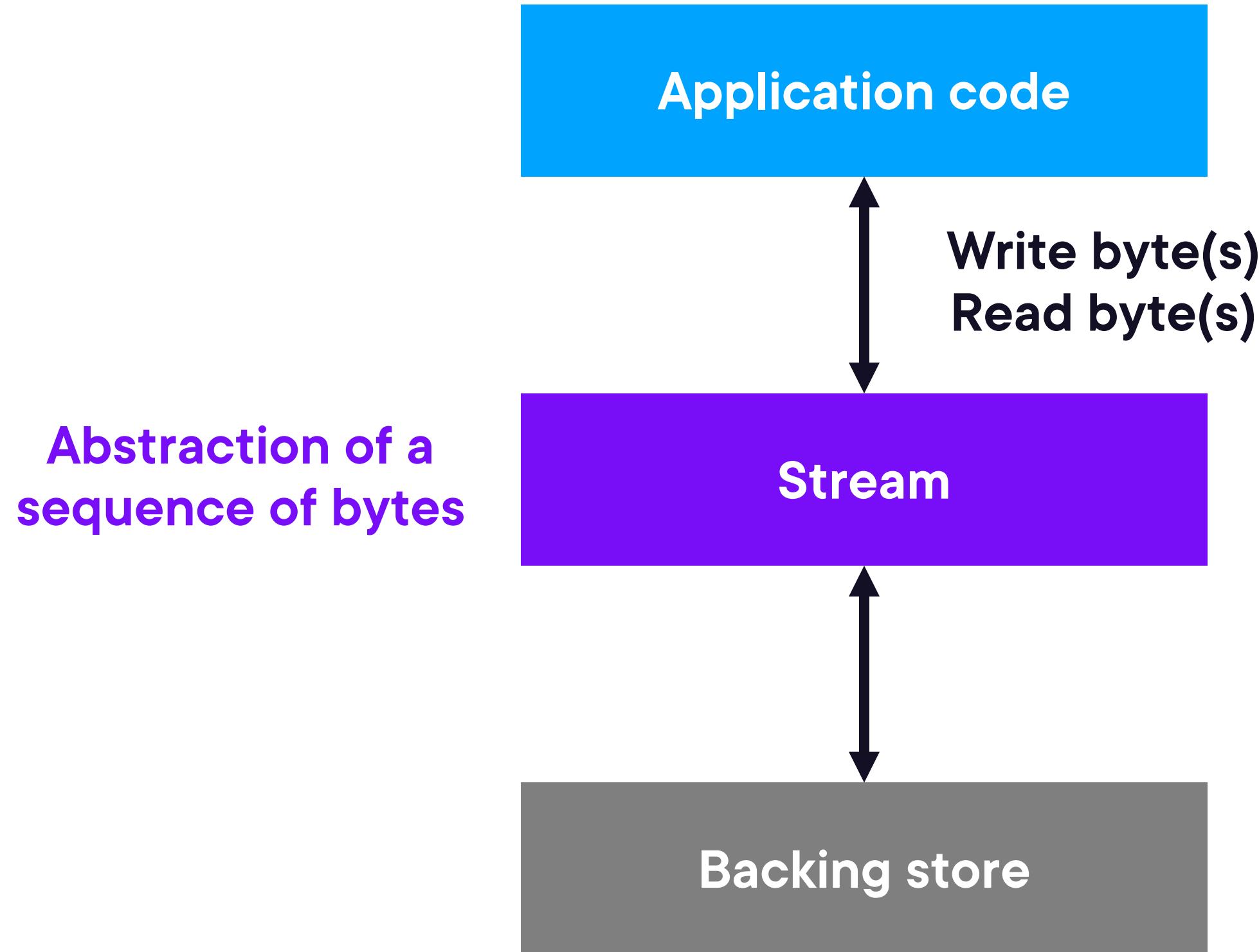
- An introduction to streams
- The benefits of streams
- .NET class hierarchy overview
- Using streams to read and write text
- Selectively processing part of stream
- Using streams to read and write binary data
- Using BinaryReader and BinaryWriter
- Specifying text encodings
- Using streams to append data
- Random FileStream access
- MemoryStream overview



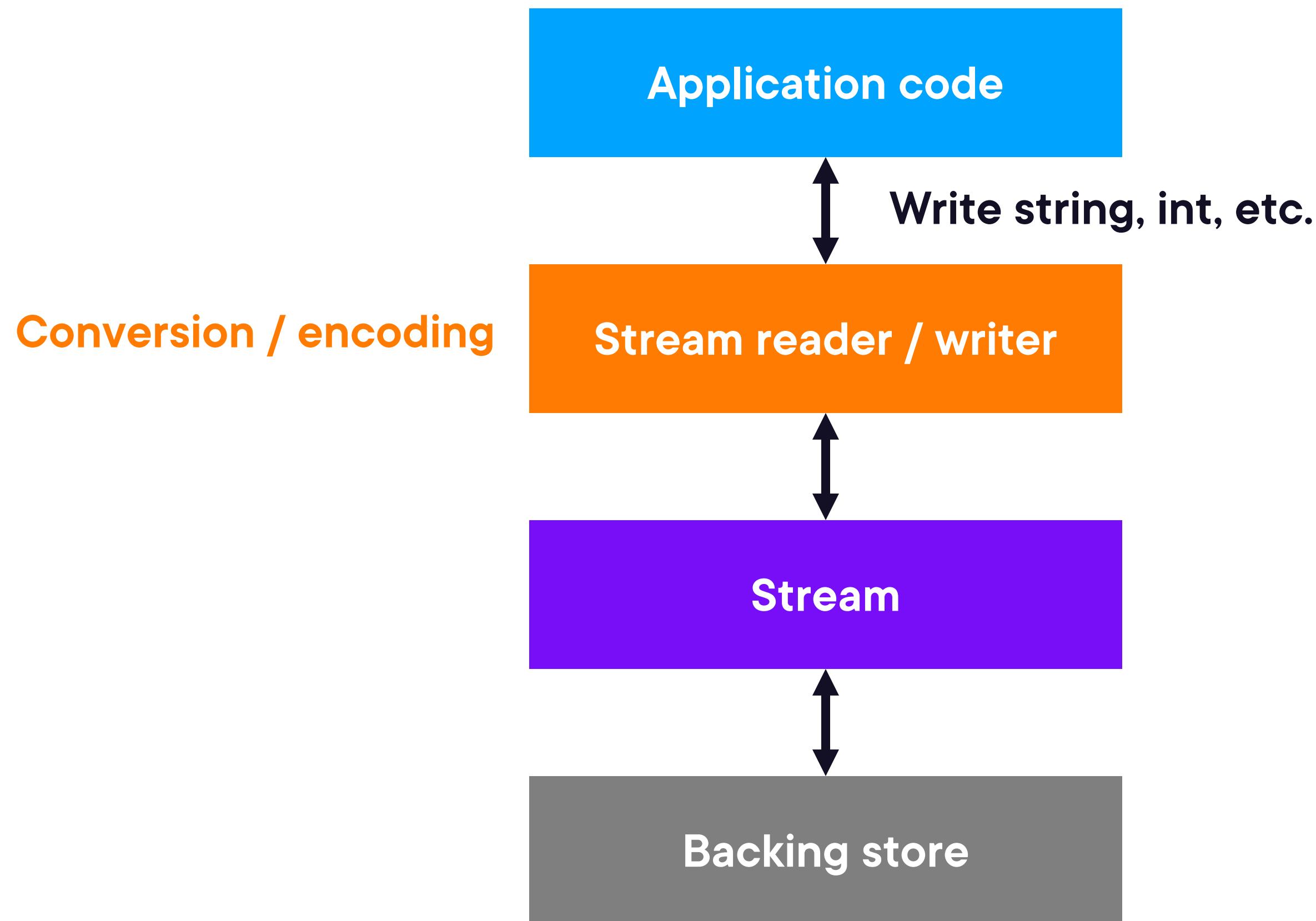
“...a stream is a sequence of bytes that you can use to read from and write to a backing store, which can be one of several storage mediums”



An Introduction to Streams



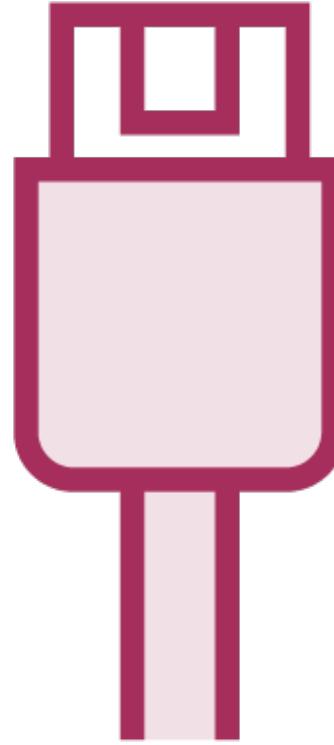
An Introduction to Streams



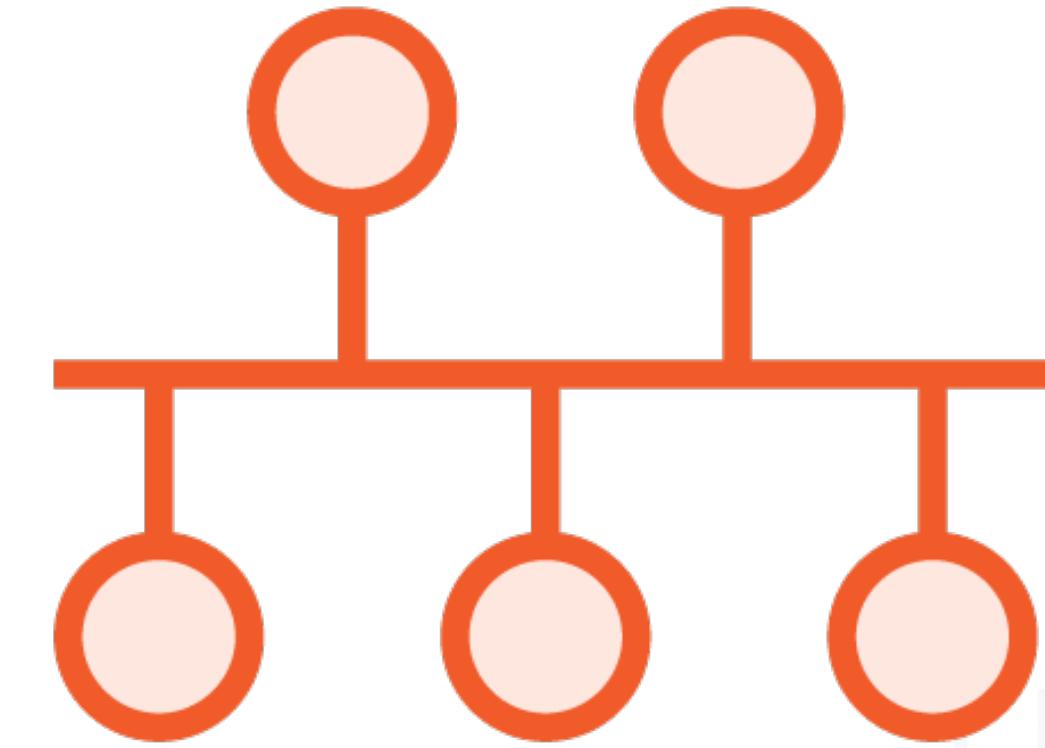
Examples of Backing Stores



Files



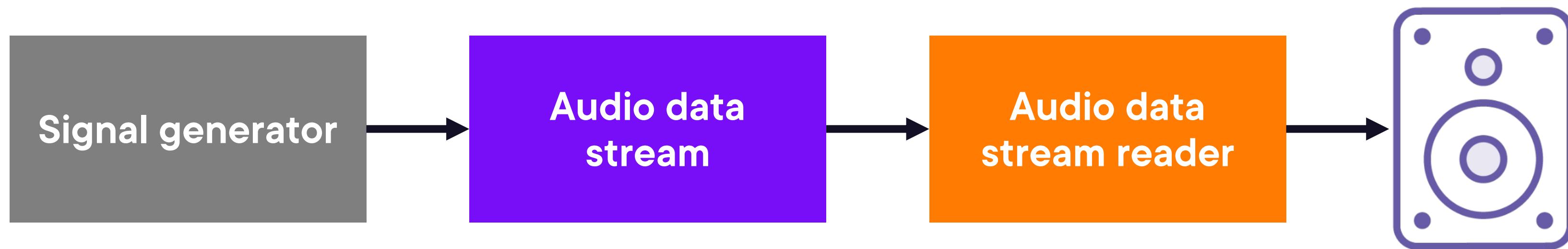
Input/output
device/hardware



TCP/IP sockets



Non-Backing Store Streams



<https://github.com/naudio/NAudio>



The Benefits of Streams

Incremental data processing

Abstraction of backing store

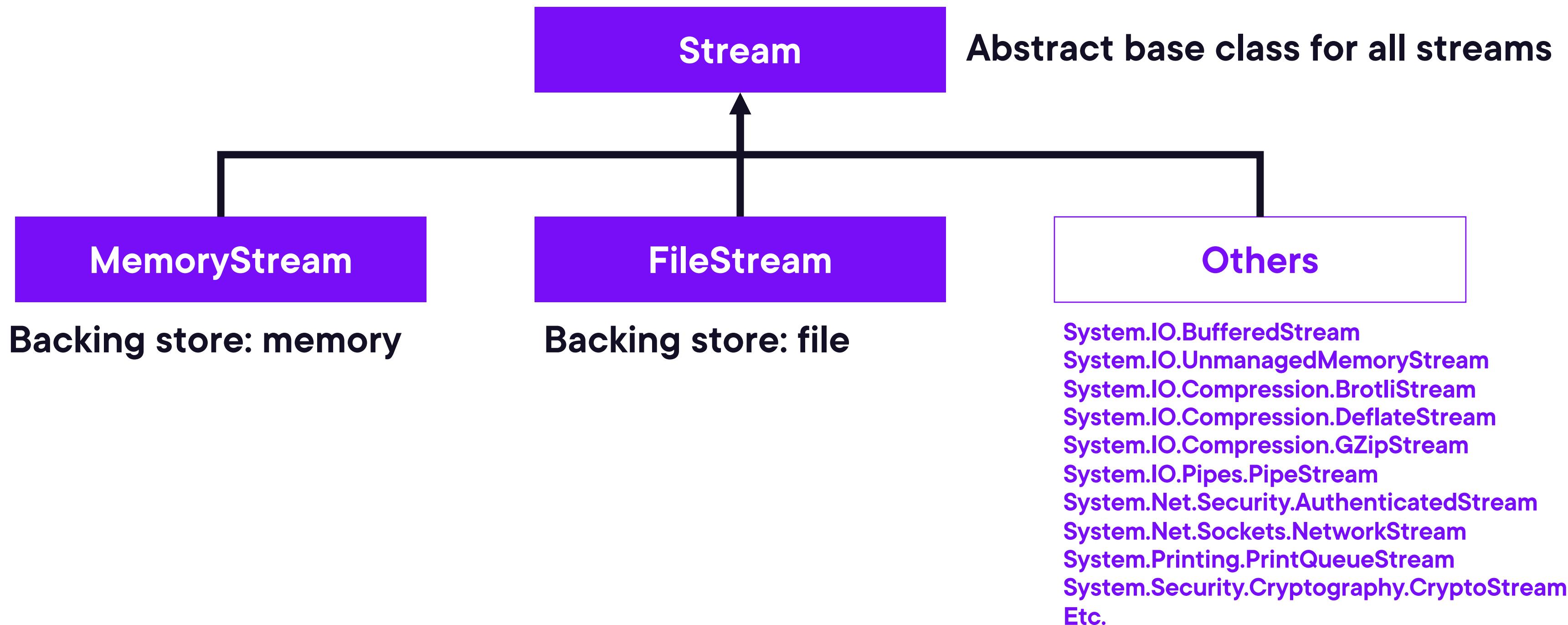
Flexibility / control

Random access / seeking

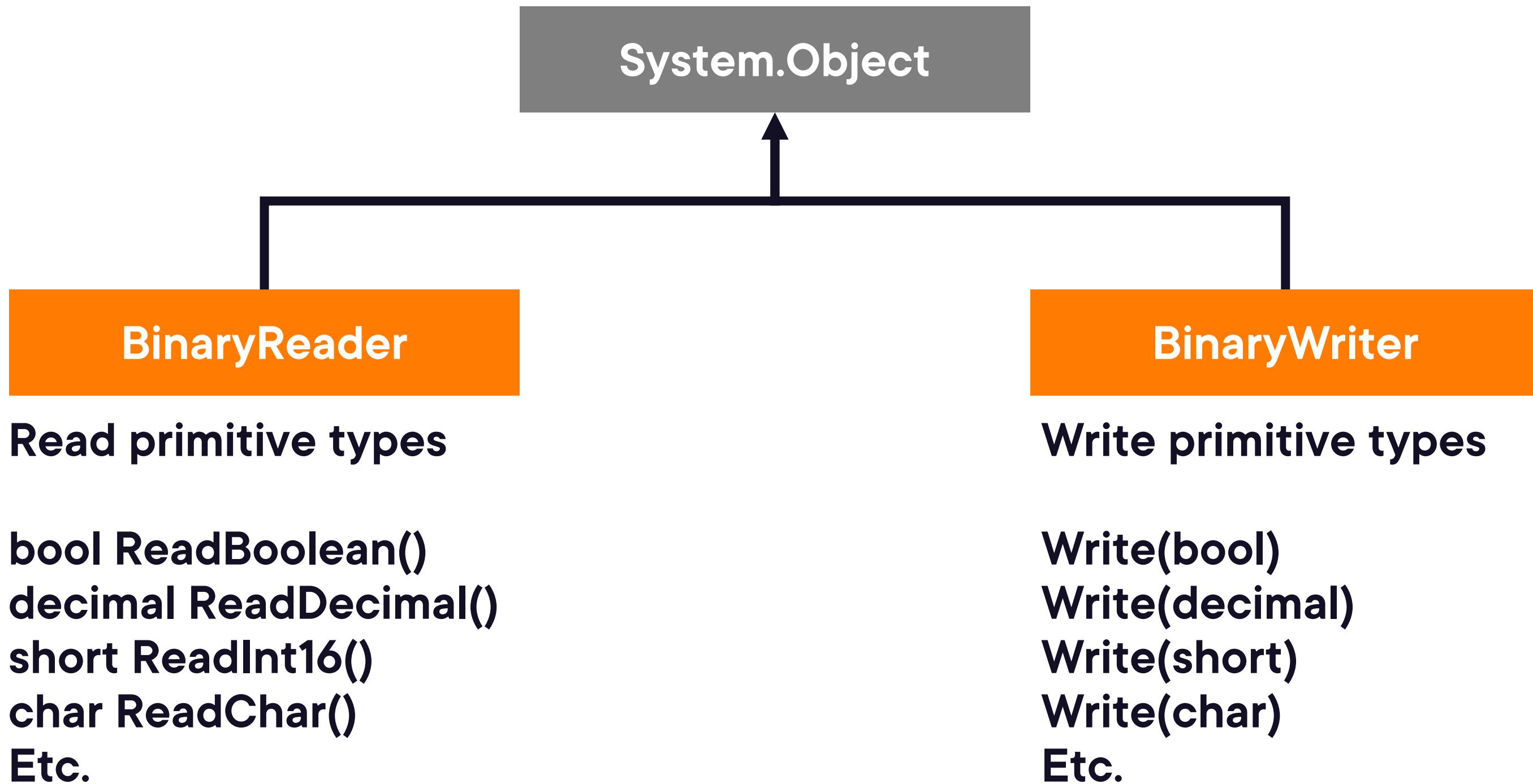
Composability / pipelines



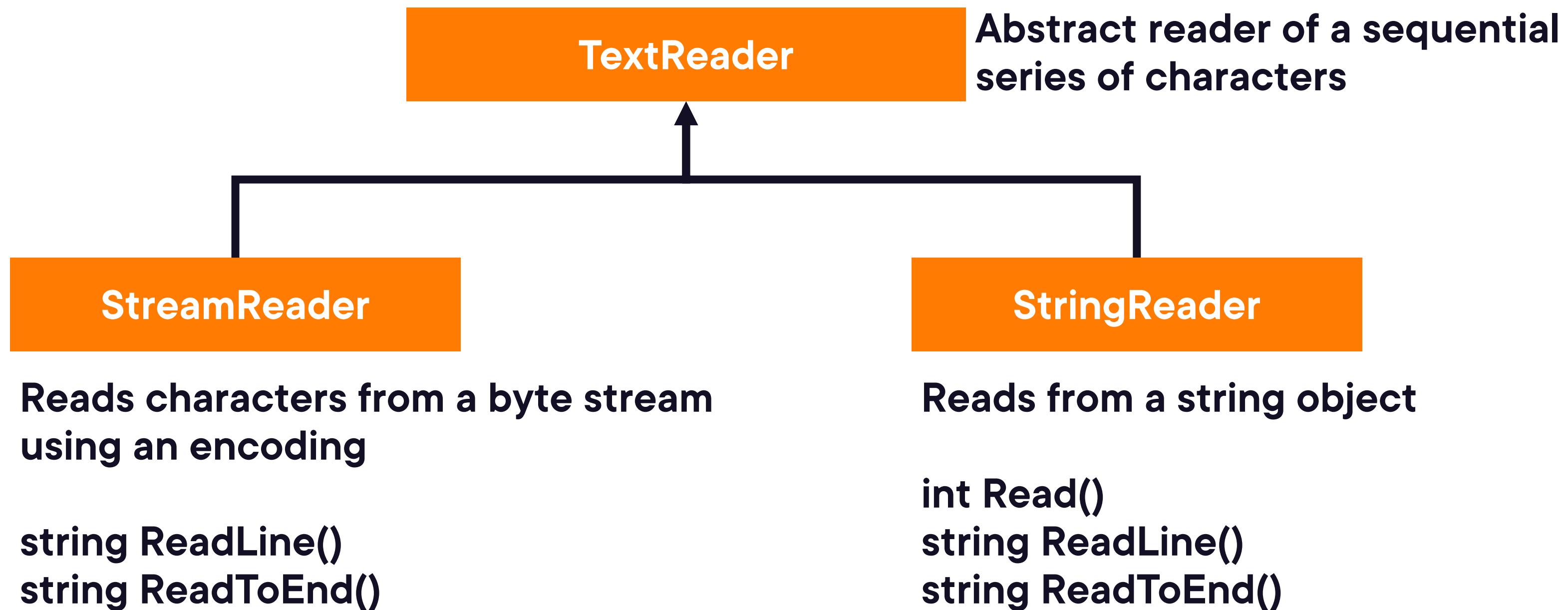
Stream Classes



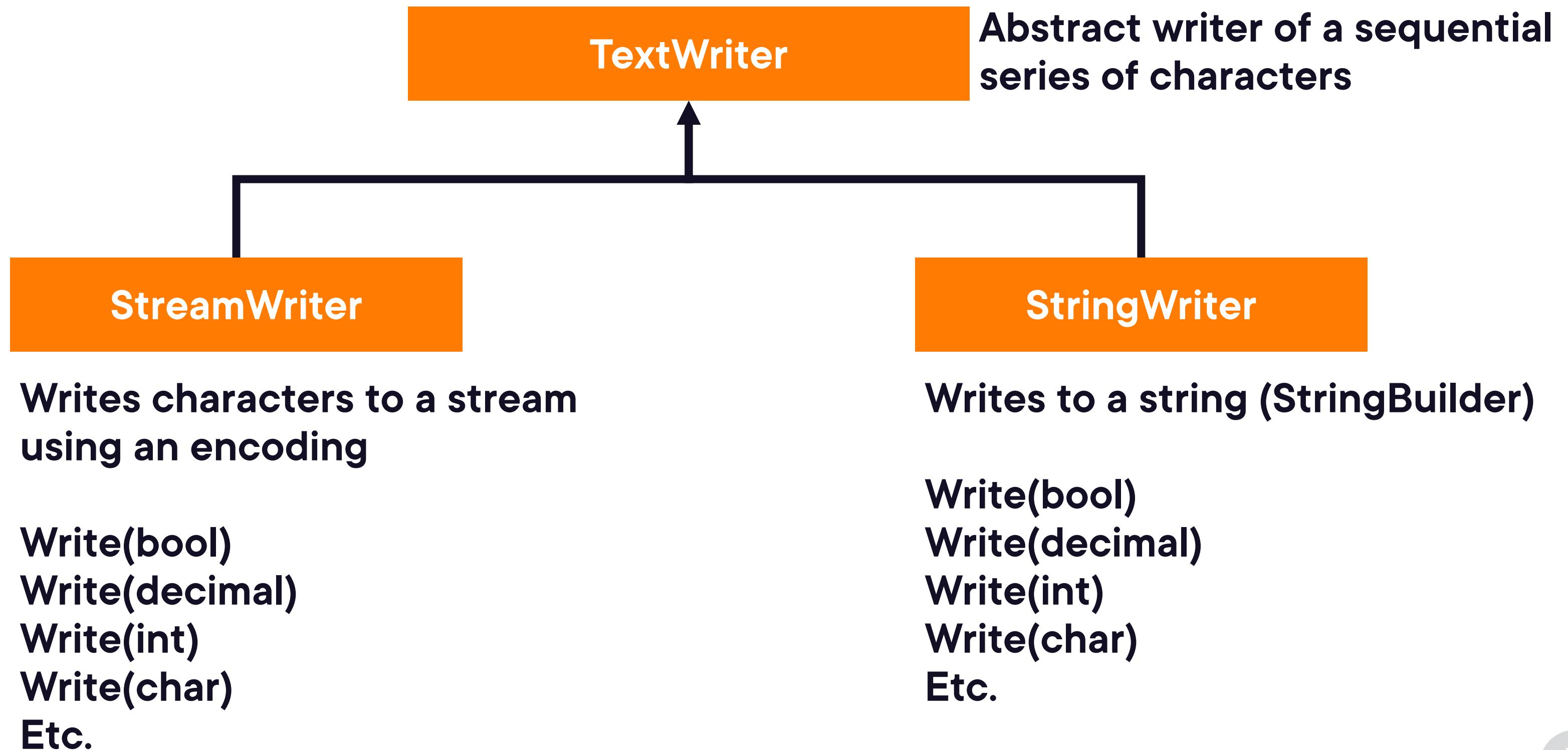
Binary Reader and Writer



TextReaders



TextWriters



FileStreamOptions Properties

Mode Property

Specifies how to open/create the file
 FileMode enum
Default: FileMode.Open



```
// Create a new file - exception if the file already exists
```

```
 FileMode.CreateNew
```

```
// Create a new file - overwrites existing file if exists
```

```
 FileMode.Create
```

```
// Open existing file - exception if file doesn't exist
```

```
 FileMode.Open
```

```
// Open file if already exists, otherwise create new file
```

```
 FileMode.OpenOrCreate
```

```
// Open file then truncate its size is zero bytes - exception if file
```

```
// doesn't exist
```

```
 FileMode.Truncate
```

```
// Open existing file then seek to the end, otherwise create new file
```

```
 FileMode.Append
```



FileStreamOptions Properties

Share Property

Specifies how the file will be shared by multiple processes when they try to open the specified file concurrently

FileShare enum (combinable)

Default: FileShare.Read

- “Allows subsequent opening of the file for reading. If this flag is not specified, any request to open the file for reading (by this process or another process) will fail until the file is closed.”

<https://docs.microsoft.com/en-us/dotnet/api/system.io.fileshare>



FileStreamOptions Properties

Options Property

Advanced/additional file options

FileOptions enum (combinable)

Default: FileOptions.None

FileOptions.RandomAccess

- "...file is accessed randomly. The system can use this as a hint to optimize file caching."

FileOptions.Aynchronous

- "... a file can be used for asynchronous reading and writing."

<https://docs.microsoft.com/en-us/dotnet/api/system.io.fileoptions>



FileStreamOptions Properties

PreallocationSize Property

Requests that the OS pre-allocates a specified amount of space on disk before creating a file

long (size in bytes)

Default: 0

Just a hint to the OS and “is not a strong guarantee.”

E.g. When creating large files (when size is known in advance) set PreallocationSize to be the file length

- May reduce disk fragmentation
- May improve performance



FileStreamOptions Properties

BufferSize Property

Specifies buffer size for reading/writing data
Allows .NET to optimize performance
Reduce the number of expensive OS read/write calls
`int`
Default: 4096 (bytes)
Set to 1 to disable buffering in .NET



Specifying Text Encodings

```
new StreamReader(inputStream, Encoding.UTF32)
new StreamReader(inputStream, new UTF8Encoding(true))
new StreamWriter(filePath, Encoding.UTF32)
new StreamWriter(filePath, new UTF8Encoding(true))
```

```
// No explicit encoding overload
File.OpenText(InputFilePath) // Expects file to be UTF-8
File.CreateText(OutputFilePath) // UTF-8 output encoding
```



Specifying Text Encodings

```
new BinaryReader(inputStream, Encoding.UTF32)  
new BinaryWriter(outputFileStream, Encoding.UTF32)
```

```
// UTF-8 encoding  
new BinaryReader(inputStream)  
new BinaryWriter(outputFileStream)
```



Using Streams to Append Data

```
var streamWriter = new StreamWriter(@"C:\data.txt", true);  
  
streamWriter.Write("Content to append...");  
streamWriter.WriteLine("Content to append with new line...");
```



Using Streams to Append Data

```
FileStream fs = File.Open(@"C:\data.data", FileMode.Append);  
var binaryWriter = new BinaryWriter(fs);  
  
binaryWriter.Write(42); // append to end of file
```



Random FileStream Access

05 0F FF 5E 2A 00
↑

```
fileStream.Position = 0; // zero-based  
int firstByte = fileStream.ReadByte(); // 05
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Position = 0; // zero-based  
int firstByte = fileStream.ReadByte(); // 05
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Position = 2;
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Position = 2;  
int thirdByte = fileStream.ReadByte(); // FF
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Position = 2;  
int thirdByte = fileStream.ReadByte(); // FF
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Seek(2, SeekOrigin.Begin);
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Seek(2, SeekOrigin.Begin);  
thirdByte = fileStream.ReadByte(); // FF
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Seek(2, SeekOrigin.Begin);  
thirdByte = fileStream.ReadByte(); // FF
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Seek(1, SeekOrigin.Current);
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Seek(1, SeekOrigin.Current);  
int fifthByte = fileStream.ReadByte(); // 2A
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Seek(1, SeekOrigin.Current);  
int fifthByte = fileStream.ReadByte(); // 2A
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Seek(-3, SeekOrigin.End);
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Seek(-3, SeekOrigin.End);  
int threeFromEnd = fileStream.ReadByte(); // 5E
```



Random FileStream Access

05 0F FF 5E 2A 00



```
fileStream.Seek(-3, SeekOrigin.End);  
int threeFromEnd = fileStream.ReadByte(); // 5E
```



**Not all streams support
random access / seeking.**

Stream.CanSeek



MemoryStream Overview

```
using var memoryStream = new MemoryStream();  
using var memoryStreamWriter = new StreamWriter(memoryStream);  
using var fileStream = new FileStream(@"C:\data.txt",  
                                      FileMode.Create);  
  
memoryStreamWriter.WriteLine("Line 1");  
memoryStreamWriter.WriteLine("Line 2");  
  
// Ensure everything's written to memory stream  
memoryStreamWriter.Flush();  
  
memoryStream.WriteTo(fileStream);
```



Asynchronous Streams

```
string currentLine = inputStreamReader.ReadLine();
string currentLine = await inputStreamReader.ReadLineAsync();

outputStreamWriter.Write(currentLine);
await outputStreamWriter.WriteAsync(currentLine);

outputStreamWriter.WriteLine(currentLine);
await outputStreamWriter.WriteLineAsync(currentLine);
```



Asynchronous Streams

```
int nextByte = inputStream.ReadByte();
int nextByte = inputStream.ReadByteAsync(); // ERROR

public Task<int> ReadAsync(byte[] buffer,
                           int offset,
                           int count,
                           CancellationToken ...);

public ValueTask<int> ReadAsync(Memory<byte> buffer, ...);

public Task WriteAsync(byte[] buffer, ...);
public ValueTask WriteAsync(ReadOnlyMemory<byte> buffer, ...);
```



Asynchronous Streams from C# 8.0

```
// Don't confuse with System.IO streams
// "Asynchronous enumerables"
// Get a "stream" of results

public interface IAsyncEnumerable<out T>
{
    IAsyncEnumerator<T> GetAsyncEnumerator(
        CancellationToken cancellationToken = default);
}
```



```
public static async IAsyncEnumerable<int> TakeSensorReadings()
{
    var rnd = new Random();
    for (int i = 0; i < 10; i++)
    {
        await Task.Delay(1_000);

        int temp = rnd.Next(minValue: -10, maxValue: 50);

        yield return temp;
    }
}
```



```
public static async IAsyncEnumerable<int> TakeSensorReadings()
{
    var rnd = new Random();
    for (int i = 0; i < 10; i++)
    {
        await Task.Delay(1_000);

        int temp = rnd.Next(minValue: -10, maxValue: 50);

        yield return temp;
    }
}
```



```
await foreach (var reading in TakeSensorReadings())
{
    Console.WriteLine($"Sensor reading: {reading}");
}
```



Perform processing during enumeration and return multiple values asynchronously in a pull-based fashion.



Thread-Safe File IO



RandomAccess Class

Low-level API

Read/write byte(s)

Offset-based

Stateless (e.g. no pointer to current position)

Thread-safe reading and writing

Methods that use a SafeFileHandle (wrapper class for OS file handles)

- GetLength(...)
- Read(...)
- ReadAsync(...)
- Write(...)
- WriteAsync(...)



```
using SafeFileHandle inputFileHandle =
    File.OpenHandle(InputFilePath, FileMode.Open);
using SafeFileHandle outputFileHandle =
    File.OpenHandle(OutputFilePath,
                    FileMode.CreateNew,
                    FileAccess.Write);
var inputFileLength = RandomAccess.GetLength(inputFileHandle);
var singleByteBuffer = new Span<byte>(new byte[1]);
byte largestByte = 0;
```



```
for (int fileOffset = 0; fileOffset < inputFileLength;
    fileOffset++)
{
    RandomAccess.Read(inputFileHandle,
                      singleByteBuffer,
                      fileOffset);
    RandomAccess.Write(outputFileHandle,
                      singleByteBuffer,
                      fileOffset);
    if (singleByteBuffer[0] > largestByte)
    {
        largestByte = singleByteBuffer[0];
    }
}
```



```
var largestByteBuffer =  
    new Span<byte>(new byte[1] { largestByte });  
  
RandomAccess.Write(outputFileHandle,  
    largestByteBuffer,  
    inputFileLength);
```

<https://docs.microsoft.com/en-us/dotnet/api/system.io.randomaccess>



Module Summary

An introduction to streams

The benefits of streams

.NET class hierarchy overview

`new StreamReader(inputStream)`

`File.OpenText(inputFilePath)`

Selectively processing part of stream

Using streams to read and write binary data

`outputFileStream.WriteByte(...)`

`new BinaryReader(inputStream)`

Text encodings & appending data

Random access & `MemoryStreams`



Up Next:

Reading and Writing CSV Data

