.NET Streaming Architecture

Mohamad Halabi
Microsoft Integration MVP
@mohamadhalabi

pluralsight
hardcore dev and IT training

Input / Output (I/O)

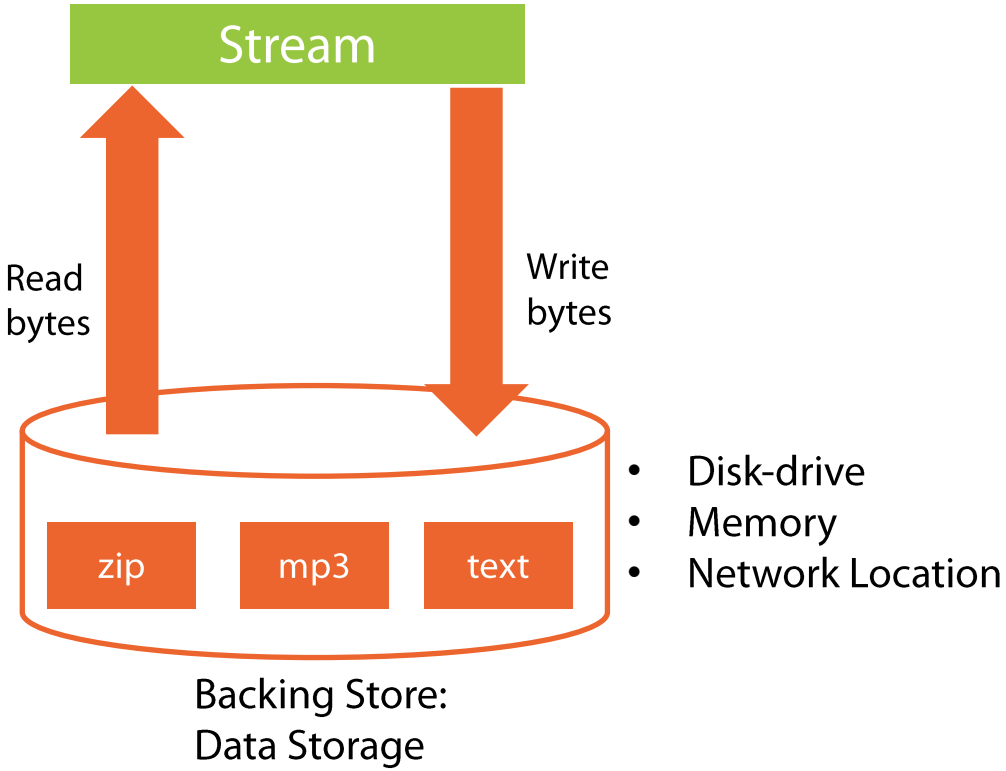
- **Transfer of data from or to applications**
- **Microsoft run-time library defines three I/O types:**
  - Stream I/O: Data is represented as a stream of bytes
    - Stream represent sources and/or destinations (ex: files, memory arrays)
  - Low-level I/O: Invoke operating system directly
  - Console and port I/O: read/write to console (ex: keyboard) or I/O (ex: printer) port

Stream I/O Programming Model

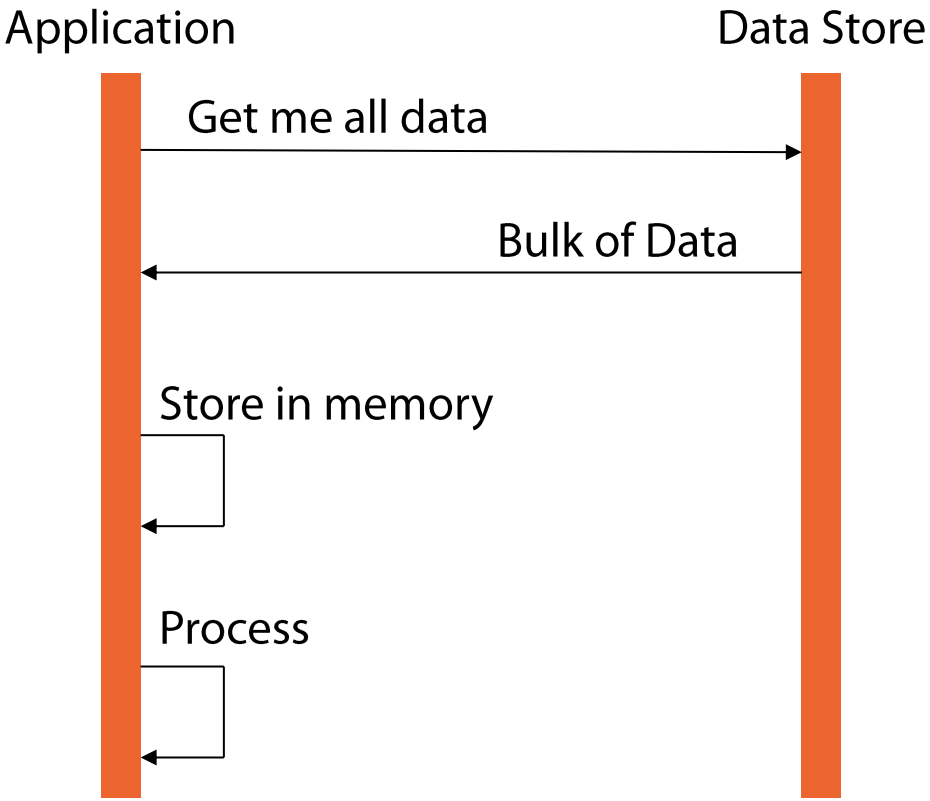
- **I/O streams have a common programming model**
  - A common reading and writing method across different I/O types
- **I/O can be against file system, network socket, memory array**
  - .NET provides a consistent programming interface
- **I/O operations can be synchronous and asynchronous**

What is a Stream?

- *"a stream is a sequence of bytes that you can use to read from or write to a backing store"*…MSDN

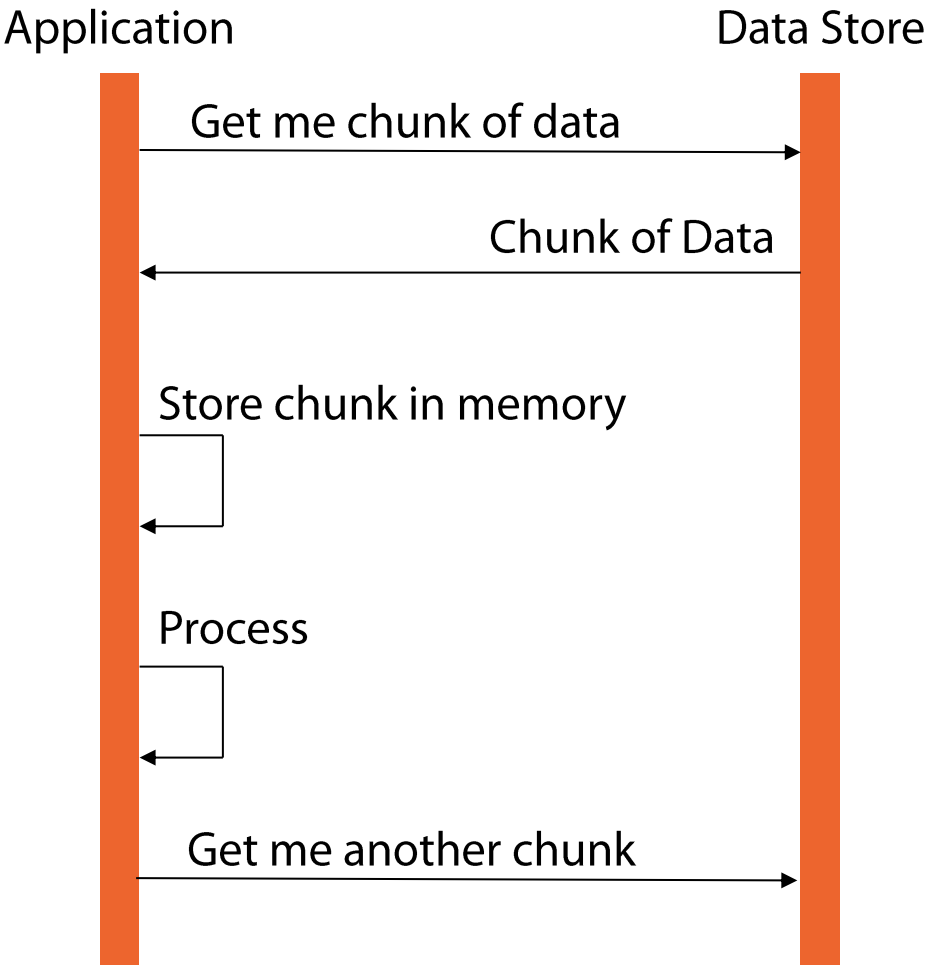


Non-Stream Data Consumption



- **Negative impact on memory**
- **Limits scalability**

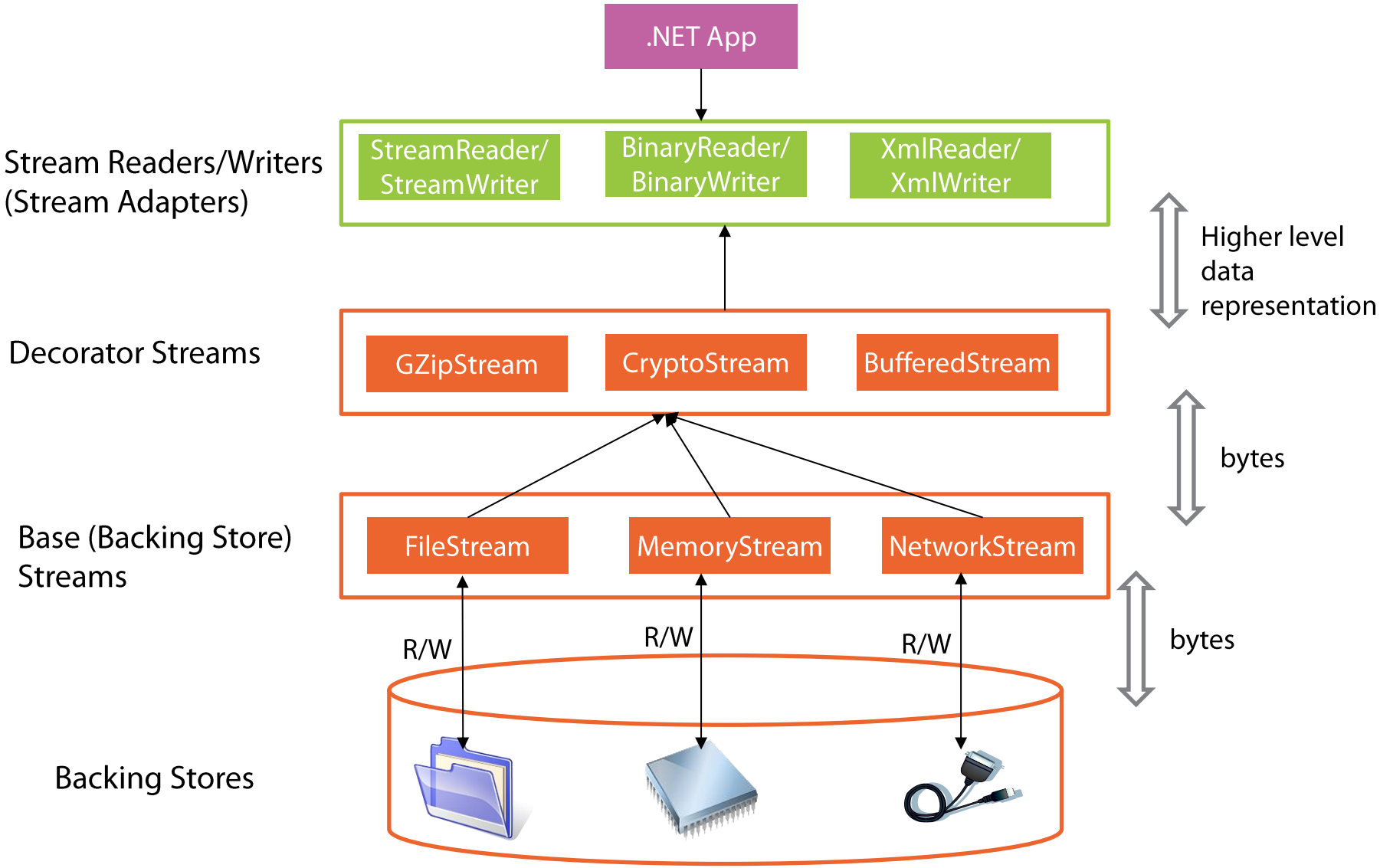
Stream Data Consumption



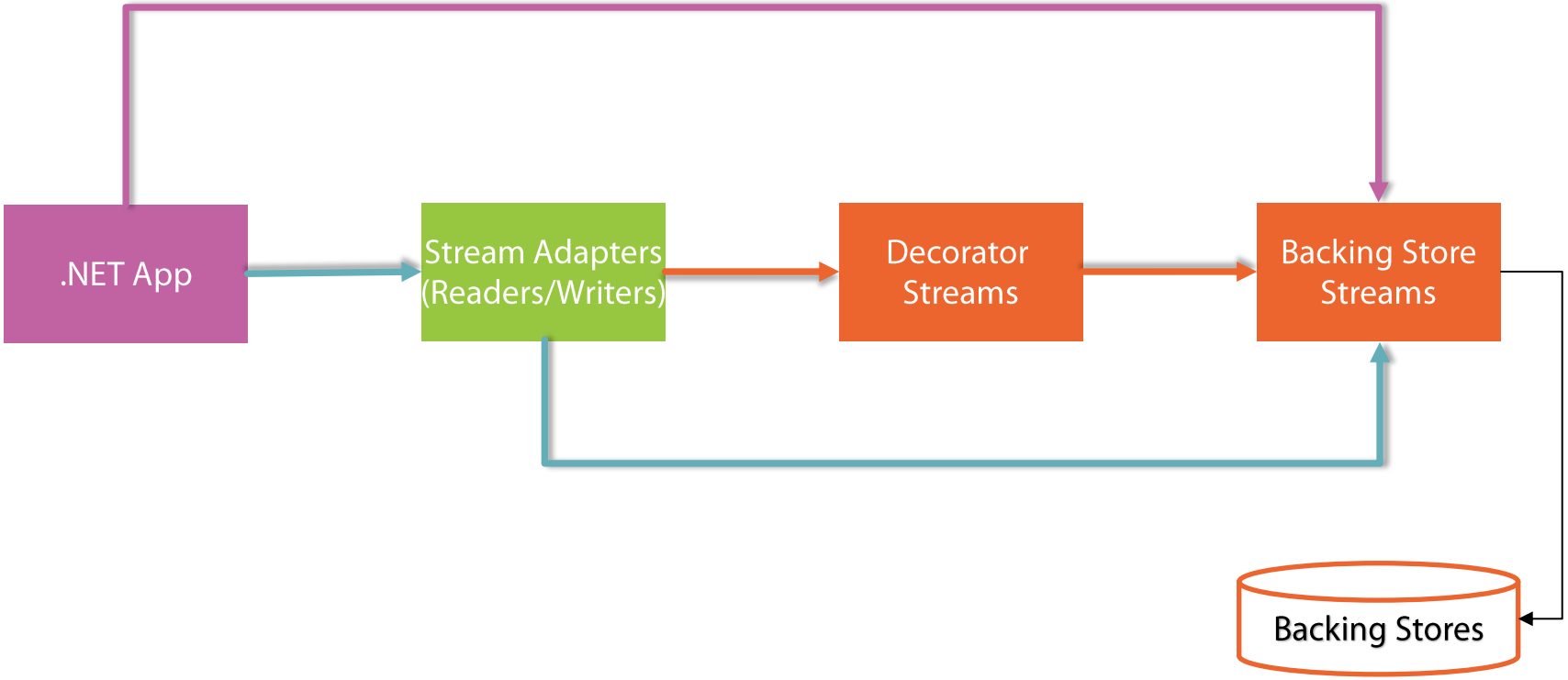
What About a Memory-Based Stream?

- **In memory-based stream, data is already in memory**
  - Ex: MemoryStream
- **How do streaming save memory in this case?**
  - It doesn't!
- **However, memory-based streams still have various usage scenarios**
  - Ex: When data size is small

The Overall Architecture

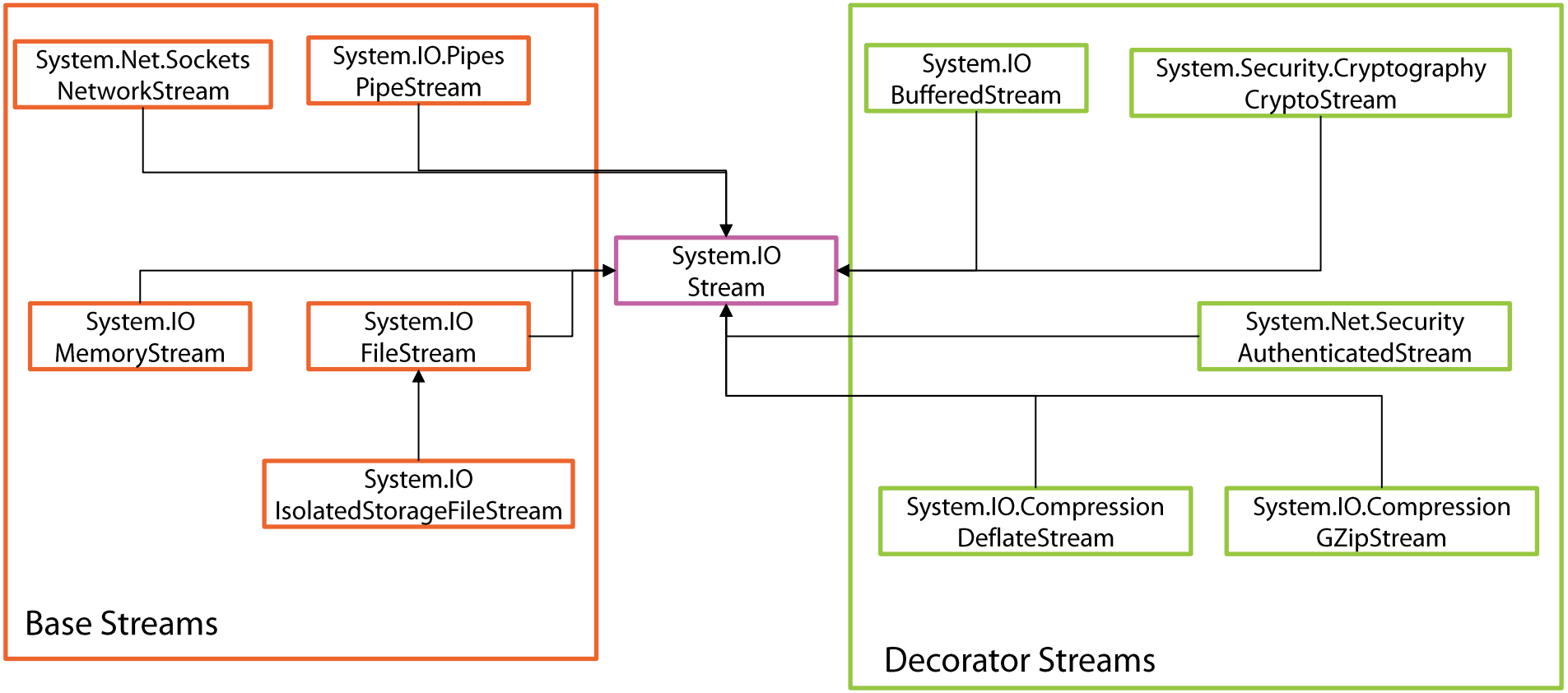


Layers Interaction



System.IO.Stream Class

- **Stream class in System.IO namespace is the base class for all streams**
  - Backing store streams & decorator streams

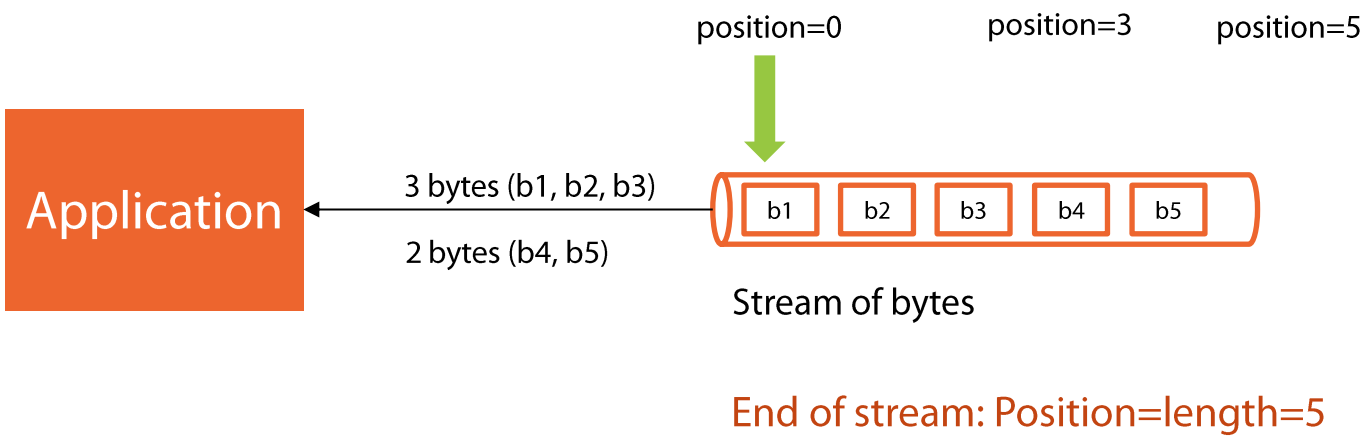


The Programming Model

- All stream implementations share a common set of methods
- However, different streams will have different characteristics and features support
- Stream class members can be grouped as follows:

Stream Pointer

- **Streams have a pointer which indicates the current position within the stream**



Reading

- **System.IO.Stream class has three members related to reading:**

Member	Description
Read	Reads a sequence of bytes into an array passed as a parameter Advances pointer position by number of bytes read Returns number of bytes read, or 0 if end of stream
ReadByte	Reads and returns a single byte Advances pointer position by 1 Returns -1 if end of stream
CanRead	Checks if a stream supports reading Used as an assurance mechanism

The Read Method Parameters

- ***Read(byte[] buffer, int offset, int count)***
- ***buffer:* the array to store the read data**
- ***offset:***
  - This is not the offset at which to start reading from the stream
  - This is the buffer offset at which to start placing read data
- ***count:* maximum number of bytes to read**
  - Actual number of bytes read might be less
    - End of stream is reached
    - Stream decided to return fewer bytes. Often happens with network streams

How Reading Works

- You can be sure that end of stream is reached, only when Read returns 0
- So the following code is not reliable:

```
byte[] dataToRead = new byte[stream.Length];
////there is no guarantee that the stream will actually read all data at once
////as specified in the count parameter (dataToRead.Length)
int bytesRead = stream.Read(dataToRead, 0, dataToRead.Length);
```

How Reading Works

- **The proper way:**

```
static byte[] ReadBytes(Stream stream)
{
    // dataToRead will hold the data read from the stream
    byte[] dataToRead = new byte[stream.Length];

    //this is the total number of bytes read. this will be incremented
    //and eventually will equal the bytes size held by the stream
    int totalBytesRead = 0;

    //this is the number of bytes read in each iteration (i.e. chunk size)
    int chunkBytesRead = 1;

    while (totalBytesRead < dataToRead.Length && chunkBytesRead > 0)
    {
        chunkBytesRead = stream.Read(dataToRead, totalBytesRead, dataToRead.Length - totalBytesRead);
        totalBytesRead = totalBytesRead + chunkBytesRead;
    }

    return dataToRead;
}
```

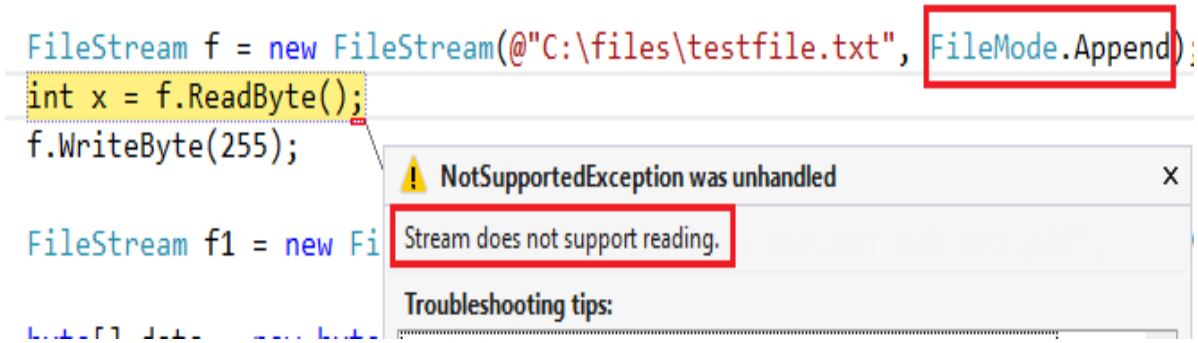
Defensive programming

ReadByte

- **Reads and returns a single byte from the current stream position**
- **A return value of -1 signals end of stream**
- **ReadByte return type is integer not byte**
  - Reason is to accommodate the return value of -1

CanRead

- **Streams might chose not to support reading**
  - A design decision for a custom stream (MyUnReadableStreamImplementation:System.IO.Stream)
  - Restriction caused by a backing store state



- **Therefore it's advisable to check CanRead property before performing reads**

Writing

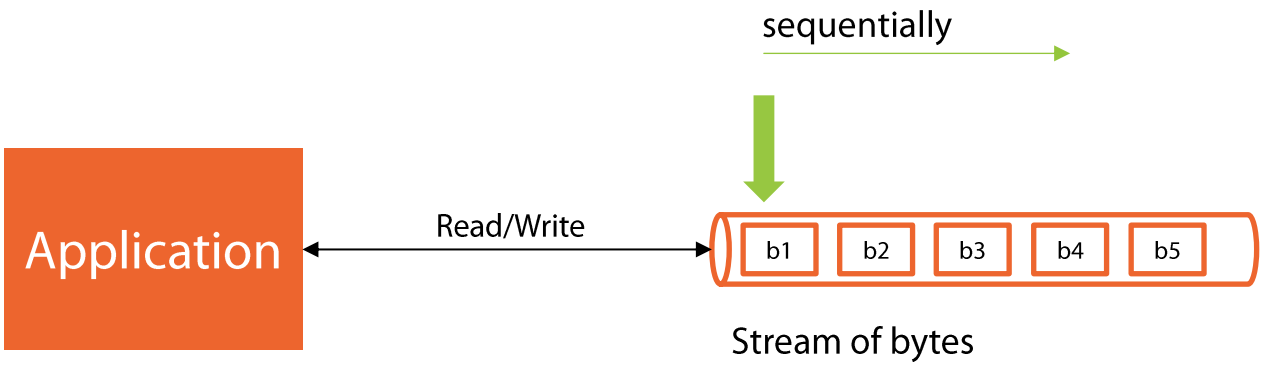
Member	Description
Write	Writes a sequence of bytes into the stream Advances pointer position by number of bytes written
WriteByte	Writes a single byte Advances pointer position by 1
CanWrite	Checks if a stream supports writing Used as an assurance mechanism

The Write Method

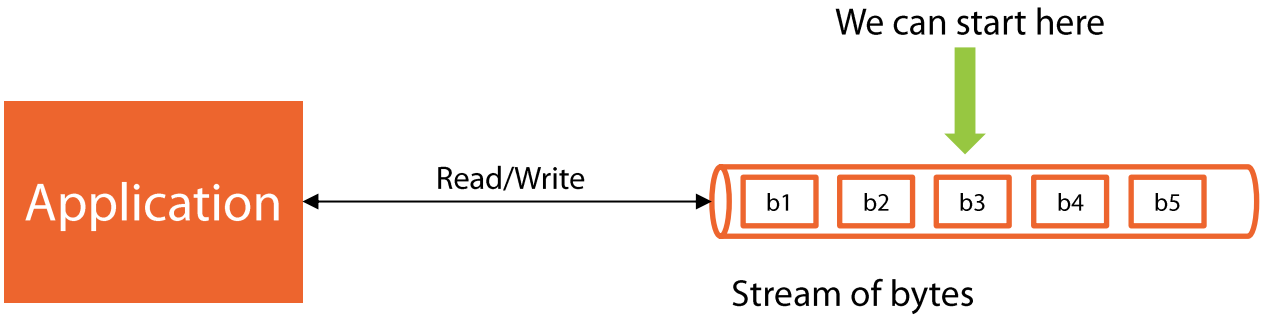
- ***Write(byte[] array, int offset, int count)***
  - *array*: the array of bytes to write to the stream
  - *offset*: index of the array to start copying bytes to the stream. Not the position within the stream
  - *count*: the maximum number of bytes to write

What is Seeking?

- **Read and Write operations move a stream pointer**



- **Seeking allows selecting a certain position within the stream**



Not All Backing Stores Support Seeking

- **Seeking support depends on the backing store type**
  - MemoryStream and FileStream support seeking
  - NetworkStream (sockets) and PipeStream (pipes) so not support seeking
  - Decorator streams do not support seeking
    - With the exception of BufferedStream
- **BufferedStream can wrap a non-seekable stream and provide seeking over the buffered portion**

Seeking Members

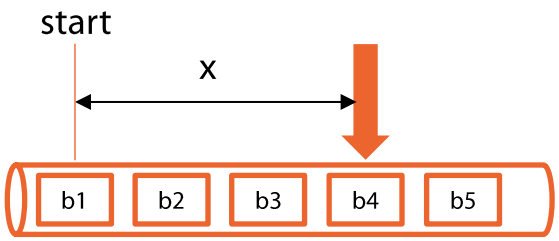
Member	Description
Seek(long Offset, SeekOrigin origin)	Set the pointer position within the stream
SetLength (long value)	Sets the length of the stream If value < stream.length then stream is truncated If value > stream.length then stream is expanded Stream must support writing and seeking
Length	Read-only. Gets the length of stream Supported only when seeking is enabled
Position	Get or set the pointer position within the stream Seeking must be supported Seek method is used implicitly Position vs. Seek?
CanSeek	Indicates if stream supports seeking (based on backing store)

Position vs. Seek

Position

Always relative to the start of the stream

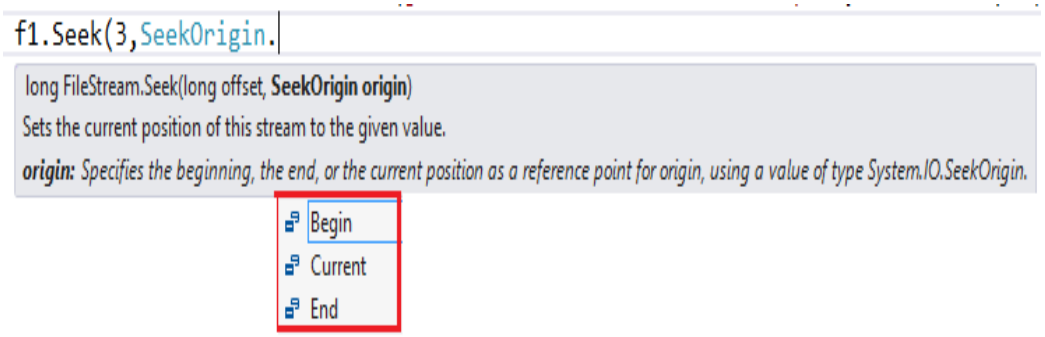
stream.Position = x



Stream of bytes

Seek

You can decide to move relative to current position, start, or end




```
f1.Seek(300, SeekOrigin.Begin);
long position = f1.Position; //300
f1.Seek(200, SeekOrigin.Current);
position = f1.Position; //500
```

Get Length of Un-Seekable Stream

- **Recall:**
  - Length property requires seeking support
  - Some streams do not support seeking; ex: NetworkStream and PipeStream
- **What if I want to get the length of a non-seekable stream?**
  - Read the entire stream
  - Store the stream in a buffer (ex: memory)
  - Query the length of this buffer

Demo: Seeking

Disposing Streams

- **Remember to dispose a stream to release its resources**
  - Ex: FileStream uses a file handle, NetworkStream uses a socket handle
- **Follow same .NET standards in disposing**
  - Explicitly call Dispose on a stream
  - Use “using” statement

```
using (FileStream s = new FileStream("myFile.txt", FileMode.Create))
{
   //perform operations
}
```

What is Buffering?

- **I/O are expensive operations**
- **To improve performance, some streams implement internal memory buffers**
- **Write operations store data into the buffer instead of the backing store**
  - When buffer is full, or explicitly commanded by code, data is written into the backing store
  - This saves the cost of multiple I/O write operations
- **Read operations can store more data in the buffer than what is asked for**
  - Subsequent read operations are done on the buffered data

Example of Buffering

- FileStream implements an internal buffer with default size of 4k

```
[ComVisible(true)]
public class FileStream : Stream
{
  private static readonly bool  canUseAsync = Environment.RunningOnWinNT;
  internal const int DefaultBufferSize = 4096;
  internal const int GENERIC_READ = -2147483648;
  internal const int ERROR_BROKEN_PIPE = 109;
  internal const int ERROR_NO_DATA = 232;
```

Flushing a Buffer

- **When you use a stream with an internal buffer:**
  - You must call the Flush method to explicitly write buffer data to the backing store
  - …Or, Flush is automatically called when a stream is disposed
- **What happens if you call Flush on a stream that does not implement an internal buffer?**
  - Flush would do nothing

```
public override void Flush()
{
}

public override Task FlushAsync(CancellationToken cancellationToken)
{
  return Task.CompletedTask;
}
```

- **For streams that do not implement buffering, you could use the BufferedStream decorator stream**

Multithreading

- Multithreading can be used to read/write into a stream in parallel
- However, as a rule, streams are not thread-safe
- Therefore, explicitly code for thread-safety

Synchronized

- Stream class has a static Synchronized method
- Returns a thread-safe wrapper around the stream

```
[HostProtection(SecurityAction.LinkDemand, Synchronization = true)]
public static Stream Synchronized(Stream stream)
{
  if (stream == null)
    throw new ArgumentNullException("stream");
  if (stream is Stream.SyncStream)
    return stream;
  else
    return (Stream) new Stream.SyncStream(stream);
}
```

Synchronized

```
[Serializable]
internal sealed class SyncStream : Stream, IDisposable
{
    public override long Length
    {
      get
      {
        lock (this._stream)
          return this._stream.Length;
      }
    }
    public override long Position
    {
      get
      {
        lock (this._stream)
          return this._stream.Position;
      }
      set
      {
        lock (this._stream)
          this._stream.Position = value;
      }
    }
    public override void Flush()
    {
      lock (this._stream)
        this._stream.Flush();
    }
    public override int Read([In, Out] byte[] bytes, int offset, int count)
    {
      lock (this._stream)
        return this._stream.Read(bytes, offset, count);
    }
    public override long Seek(long offset, SeekOrigin origin)
    {
      lock (this._stream)
        return this._stream.Seek(offset, origin);
    }
    public override void Write(byte[] bytes, int offset, int count)
    {
      lock (this._stream)
        this._stream.Write(bytes, offset, count);
    }
```

Async Support

- Asynchronous support goes back to pre .NET 4.5
- In .NET 4.5 you can use the “async” keyword

Non async method	async equivalent
Flush	FlushAsync
Read	ReadAsync
Write	WriteAsync

Stream.Null

- Stream.Null returns a stream with no backing store
- Might be useful for testing code to write huge data without actually consuming resources
- Write or WriteByte on a Null stream do nothing

```
Stream s = Stream.Null; //return a null stream
s.WriteByte(66); //data is ignored
//My core functionality test goes here
```

- Reading from a Null stream returns 0

Summary

- **I/O**
- **Streams**
- **.NET Streaming architecture**
  - Backing stores
  - Backing store (Base) streams
  - Decorator streams
  - Stream adapters
- **Common operations**
  - Read/Write
  - Seek
  - Flush
- **Multithreading**
- **Null Streams**