1. What does the compiler do when it encounters the await keyword?

My answer: delegetas the “async” code to the thread pool, thread pool then yields one of the threads (might be a new one, might not be), stages the current state of the state machine (held in the Task object), returns the Task object to the thread calling context whose state is at that moment “captured” so the execution may return to it

In essence the execution will “continue” (continuation context) right after the calling context was “captured and suspended” by means of a callback (through the Task object)

1. What is ConfigureAwait(false)

My answer: this one is relevant only in pre-netcore contexts, and for “UI thread” apps like webforms, where you caller (thread) is the one that the execution will return to after the async task has been completed, the default is ConfigureAwait(true) meaning - return to the exact calling thread, this can cause deadlocks/not responding because the returning context encounters a blocked calling thread and is unable to return, whilst the calling context is stuck waiting for the called context to return (deadlock)

1. How are errors handled in async await?

My answer: the compiler internally creates a try catch block and uses the Task object (which is a form of a promise) to hold the error (if it were to happen) so the calling context can evaluate it, if theres is no Task object, no errors will be caught (which is a classic junior level error)

This is the what they wanted to have a discussion about (the relevant compiler output)

URL: <https://gist.github.com/brunobozic/5df4e9f3a6533479cff87da97ea564cf>

[CompilerGenerated]

private sealed class <GetValueAsync3>d\_\_2 : IAsyncStateMachine

{

public int <>1\_\_state;

public AsyncTaskMethodBuilder<int> <>t\_\_builder;

public C <>4\_\_this;

private int <>s\_\_1;

private TaskAwaiter<int> <>u\_\_1;

private void MoveNext()

{

int num = <>1\_\_state;

int result;

try

{

TaskAwaiter<int> awaiter;

if (num != 0)

{

awaiter = Task.FromResult(42).GetAwaiter();

if (!awaiter.IsCompleted)

{

num = (<>1\_\_state = 0);

<>u\_\_1 = awaiter;

<GetValueAsync3>d\_\_2 stateMachine = this;

<>t\_\_builder.AwaitUnsafeOnCompleted(ref awaiter, ref stateMachine);

return;

}

}

else

{

awaiter = <>u\_\_1;

<>u\_\_1 = default(TaskAwaiter<int>);

num = (<>1\_\_state = -1);

}

<>s\_\_1 = awaiter.GetResult();

result = <>s\_\_1;

}

catch (Exception exception)

{

<>1\_\_state = -2;

<>t\_\_builder.SetException(exception);

return;

}

<>1\_\_state = -2;

<>t\_\_builder.SetResult(result);

}

void IAsyncStateMachine.MoveNext()

{

//ILSpy generated this explicit interface implementation from .override directive in MoveNext

this.MoveNext();

}

[DebuggerHidden]

private void SetStateMachine(IAsyncStateMachine stateMachine)

{

}

void IAsyncStateMachine.SetStateMachine(IAsyncStateMachine stateMachine)

{

//ILSpy generated this explicit interface implementation from .override directive in SetStateMachine

this.SetStateMachine(stateMachine);

}

}

Also important:

But, you might ask, what about what's often called "**fire and forget**"? This refers to the situation where you do not want to await an asynchronous method and you're not particularly concerned about when it finishes. In that case, consider, at the very least, adding a **ContinueWith** with **TaskContinuationOptions.OnlyOnFaulted** where you can **log any exceptions that may arise**.

1. How exactly does it return the captured context (compiler output inspection)?

My answer: did not know this one

All I knew was it had captured the “before async” context somehow, embedded the info somewhere (Task object?) and will use this info to return to the exact line where it needs to

Probable answer because they mentioned the **UnsafeStart** at one point:

**Thread.UnsafeStart (new in .NET 6)**

The new function for starting threads is called “unsafe” because it does not capture the execution context. David Fowler explains,

We added [UnsafeStart](https://github.com/dotnet/runtime/issues/46594) in this PR #46181 because we needed to lazily create thread pool threads and the timer thread **on the default execution context**. UnsafeStart **avoids capturing the current execution context and restoring it when the thread runs**. There are other places where we create threads that could use similar logic

Found some more information on how exactly the compiler does its thing

at OnButtonClick()

at System.Threading.ExecutionContext.RunInternal(ExecutionContext executionContext, ContextCallback callback, Object state)

at System.Runtime.CompilerServices.AsyncTaskMethodBuilder`1.AsyncStateMachineBox`1.MoveNext()

at System.Threading.Tasks.AwaitTaskContinuation.RunOrScheduleAction(IAsyncStateMachineBox box, Boolean allowInlining)

at System.Threading.Tasks.Task.RunContinuations(Object continuationObject)

at System.Threading.Tasks.Task`1.TrySetResult(TResult result)

at System.Runtime.CompilerServices.AsyncTaskMethodBuilder`1.SetExistingTaskResult(TResult result)

at System.Runtime.CompilerServices.AsyncTaskMethodBuilder.SetResult()

at DownloadAndBlur()

at System.Threading.ExecutionContext.RunInternal

Notice that there are a lot of methods being called here that we did not define in our code, including AsyncStateMachineBox.MoveNext() and AsyncTaskMethodBuilder.SetResult. It's apparent that the compiler is generating a bunch of code on our behalf to keep track of the execution state.

The details of the generated code are outside the scope of this guide (and vary depending on the C# compiler and version), but suffice it to say that there is a state machine produced that uses goto statements in combination with the Task Parallel Library methods, along with some exception and context (i.e. thread) tracking.

1. How to create a stack overflow condition using for loop?

My answer: concatenate strings, you will end up having a stack overflow

* 1. Why? My answer: reference type
  2. How to fix this?

My answer: use stringbuilder

1. Why is a string class/object made immutable?

My answer: design choice as strings are most often mutated

1. Can we iterate over a set of data without using window functions?

My answer: yes, via cursors

1. How do cursors work (implemented on db engine)?

My answer: didn't know, said I didn't care much how they work, I just use them and they work

need to look it up

1. What is the core difference between triggers on MSSQL vs Oracle

My answer: did not know this one, said I just use triggers, I don't care how they work internally

Looked it up later: Oracle (as most other vendors) has a different trigger concept as MS SQL Server. MS SQL (such as Sybase) has a set based approach. Rows that are affected by a data modification (insert, update, delete) are stored in the inserted and deleted tables. A regular DML trigger in MS SQL is always executed after the statement. The before image is stored in the deleted table, the after image in the inserted table. Both can be accessed from within the trigger. It is also possible to join the inserted and deleted table and use them to update the table on which the trigger was executed.

In Oracle there are before and after triggers and a trigger can be defined to be executed per row or per statement (there are also compound triggers that can have a section for per row and per statement). Before per statement triggers are executed before the statement (insert, update, delete) is executed and only have access to the table before the modification. After per statement triggers are executed after the statement has completed (which is why they are sometimes called deferred, which is a bit misleading as deferred in SQL usually means that an execution is deferred till commit in a transaction that may consist of more than one statement) and only have access to the table after the modification.

1. You have database (single instance), you want to scale it, the callers (rest api apps) want to get immediate results and you \*have\* to make it work.

The interviewers are well aware of eventual consistency problems and the CAP theorem, still you have to make it work, how would you do it?

**So both immediate consistency and scale out are a must.**

My answer: shard the data, create n clusters where each cluster has one master and n read slaves, and each shard holds a portion (shard) of data

1. How is the data organized in a clustered index as opposed to non clustered?

My answer: it's always a IOT (index organized table) hit in that case, without using a heap, this is considered fastest

1. Elaborate on that (IoT tables and heap tables)

My answer: couldn't, really have no idea what happens at that level (really low level)

need to look it up

1. When do we use heap tables?

My answer: tables with one index only are best implemented as clustered indexes or index-organized tables.. tables with more indexes can often benefit from heap tables

need to look it up

1. Why?

My answer: didn't know, couldn't explain

Expected answer: heap tables have the benefit of providing a stationary master copy that can be easily referenced

Need to look up further.

1. How do we turn a clustered index into a heap table

My answer: dropping a clustered index transforms the table into a heap table

1. What is ValueTask, why do we have it?

My answer: didn't know (read a bit on ValueTasks, but haven't remembered anything)

Expected answer: we can utilize ValueTask<T> to **avoid allocations when methods complete synchronously**

Explain - couldn't, has something to do with GC probably there are no allocations so there's less of an overhead (and I was right, just couldn't explain)

Expected answer (I guess):

Task is a class. As a class, that means that any operation which needs to create one needs to allocate an object, and the more objects that are allocated, the more work the garbage collector (GC) needs to do, and the more resources we spend on it that could be spent doing other things.

When ValueTask<TResult> was introduced in .NET Core 2.0, it was purely about optimizing for the synchronous completion case, in order to avoid having to allocate a Task<TResult> to store the TResult already available.

ValueTask<TResult> was introduced in .NET Core 2.0 as a struct capable of wrapping either a TResult or a Task<TResult>. This means it can be returned from an async method, and if that method completes **synchronously and successfully, nothing need be allocated: we can simply initialize this** ValueTask<TResult> struct with the TResult and return that.

Only if the method completes **asynchronously** does a Task<TResult> need to be allocated, with the ValueTask<TResult> created to wrap that instance (to minimize the size of ValueTask<TResult> and to optimize for the success path, an async method that faults with an **unhandled exception will also allocate** a Task<TResult>, so that the ValueTask<TResult> can simply wrap that Task<TResult> rather than always having to carry around an additional field to store an Exception).

Caveats (do not use this type when):

* Awaiting a ValueTask / ValueTask<TResult> **multiple** times. The underlying object may have been recycled already and be in use by another operation. In contrast, a Task / Task<TResult> will never transition from a complete to incomplete state, so you can await it as many times as you need to, and will always get the same answer every time.
* Awaiting a ValueTask / ValueTask<TResult> **concurrently**. The underlying object expects to work with only a single callback from a single consumer at a time, and attempting to await it concurrently could easily introduce race conditions and subtle program errors. It’s also just a more specific case of the above bad operation: “awaiting a ValueTask / ValueTask<TResult> multiple times.” In contrast, Task / Task<TResult> do support any number of concurrent awaits.
* Using .GetAwaiter().GetResult() when the operation hasn’t yet completed. The IValueTaskSource / IValueTaskSource<TResult> implementation need not support blocking until the operation completes, and likely doesn’t, so such an operation is inherently a **race condition** and is unlikely to behave the way the caller intends. In contrast, Task / Task<TResult> do enable this, blocking the caller until the task completes.

However, ValueTask / ValueTask<TResult> are great choices when a) you expect consumers of your API to only await them directly, b) allocation-related overhead is important to avoid for your API, and c) either you expect synchronous completion to be a very common case, or you’re able to effectively pool objects for use with asynchronous completion. When adding abstract, virtual, or interface methods, you also need to consider whether these situations will exist for overrides/implementations of that method.

**In fact, the C# compiler takes advantage of this when implementing async iterators to make async iterators as allocation-free as possible.**

At some point they asked about iterators, so this is what they were getting at -> implementing allocation free iterators

In essence, the issue the .net devs were trying to solve by ValueTaskis is -> how to make async/await (both sync and async situations) allocate less (or none) Task objects on the heap in order to save space - which is especially important in all large “for each” (iteration) scenarios.

Basically in **sync** scenario - you have a **cached Task object to return**, so the same object gets returned, whereas in **async** scenarios the .net runtime will make a “best effort” to **reuse already created Task classes** (obviously this is not viable for multithreaded access - it simply would not work in that case).

You can approach the performance problem by using **Span<>** as an object of iteration (the thing that gets iterated over), and by using **ValueTask<>** when using async/await pattern.

The perfect structure (that is also thread safe) for producing and consuming a huge number of async tasks is the new **Channel** construct, it already uses ValueTasks and is already made thread safe by design. Also, it implements a form of **backpressure (a form of load balancing)**.

TODO: async iterators

1. Why would we use Span<>? **(new in .NET 6)**

My answer: didn't know (did read something on the topic at some point, couldn't remember)

Expected answer:

Span<T> is a [ref struct](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/struct#ref-struct) that is allocated on the stack rather than on the managed heap. Ref struct types have a number of restrictions to ensure that they cannot be promoted to the managed heap, including that they can't be boxed, they can't be assigned to variables of type [Object](https://docs.microsoft.com/en-us/dotnet/api/system.object?view=net-5.0), dynamic or to any interface type, they can't be fields in a reference type, and they can't be used across await and yield boundaries. In addition, calls to two methods, [Equals(Object)](https://docs.microsoft.com/en-us/dotnet/api/system.span-1.equals?view=net-5.0#system-span-1-equals(system-object)) and [GetHashCode](https://docs.microsoft.com/en-us/dotnet/api/system.span-1.gethashcode?view=net-5.0), throw a [NotSupportedException](https://docs.microsoft.com/en-us/dotnet/api/system.notsupportedexception?view=net-5.0).

A Span<T> represents a contiguous region of arbitrary memory. A Span<T> instance is often used to hold the elements of an array or a portion of an array. Unlike an array, however, a Span<T> instance can point to managed memory, native memory, or memory managed on the **stack**.

using System;

var array = new int[] { 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 };

var slice = new Span<int>(array, 2, 5);

for (int ctr = 0; ctr < slice.Length; ctr++)

slice[ctr] \*= 2;

// Examine the original array values.

foreach (var value in array)

Console.Write($"{value} ");

Console.WriteLine();

// The example displays the following output:

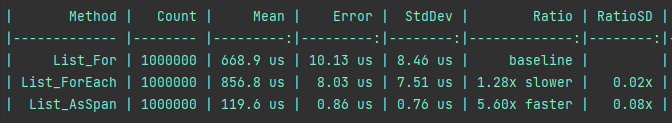
// 2 4 12 16 20 24 28 16 18 20

The [CollectionsMarshal.AsSpan method](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.collectionsmarshal.asspan?WT.mc_id=DT-MVP-5003978) is **unsafe** and should be used only if you know what you're doing. CollectionsMarshal.AsSpan returns a Span<T> on the private array of List<T>. Iterating over a Span<T> is fast as the JIT uses the same tricks as for optimizing arrays. Using this method, it won't check the list is not modified during the enumeration.

As said, this method is unsafe and you may get weird behaviors if you don't know what you are doing. For instance, you may iterate over items that are not part of the list anymore in some cases

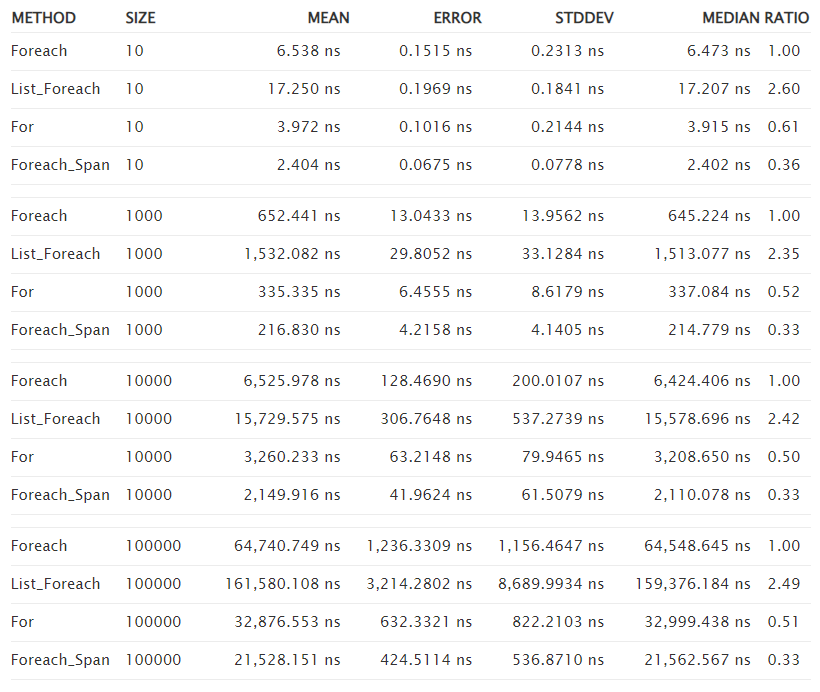
Sample code:

public int Sum(List<int> source) => Sum(CollectionsMarshal.AsSpan(source));



Using foreach makes it 1.28 times slower, while using AsSpan() makes it 5.6 times faster.

Benchmark (source: <https://www.meziantou.net/fastest-way-to-enumerate-a-list-t.htm> ):



A lot of their questions resolved around array iteration, iterators, yield, benchmarks etc. so I think this might be relevant as well: <https://antao-almada.medium.com/array-iteration-performance-in-c-f9801a2c7a3c>

<https://ikbalkazanc.medium.com/c-6-performance-tricks-ae749a7a8c45>

From these two articles it becomes apparent that the idea they were trying to put light on was the idea that **foreach will always be slower** because it implements IEnumerator, the compiler output for foreach shows this and this is the reason why foreach will almost always be a tiny bit slower UNLESS you turn the array you're iterating over into a Span<> -> if you can live with the side effects ([CollectionsMarshal.AsSpan method](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.interopservices.collectionsmarshal.asspan?WT.mc_id=DT-MVP-5003978)). You can expect as much a 6x speed improvement when iterating over an array if you first Marshal it unsafely as a Span.

Compiler output for sum, take notice of enumerator.MoveNext and enumerator.Dispose() -> this is what makes iterating via foreach a bit slower…

static int Sum(IEnumerable<int> source)

{

var sum = 0;

IEnumerator<int> enumerator = source.GetEnumerator();

try

{

while(enumerator.MoveNext())

sum += enumerator.Current;

}

finally

{

enumerator.Dispose()

}

return sum;

}

1. Explain double lock pattern in regards to processor architecture?

My answer: did the best I could, haven't been using locks in a really long time, the lock is a keyword that enables, well locking a reference type (usually its a private readonly object) to “serialize” access to a mutable property to ensure it behaves nicely in a multithreaded scenario (where you have multiple threads mutating the same property), the access is anchored (blocked/released) via the object, sometimes we have to use 2 locks due to processor optimizations that may “go around” a single lock (couldn't explain further)

lock statement is syntactic sugar for a try/finally statement with Monitor.Enter and Monitor.Exit

A thread blocked while awaiting a contended lock has a [ThreadState](https://www.albahari.com/threading/part2.aspx#_ThreadState) of **WaitSleepJoin**

Expected answer: there is no double lock but there is a double boolean check one implements manually to check if a lock has been already set/evaluated to prevent fringe case race conditions on certain processor architectures

Sample of a simple locking code:

class ThreadSafe

{

static readonly object \_locker = new object();

static int \_val1, \_val2;

static void Go()

{

lock (\_locker)

{

if (\_val2 != 0) Console.WriteLine (\_val1 / \_val2);

\_val2 = 0;

}

}

}

See the compiler output for this: <https://gist.github.com/brunobozic/898fc52fdc6e8753e3dd3b6bca706a76>

Take notice of Monitor.Enter and Exit.

1. Why can't we use value type as a lock object?

My answer: boxing would happen automatically and you would get a **different object each time**, preventing lock

1. Are “finally” clauses run when a thread is aborted?

My answer: they don't

1. Are “finally” clauses run when you kill an app via Task Manager?

My answer: they don't

1. Explain semaphore structure?

My answer: structure to manually control threading to avoid deadlocks, if implemented correctly

Expected answer:

A semaphore with a capacity of one is similar to a Mutex or lock, except that the semaphore has no “owner” — **it’s thread-agnostic**. Any thread can call Release on a Semaphore, whereas with Mutex and lock, only the thread that obtained the lock can release it. Semaphores can be useful in limiting concurrency — preventing too many threads from executing a particular piece of code at once.

1. How is semaphore implemented in .net (low level)? - didn't know, did know about the lock keyword thou

My answer: the lock has “under the cover” a Monitor.Enter and Monitor.Exit (inside a finally block)

the semaphore on the other hand .. TBD need to look it up

1. Why do we have .GetAwaiter()?

My answer: so we can get a handle on internal “state machine” to read/interact with the current state of the “promise” structure that is Task

Proper answer (what they wanted to hear):

Task.GetAwaiter().GetResult() is preferred over Task.Wait and Task.Result because it **propagates exceptions** rather than wrapping them in an AggregateException, there's probably more…

1. How is the stack overflow exception handled by the .net runtime, and how does it arrive at userland?

My answer: no idea

1. What is a channel construct in TPL **(new in .NET 6)**

My answer: didn't know

It's a brand new thing in newest .net

This is the relevant code sample, special attention to ValueTask and Channel

using BenchmarkDotNet.Attributes;

using BenchmarkDotNet.Running;

using System.Threading.Channels;

using System.Threading.Tasks;

using System.Threading.Tasks.Dataflow;

[MemoryDiagnoser]

public class Program

{

static void Main() => BenchmarkRunner.Run<Program>();

private readonly Channel<int> \_channel = Channel.CreateUnbounded<int>();

private readonly BufferBlock<int> \_bufferBlock = new BufferBlock<int>();

[Benchmark]

public async Task Channel\_ReadThenWrite()

{

ChannelWriter<int> writer = \_channel.Writer;

ChannelReader<int> reader = \_channel.Reader;

for (int i = 0; i < 10\_000\_000; i++)

{

ValueTask<int> vt = reader.ReadAsync();

writer.TryWrite(i);

await vt;

}

}

[Benchmark]

public async Task BufferBlock\_ReadThenWrite()

{

for (int i = 0; i < 10\_000\_000; i++)

{

Task<int> t = \_bufferBlock.ReceiveAsync();

\_bufferBlock.Post(i);

await t;

}

}

}

Expected answer:

**Channels give us a way to communicate between concurrent (async) operations in .NET.**

We can think of a channel as a sort of thread-safe queue. This means that we can add items to a channel in one concurrent operation and read those items in another. (We can also have multiple readers and multiple writers.)

Since channels are thread-safe, we can have multiple processes writing to and reading from the channel without worrying about missed writes or duplicate reads

Here is how these look on the inside:

public abstract class ChannelWriter<T>

{

public abstract bool TryWrite(T item);

public virtual ValueTask WriteAsync(T item, CancellationToken cancellationToken = default);

public abstract ValueTask<bool> WaitToWriteAsync(CancellationToken cancellationToken = default);

public void Complete(Exception error);

public virtual bool TryComplete(Exception error);

}

public abstract class ChannelReader<T>

{

public abstract bool TryRead(out T item);

public virtual ValueTask<T> ReadAsync(CancellationToken cancellationToken = default)

public abstract ValueTask<bool> WaitToReadAsync(CancellationToken cancellationToken = default);

public virtual IAsyncEnumerable<T> ReadAllAsync([EnumeratorCancellation] CancellationToken cancellationToken = default);

public virtual Task Completion { get; }

}

WriteAsync can be used, with the producer awaiting the result of WriteAsync and only being allowed to continue when room becomes available

A producer can await for WaitToWriteAsync to return true, and only then choose to produce a value that it then TryWrites or WriteAsyncs to the channel.

channels by default can be used by any number of producers and any number of consumers concurrently

Channels support the notion of completion, where a producer can signal to a consumer that there won’t be any further items produced, enabling the consumer to gracefully stop trying to consume. This is done via the Complete or TryComplete methods previously shown on ChannelWriter<T> (Complete is just implemented to call TryComplete and throw if it returns false). But if one producer marks the channel as complete, other producers need to know they’re no longer welcome to write into the channel; in that case, TryWrite returns false, WaitToWriteAsync also returns false, and WriteAsync throws a ChannelCompletedException.

The final member of ChannelReader<T> is Completion. This simply returns a Task that will complete when the channel reader is completed, meaning the channel was marked for completion by a writer and all data has been consumed.

You may have noticed that many of the methods in the surface area return ValueTask and ValueTask<T> rather than Task and Task<T>. As we saw in our trivial example implementation at the beginning of this article, we can utilize ValueTask<T> to avoid allocations when methods complete synchronously, but the System.Threading.Channels implementation also takes advantage of the advanced IValueTaskSource and IValueTaskSource<T> interfaces to avoid allocations even when the various methods complete asynchronously and need to return tasks.

**Need to be careful because**:

while ChannelReader<T> bears many similarities to IEnumerator<T>, this example can’t be implemented well on top of IEnumerator<T> (or IAsyncEnumerator<T>). I{Async}Enumerator<T> exposes a MoveNext{Async} method, which moves the cursor ahead to the next item, which is then exposed from Current. If we tried to implement such a WhenAny on top of IAsyncEnumerator<T>, we would need to invoke MoveNextAsync on each. In doing so, we would potentially move both ahead to their next item. If we then used that method in a loop, we would likely end up missing items from one or both enumerators, because we would potentially have advanced the enumerator that we didn’t return from the method.

Also take notice that using channels you can also construct arbitrary pipelines as in the code that follows:

var generator = new ThumbnailGenerator();

await Channel

.CreateBounded<string>(50000)

.Source(Directory.GetFiles(Constants.PostsDirectory))

.PipeAsync(

maxConcurrency: 2,

capacity: 100,

transform: async postPath =>

{

var frontMatter = await generator.ReadFrontMatterAsync(postPath);

return (postPath, frontMatter);

})

.Filter(tuple => tuple.Item2 != null)

.PipeAsync(

maxConcurrency: 10,

capacity: 20,

transform: async tuple =>

{

var (postPath, frontMatter) = tuple;

var cardImage = await generator.CreateImageAsync(frontMatter!);

return (postPath, frontMatter, cardImage);

})

.ReadAllAsync(async tuple =>

{

var (postPath, \_, cardImage) = tuple;

await generator.SaveImageAsync(

cardImage, Path.GetFileName(postPath) + ".png");

});

1. How would you write a combinator?

My answer: didn't know

Answer (googled afterwards):

public static async ValueTask<ChannelReader<T>> WhenAny<T>(

ChannelReader<T> reader1, ChannelReader<T> reader2)

{

var cts = new CancellationTokenSource();

Task<bool> t1 = reader1.WaitToReadAsync(cts.Token).AsTask();

Task<bool> t2 = reader2.WaitToReadAsync(cts.Token).AsTask();

Task<bool> completed = await Task.WhenAny(t1, t2);

cts.Cancel();

return completed == t1 ? reader1 : reader2;

}

1. What is the latest C# you know and what are your thoughts on what if offers?

My answer: managed to get around the question by babbling about the new Record type (the immutable one) it's basically this:

public Task<IEnumerable<string>> ReadStreamsAsync(IEnumerable<Request> requests) =>

requests

.Select(r => r.Stream.ReadToEndAsync())

.WhenAllAsync();

public **record** Request(Stream Stream); // <= **immutable**

Expected anwer: they probably wanted to know if I had worked on C# 10 (I didn't have that opportunity), and simply proceeded asking questions pertaining C# 10 (Span<>, Channels, ValueTask<> etc.)

Should have hijacked the discussion by talking about these things that are new to c# 10:

1. Global Using Statements
2. Implicit Using statements
3. FileScoped Namespaces

// Namespace without parentheses but with a semicolon

namespace Medium;

public class Program

{

public static void Main() {

Console.WriteLine(new Reader( "Arnold", "Abraham"));

}

}

public record Reader(string firstName, string lastName);

1. Extended Property Patterns
2. Constant Interpolated Strings
3. DateOnly
4. TimeOnly
5. Check Strings for Null

// Safer check through the CallerArgumentExpression

public void ProcessName(string name, string firstName)

{

ArgumentNullException.ThrowIfNull(name);

ArgumentNullException.ThrowIfNull(firstName);

}

1. PriorityQueue

using System;

class Program

{

public static void Main()

{

var queue = new PriorityQueue<string, int>();

queue.Enqueue("Element 1", 9);

queue.Enqueue("Element 2", 11);

queue.Enqueue("Element 3", 27);

queue.Enqueue("Element 4", 7);

queue.Enqueue("Element 5", 13);

queue.Enqueue("Element 6", 0);

while (queue.TryDequeue(out string element, out int priority))

Console.WriteLine($"element: {element} - priority: {priority}");

}

}

1. Process data in Chunks

using System;

class Program

{

public static void Main()

{

// Iterate with chunks

var list = Enumerable.Range(1, 30);

const int size = 10;

int chunkCounter = 0;

foreach (var chunk in list.Chunk(size))

{

Console.WriteLine($"CHUNK: #{chunkCounter++}");

foreach (var item in chunk)

{

Console.WriteLine($"-> {item}");

}

}

}

}

1. What happens in runtime if GC is getting choked, how would you troubleshoot that?

My answer: someone is probably creating ton load of classes in rapid succession, probably in a for loop, that are not being collected

This question has a connection to ValueTask<> questions, for more info see there.

1. How does the GC decide if an object can be cleaned up?

My answer: root graphing then propagating gen0 -> gen2, gen0 get popped off the stack immediately, long lived ones survive all the way to gen2

1. Any fringe cases?

My answer: yes, the finalizer queue and finalizer thread where we can have objects “living a little bit longer” so that the finalizer can complete, does not work all that well combined with GC.Collect

1. What does the yield keyword signify?

My answer: Ienumerable implementation

1. What are statistics?

My answer: The query optimizer uses these to decide on the strategy

1. Explain?

My answer: didn't know

1. Are statistics row based or column based?

My answer: didn't know

Expected answer: they are **column** based need to look it up

1. Are null values calculated into statistics

My answer: didn't know

Expected answer: need to look it up

1. How are function based indexes shown in statistics

My answer: didn't know,

Expected answer: they are kept on TABLE level as VIRTUAL COLUMNS

don't really know, nor care, why this is so, need to look it up

1. Are they ever renewed automatically by db?

My answer: didn't know (answer: it depends)

1. Do statistics auto-update when a new index is created

My answer: yes (but -> no)

1. On all databases?

My answer: yes (wrong, Oracle does **not** recalculate)

1. What is an index?

My answer: explained root, leaf nodes, page, b-tree, doubly linked list and why those as used, what problems to they solve (all boils down to how data is laid out on the disk level, need to be able to explain why leaf nodes are a linked list (for fast inserts))

1. What is a covering index?

My answer: an index that includes all the attributes from where part and the select

1. Does a covering index read data from the disk?

My answer: nope, that's their advantage, the data they need is already contained at the leaf level so no actual disk access is actually needed

1. Is there a limit?

My answer: Obviously, the size

* 1. what is the max allowed size?

My answer: Depends

* 1. for MSSQL?

My answer: didn't know, said 8000 bytes but that's the size of a page not the limit for a covering index - should I be memorizing these things? For Oracle, for MSSQL, for Postgresql and every other database etc.? Why?

1. What is a secondary index and why do we use it?

My answer: didn't know need to look it up

1. Does it (the secondary index) contain ROWID?

My answer: didn't know

Expected answer: contains only the primary key of the primary index, this might be Oracle specific though, need to read up on this

1. When would you use constructor injection (IoC/DI) and when property injection and why?

My answer: constructor injection is better because by the constructor signature you can dictate all required dependencies and the container will immediately tell you if any are not resolvable, makes coding easier, also convention, I read code faster if all dependencies are listed inside constructor signature

Expected answer: we usually advise people to use constructor injection for all mandatory collaborators and setter injection for all other properties. Again, constructor injection ensures all mandatory properties have been satisfied, and it is simply not possible to instantiate an object in an invalid state (not having passed its collaborators). In other words, when using constructor injection you do not have to use a dedicated mechanism to ensure required properties are set (other than normal Java mechanisms).

One of the arguments for not using constructor injection is the lack of argument names in constructors and the fact that these do not appear in the XML. I would argue that in **most** applications, this does not matter that much.