Design Principles vs. Performance

Wie mein Wissen über Interna und Performance das Design meiner Anwendungen verändert hat – ein anekdotischer Vortrag

Digital Craftsmanship Nordoberpfalz
16.02.2021

Agenda

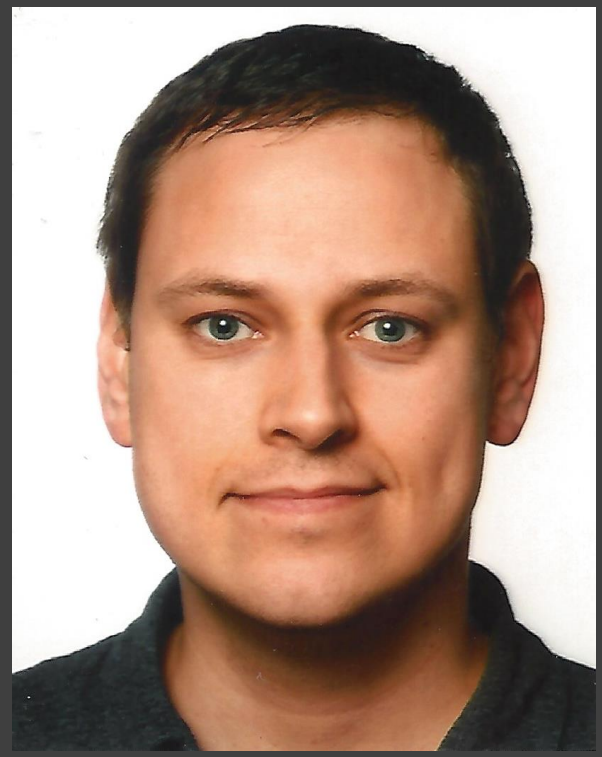
- Wie habe ich bis 2015 meine Software-Applikationen gestaltet?
- Was habe ich über Performance gelernt?
  - Speichermanagement und GC in der .NET Runtime
  - async await, Threading und Thread Pool in der .NET Runtime
- Auswirkungen auf Software-Design

Über mich

Kenny Pflug

- Senior Software Developer bei Synnotech
- Doktorand an der Universität Regensburg

- Twitter: @feO2x
- GitHub: feO2x
- YouTube: youtube.com/c/kennypflug



Anno 2015…

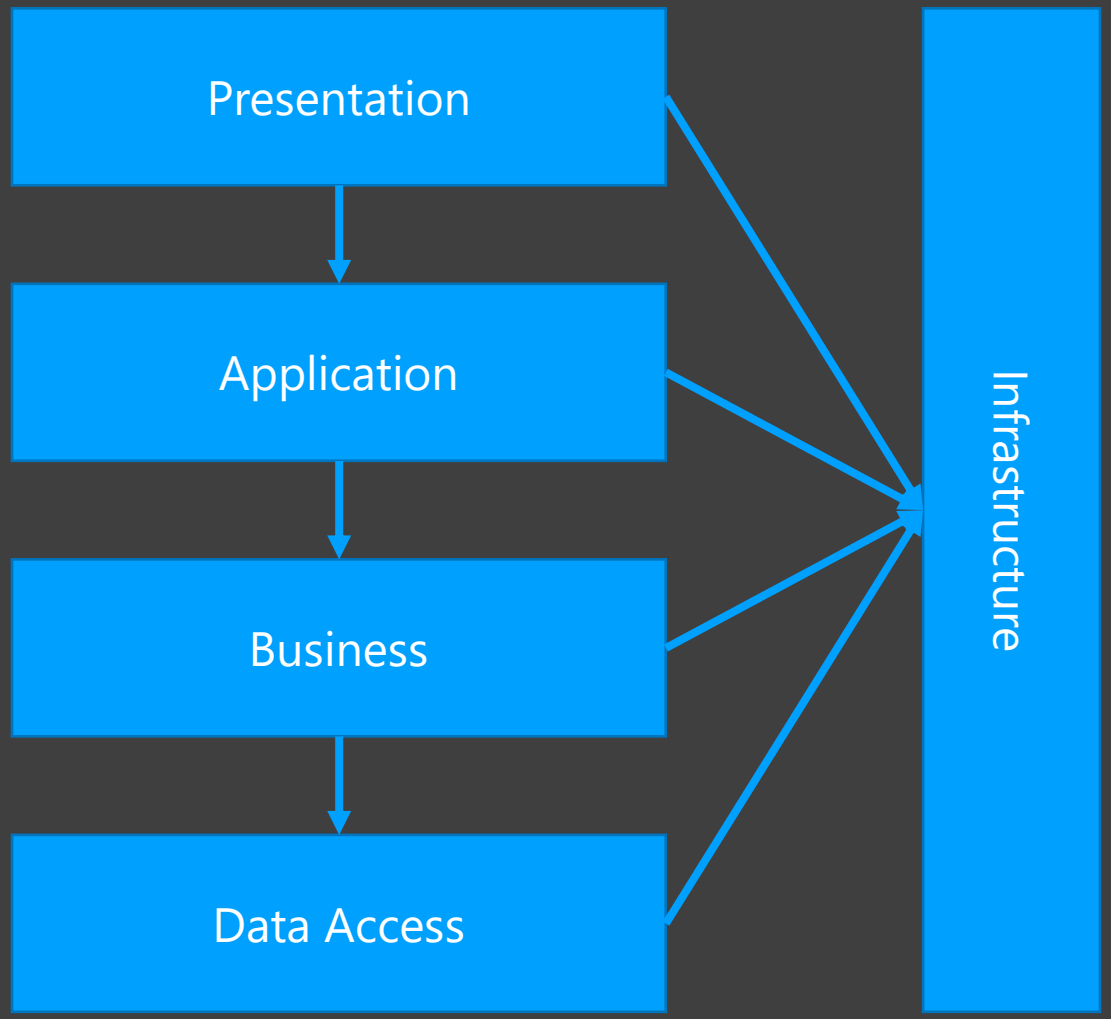
Wie ich bis 2015 meinen C# Code gestaltet habe

- Viele kleine Klassen, welche jeweils genau eine Aufgabe übernehmen
- Interfaces / abstrakte Basisklasse zwischen Aufrufer und Aufgerufenen
- Objektgraphen werden über Dependency Injection (DI) aufgelöst – normalerweise mithilfe eines DI Containers
- If-Else- oder Switch-Blöcke werden ersetzt durch Objekte mit Abstraktion
- Einsatz etablierter Design Patterns
- Test Driven Development

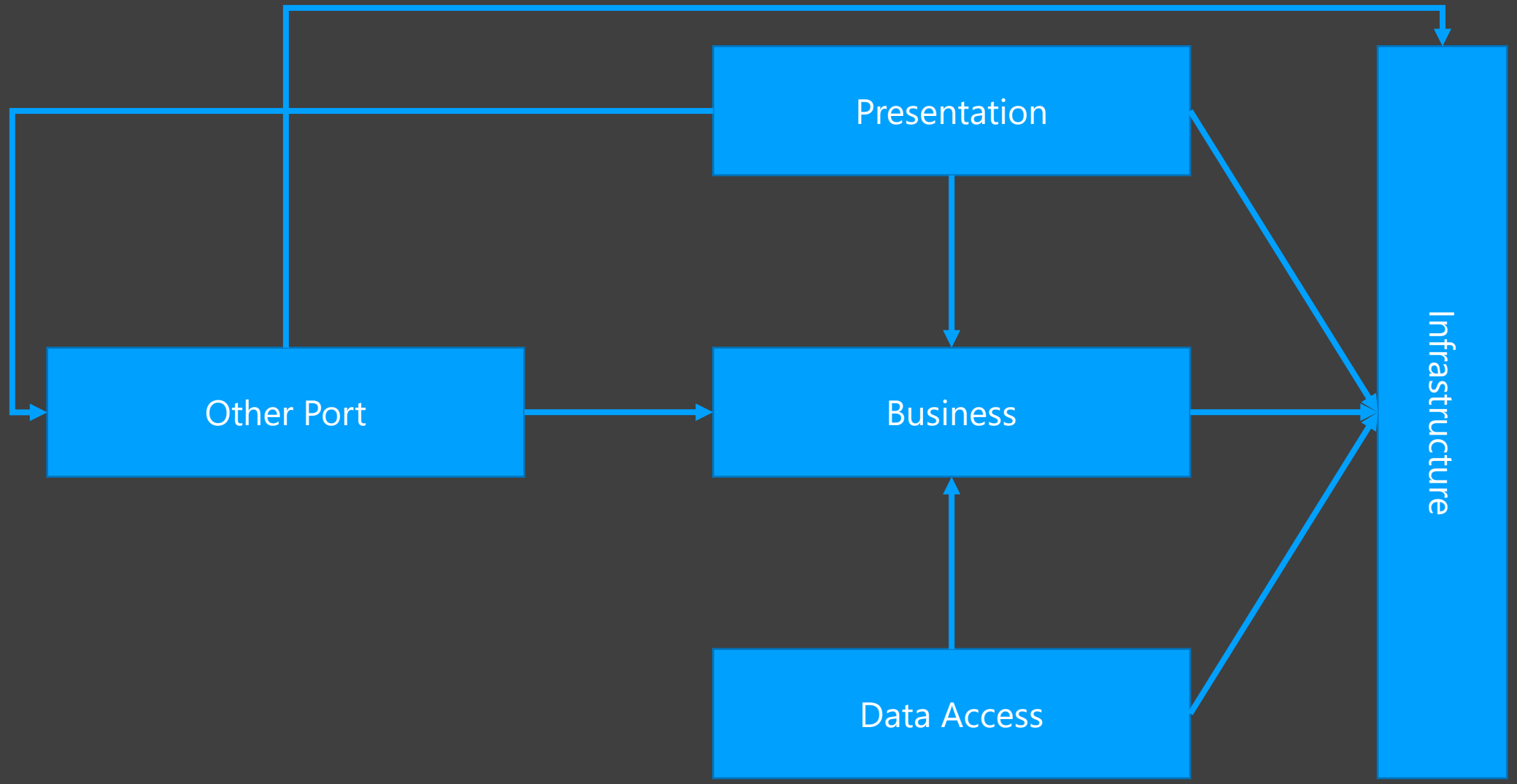
2015 – Design Patterns

- Factory, Abstract Factory
- Builder
- Singleton
- Prototype
- Adapter
- Composite
- Decorator
- Facade
- Proxy
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- Strategy
- Visitor
- Immutable Objects
- Arrange – Act – Assert (– Cleanup)
- Dummy, Stub, Spy, Mock
- Model – View – View Model
- Model – View – Controller
- Object Pooling
- und viele mehr…

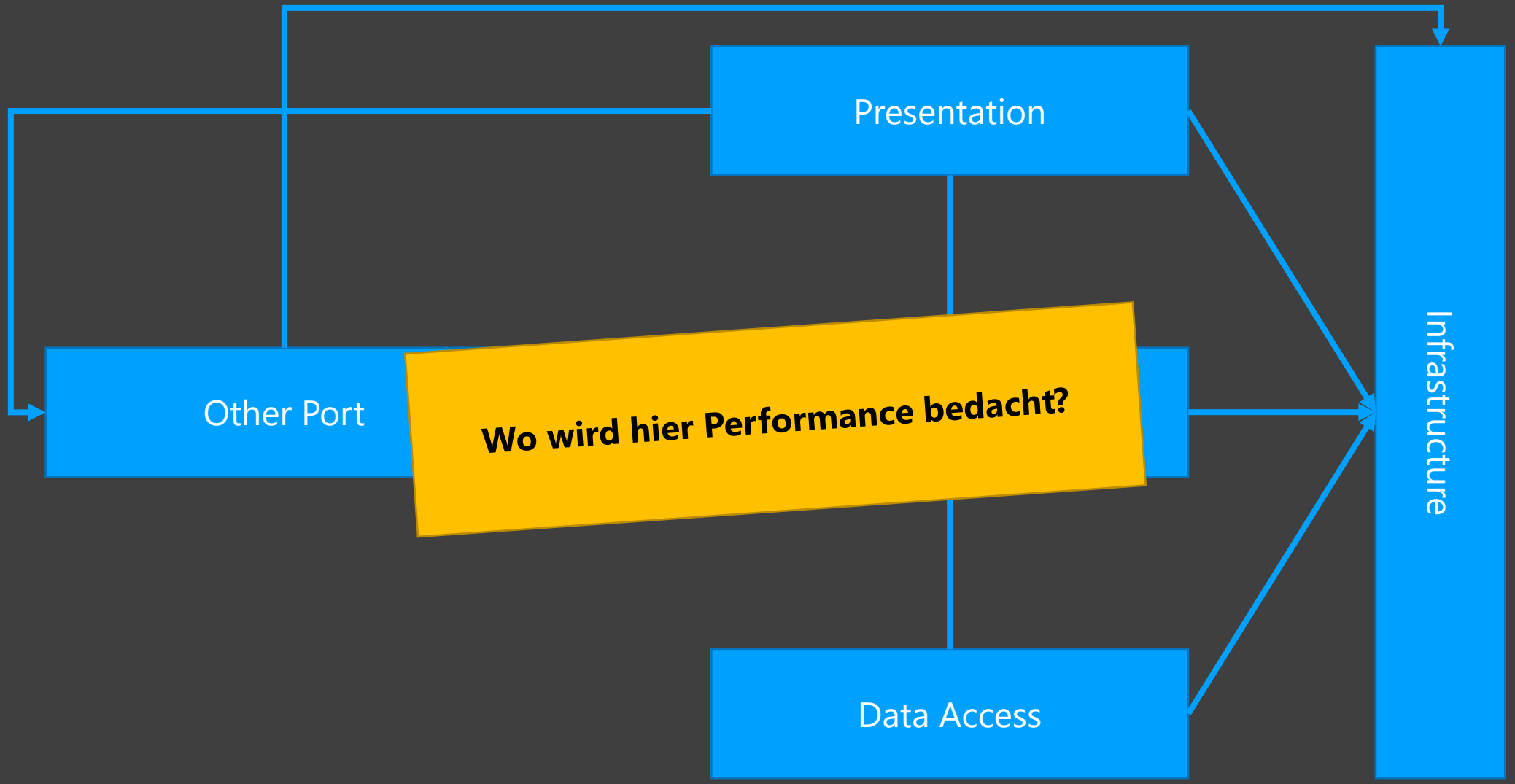
2015 – Software Architectures: N-Layer / N-Tier



2015 – Software Architectures: Ports and Adapters / Onion Architecture

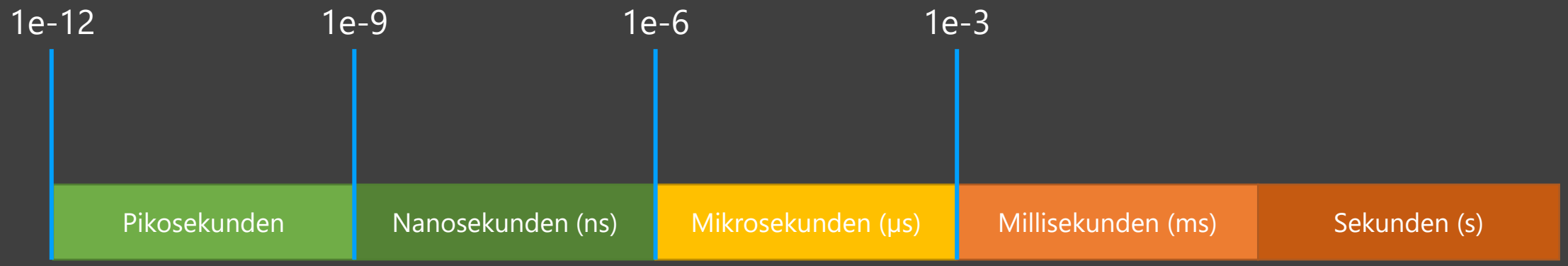


2015 – Software Architectures: Ports and Adapters / Onion Architecture

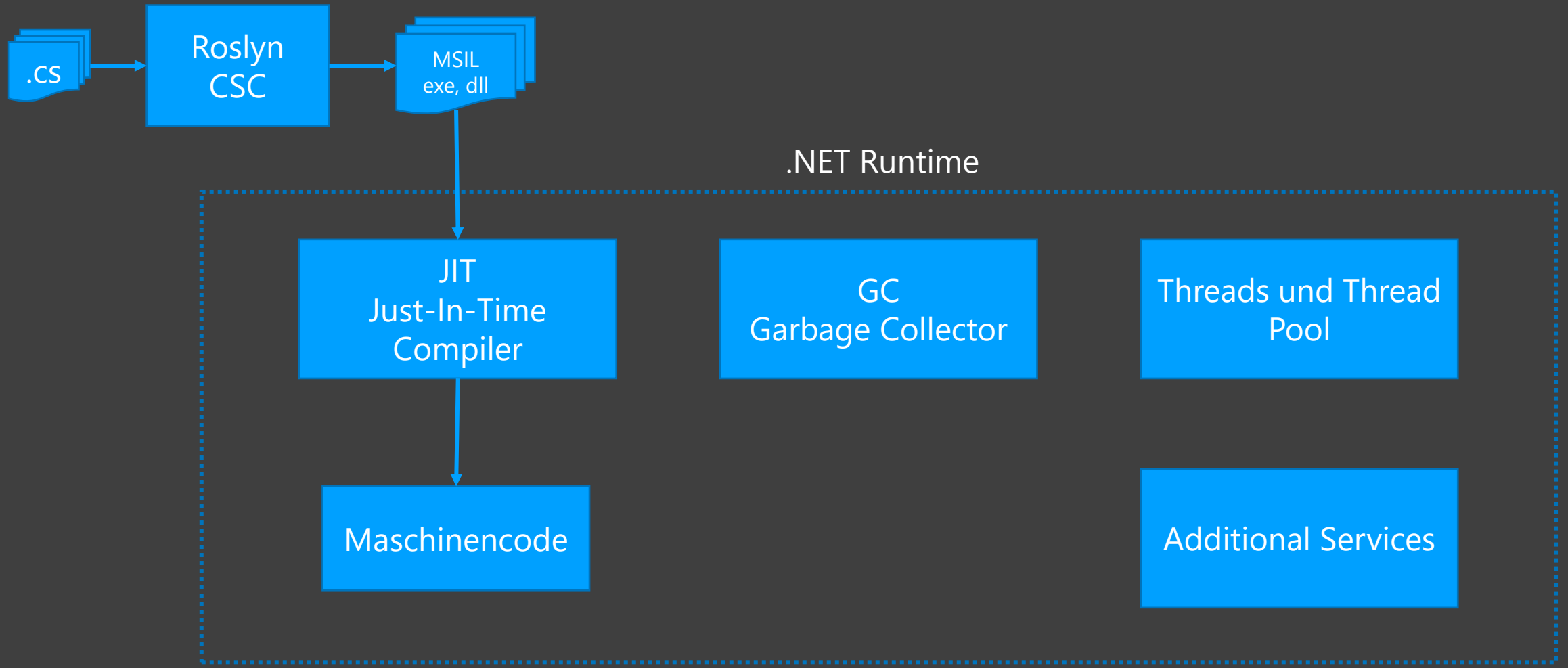


Performance of Everyday Things

Die Zeit, die Zeit...

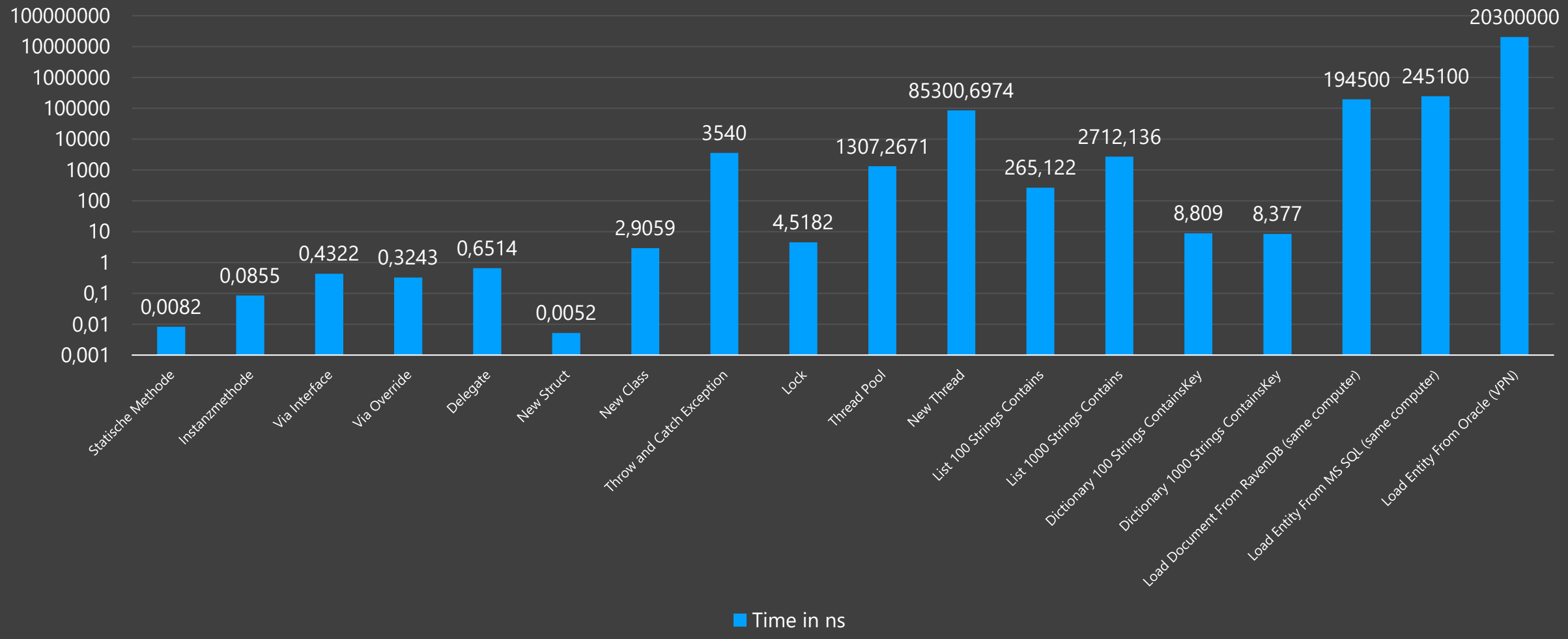


Noch ein paar Worte zur .NET Runtime



Ein paar Zeiten zum Vergleich

BenchmarkDotNet=v0.12.1, OS=Windows 10.0.19042
AMD Ryzen 9 5950X, 1 CPU, 32 logical and 16 physical cores
.NET Core SDK=5.0.103
[Host] : .NET Core 5.0.3 (CoreCLR 5.0.321.7212, CoreFX 5.0.321.7212), X64 RyuJIT
Job-SWHTBI : .NET Core 5.0.3 (CoreCLR 5.0.



Welche Schlüsse ziehen wir daraus?

- Multithreading macht erst ab einer gewissen Anzahl Operationen Sinn
- Neue Threads sind teuer, der Thread Pool verwaltet automatisch diese für uns
- I/O ist deutlich teurer als In-Memory-Operationen
- Absolute Schnelligkeitswerte sind an die jeweilige Hardware und Plattform gebunden, wichtig sind die relativen Ergebnisse zueinander

Wie gut skaliert mein Code?

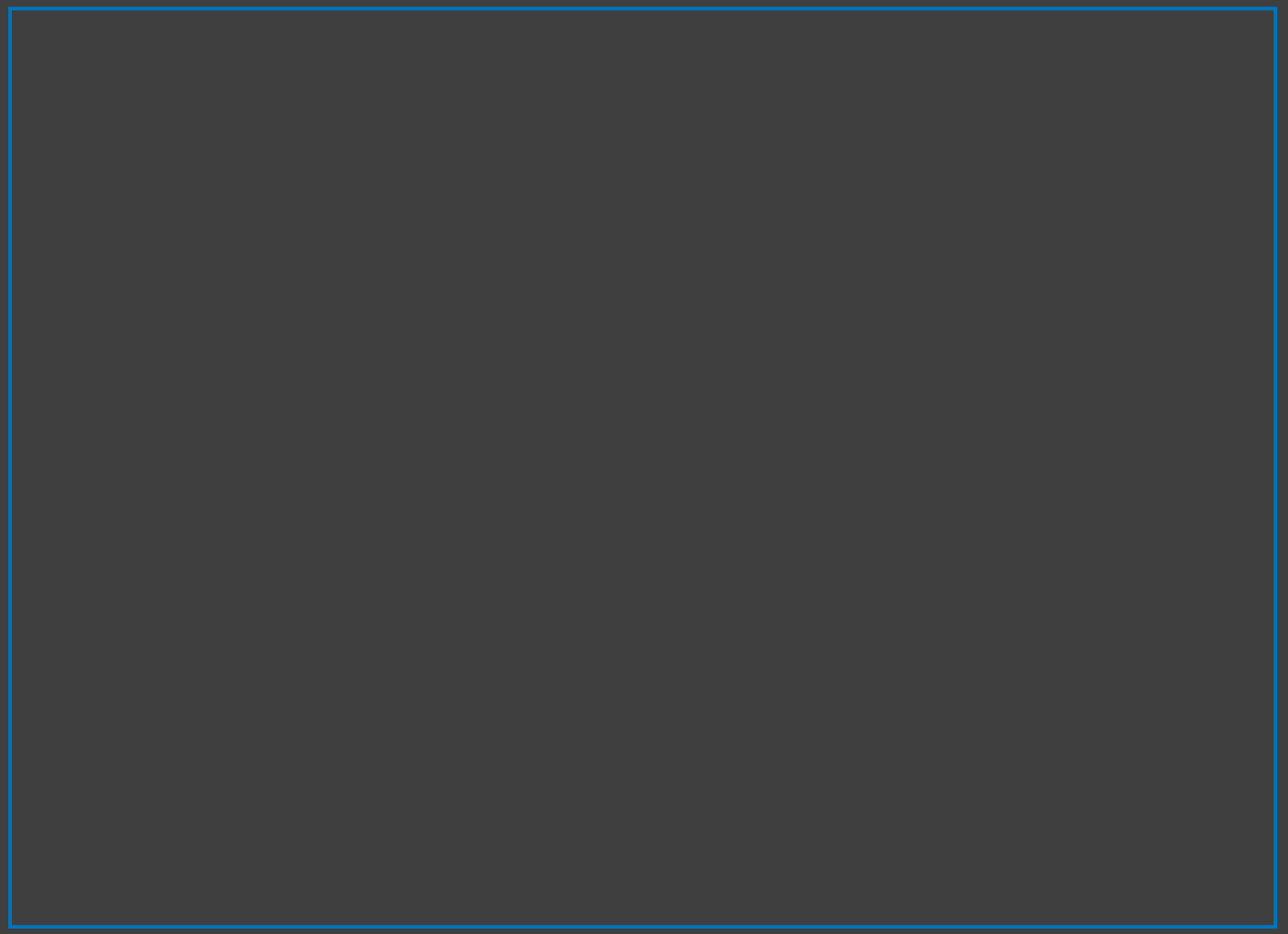
Memory Management in .NET

Speicherabbilder und der Garbage Collector

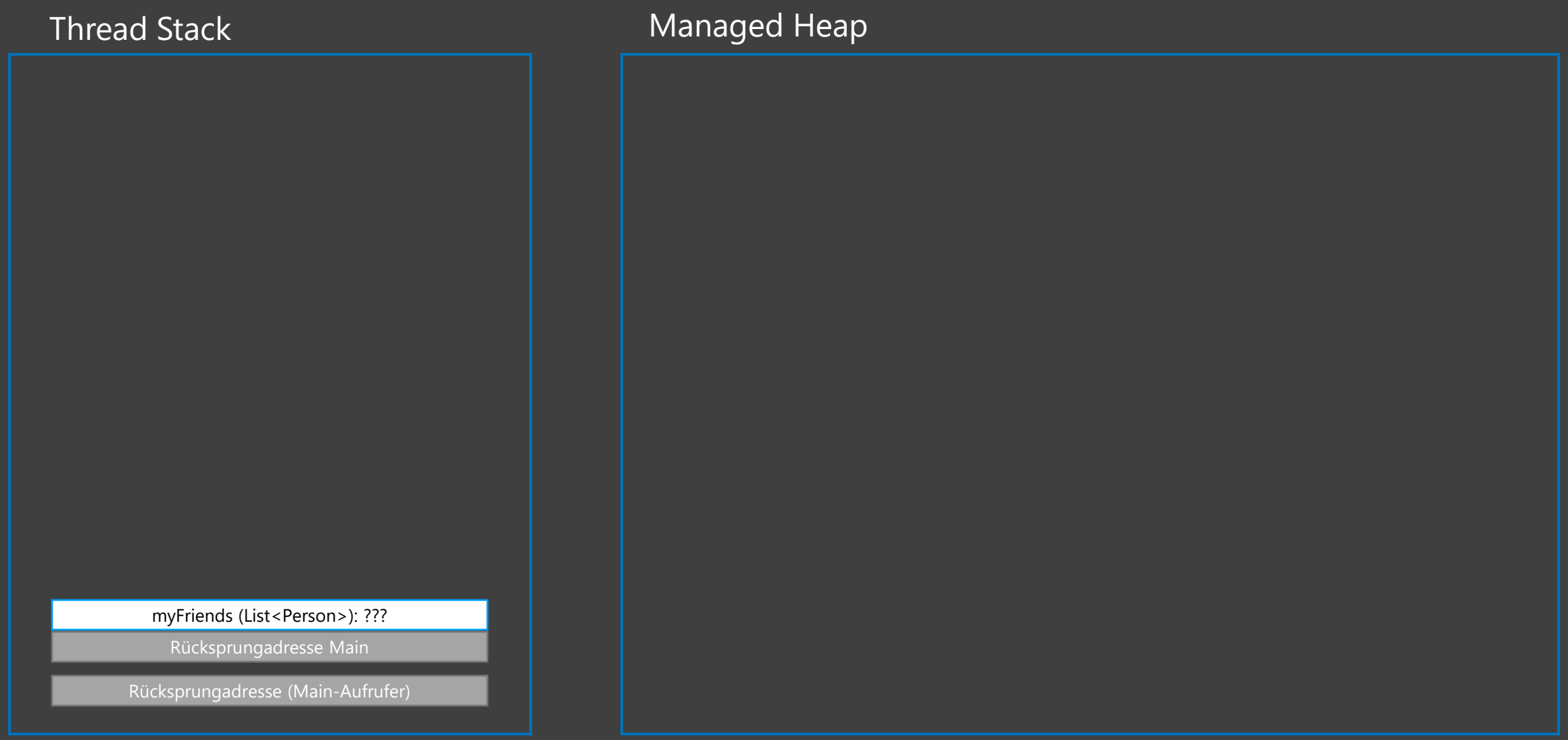
Thread Stacks und Managed Heap am simplen Beispiel (1)

Thread Stack

Managed Heap



Thread Stacks und Managed Heap am simplen Beispiel (2)

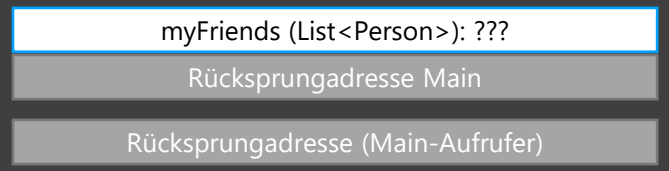


Thread Stacks und Managed Heap am simplen Beispiel (3)

Thread Stack

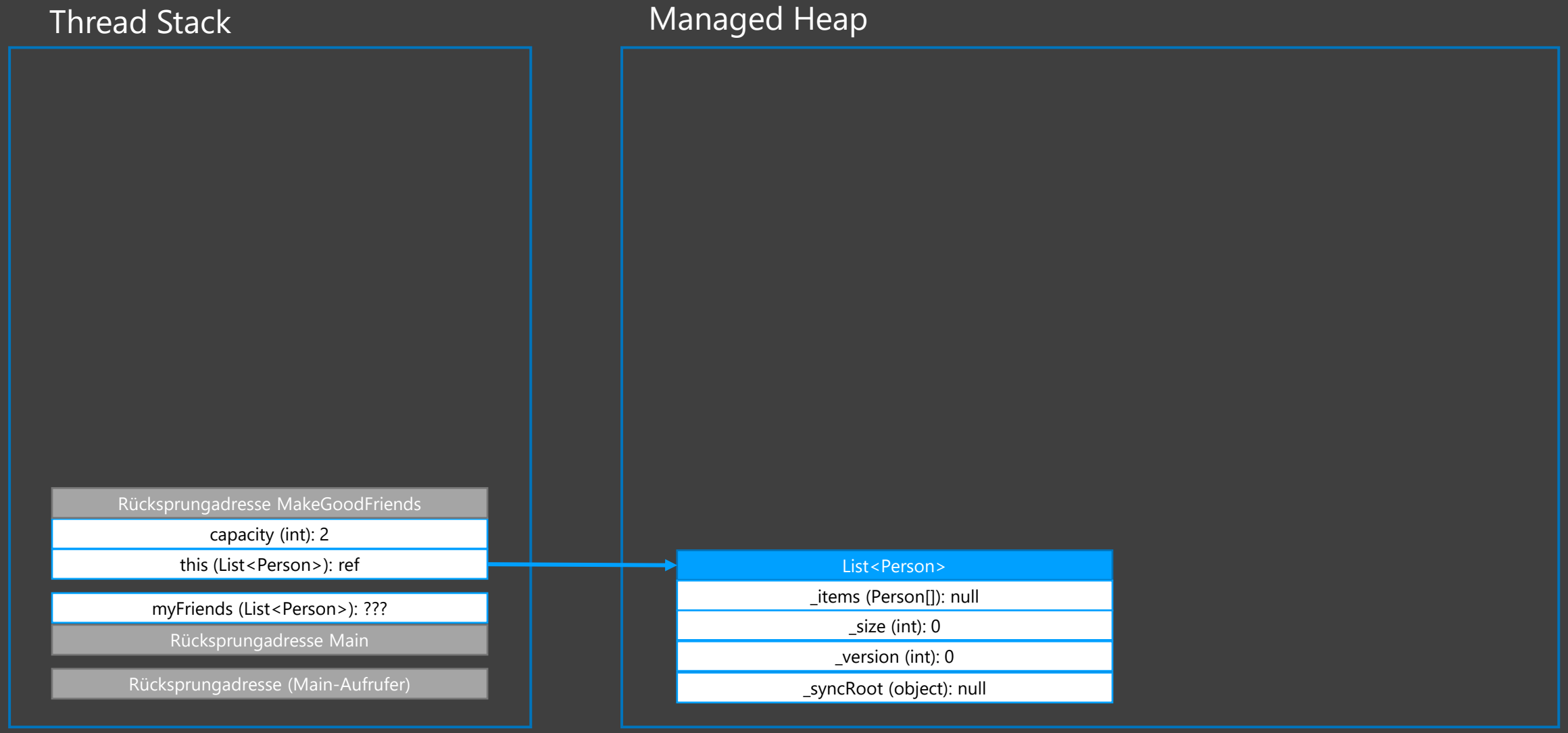
Activation Frame / Stack Frame

- Erst die Parameter
- Dann die Rücksprungadresse
- Dann die Variablen

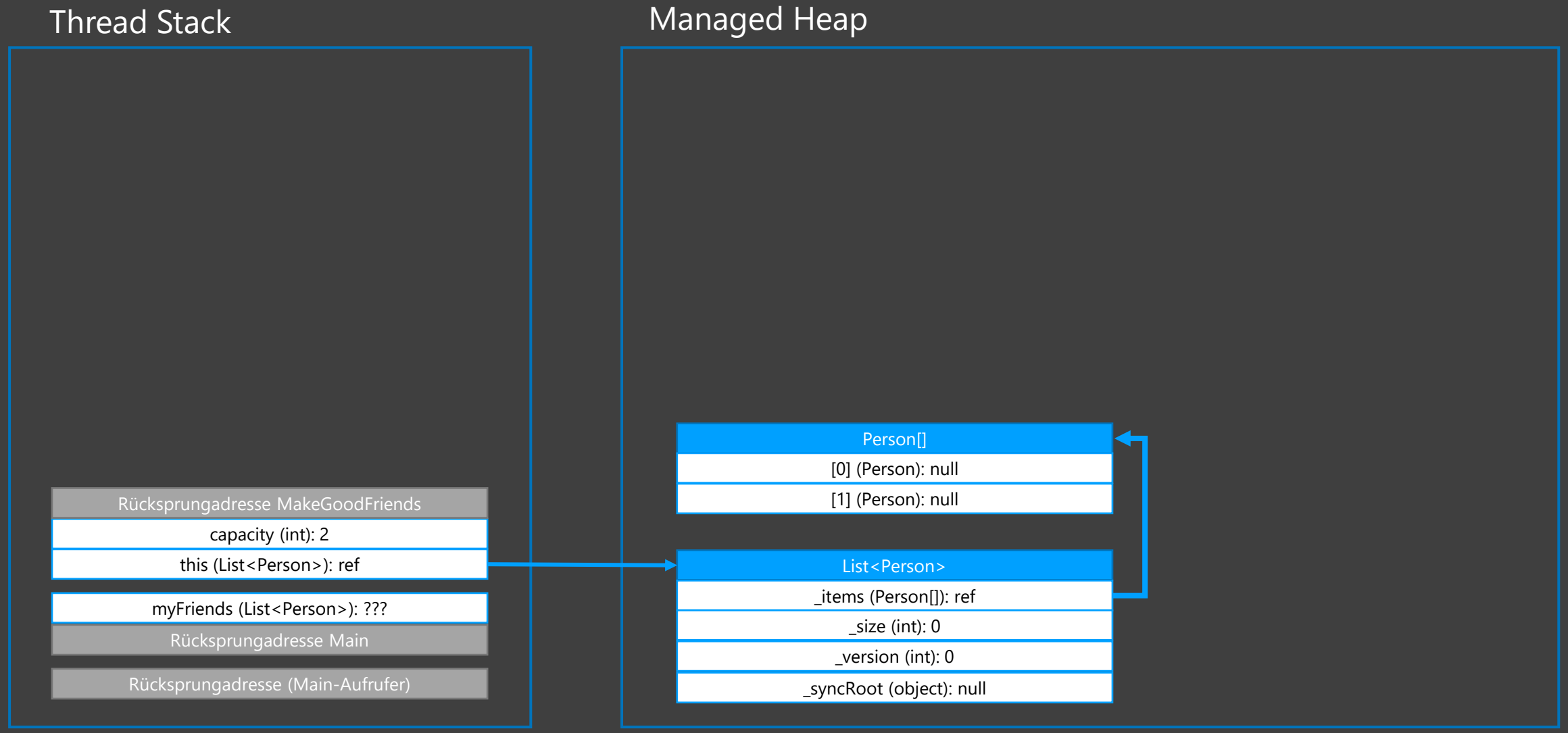


Managed Heap

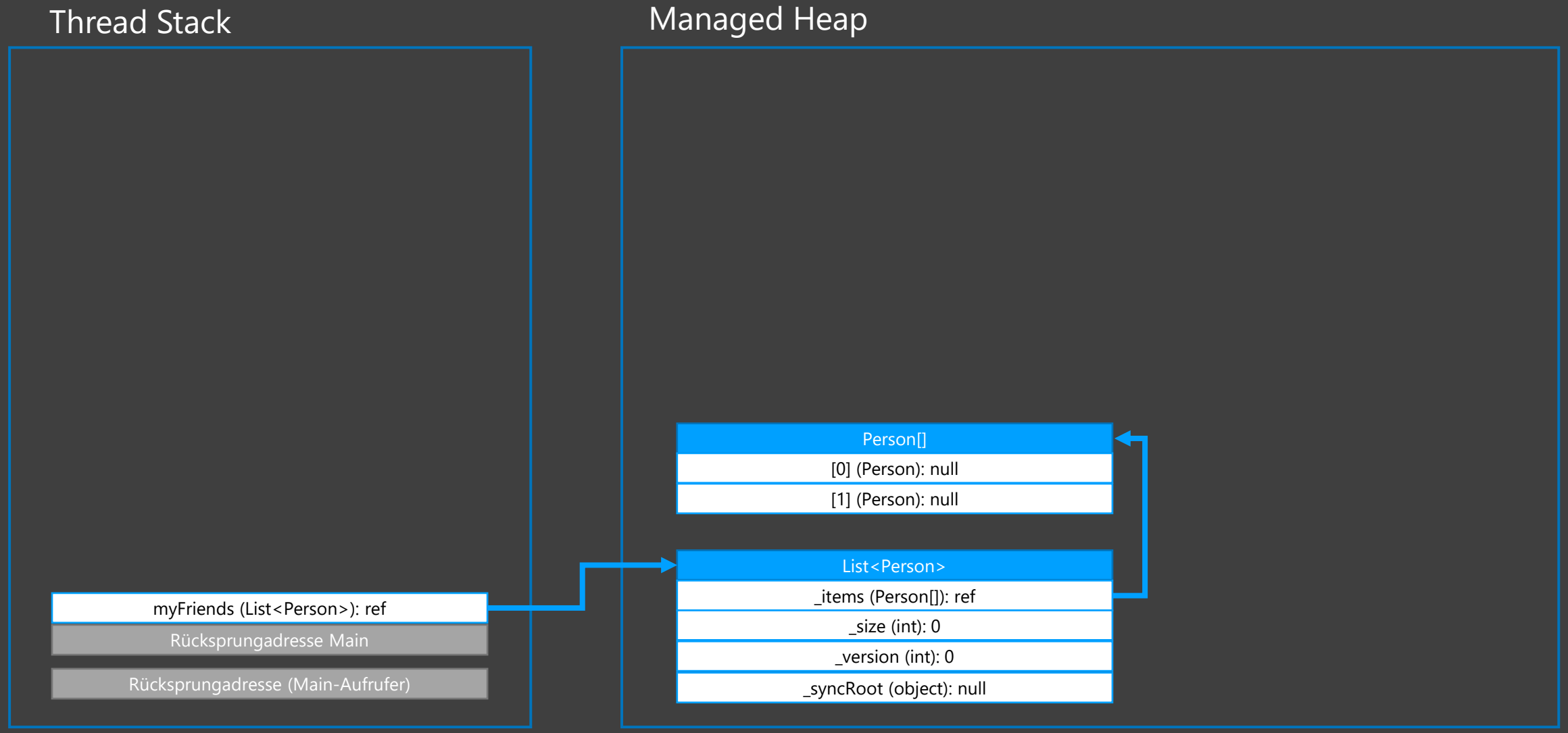
Thread Stacks und Managed Heap am simplen Beispiel (4)



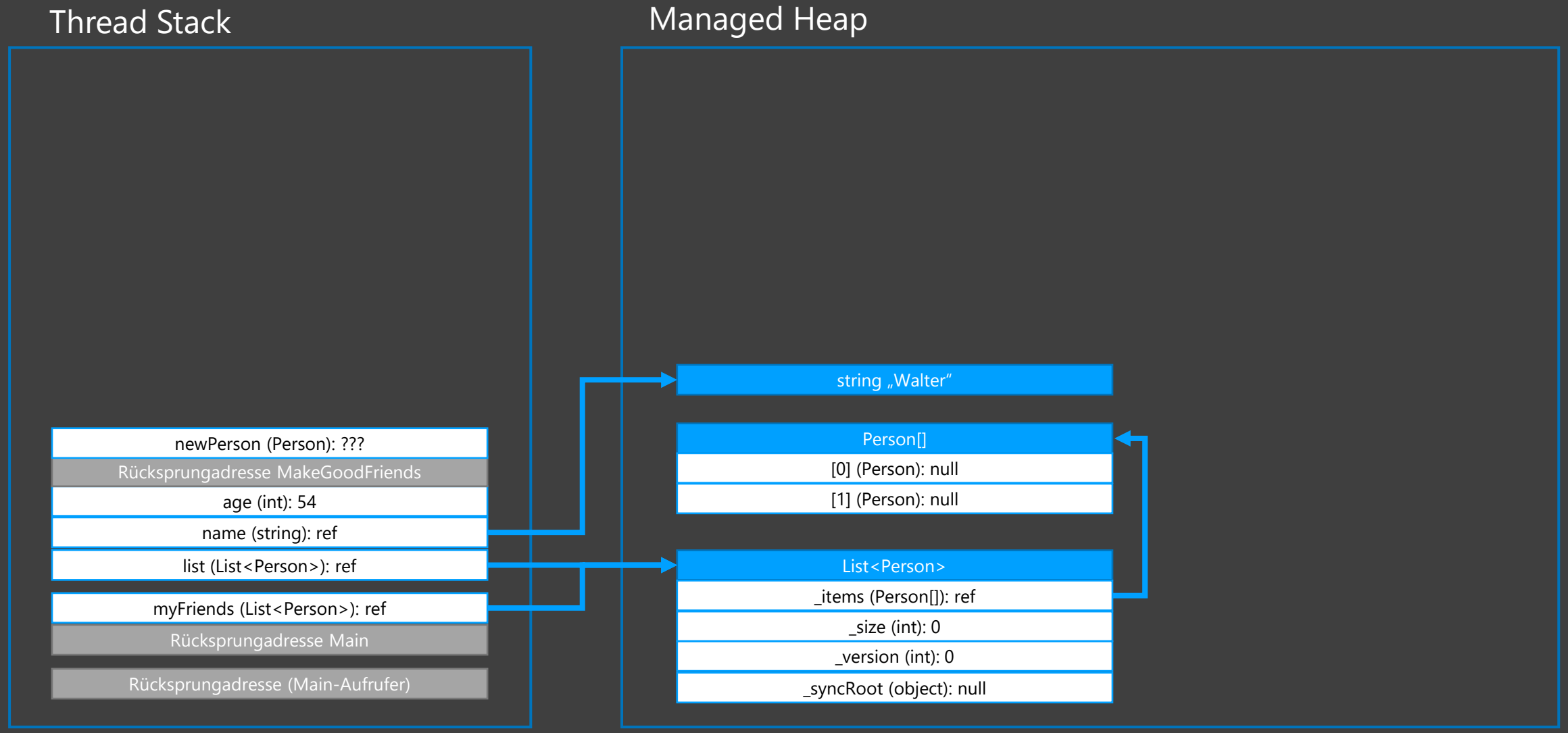
Thread Stacks und Managed Heap am simplen Beispiel (5)



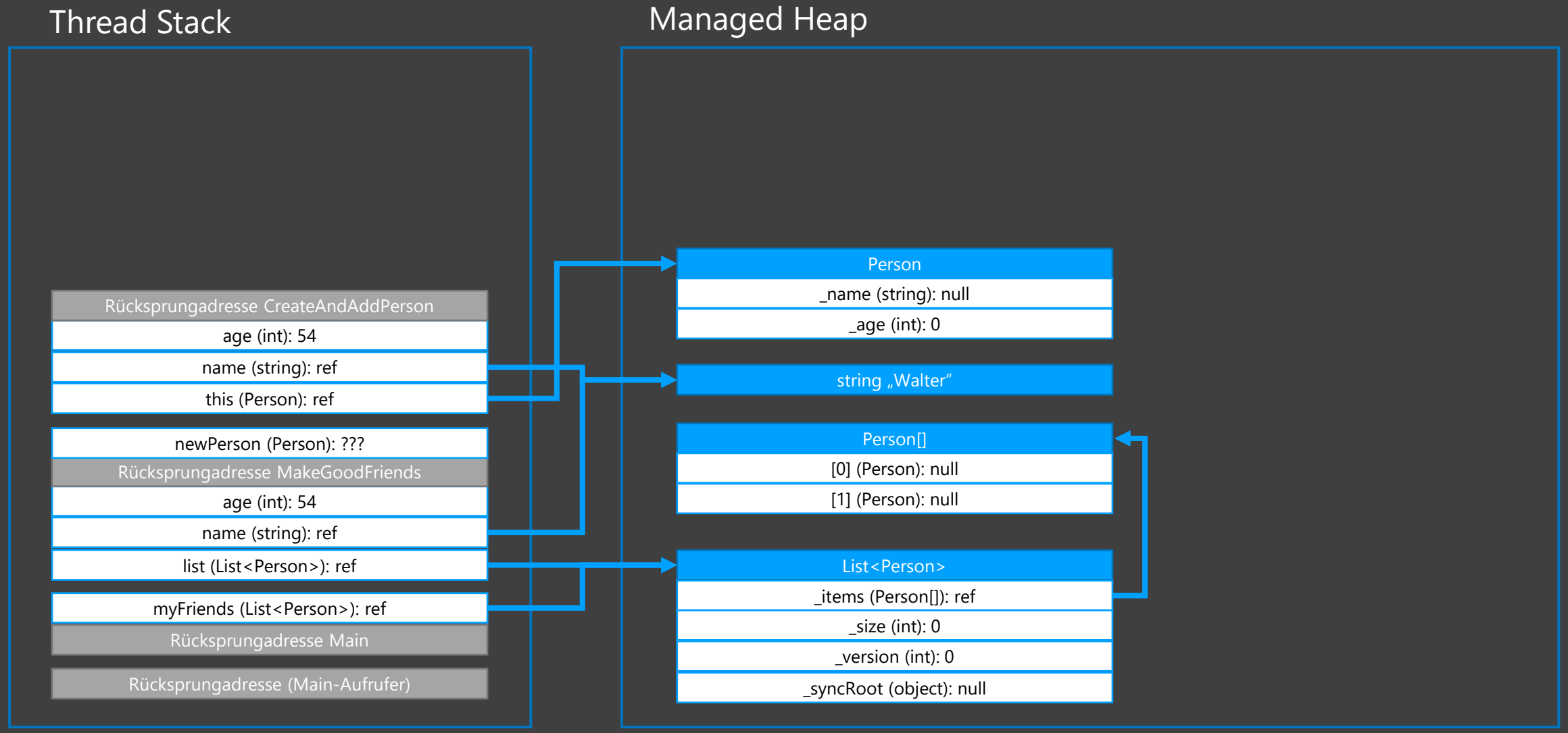
Thread Stacks und Managed Heap am simplen Beispiel (6)



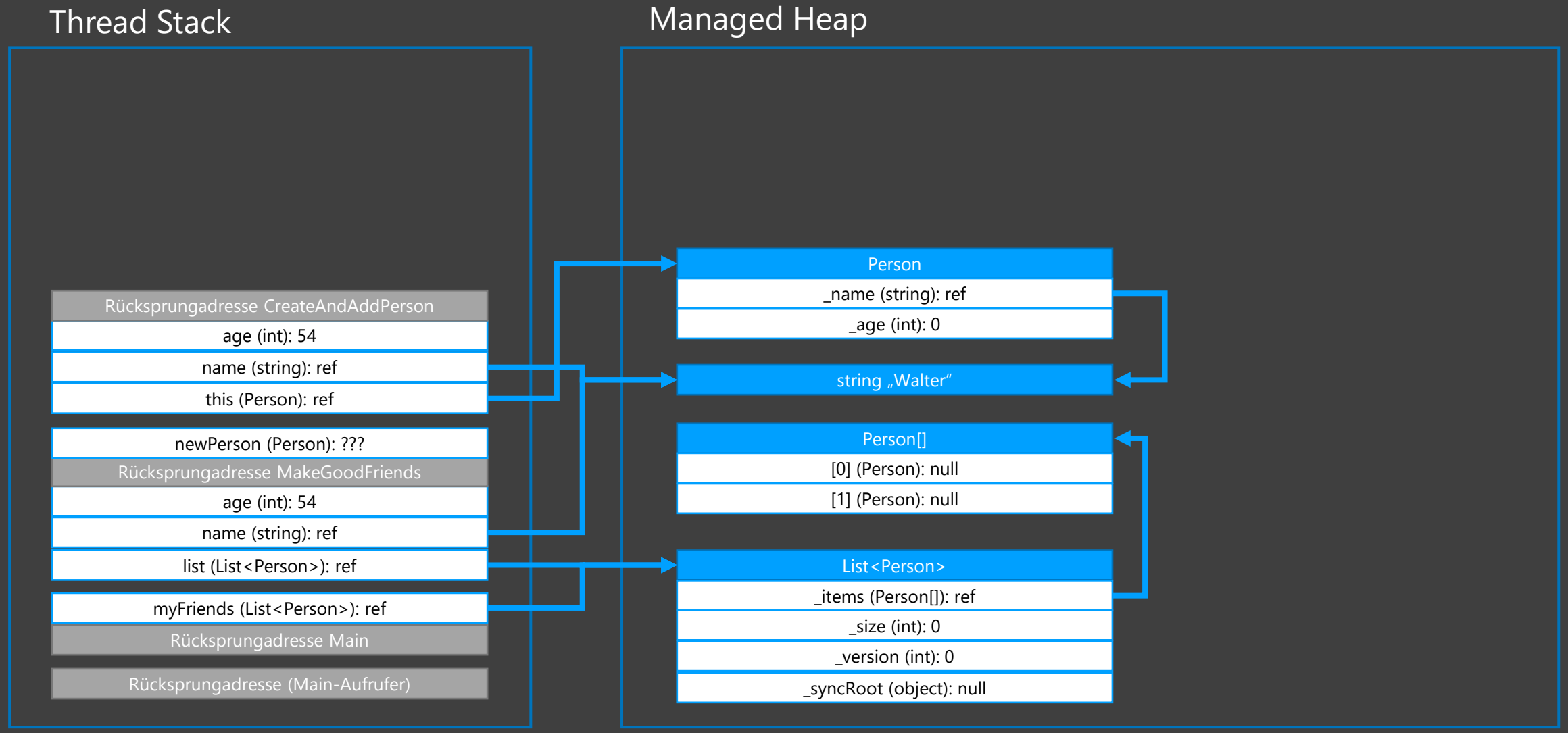
Thread Stacks und Managed Heap am simplen Beispiel (7)



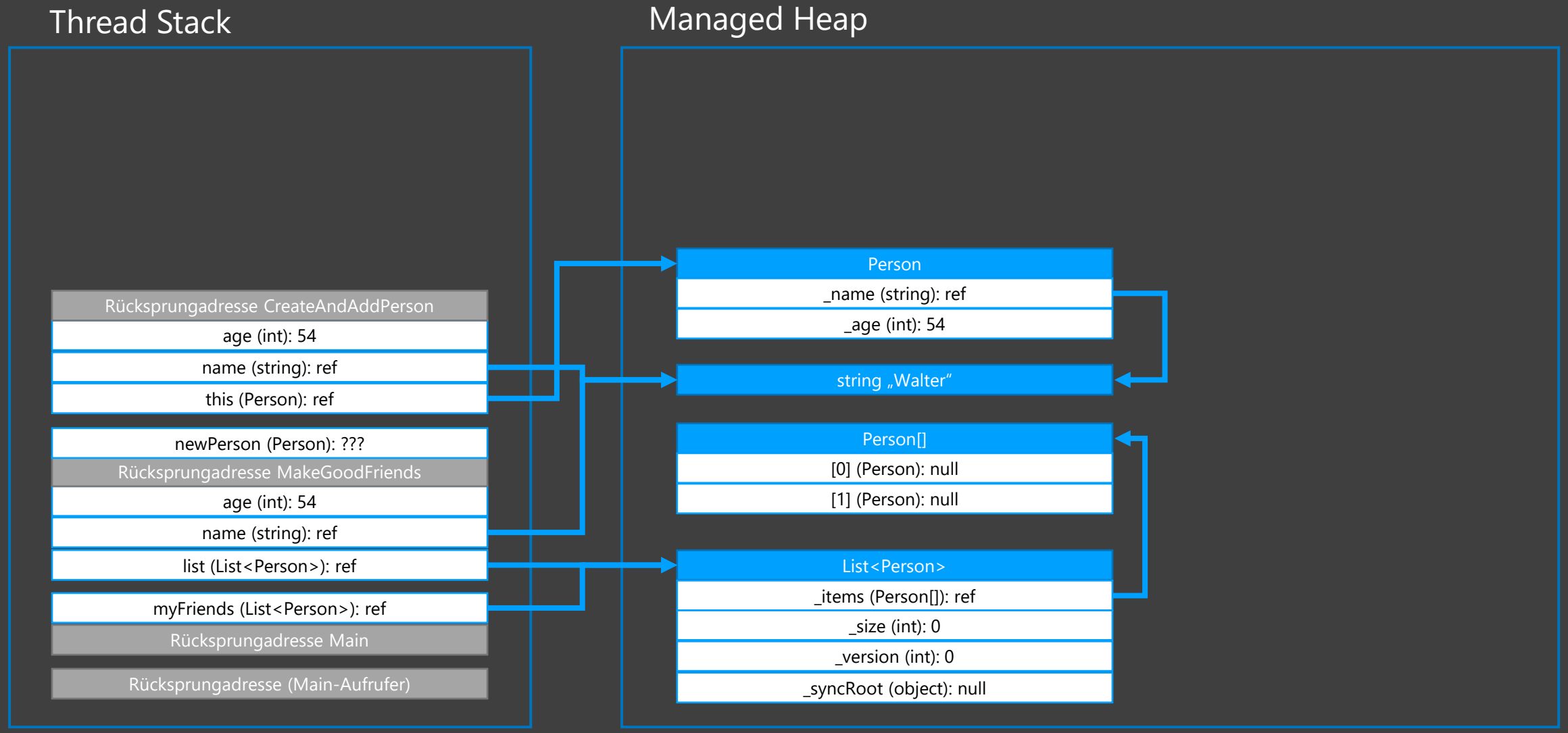
Thread Stacks und Managed Heap am simplen Beispiel (8)



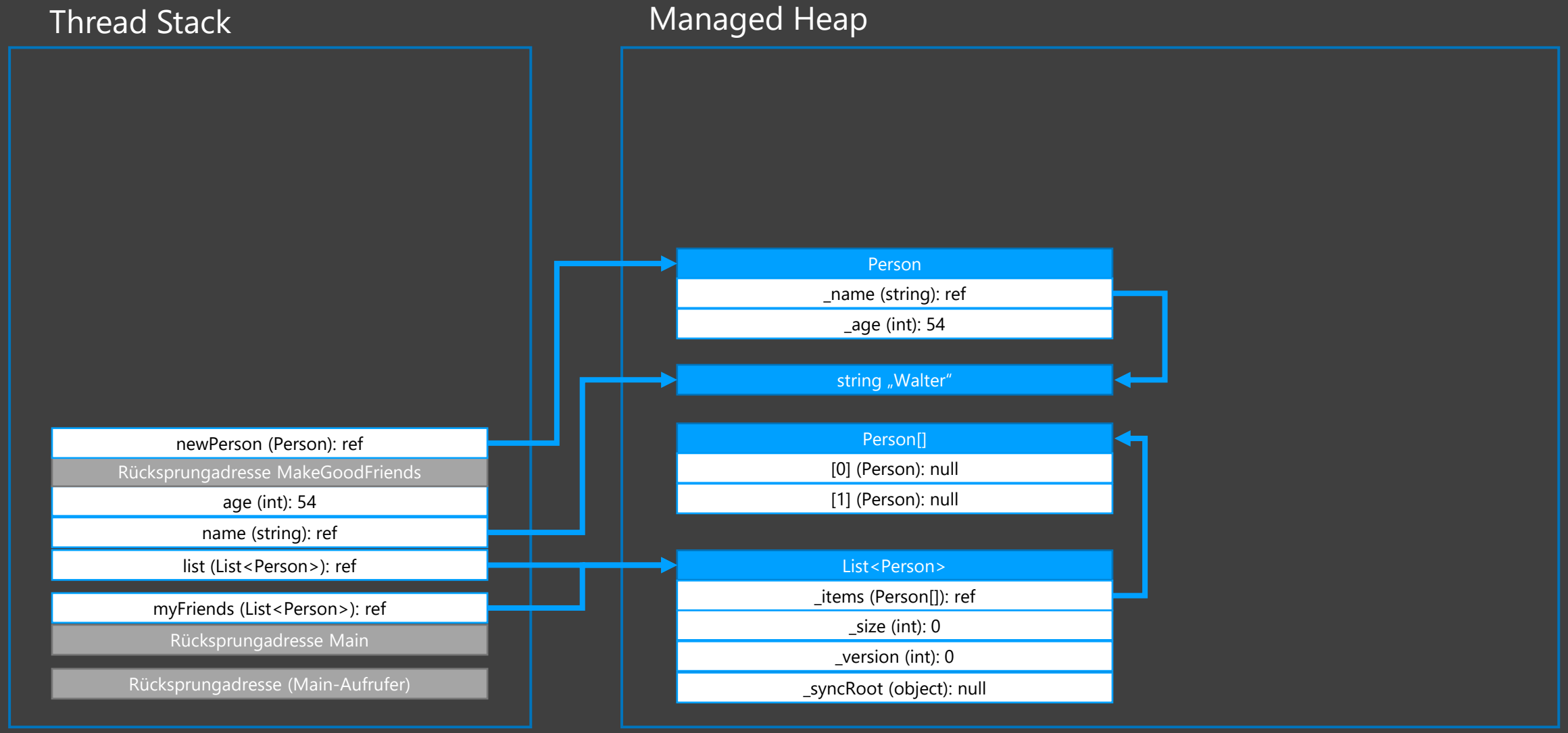
Thread Stacks und Managed Heap am simplen Beispiel (9)



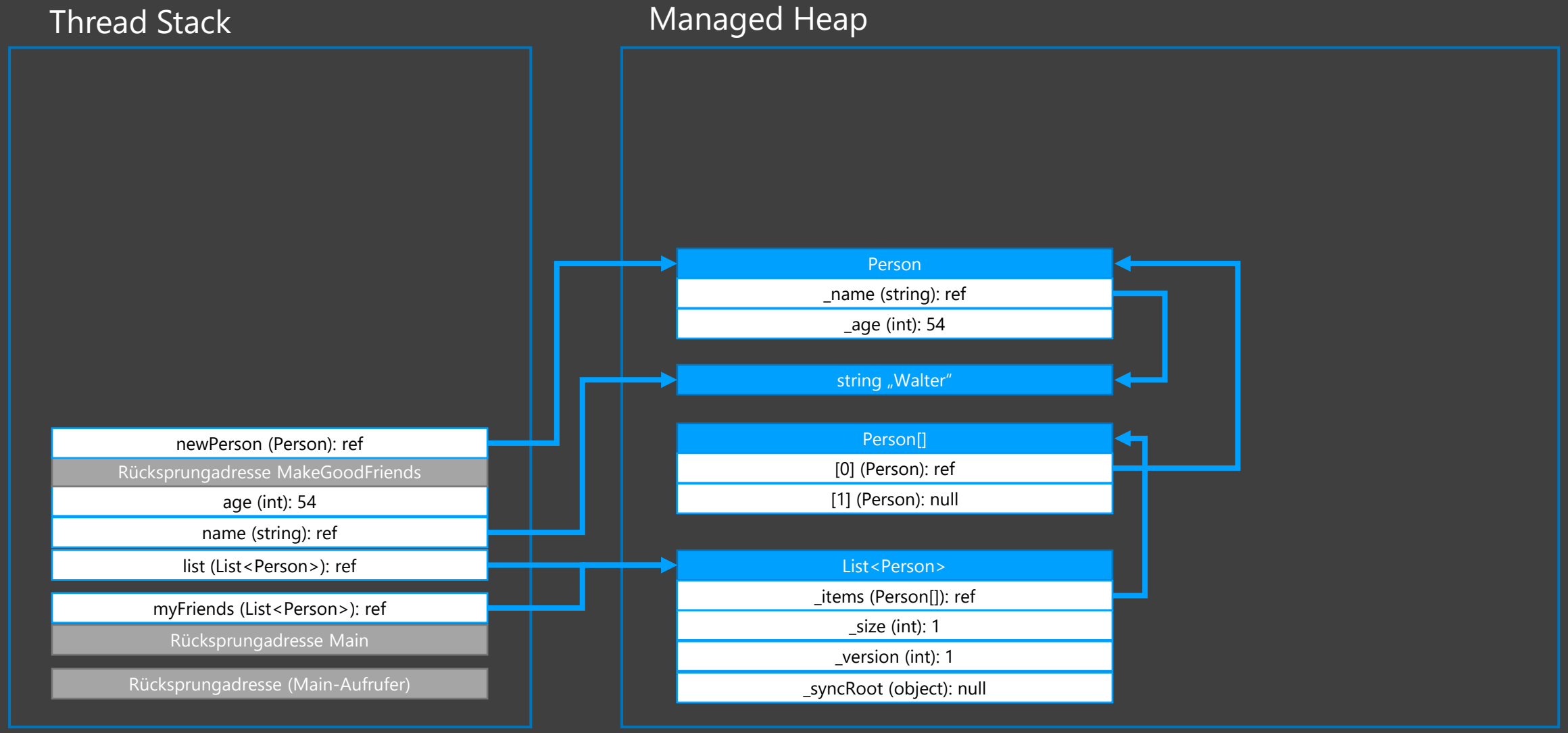
Thread Stacks und Managed Heap am simplen Beispiel (10)



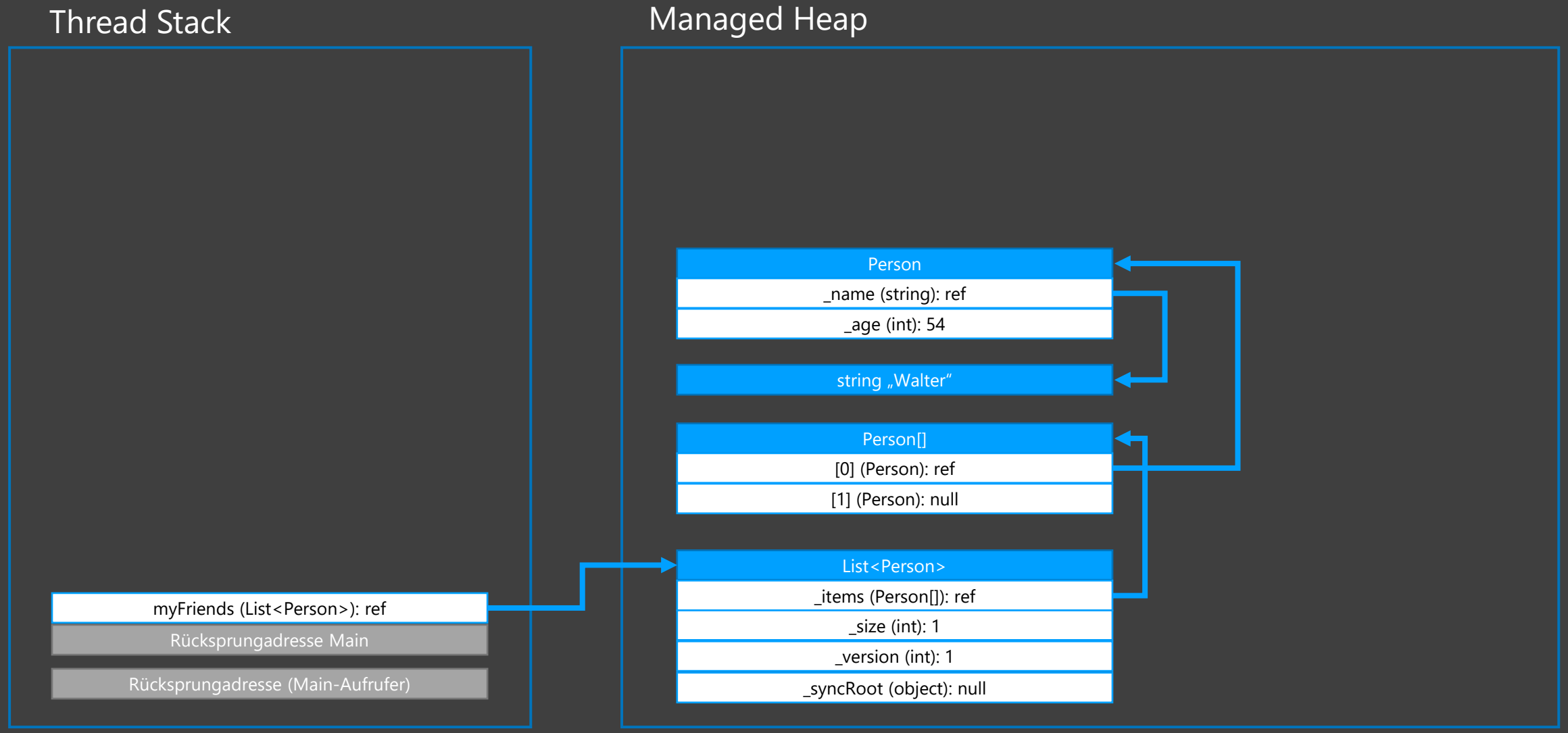
Thread Stacks und Managed Heap am simplen Beispiel (11)



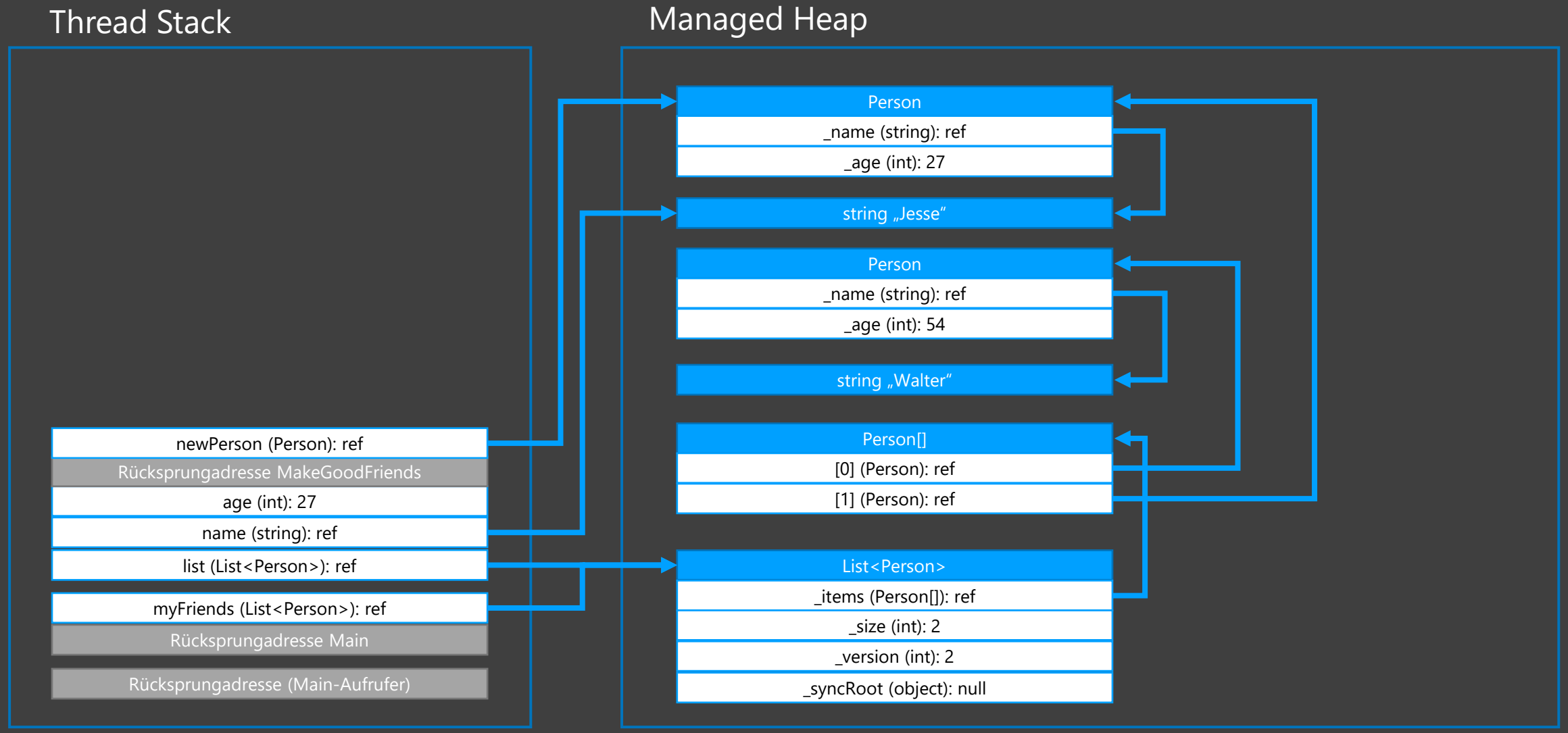
Thread Stacks und Managed Heap am simplen Beispiel (12)



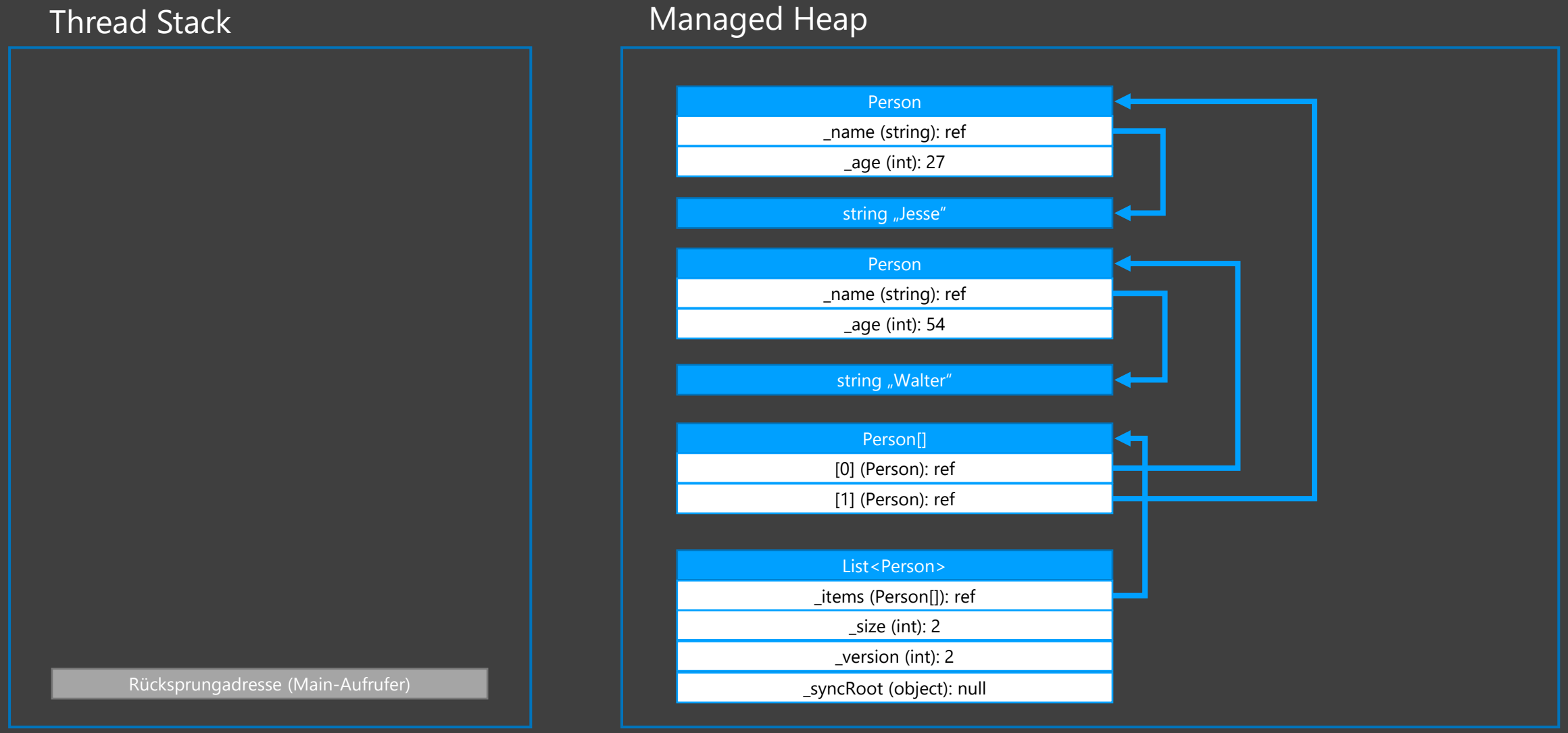
Thread Stacks und Managed Heap am simplen Beispiel (13)



Thread Stacks und Managed Heap am simplen Beispiel (14)



Thread Stacks und Managed Heap am simplen Beispiel (15)



Thread Stacks - Eigenschaften

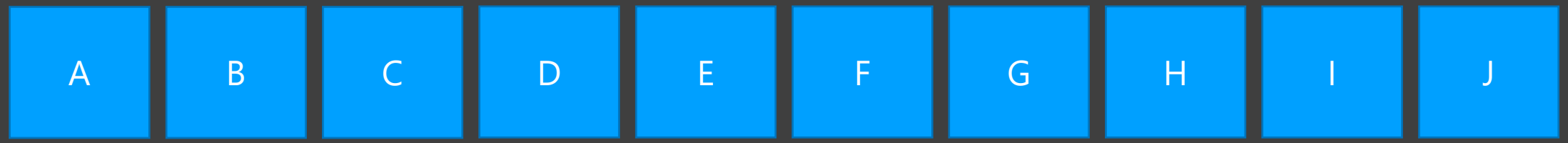
- Jeder Thread hat seinen eigenen Thread Stack. Auf ihnen werden die Daten gehalten, die bei der Ausführung von Methoden benötigt werden.
- Wird eine Methode aufgerufen, wird der sog. Activation Frame (auch Stack Frame) auf den Thread Stack gepusht. Dieser besteht aus:
  - Allen Parametern
  - Rücksprungadresse zum Aufrufer
  - Allen Variablen der Methode (auch die von Sub-Scopes wie foreach-Schleifen)
- Anschließend werden alle Statements der Methode ausgeführt. Dabei können alle Parameter (außer mit in gekennzeichnete) und Variablen von den Statements mutiert werden.
- Endet eine Methode, wird der Activation Frame abgebaut. Dabei werden einfach Pointer verschoben, d.h. die vorherigen Werte bleiben eigentlich auf dem Thread Stack stehen, werden aber von den folgenden Activation Frames überschrieben.

WICHTIG: Thread Stacks != Call Stack

Wie arbeitet der Garbage Collector beim Deallokieren von Memory?

- Der Garbage Collector führt sog. GC Runs aus, um nicht mehr referenzierte Objekte im Managed Heap zu finden, diese zu deallokieren und ggfs. den Speicher zu defragmentieren.
- Er geht dabei in drei oder vier Phasen vor
  - Mark
  - Plan
  - Sweep
  - Compact (optional)
- GC Runs können in verschiedenen Modi ausgeführt werden: Foreground und Concurrent Background. Nur Foreground GC Runs können kompaktieren.
- Der Managed Heap ist in aufgeteilt in einen Small Object Heap (SOH) und einen Large Objekt Heap (LOH). Objekte größer als 85.000 bytes landen im LOH. Im SOH wird weiterhin zwischen 3 Generationen unterteilt.
- Sog. GC Roots bilden die Einstiegspunkte für den GC in der Mark-Phase. Dies sind alle
  - Variablen und Parameter auf allen Thread Stacks
  - Statische Felder
  - Objekte mit Finalizer / Destruktor
  - Pinned Objects

GC Plan Phase



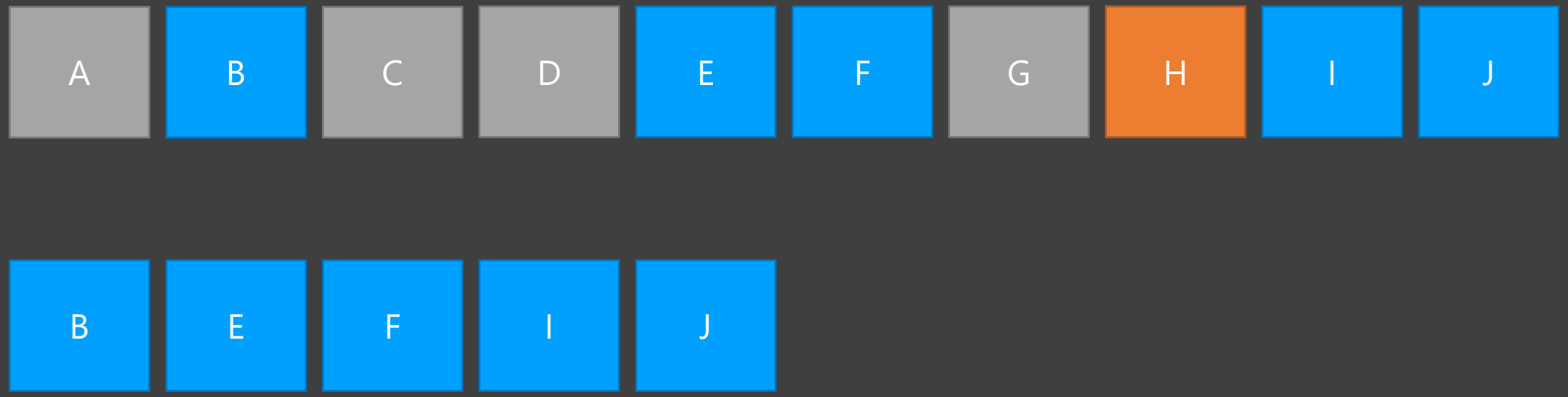
GC Plan Phase

GC Plan Phase

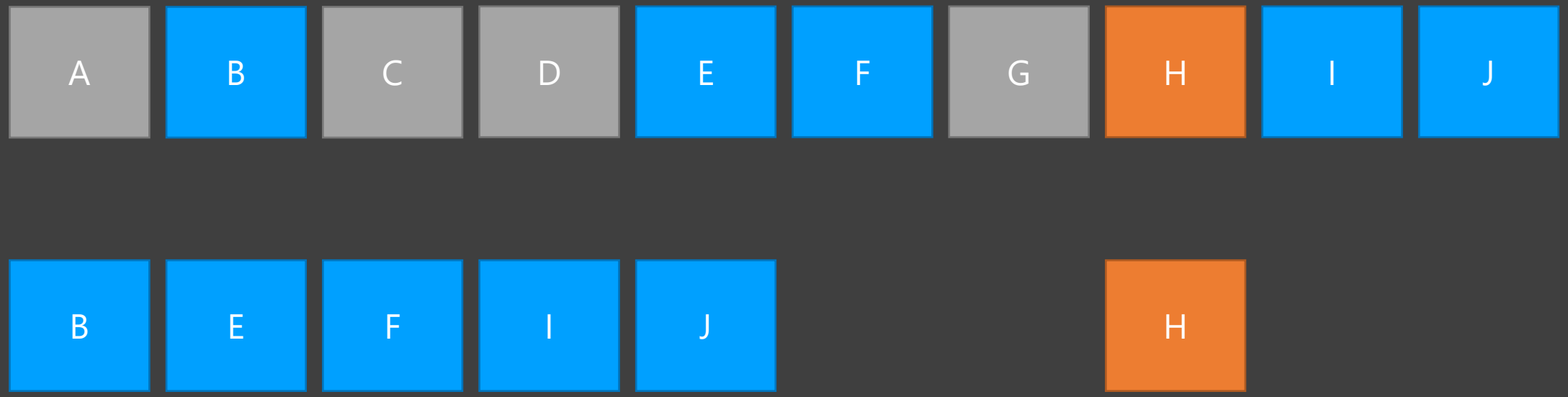
GC Plan Phase

GC Plan Phase

GC Plan Phase



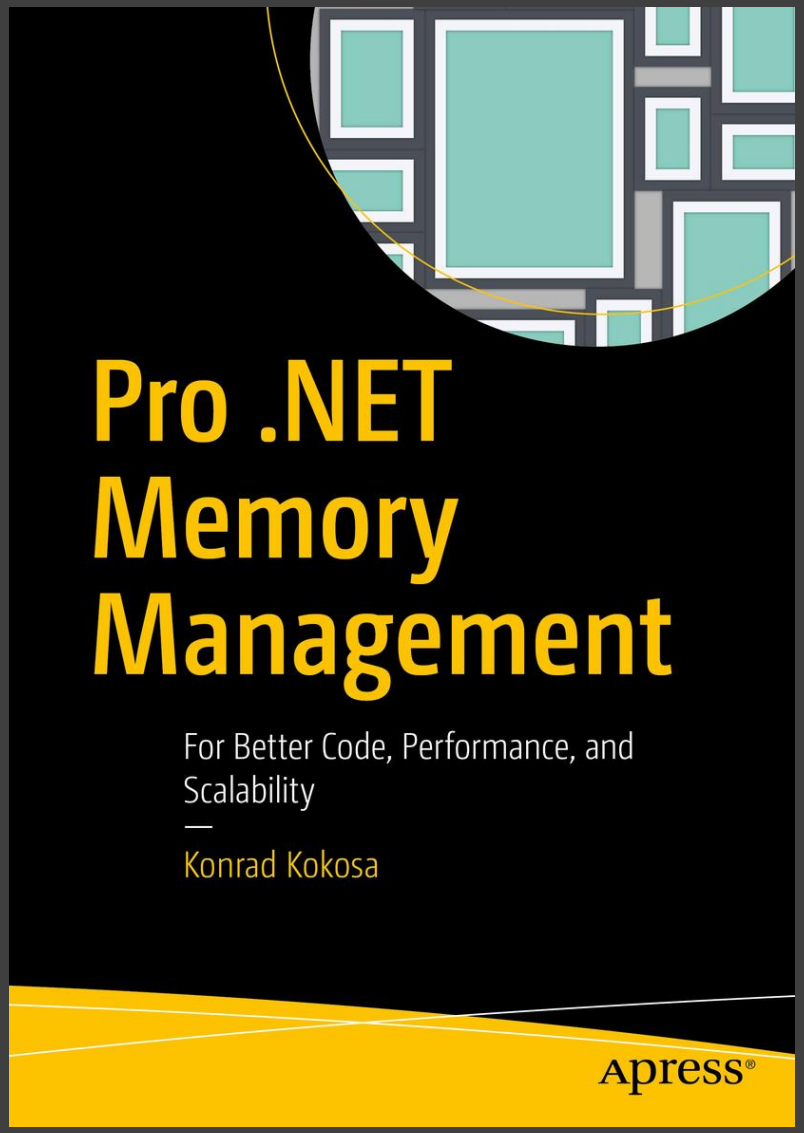
GC Plan Phase



Wie programmieren wir mit und nicht gegen den GC?

- Unnötige Allokationen vermeiden – wer nicht allokiert, löst auch keine GC Runs aus
- Berechnungen, die performancekritisch sind, sollten weitestgehend auf einem / mehreren Thread Stacks durchgeführt werden
- Die Anzahl der Threads möglichst gering halten
- Hidden Allocations vermeiden:
  - Closure / Lexical Scoping bei anonymen Methoden
  - Häufige Stringkonkatenierungen ggfs. durch StringBuilder ersetzen
  - Reflection-basierte Funktionalität reduzieren bzw. vermeiden, diese allokieren auf dem Heap -> in den meisten Fällen ist Code-Generierung besser
    - Data Binding in WPF
    - Unity Container kleiner Version 5
    - JSON.NET
    - Entity Framework (nicht Core)

Pro .NET Memory Management



Kurze Pause

Asynchrones Programmieren in .NET

Was ist Asynchrones Programmieren?

Asynchrones Programmieren bedeutet, dass man an bestimmten Stellen in seinem Source Code Funktionen aufruft, **deren Ergebnis** (Rückgabewerte oder Seiteneffekte) **beim Rücksprung zum Aufrufer noch nicht fertigberechnet** sind. Das Ergebnis wird dem Aufrufer später mitgeteilt (typischerweise über einen Event-Mechanismus). Währenddessen kann der **aufrufende Thread andere Berechnungen durchführen**.

Async Multithreading
(CPU-Bound)

Async I/O
(I/O-Bound)

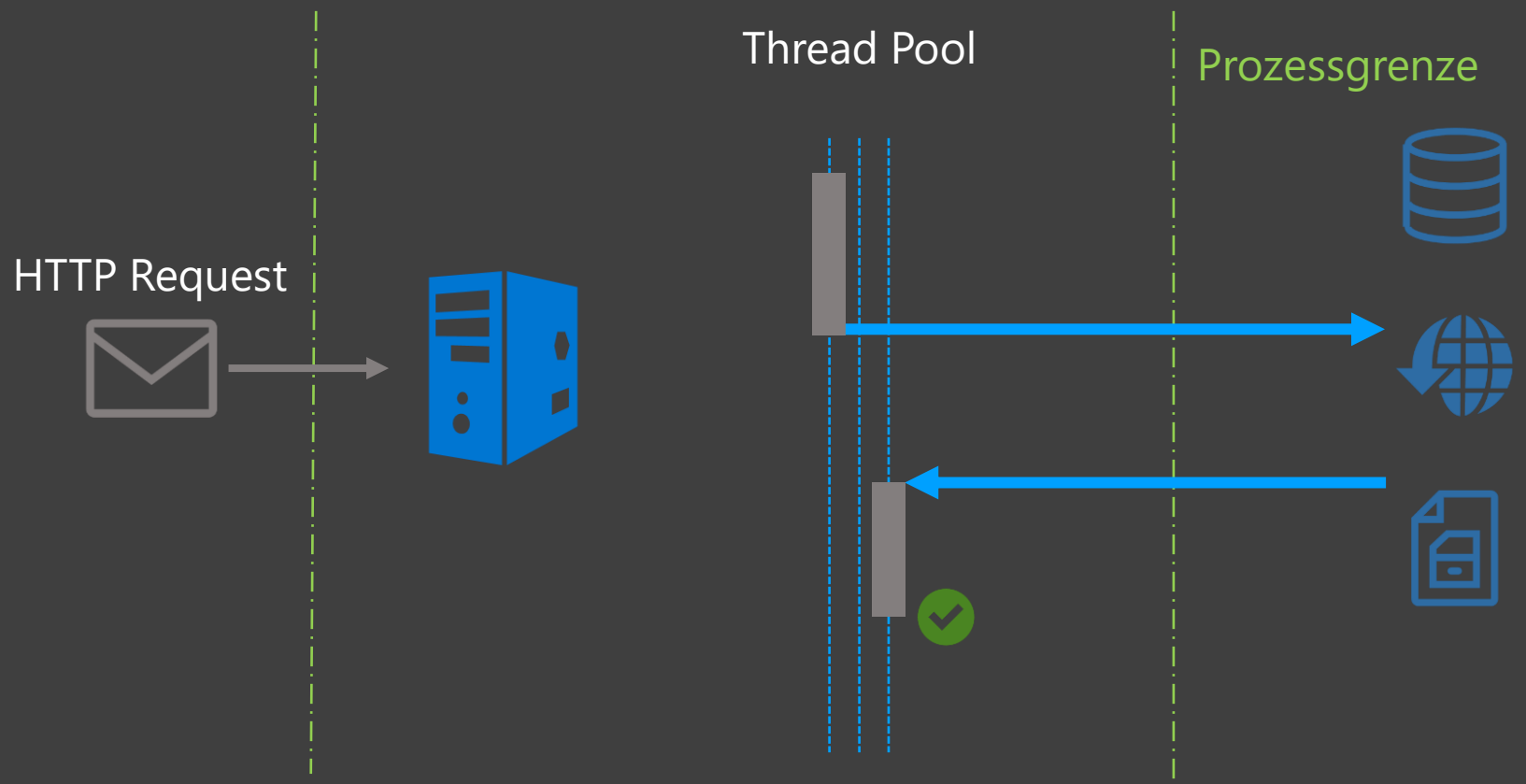
Ein paar Sachen über Threads

- Speicherverbrauch:
  - standardmäßig 1 MB (x86) oder 4 MB (x64) Thread Stack (User Mode)
  - 12 KB (x86) oder 14 KB (x64) Kernel Mode Object
- Context Switching
- Lx Cache Misses

Warum ist Async I/O wichtig?

- Wenn Sie I/O über synchrone APIs ausführen, blockiert der aufrufende Thread, bis das Ergebnis da ist.
- Wenn der Thread Pool blockierte Threads sieht, erzeugt er neue
- Threads sind teuer

Threading in Services in .NET



Overhead von async await

- Der genaue Overhead ist schwierig zu bestimmen.
- Wenn man eine Methode async macht und diese tatsächlich Async Compute oder Async I/O ausführt, ist man aber mindestens im Bereich µs.

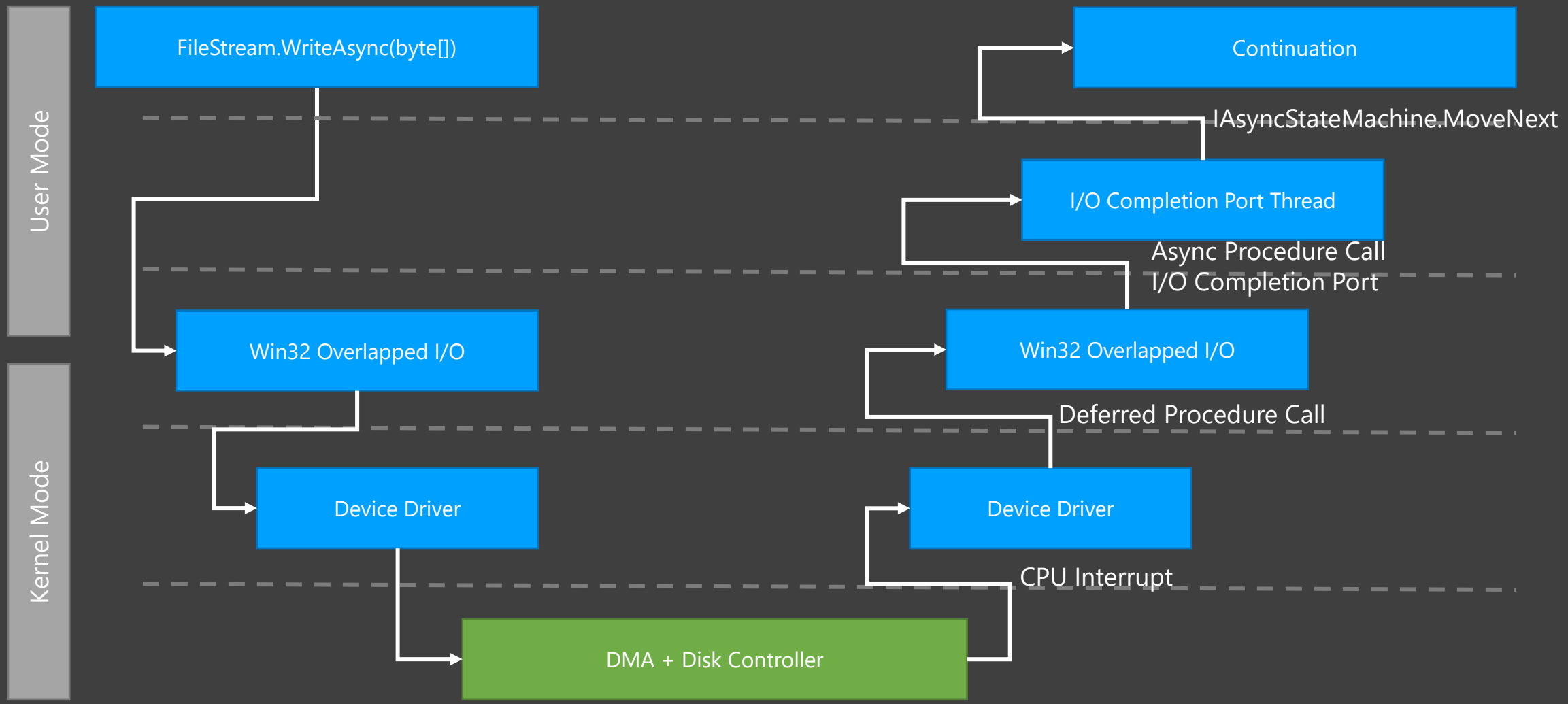
Method	Mean	Error	StdDev	Ratio	RatioSD	Gen 0	Gen 1	Gen 2	Allocated
Increment	0.2666 ns	0.0204 ns	0.0190 ns	1.00	0.00	-	-	-	-
IncrementAsync	1,181.3835 ns	18.4261 ns	20.4805 ns	4,465.72	356.49	0.0534	-	-	256 B

Erkenntnisse

- Der Compiler sorgt bei async await dafür, dass eine (synchron erscheinende) Methode in eine State Machine umgeformt wird.
- Diese State Machine kehrt zum Aufrufer zurück, wenn auf eine asynchrone Operation gewartet wieder.
- Feste Komponenten des .NET Frameworks sorgen dafür, dass die State Machine erneut angestoßen wird, wenn ein Task abgeschlossen ist.
- Hat der anstoßende Thread einen Synchronization Context, wird eine Continuation standardmäßig auf diesem Thread wieder eingereiht.
- async await hat Overhead. Überlegen Sie genau, welche Methoden async sein müssen.

Das übergeordnete Ziel: Kehre zum Aufrufer zurück, damit Threads nicht blockiert werden (besonders wichtig für async I/O in Services).

Async I/O am Beispiel Dateischreiben



Auswirkungen auf Software-Design

Achtung: subjektiv

Was wir aus den vorherigen Abschnitten lernen sollten

- Eindeutig unterscheiden zwischen I/O und In-Memory Operationen
- I/O sollte asynchron ausgeführt werden, um UI Freezes und unnötige Thread-Allokationen zu vermeiden
- Unnötigen I/O vermeiden
  - Mehrere Abfragen in eine zusammenfassen, falls möglich
  - Fail-Fast-Prinzip
- Unnötige Objektallokationen vermeiden – Indirektion nur dann einsetzen, falls notwendig
- Auch stark gekoppelter Code kann (leicht) automatisiert getestet werden, sofern dieser ausschließlich In-Memory läuft
- Speicherabbilder helfen beim Design von Mengengerüsten

Über die SOLID Prinzipien: DIP und OCP

Dependency Inversion Principle

- High-level modules should not depend upon low-level modules. Both should depend upon abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

Open / Closed Principle

Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

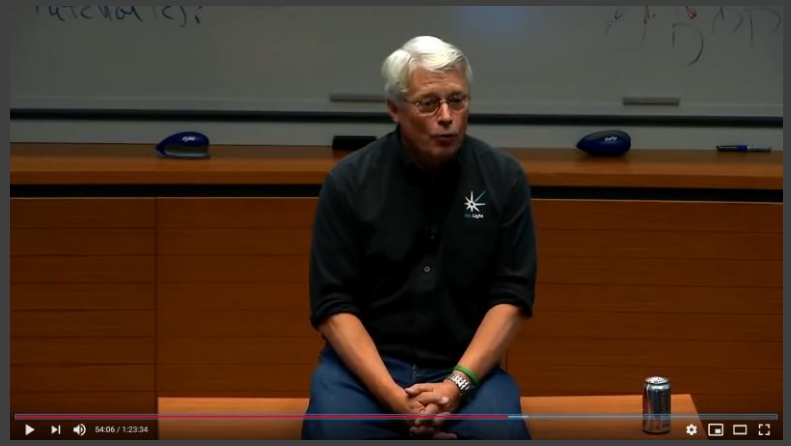
Wo ist hier der Unterschied?

Über die SOLID Prinzipien: SRP

Single Responsibility Principle:

A class should only have one reason to change

Sollte besser Single Problem Area Principle heißen.



Neues Programmierprinzip: LIT

Learn-The-Internals Principle (LIT)

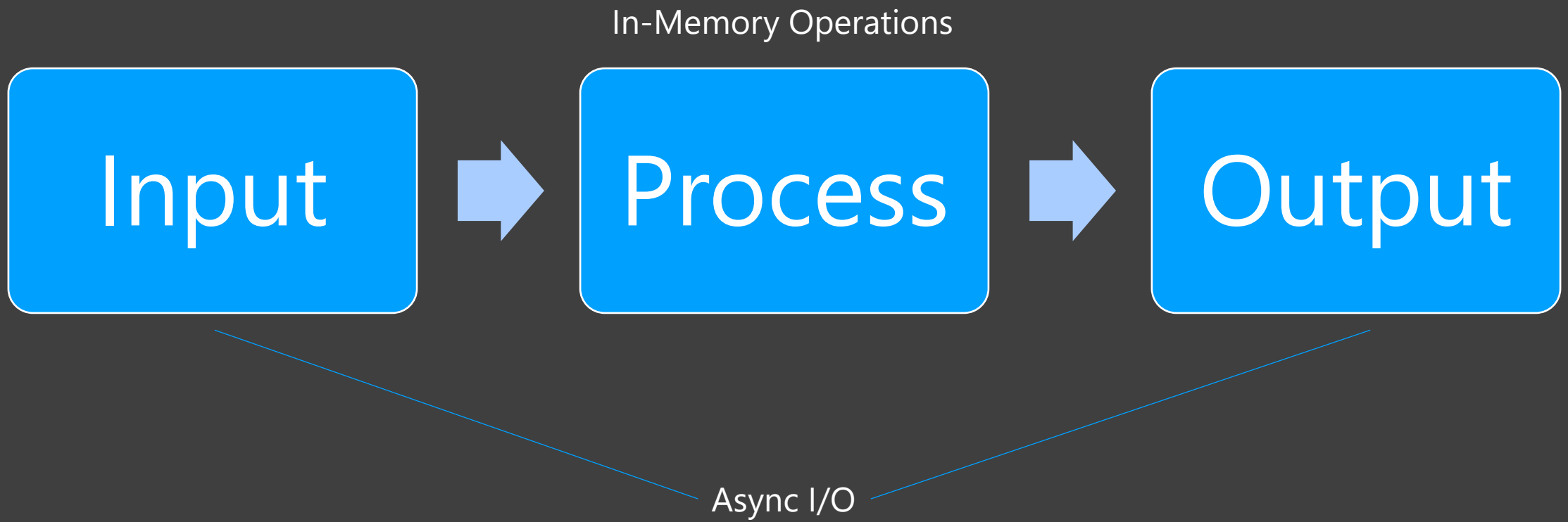
Setze dich mit den internen Mechanismen der Runtimes und Frameworks / Bibliotheken, die du einsetzt, auseinander und verstehe, wie sie wiederkehrende Probleme lösen (zumindest auf der obersten Abstraktionsschicht). Mache ausfindig, welche Aufruf-Muster Probleme erzeugen können und stelle sicher, dass diese im aufrufenden Code vermieden werden.

Neues Programmierprinzip: RPB

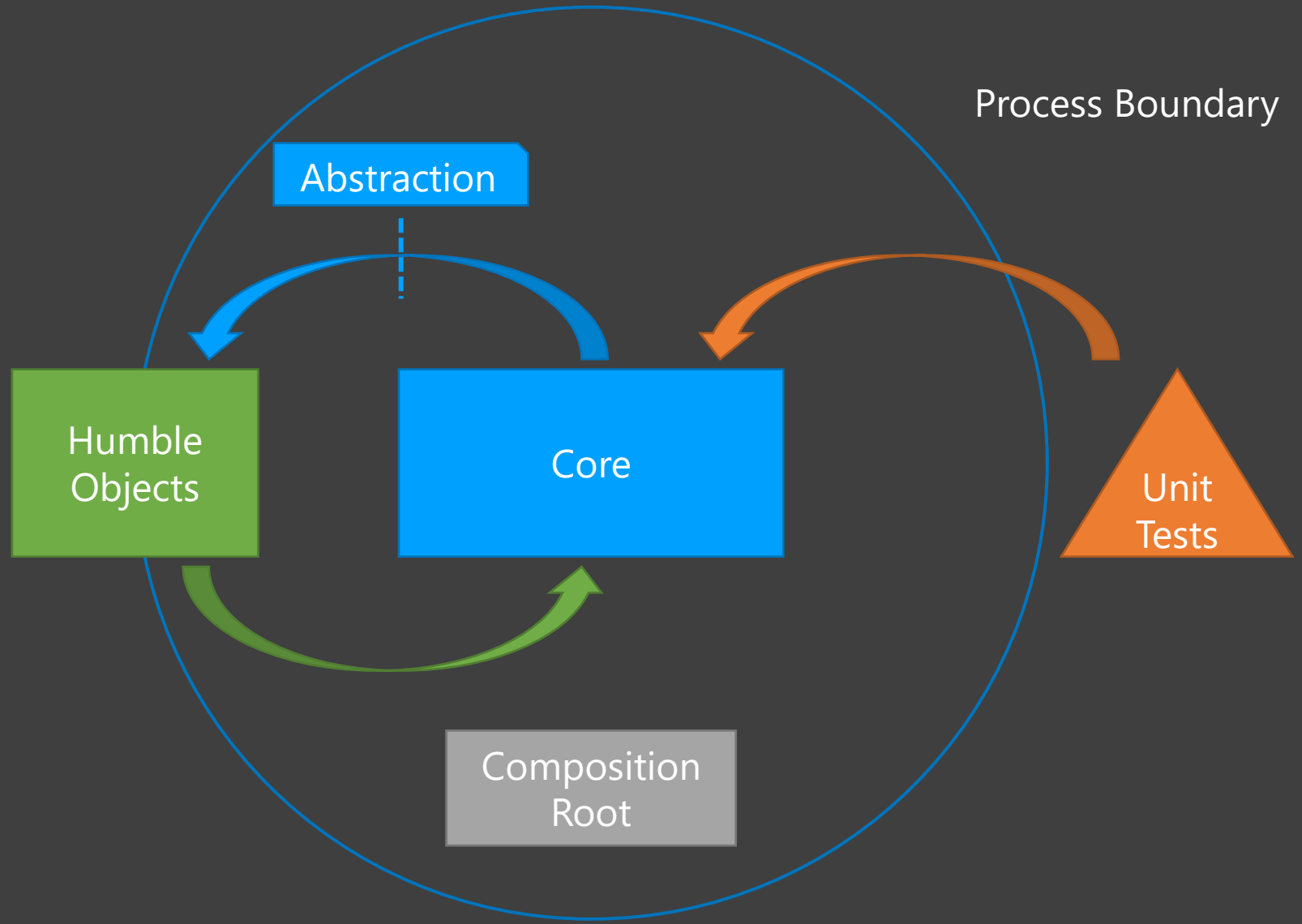
Respect-the-Process-Boundary Principle (RBP)

Unterscheide zwischen In-Memory Operationen und I/O Operationen, da letztere deutlich höhere Ausführungszeiten haben. I/O Operationen sollten über asynchrone APIs umgesetzt werden, um blockierende Threads zu vermeiden.

Eingabe – Verarbeitung – Ausgabe



CHUC: Core – Humble Objects – Unit Tests – Composition Root



Quellen

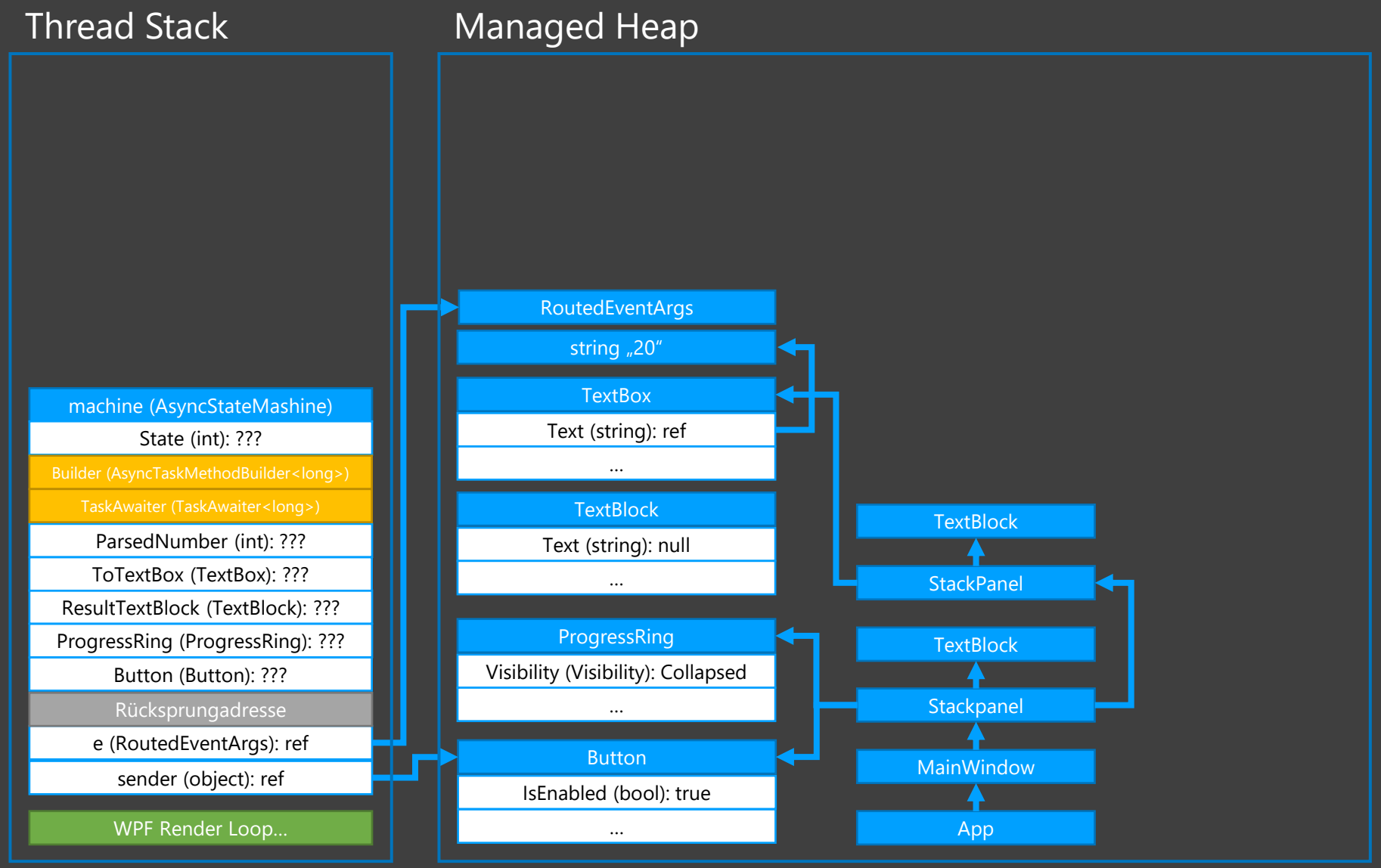
- Matt Waren: The 68 things the CLR does before executing a single line of your code (*)
- Konrad Kokosa: Pro .NET Memory Management
- Jeffrey Richter: Advanced Threading in .NET
- Daniel Palme: IoC Container Benchmark - Performance comparison
- Joseph Albahari: Threading in C#
- John Skeet: Asynchronous C# 5.0
- Robert C. Martin: Agile Principles, Patterns, and Practices in C#
- Andrey Akinshin: Pro .NET Benchmarking – The Art of Performance Measurement
- Martin Fowler: Patterns of Enterprise Application Architecture
- Mark Seemann: Dependency Injection in .NET

Vielen Dank!

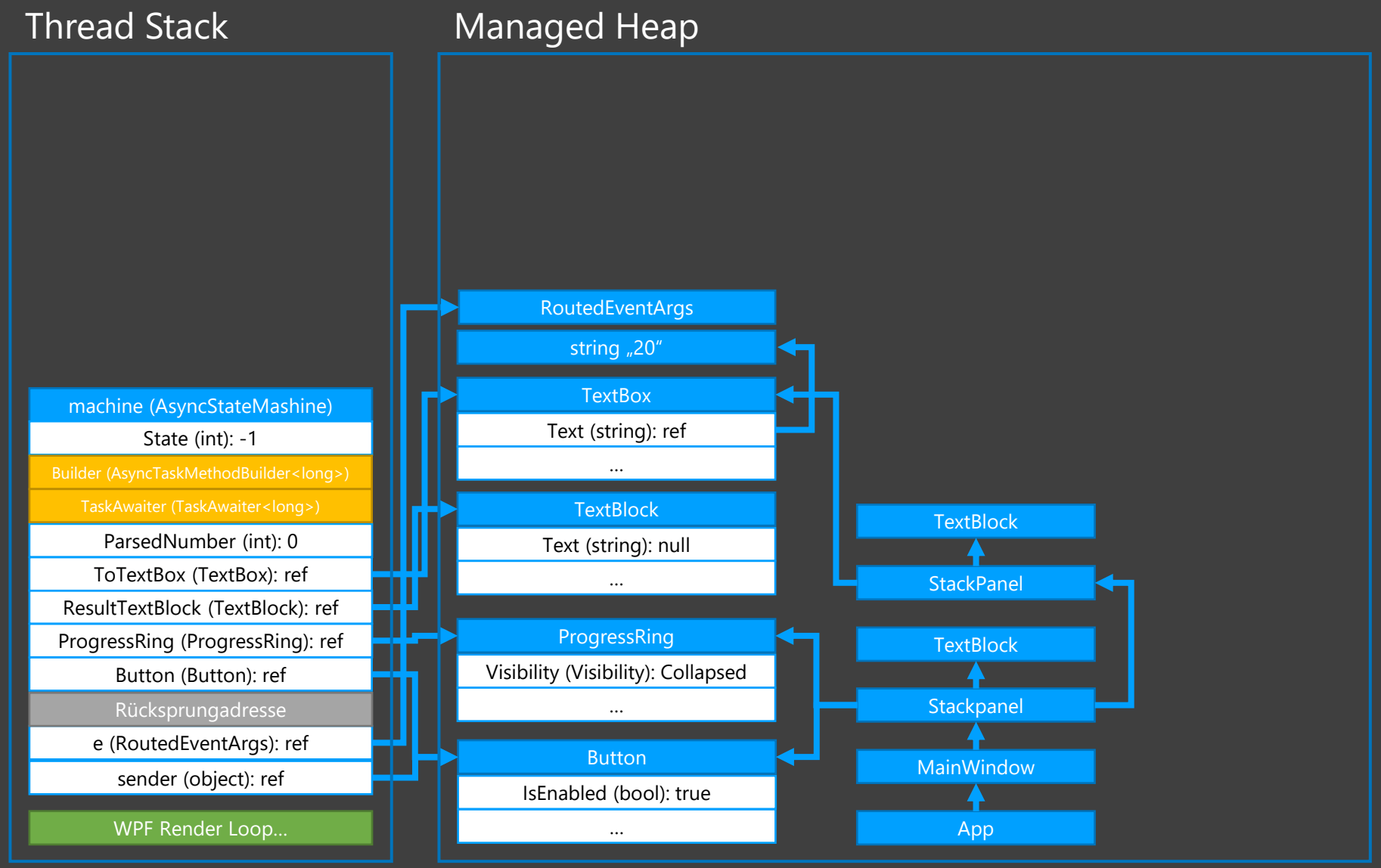
Haben Sie Fragen?

Speicherabbild einer async Methode

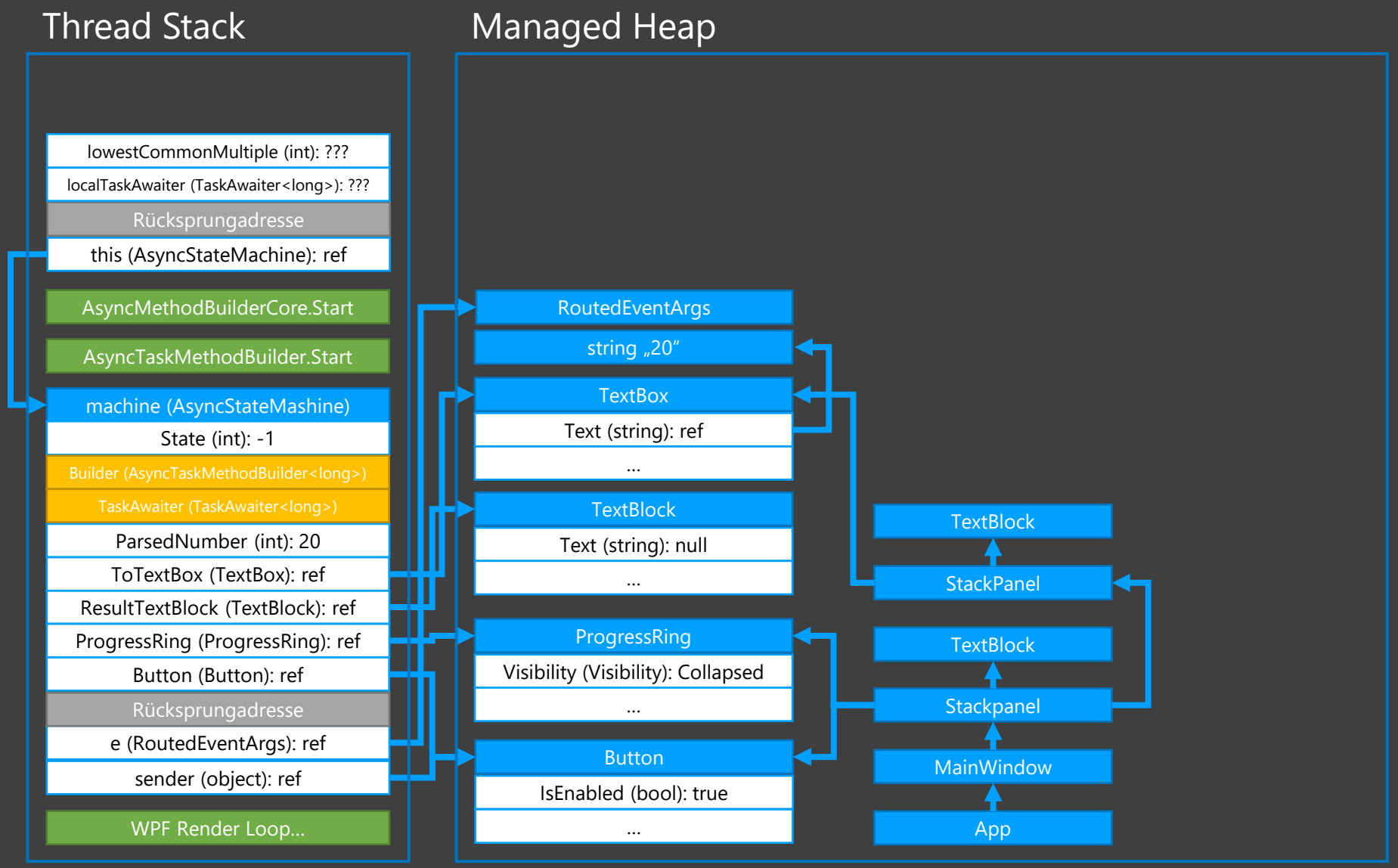
Speicherabbild async: am Start der async-Methode



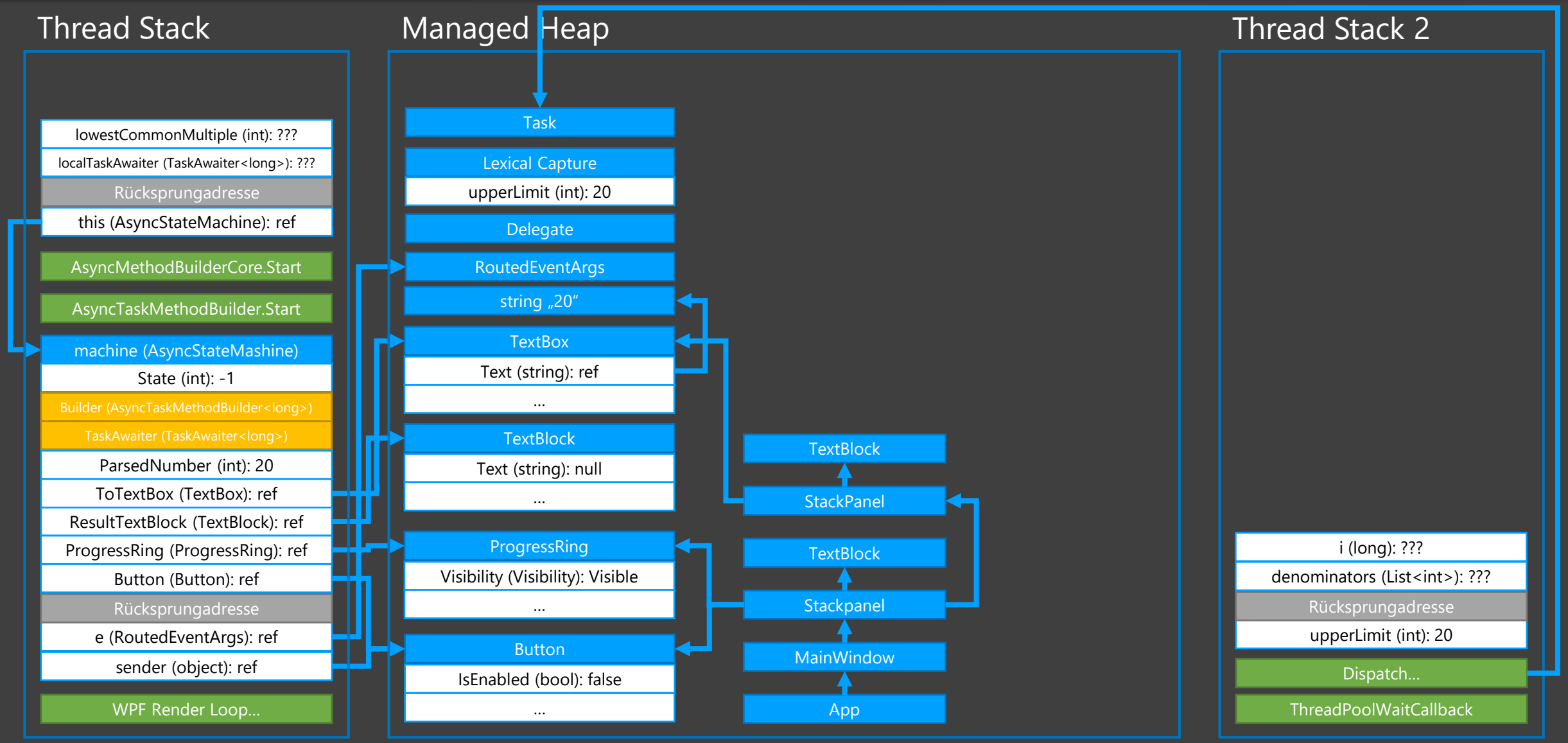
Speicherabbild async – vor AsyncTaskMethodBuilder.Start



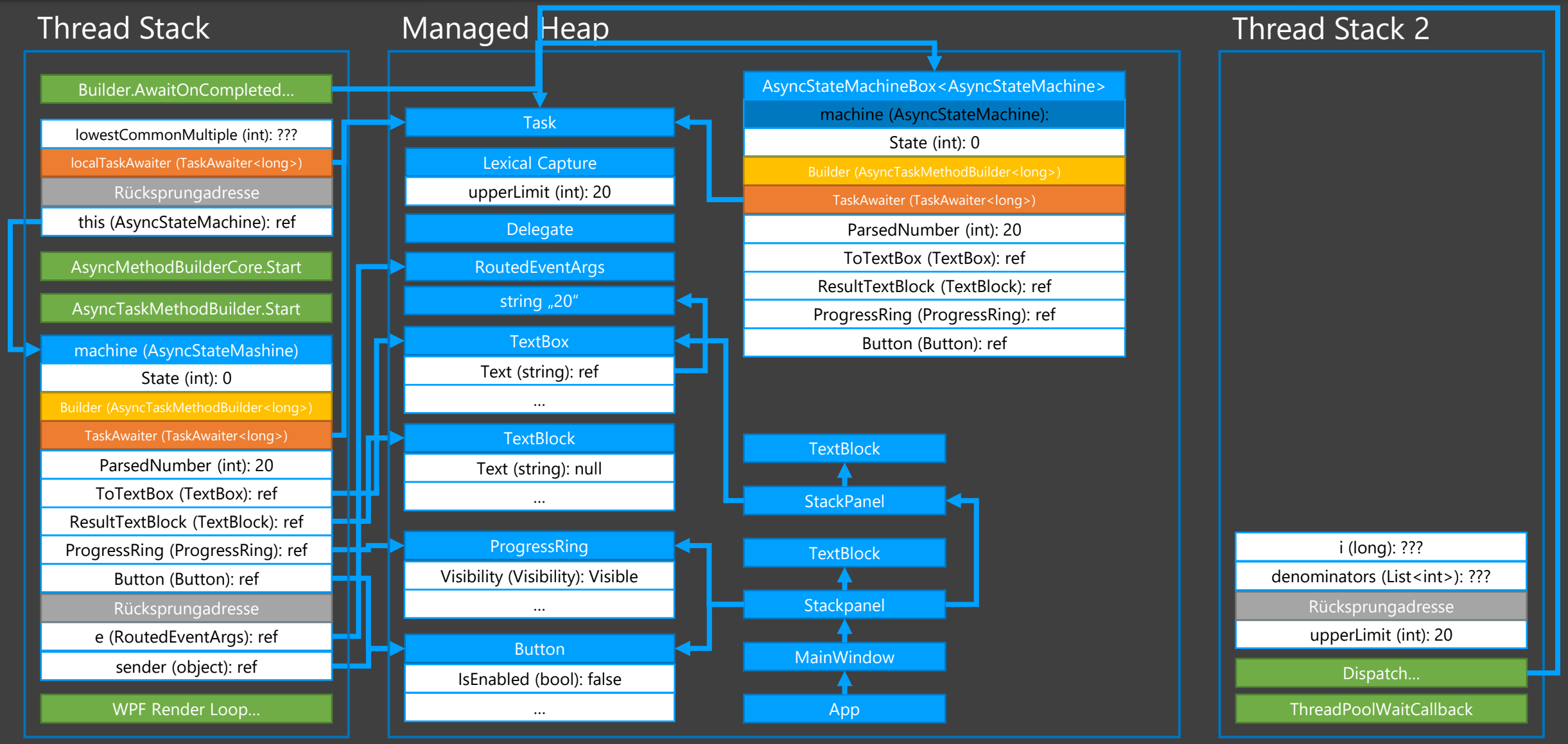
Speicherabbild async – Beim ersten Eintreten in MoveNext



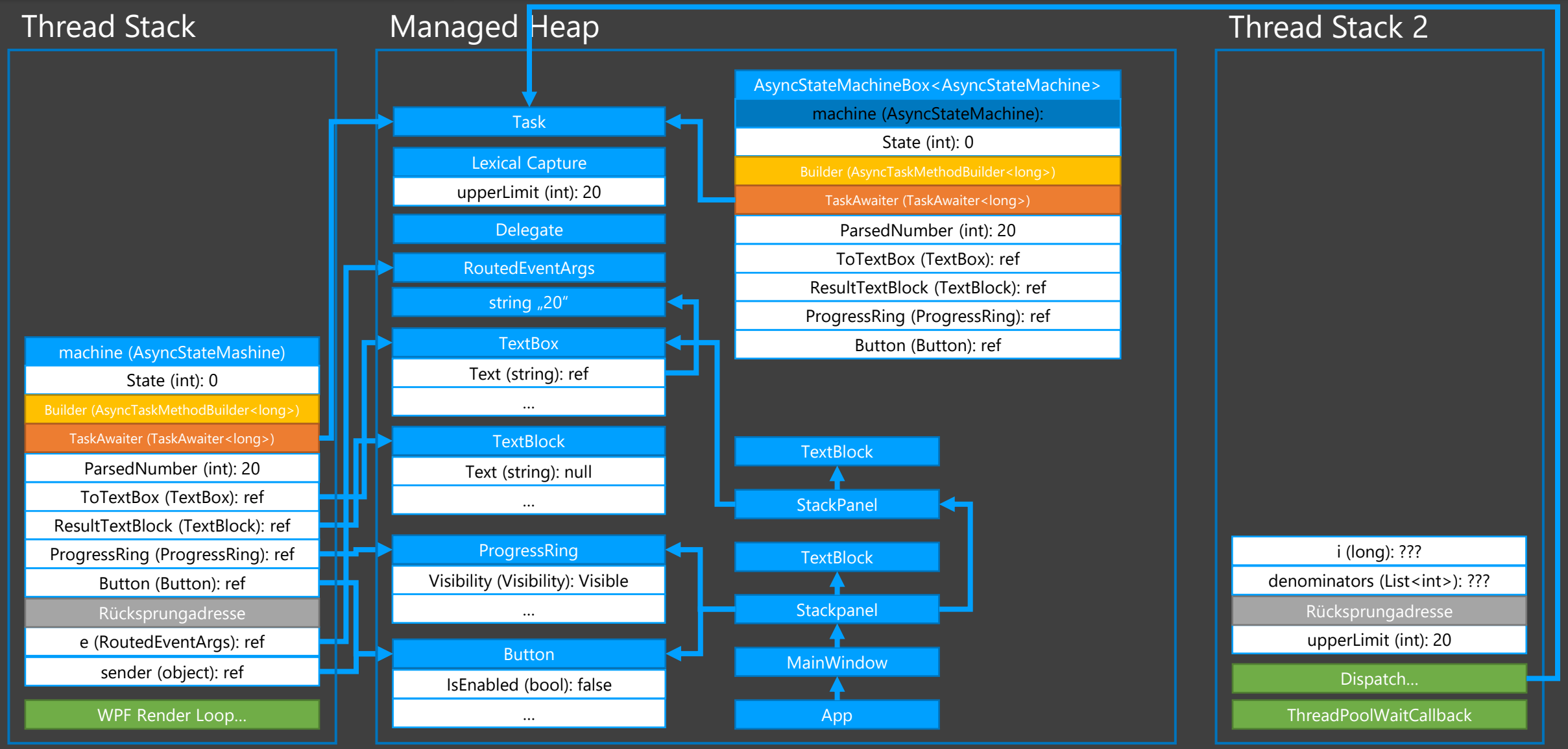
Speicherabbild async – nach Task.Run



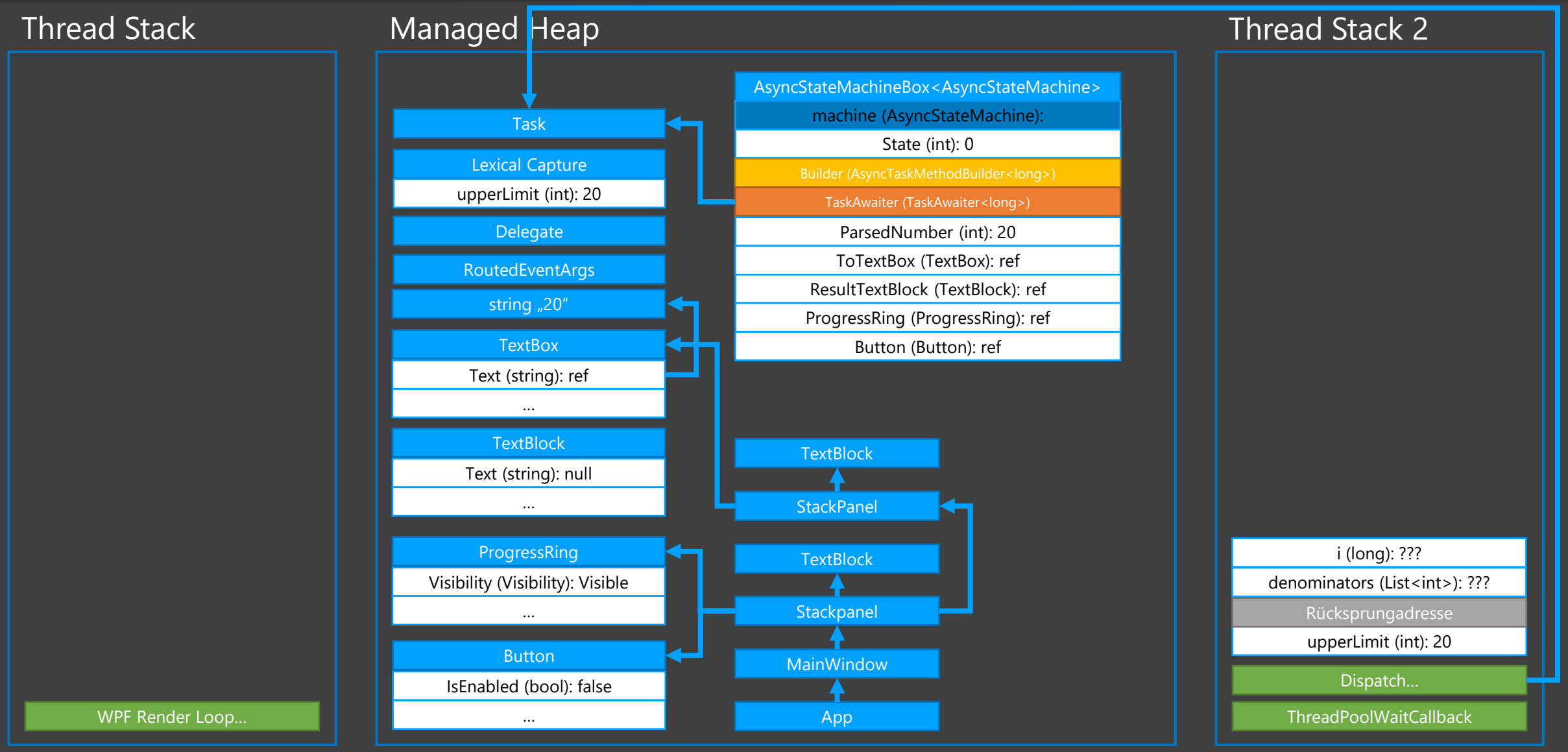
Speicherabbild async – Am Ende von Builder.AwaitOnCompleted



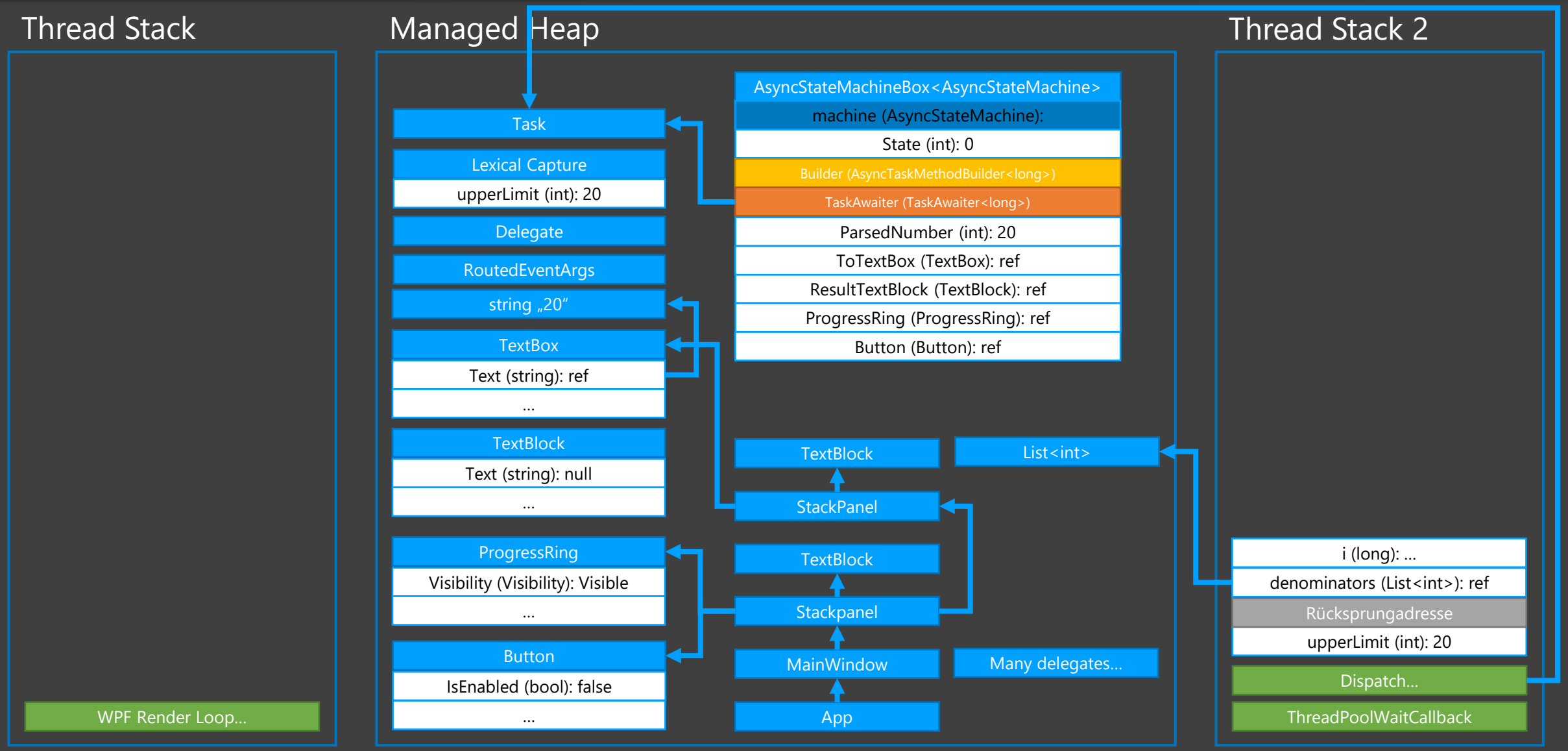
Speicherabbild async – Zurückkehren zur ursprünglichen Methode



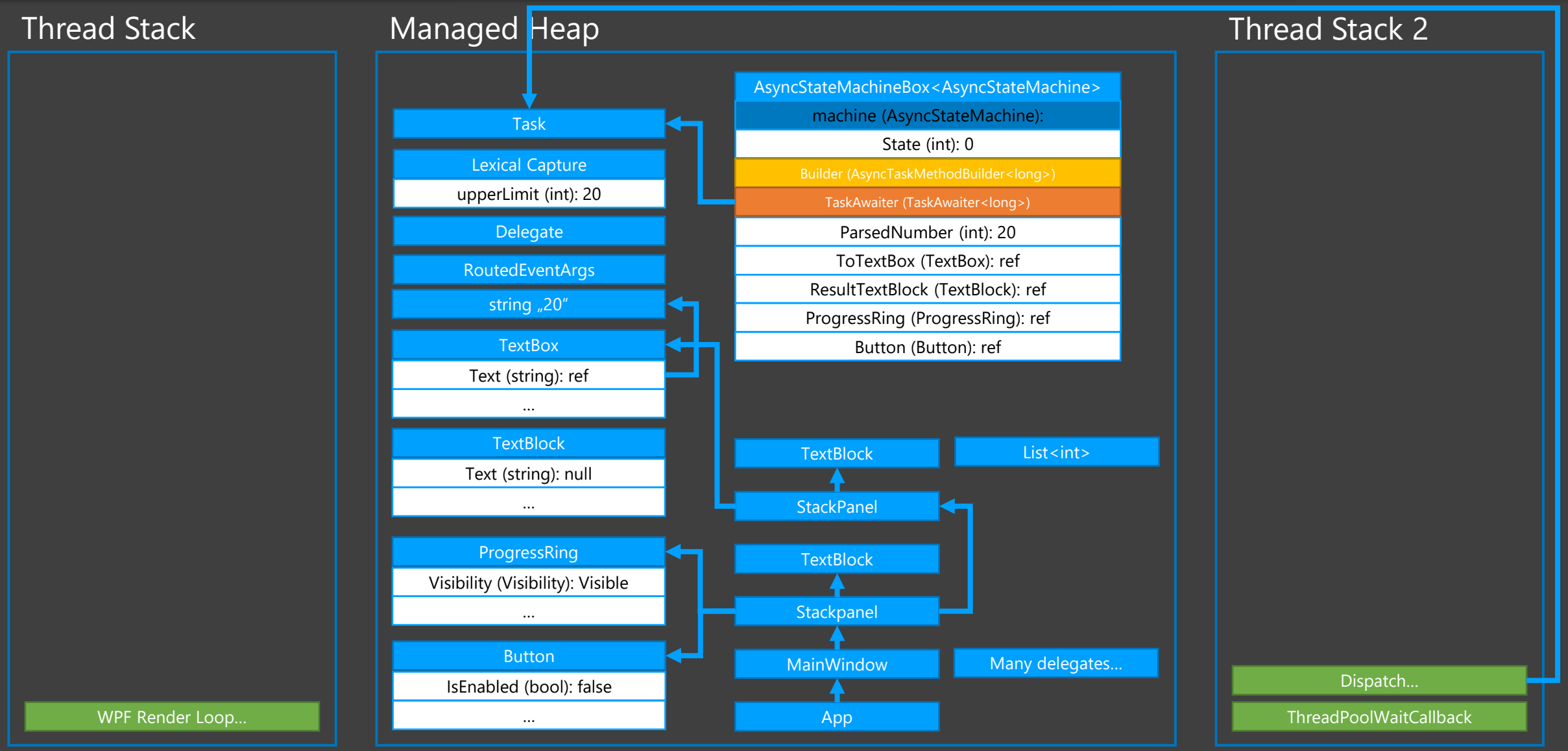
Speicherabbild async – Rückkehr zur Renderloop



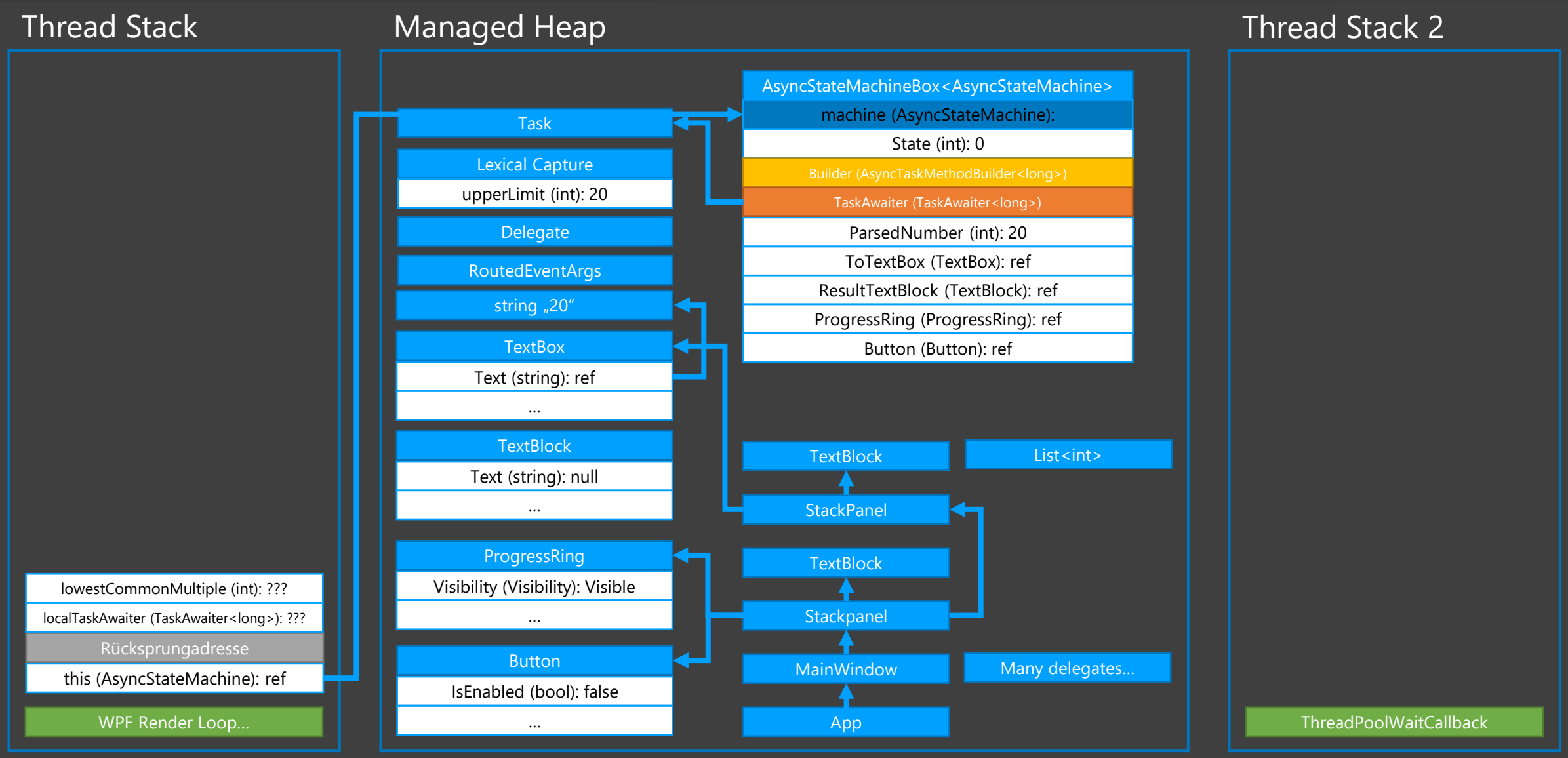
Speicherabbild async – Fortschritt auf dem Background Thread



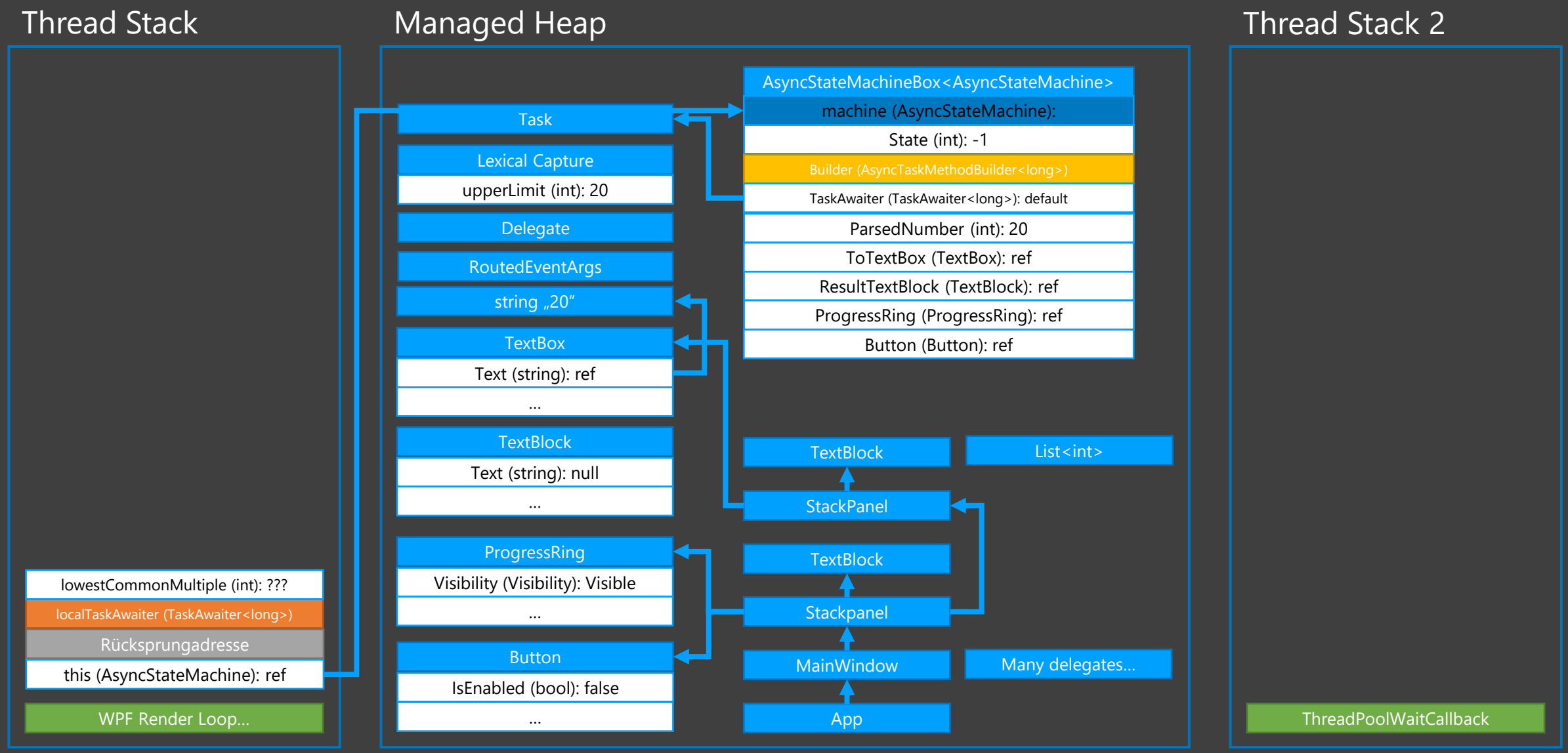
Speicherabbild async – den Task abschließen



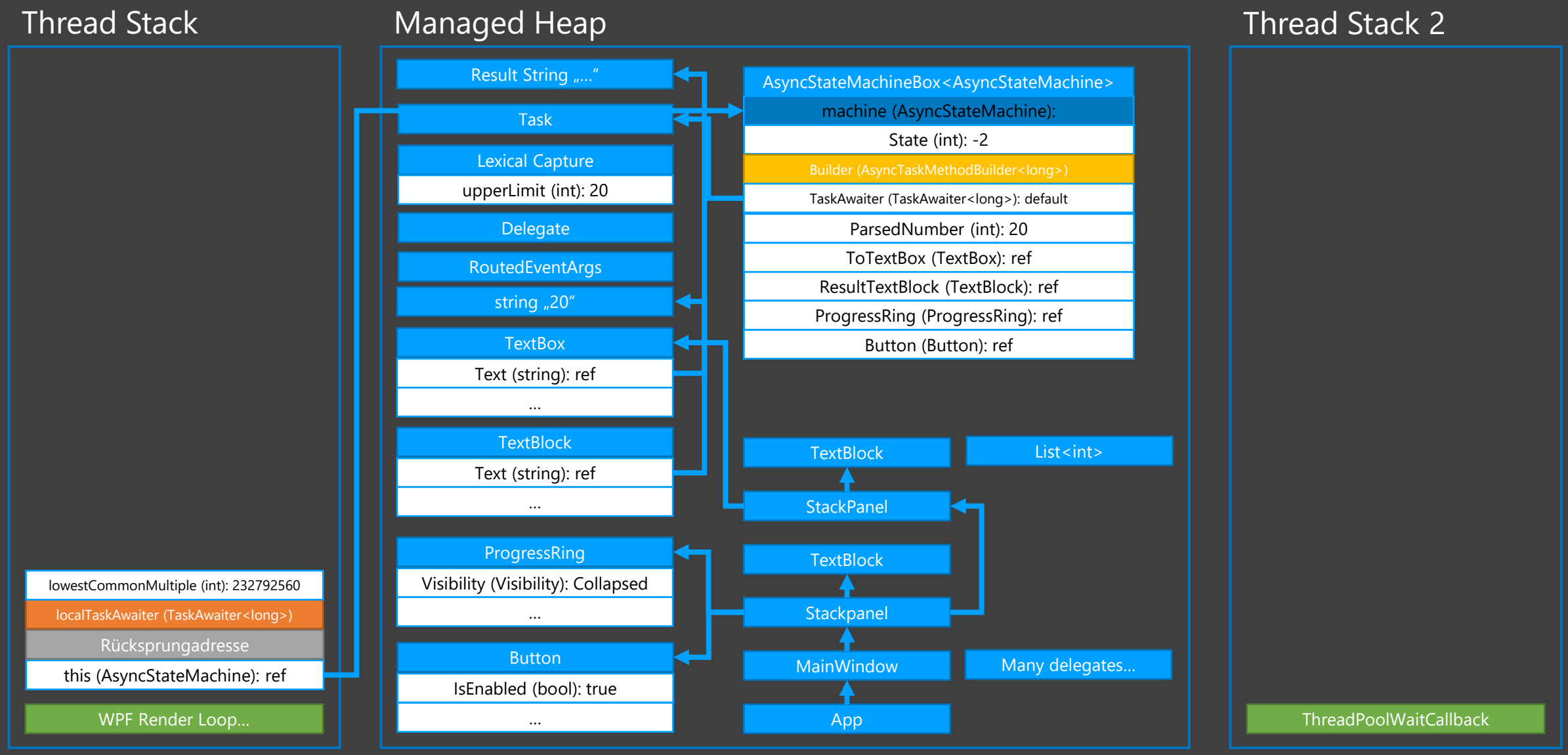
Speicherabbild async – Fortsetzung auf dem UI Thread



Speicherabbild async – Über LocalTaskAwaiter das Ergebnis holen (1)



Speicherabbild async – finales Aktualisieren der Controls



Speicherabbild async – Rückkehr zur UI Loop