Оглавление

[Async/Await Method 1](#_Toc203561228)

[State Machine 1](#_Toc203561229)

[Deadlock Prevention Technique 1](#_Toc203561230)

[Task Scheduler 1](#_Toc203561231)

[Synchronization Context 2](#_Toc203561232)

[Cancellation of Async Operation 2](#_Toc203561233)

[Task Parallelism/Continuation 2](#_Toc203561234)

[Performance 2](#_Toc203561235)

[TaskCompletionSource 2](#_Toc203561236)

[Async Streams 3](#_Toc203561237)

[Async Local 3](#_Toc203561238)

# Async/Await Method

* Async Method Signature: Methods marked with the async keyword return Task, Task<T>, or void (rarely). Signature includes async modifier, e.g., public async Task<int> MethodAsync().
* Release of Calling Thread with Await: await yields control to the calling thread, allowing it to process other tasks while the awaited operation (e.g., I/O) completes. The thread resumes execution when the awaited task finishes.
* Task/Task<T> Role: Task represents an asynchronous operation; Task<T> returns a result of type T. They encapsulate the state and result of async work.
* Async Void Disadvantages: async void methods can't be awaited, making it hard to track completion or handle exceptions. They’re prone to unhandled exceptions crashing the app. Use only for event handlers.

# State Machine

* Structure (Switch-Case and If-Else): The C# compiler transforms async methods into state machines using switch-case or if-else logic to manage state transitions (e.g., paused, completed, or faulted).
* IAsyncStateMachine: Interface implemented by the compiler-generated state machine struct/class, defining the async method's execution flow.
* MoveNext(): Core method of IAsyncStateMachine, invoked to progress the state machine through states, resuming execution after await.
* AsyncTaskMethodBuilder/TaskAwaiter: AsyncTaskMethodBuilder manages the creation and completion of Task objects. TaskAwaiter handles awaiting tasks, ensuring the state machine resumes correctly.

# Deadlock Prevention Technique

* Avoid .Result and .Wait(): Using .Result or .Wait() on a Task in a context with a SynchronizationContext (e.g., UI thread) can cause deadlocks by blocking the thread waiting for the task to complete.
* Async Method Propagation: Ensure all methods in the call chain are async and use await instead of blocking calls to prevent deadlocks and maintain async flow.

# Task Scheduler

* Role: Manages the low-level queuing and execution of tasks on threads, typically via the ThreadPool in .NET.
* Usage: Integral in .NET Core and newer for efficient task scheduling. Custom schedulers can be implemented for specific needs (e.g., UI thread scheduling).
* ConfigureAwait(): ConfigureAwait(false) bypasses capturing the SynchronizationContext, improving performance in non-UI scenarios by avoiding thread marshaling. Use ConfigureAwait(true) (default) when context (e.g., UI) must be preserved.

# Synchronization Context

* Role: Captures and propagates the execution context (e.g., UI thread or ASP.NET request context) across async operations to ensure code runs in the correct environment.
* Usage: Commonly used in .NET Framework for UI apps (e.g., WPF, WinForms) or ASP.NET. Less critical in .NET Core for non-UI scenarios.

# Cancellation of Async Operation

* CancellationTokenSource: Creates a CancellationToken and allows triggering cancellation via .Cancel().
* .Cancel(): Signals cancellation, notifying all operations monitoring the associated CancellationToken.
* CancellationToken: Passed to async methods to monitor cancellation requests, enabling graceful termination of operations (e.g., Task.Delay(token) or custom logic with token.ThrowIfCancellationRequested()).

# Task Parallelism/Continuation

* Task.Run(): Schedules a delegate to run on the ThreadPool, ideal for CPU-bound work.
* Task.Factory.StartNew(): More customizable than Task.Run(), allowing options like custom schedulers or creation flags, but less common.
* ContinueWith(): Specifies a continuation task to execute after a task completes, allowing chained operations.
* WaitAll(): Blocks the calling thread until all specified tasks complete; avoid in async code to prevent blocking.
* WhenAll(): Returns a Task that completes when all provided tasks complete, non-blocking, suitable for async code.
* WhenAny(): Returns a Task that completes when any of the provided tasks complete, useful for racing conditions.

# Performance

* Disadvantages of Using Async Methods: Overhead from state machine creation and context switching, especially for small, quick operations. Avoid overusing async for CPU-bound tasks.
* ValueTask vs Task: ValueTask<T> reduces allocations for short-lived async operations (e.g., cached results). Use when performance is critical, but requires careful handling to avoid reusing consumed ValueTasks.

# TaskCompletionSource

* Role: Allows manual control over a Task's lifecycle, enabling creation of a Task without immediately starting it. Useful for wrapping non-async APIs or events into Task-based APIs.
* Example: TaskCompletionSource<T> can be used to signal completion with .SetResult(), .SetException(), or .SetCanceled().

# Async Streams

* IAsyncEnumerable<T>: Enables asynchronous iteration over a sequence of items, ideal for streaming data.
* Await Foreach: Used to iterate over IAsyncEnumerable<T> with await foreach, processing items as they arrive.
* Use Cases: Processing data from I/O sources (e.g., reading database rows, streaming API responses, or file chunks) without loading all data into memory.

# Async Local

* AsyncLocal<T>: Stores data that flows with the async execution context, persisting across await boundaries.
* Use Cases: Maintaining context (e.g., user ID, request ID) across async operations in a call stack, useful for logging or tracing in distributed systems.