## **Principles of Asynchrony**

This section aims to precisely define what constitutes an asynchronous operation and how this leads to the design principles of asynchronous programming.

### **Synchronous Versus Asynchronous Operations**

To truly grasp asynchronous programming, it is essential to distinguish between synchronous and asynchronous operations:

* **Synchronous Operation:** A synchronous operation completes all of its work *before* returning control to the caller. The calling thread remains blocked until the operation concludes. Common examples include List.Add(), Console.WriteLine(), or Thread.Sleep().
* **Asynchronous Operation:** An asynchronous operation initiates its work and then returns control to the caller *before* the work is fully completed. The actual work continues in parallel with the caller, enabling non-blocking behavior. Asynchronous methods typically return quickly or immediately, allowing the calling thread to remain responsive.

We have already encountered several general-purpose asynchronous methods:

* Thread.Start()
* Task.Run()
* Methods that attach continuations to Task objects (GetAwaiter().OnCompleted(), ContinueWith()) Additionally, UI-specific methods like Dispatcher.BeginInvoke() and Control.BeginInvoke(), as well as SynchronizationContext.Post(), are asynchronous, as is the Delay() method we implemented using TaskCompletionSource.

### **What Is Asynchronous Programming?**

The fundamental principle of asynchronous programming is to design and implement long-running (or potentially long-running) functions in an **asynchronous manner from their inception**. This contrasts with the traditional approach, where a synchronous long-running function is wrapped in a new Thread or Task externally to achieve concurrency.

The key distinction lies in **where concurrency is initiated**: in asynchronous programming, concurrency is initiated *inside* the long-running function itself. This paradigm offers two significant benefits:

1. **I/O-Bound Concurrency without Thread Blocking:** For operations heavily dependent on input/output (e.g., network requests, file access), asynchronous programming allows these operations to proceed without tying up valuable threads. This significantly improves application scalability and efficiency, as threads are not idly waiting for external resources.
2. **Simplified Thread Safety in Rich-Client Applications:** By executing potentially long-running operations asynchronously, rich-client applications can minimize the amount of code running on worker threads. This ensures that the bulk of the application's logic, particularly UI updates, remains on the main UI thread, vastly simplifying thread safety concerns related to shared state and UI controls.

These benefits lead to two primary use cases for asynchronous programming:

* **Server-Side Applications (I/O-Bound Focus):** The goal here is to efficiently handle a large volume of concurrent I/O operations (e.g., numerous network requests in a web server). The challenge is not typically thread safety (as shared state is often minimal) but rather **thread efficiency** – avoiding the consumption of a dedicated thread per network request. In this context, only I/O-bound operations derive significant benefit from asynchrony.
* **Rich-Client Applications (UI Responsiveness and Thread Safety Focus):** This scenario addresses the complexity of maintaining UI responsiveness and simplifying thread safety as an application grows. In a traditional synchronous call graph, if any part of a complex operation is long-running, the entire call graph must be executed on a worker thread to prevent UI freezing. This leads to **coarse-grained concurrency**, where a single concurrent operation spans many methods, necessitating thread-safety considerations across the entire graph.

In contrast, with an asynchronous call graph, threads are only initiated when absolutely necessary, often deep within the call graph, or not at all for purely I/O-bound operations. All other methods can execute on the UI thread with simplified thread-safety. This results in **fine-grained concurrency**, a series of smaller concurrent operations interleaved with execution returning to the UI thread.

As a general guideline, any operation that might take longer than **50 milliseconds** should be considered for asynchronous implementation. It is important to note that excessively fine-grained asynchrony can introduce its own overhead, potentially impacting performance. Modern frameworks, such as the Universal Windows Platform (UWP), strongly advocate for asynchronous programming, often providing only asynchronous versions of long-running methods.

### **Asynchronous Programming and Continuations**

The Task class is ideally suited for asynchronous programming due to its robust support for **continuations**. As we observed with our Delay method implemented via TaskCompletionSource, continuations are essential for defining what happens after an asynchronous operation completes without blocking the initiating thread.

* For **I/O-bound asynchronous methods** at the lowest level of a call graph, TaskCompletionSource is the standard implementation mechanism. It allows you to create a Task that completes based on an external event, rather than a thread executing a delegate.
* For **compute-bound asynchronous methods**, Task.Run() is typically used to offload the computation to a thread pool thread. By returning the Task generated by Task.Run(), we effectively create an asynchronous method.

The core distinction in asynchronous programming is the aim to initiate concurrency **lower in the call graph**. This design principle allows higher-level methods, particularly in rich-client applications, to remain on the UI thread. Consequently, these methods can safely access UI controls and shared state without the complexities of explicit thread-safety mechanisms.

Consider a computationally intensive method, GetPrimesCount(), that calculates prime numbers using parallel processing:

| int GetPrimesCount(int start, int count) {  // Computes prime numbers using ParallelEnumerable  return ParallelEnumerable.Range(start, count).Count(n =>  Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0)); }  void DisplayPrimeCounts() {  for (int i = 0; i < 10; i++)  Console.WriteLine(GetPrimesCount(i \* 1000000 + 2, 1000000) +  " primes between " + (i \* 1000000) + " and " + ((i + 1) \* 1000000 - 1));  Console.WriteLine("Done!"); } |
| --- |

In this synchronous example, calling DisplayPrimeCounts() would block the calling thread until all prime counts are calculated. To introduce **coarse-grained concurrency**, one might wrap the entire call to DisplayPrimeCounts() within Task.Run(): Task.Run(() => DisplayPrimeCounts());.

However, for a **fine-grained asynchronous approach**, we would create an asynchronous version of GetPrimesCount():

| Task<int> GetPrimesCountAsync(int start, int count) {  return Task.Run(() => // Offloads computation to a thread pool thread  ParallelEnumerable.Range(start, count).Count(n =>  Enumerable.Range(2, (int)Math.Sqrt(n) - 1).All(i => n % i > 0))); } |
| --- |

### **Why Language Support (async and await) Is Important**

Modifying DisplayPrimeCounts() to call GetPrimesCountAsync() manually using GetAwaiter().OnCompleted() becomes remarkably complex. A naive loop like this:

| for (int i = 0; i < 10; i++) {  var awaiter = GetPrimesCountAsync(i \* 1000000 + 2, 1000000).GetAwaiter();  awaiter.OnCompleted(() =>  Console.WriteLine(awaiter.GetResult() + " primes between... ")); } Console.WriteLine("Done"); |
| --- |

would result in all 10 GetPrimesCountAsync operations starting in parallel (as they are non-blocking) and "Done" appearing prematurely. To serialize these operations (e.g., if one task depends on the result of a previous one, like a DNS lookup before an HTTP request), one would have to resort to a cumbersome recursive pattern within the continuations, essentially building a state machine manually:

| void DisplayPrimeCounts() {  DisplayPrimeCountsFrom(0); }  void DisplayPrimeCountsFrom(int i) {  var awaiter = GetPrimesCountAsync(i \* 1000000 + 2, 1000000).GetAwaiter();  awaiter.OnCompleted(() =>  {  Console.WriteLine(awaiter.GetResult() + " primes between...");  if (++i < 10) DisplayPrimeCountsFrom(i); // Recursive call for next iteration  else Console.WriteLine("Done");  }); } |
| --- |

Furthermore, making DisplayPrimeCountsFrom itself asynchronous (returning a Task that signals completion) would require manual management of a TaskCompletionSource and an explicit state machine class, significantly increasing boilerplate code.

Fortunately, C#’s **async and await keywords** abstract away this complexity. They allow developers to write asynchronous code in a sequential, synchronous-looking style, with the compiler generating the necessary state machine, continuations, and TaskCompletionSource calls implicitly.

| async Task DisplayPrimeCountsAsync() {  for (int i = 0; i < 10; i++)  Console.WriteLine(await GetPrimesCountAsync(i \* 1000000 + 2, 1000000) + // 'await' pauses execution here  " primes between " + (i \* 1000000) + " and " + ((i + 1) \* 1000000 - 1));  Console.WriteLine("Done!"); } |
| --- |