## **Asynchronous Patterns**

Beyond merely initiating asynchronous operations, building robust applications requires effective strategies for managing their lifecycle, reporting progress, and combining their outcomes. This section covers key asynchronous patterns.

### **Cancellation**

The ability to **cancel a concurrent operation** is critical, whether in response to a user's request, a system-imposed timeout, or other dynamic conditions.

A basic approach to cancellation involves a shared flag. The Common Language Runtime (CLR) formalizes this concept through two primary types:

* **CancellationTokenSource**: This class is responsible for *initiating* the cancellation. It provides the Cancel() method to signal a cancellation request.
* **CancellationToken**: This struct represents the actual token that an asynchronous operation monitors. It exposes properties like IsCancellationRequested to check for cancellation and ThrowIfCancellationRequested() to throw an OperationCanceledException (a predefined .NET exception for this purpose) if cancellation has been requested. This separation ensures that a method consuming a CancellationToken can observe, but not initiate, a cancellation.

**Implementing Cancellation:**

To support cancellation, an asynchronous method typically accepts a CancellationToken as a parameter. The method's logic periodically checks the token or passes it to other asynchronous methods that support cancellation.

| async Task Foo(CancellationToken cancellationToken) {  for (int i = 0; i < 10; i++)  {  Console.WriteLine(i);  // Pass the token to Task.Delay, which internally monitors for cancellation  await Task.Delay(1000, cancellationToken);  // No explicit ThrowIfCancellationRequested is needed here if Task.Delay handles it  } } |
| --- |

When the caller wants to cancel, it calls Cancel() on its CancellationTokenSource instance. If Task.Delay() (or any other method accepting the CancellationToken) detects the cancellation, it will immediately cease execution and throw an OperationCanceledException, which propagates up the call stack. This ensures that cancellation requests cascade effectively through the asynchronous call graph.

CancellationTokenSource can also be constructed with a time interval, automatically initiating cancellation after a specified duration, making it highly useful for implementing timeouts in both synchronous and asynchronous operations. The CancellationToken struct also offers a Register() method to attach callback delegates that execute upon cancellation.

When an OperationCanceledException is unhandled by an async function, the Task it returns will enter a **"Canceled" state** (IsCanceled will be true, IsFaulted will be false). While in most asynchronous scenarios, a canceled task behaves similarly to a faulted task (both throw OperationCanceledException when awaited), this distinction can be important in advanced parallel programming contexts.

### **Progress Reporting**

For long-running asynchronous operations, it is often desirable to provide the user with **progress updates** as the operation proceeds.

A naive approach might involve passing an Action delegate to the asynchronous method to report progress. However, this method faces challenges in rich-client applications, as the Action delegate would be invoked on a worker thread, potentially leading to thread-safety issues when trying to update UI elements directly.

The CLR provides a specialized pair of types to address this problem safely and effectively:

* **IProgress<T>**: A generic interface defining a single method, void Report(T value), used by the asynchronous operation to publish progress updates. T represents the type of the progress data (e.g., int for percentage, or a custom class for more detailed information).
* **Progress<T>**: A class that implements IProgress<T>. When Progress<T> is instantiated, it automatically **captures the current SynchronizationContext** (if one exists, as in UI applications). When the asynchronous operation calls Report(), the Progress<T> instance ensures that the provided progress delegate (passed in its constructor) or ProgressChanged event handlers are invoked on the captured synchronization context, thereby preventing cross-thread UI access violations.

**Implementing Progress Reporting:**

| async Task Foo(IProgress<int> onProgressPercentChanged) {  await Task.Run(() =>  {  for (int i = 0; i < 1000; i++)  {  if (i % 10 == 0) onProgressPercentChanged.Report(i / 10);  // Simulate compute-bound work  Thread.Sleep(1);  }  }); }  // In a UI thread context: var progress = new Progress<int>(i => Console.WriteLine($"Progress: {i}%")); await Foo(progress); // This call will report progress safely to the console (or UI) |
| --- |

This pattern allows the asynchronous method to remain agnostic of the UI specifics, focusing solely on reporting data, while Progress<T> handles the necessary context marshalling for safe consumption.

### **The Task-Based Asynchronous Pattern (TAP)**

The **Task-Based Asynchronous Pattern (TAP)** is the recommended and formalized pattern in .NET for exposing asynchronous operations. It provides a consistent programming model that facilitates the use of C#'s async and await keywords.

A method conforming to the TAP pattern typically exhibits the following characteristics:

* **Returns a Task or Task<TResult>**: The method returns a "hot" (already running) Task for operations that do not produce a result, or a Task<TResult> for operations that do.
* **"Async" Suffix**: The method name conventionally ends with "Async" (e.g., DownloadStringAsync), with few exceptions for task combinators.
* **CancellationToken and IProgress Overloads**: If the operation supports cancellation or progress reporting, it provides overloads that accept a CancellationToken and/or IProgress<T> respectively.
* **Quick Return**: The method returns quickly to the caller, performing only a small initial synchronous phase before initiating the asynchronous work.
* **Thread Efficiency**: For I/O-bound operations, the method does not tie up a thread during the asynchronous wait periods.

The simplicity of writing TAP methods is greatly enhanced by the async and await keywords in C#.

### **Task Combinators**

**Task combinators** are functions that take one or more Task objects as input and return a new Task that represents the combined result of those inputs. They allow for powerful, declarative control over the execution flow of multiple asynchronous operations without needing to know the specifics of each individual task. The CLR provides two fundamental task combinators: Task.WhenAny and Task.WhenAll.

#### **Task.WhenAny**

Task.WhenAny returns a Task that completes when *any one* of the provided input tasks completes. The returned Task wraps the specific Task that finished first.

| async Task<int> Delay1() { await Task.Delay(1000); return 1; } async Task<int> Delay2() { await Task.Delay(2000); return 2; } async Task<int> Delay3() { await Task.Delay(3000); return 3; }  // Example usage: Task<int> winningTask = await Task.WhenAny(Delay1(), Delay2(), Delay3()); Console.WriteLine(await winningTask); // Output: 1 (after ~1 second) |
| --- |

Task.WhenAny is particularly useful for scenarios such as applying a timeout to an operation that doesn't inherently support it, or when you need to react to the first of several possible asynchronous events. If the winning task faults, awaiting it directly (await winningTask) rethrows its exception without AggregateException wrapping. Note that if a non-winning task subsequently faults, its exception will go unobserved unless explicitly awaited or its Exception property is queried.

#### **Task.WhenAll**

Task.WhenAll returns a Task that completes only when *all* of the provided input tasks have completed. This embodies the "fork/join" pattern, allowing multiple asynchronous operations to run in parallel and then waiting for all of them to finish before proceeding.

| // Example usage: await Task.WhenAll(Delay1(), Delay2(), Delay3()); // Completes after ~3 seconds Console.WriteLine("All delays complete!"); |
| --- |

A key distinction from simply awaiting tasks sequentially is in exception handling. Task.WhenAll continues to run all tasks even if some fault. If multiple tasks fault, their exceptions are aggregated into an AggregateException associated with the Task returned by WhenAll. Awaiting this combined task will rethrow only the *first* exception encountered, but all inner exceptions can be inspected via the Exception.InnerExceptions property.

When Task.WhenAll is used with Task<TResult> inputs, it returns a Task<TResult[]>, allowing you to conveniently collect all results as an array:

| Task<int> task1 = Task.Run(() => 1); Task<int> task2 = Task.Run(() => 2); int[] results = await Task.WhenAll(task1, task2); // results will be {1, 2} |
| --- |

A practical application is downloading multiple URIs in parallel:

| async Task<long> GetTotalSize(string[] uris) {  // Create a collection of tasks that return the content length  IEnumerable<Task<int>> downloadTasks = uris.Select(async uri =>  (await new WebClient().DownloadDataTaskAsync(uri)).Length); // Async lambda for efficiency   int[] contentLengths = await Task.WhenAll(downloadTasks); // Wait for all downloads  return contentLengths.Sum(); // Sum their lengths } |
| --- |