Async programming

Asynchronous programming in C# is a technique used to perform non-blocking operations, allowing an application to perform tasks concurrently without waiting for each operation to complete before moving on to the next one. This improves the responsiveness of the application, especially when dealing with tasks that may take time, such as I/O operations or waiting for external resources.

Async programming use cases

Synchronous

```
private void Search_Click(...)
{
    var client = new WebClient();

    var content =
              client.DownloadString(URL);

}
```

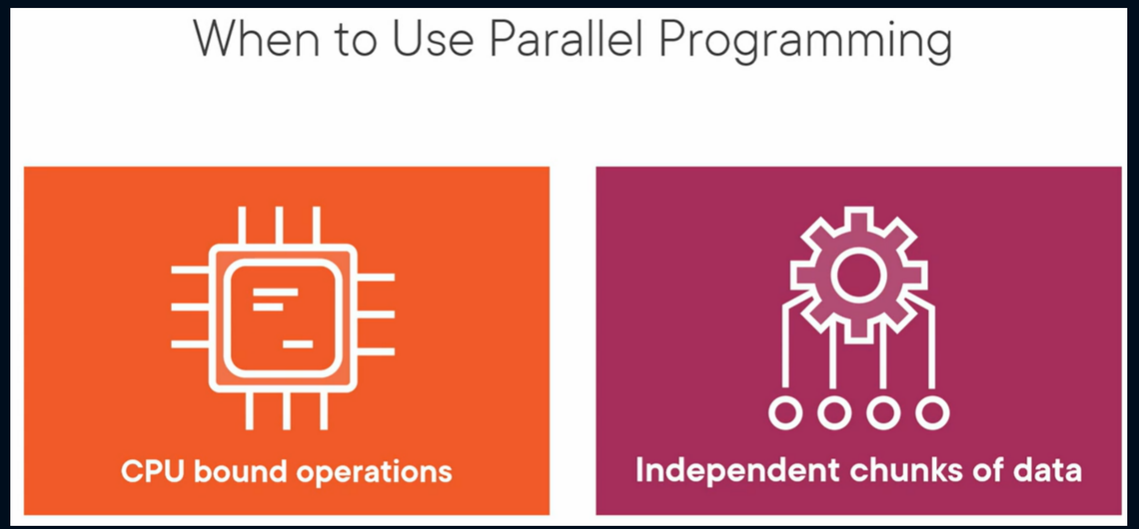
Asynchronous

```
private async void Search_Click(...)
{
    var client = new HttpClient();

    var response = await
              client.GetAsync(URL);

    var content = await response.
             Content.ReadAsStringAsync();
}
```

Example

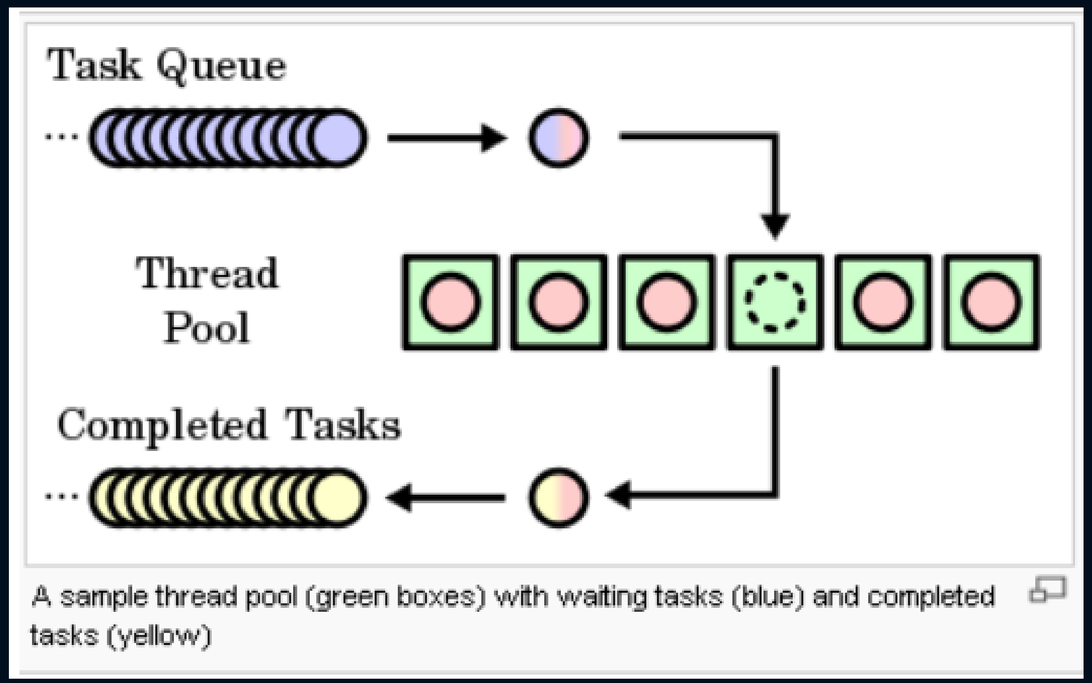
Async vs parallel programming

1. Asynchronous Programming:
   - Is a technique used to perform tasks concurrently without blocking the execution of the main program.
   - Tasks that may take time to complete (such as I/O operations or waiting for external resources) are initiated, and the program continues to execute other tasks while waiting for the completion of those asynchronous tasks.
   - Asynchronous operations typically use features like callbacks, promises, or async/await in C# to handle concurrency without blocking the main program's execution.
   - Is well-suited for I/O-bound tasks, where the program can perform other work while waiting for the I/O operation to complete.
2. Parallel Programming:
   - Is a technique used to execute multiple tasks simultaneously by dividing them into smaller subtasks and processing them concurrently.
   - In parallel programming, multiple threads or processes are used to perform the tasks simultaneously, taking advantage of multiple CPU cores to achieve parallelism.
   - Parallel programming is ideal for CPU-bound tasks, where the program can divide the workload across multiple cores to speed up the execution of computationally intensive tasks.
   - Task Parallel Library (TPL) in C# is used to implement parallel programming.



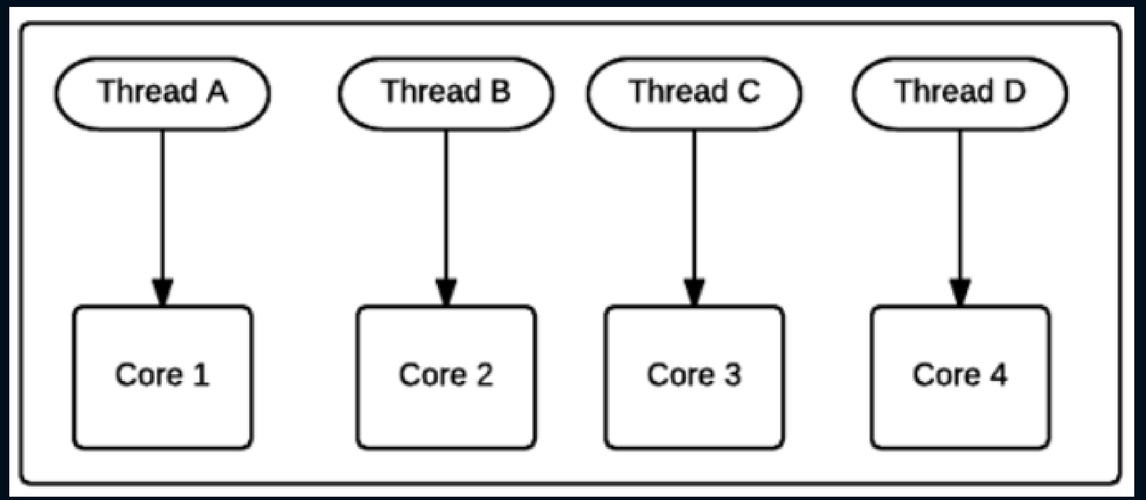
Async vs parallel programming

A thread represents an independent flow of execution within a process, and multiple threads can run simultaneously, performing different tasks concurrently. Threads can be used to accomplish various goals, including parallelizing work, handling I/O operations asynchronously, and improving application responsiveness.
The main thread is the first thread created when a C# application starts running. In GUI-based applications (e.g., Windows Forms, WPF, UWP), the main thread is often referred to as the UI thread because it handles user interface interactions and updates.



A sample thread pool (green boxes) with waiting tasks (blue) and completed tasks (yellow)

Tasks



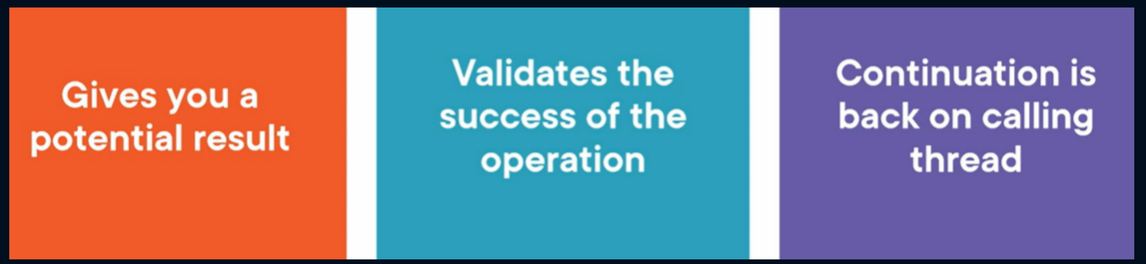
Threads

Await keyword

C# provides the async and await keywords to facilitate asynchronous programming. The async keyword is used to mark a method as asynchronous, while the await keyword is used to asynchronously wait for the completion of an asynchronous operation.
When used with asynchronous methods, the await keyword provides a non-blocking way to handle time-consuming operations, such as I/O-bound tasks or waiting for external resources, without freezing the application's user interface or main thread.

```
private async void Search_Click(...)
{
    var store = new DataStore();

    var responseTask = store.GetStockPrices("MSFT");

    var data = await responseTask;
    // Code below will run
    // when responseTask has completed
    Stocks.ItemsSource = data;
}
```

Example



Await characteristics

Threads

Threads may need to coordinate and share data while running concurrently.
Proper synchronization mechanisms, such as locks, semaphores, and monitors, are used to prevent data corruption or race conditions when multiple threads access shared resources.

```
var response = await client.GetAsync(URL);
                    ↓ Continuation executed when GetAsync completes
var content =  await response.Content.ReadAsStringAsync();
                    ↓ Continuation executed when ReadAsStringAsync completes
var data = JsonConvert.DeserializeObject(...)
```

Continuation

Await characteristics

Exceptions with Async

When working with asynchronous code, it's essential to handle exceptions properly to prevent unhandled exceptions from crashing the application. Since asynchronous methods return Task or Task<T> objects representing the ongoing operations, exceptions that occur in asynchronous methods are wrapped inside the returned Task.
To handle exceptions in async/await code, use try-catch blocks inside the asynchronous method or rely on the caller to handle exceptions using await.

```
private async Task GetStocks()
{
    try
    {
        var store = new DataStore();

        var responseTask = store.GetStockPrices(StockIdentifier.Text);

        Stocks.ItemsSource = await responseTask;
    }
    catch (Exception ex)
    {
        throw;
    }
}
```

Exception Handler

```
private async void Search_Click(object sender, RoutedEventArgs e)
{
    try
    {
        BeforeLoadingStockData();

        await GetStocks();
    }
    catch (Exception ex)
    {
        Notes.Text = ex.Message;
    }
    finally
    {
        AfterLoadingStockData();
    }
```

Exception Handler

Async and Void

In the context of asynchronous programming, it's essential to avoid using async void methods unless they are event handlers or top-level entry points (e.g., Main method in console applications or event handlers in GUI applications). The reason is that async void methods are not easily awaited, and exceptions thrown within them cannot be caught directly by the calling code.

Avoid using async void

```
async Task Good()
{
    throw new Exception("Find me on the Task");
}

async void Bad()
{
    throw new Exception("No one can catch me");
}
```

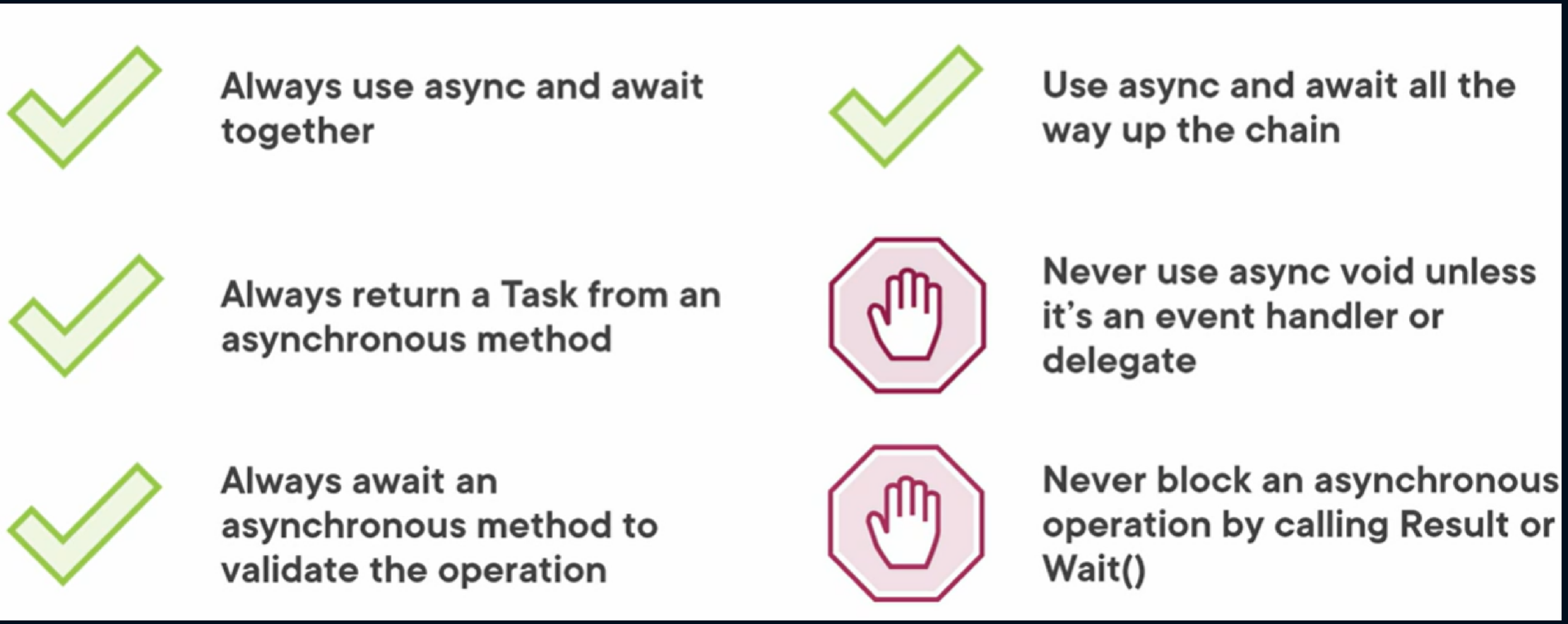
Async Void

```
private async void Search_Click(...)
{
    GetStocks().Wait();
}
private async Task GetStocks()
{
    ...
}
```

Causes a deadlock!

Example

Some best practices with Async



Async programming use cases

Await cons

The “cost” of await

Await creates a “state-machine” behind the scenes. One might tempted to “save” that, but not awaiting can make course problems with IDisposable or lead to a less traceable stacktrace.

```
[MemoryDiagnoser]
public class AwaitVsElide
{
    private static readonly Task<string> ExampleTask
        = Task.FromResult("Hello foo bar");

    [Benchmark]
    public async Task<string> GetStringWithoutAsync()
        => await GetStringWithoutAsyncCore();
    [Benchmark]
    public async Task<string> GetStringWithAsync()
        => await GetStringWithAsyncCore();

    private static Task<string> GetStringWithoutAsyncCore() => ExampleTask;
    private static async Task<string> GetStringWithAsyncCore() => await ExampleTask;
}
```

```
|                 Method |     Mean |    Error |   StdDev |   Gen0 | Allocated |
|----------------------- |---------:|---------:|---------:|-------:|----------:|
| GetStringWithoutAsync | 12.30 ns | 0.050 ns | 0.047 ns | 0.0115 |      72 B |
|    GetStringWithAsync | 24.96 ns | 0.108 ns | 0.101 ns | 0.0229 |     144 B |
```

Steven Giesel

Demo

We will check the following:

Asynchronous programming implementation

How async and await are compiled to IL

You can check the following Repository(ies) for some examples:

c-sharp-async-programming

You can check the following Repository for some examples:

C# fundamentals

Task

Quick Exercises

Using the gRPC Client Project from C# fundamentals repository, currently we are using the gRPC server to send requests, now alongside sending the request I need to save the request and the response with the time they were sent in a File, you can use a .txt file.

Add a stopwatch timer and measure the time it takes to send the request and save it and the response in a file. The timer should print the result in Console and should also log that information in the file.

Every new request made a new instance of the timer is created. More information on the stopwatch class is here

It is important to use asynchronous operation that way we don't block the thread in which the stopwatch is running