Threading in C#: From Threads to Async/Await

Slide 1: Title Slide

Threading in C#: From Threads to Async/Await

A Practical Guide for Beginners

Slide 2: What is Threading?

A thread is the smallest unit of execution within a process.

Each process starts with a single main thread.

Multithreading allows multiple tasks to run concurrently.

Slide 3: Why Use Threading?

- To keep applications responsive (e.g., UI apps)
- To run tasks concurrently
- To utilize multiple cores
- For background or long-running operations

Slide 4: Manual Threading

```
Thread thread = new Thread(() => {
    Console.WriteLine("Running in a new thread");
});
thread.Start();
```

Slide 5: Threading Challenges

- Hard to manage and debug
- Risk of race conditions

Need for synchronization (e.g., lock, Monitor)

Threads consume memory and CPU

Slide 6: Task Parallel Library (TPL)

```
await Task.Run(() => {
    Console.WriteLine("Task running on background thread");
});
```

Slide 7: Async and Await

```
async Task<string> ReadFileAsync()
{
    await Task.Delay(2000); // Simulated I/O
    return "File content";
}
```

Slide 8: Live Demo

Demonstrate:

1. Manual Thread
2. Task.Run
3. async/await I/O simulation

Slide 9: Comparing the Approaches

Feature	Thread	Task	Async/Await
Control	Full	Medium	Low

Abstraction	Low	Medium	High
Complexity	High	Medium	Low
Use Case	CPU	Mixed	I/O
Blocking	Yes	Optional	No

Slide 10: Best Practices

- Dont block async code with .Wait() or .Result
- Use Task.WhenAll for parallel async operations
- Avoid unnecessary threads
- Use CancellationToken
- Use lock/SemaphoreSlim for shared resources

Slide 11: Summary

- Threads enable concurrency, but are low-level
- TPL simplifies parallelism
- Async/await is for I/O
- Know the trade-offs

Slide 12: Questions & Resources

Books:

- Concurrency in C# Cookbook by Stephen Cleary

Docs:

- docs.microsoft.com