Threading & Asynchronous Development in C#

1. Introduction

- **Multithreading** and **asynchronous programming** allow programs to perform multiple operations concurrently.
- Improves responsiveness, performance, and scalability.
- Essential for UI apps, server-side development, I/O-bound tasks.

2. Threading Basics

`System.Threading.Thread`

- Used to create and manage threads manually.
- Example:

```
Thread t = new Thread(SomeMethod);
t.Start();
```

Thread Lifecycle

- **Unstarted → Running → WaitSleepJoin → Stopped**
- Methods:
  - `.Start()`, `.Join()`, `.Abort()` *(not recommended)*

Limitations

- Manual thread management is **error-prone**.
- Use **ThreadPool**, **Tasks**, or `async/await` for simplicity and efficiency.

3. ThreadPool

`System.Threading.ThreadPool`

- Provides a pool of worker threads.
- Suitable for **short-lived background operations**.

```
ThreadPool.QueueUserWorkItem(state => {
    Console.WriteLine("Running in thread pool!");
});
```

4. Tasks (`System.Threading.Tasks.Task`)

- **Higher-level abstraction** over threads.
- Easy to chain operations and manage continuations.

Example:

```
Task t = Task.Run(() => DoWork());
```

Task Chaining:

```
Task.Run(() => DoWork())
    .ContinueWith(t => MoreWork());
```

5. Asynchronous Programming (`async` / `await`)

- Introduced in **C# 5.0**.
- Makes asynchronous code **readable and maintainable**.
- Used for **I/O-bound operations**, e.g., file access, web requests.

Example:

```
public async Task<string> GetDataAsync()
{
    HttpClient client = new HttpClient();
    string result = await client.GetStringAsync("https://example.com");
    return result;
}
```

Benefits:

- Non-blocking.
- Scales better than threads for I/O-bound operations.

6. Common Patterns

Pattern	Use Case
`Task.Run`	Run CPU-bound work on background threads
`async/await`	Handle I/O without blocking the thread
`Parallel.ForEach`	Simple parallel loops
`ThreadPool`	Short tasks with minimal overhead

7. Thread Safety & Synchronization

- Avoid race conditions using:
  - `lock`
  - `Monitor`
  - `Mutex`, `Semaphore`
  - `Concurrent collections`
  - `Interlocked`

```
lock(myLockObj)
{
    sharedResource++;
}
```

8. Best Practices

- Avoid unnecessary threads.
- Use `async` for I/O-bound, `Task.Run` for CPU-bound.
- Never block on `async` code (e.g., `.Result`, `.Wait()`).
- Use cancellation with `CancellationToken`.
- Use `ConfigureAwait(false)` in libraries.

10. Summary Table

Feature	Use When	Abstraction Level
Thread	Full control over execution	Low
ThreadPool	Fire-and-forget background work	Medium
Task	Chaining, better exception handling	Medium-High
async/await	Simplify async I/O code	High

Q & A