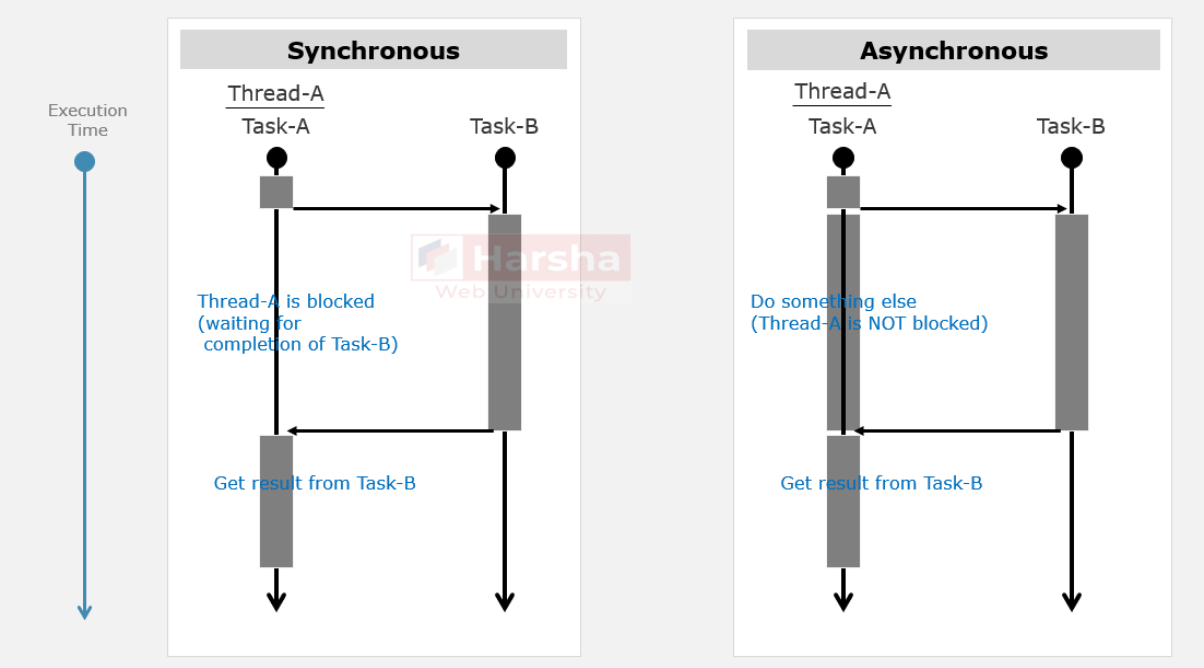
# C# - Ultimate Guide - Beginner to Advanced | Master class

## Section 33 – Asynchronous Programming

**Introduction to Asynchronous Programming**




Asynchronous programming is a programming paradigm that allows tasks to be executed independently and concurrently, without blocking the main thread of execution.

**Non-Blocking I/O operations:** The asynchronous read / write operations with external resources such as files, databases or network requests that do not block the current (caller) thread.

**'async' and 'await'**

**'async' Keyword**

The '**async**' keyword is used to make a method, lambda expression or anonymous method as '**asynchronous method**'.

The return type of an async method is typically **Task** or **Task<T>** where **T** is the type of the result (actual return value).

The **asynchronous method** can be called with '**await**' keyword.

**'await' Keyword**

The '**await**' keyword indicates a point where the method should pause its execution and wait for result of an **asynchronous method**; and also it yields the 'execution control' back to its caller until the awaited **asynchronous method** is complete.

The '**await**' keyword can only be used within another **asynchronous method**.

**How 'async' and 'await' work?**

**Invoke Asynchronous Method**

```
Task<TResult> task = MethodAsync(); //Invoke the asynchronous method;
… //and also wait until the asynchronous method completes (without blocking the thread)
```

**Asynchronous Method**

```
public async Task<TResult> MethodAsync(parameters)
{
  …
  return some_value; //sets the returned value into Task.Result property
}
```

**Receive the task result**

```
TResult result = await task; //sets the Task.Result into "result" variable
… //continue execution.
```

**1. Suspend Execution**

When you await an asynchronous operation, the runtime (CLR) suspends the execution of the current method (but not the current thread).

It ensures that the method's local state is preserved, so when the awaited task completes, the method can resume execution from where it left off.

**2. Continuation**

While the awaited task is running asynchronously (e.g., making a network request or reading a file or a database call), the current thread is freed up to perform other work. The method doesn't block, making your program responsive.

**3. Completion Notification**

When the awaited task completes, whether successfully or with an exception, the runtime captures the result or exception information and the caller method resumes execution at the point where it was suspended, and assigns the return value of the asynchronous method at the place where the asynchronous method was called; and then it continues execution of the caller method.

**4. Error Handling**

If the awaited task throws an exception, that exception is propagated up the call stack, allowing you to handle errors using standard exception handling mechanisms

**Writing 'async' and 'await' code in better way**

**Caller Method**

```
public async Task CallerMethod()
{
  await MethodAsync(); //Invoke the asynchronous method;
  … //and also wait until the asynchronous method completes (without blocking the thread)
}
```

**Asynchronous Method**

```
public async Task<TResult> MethodAsync(parameters)
{
  …
  return some_value; //sets the returned value into Task.Result property
}
```

**Caller Method**

```
TResult result = await MethodAsync(); //sets the Task.Result into "result" variable
… //continue execution.
```

**Writing 'async' and 'await' code in lengthy way**

**Invoke Asynchronous Method**

```
Task<TResult> task = MethodAsync(); //Invoke the asynchronous method;
… //and also wait until the asynchronous method completes (without blocking the thread)
```

**Asynchronous Method**

```
public async Task<TResult> MethodAsync(parameters)
{
  …
  return some_value; //sets the returned value into Task.Result property
}
```

**Receive the task result**

```
TResult result = await task; //sets the Task.Result into "result" variable
… //continue execution.
```

**'async' and 'await' - Best Practices**

**Avoid "async void"**

Avoid using async void methods, especially in top-level event handlers or constructors.

Async void methods can't be awaited and can make it difficult to catch the unhandled exceptions.

**Return Task or Task<TResult>**

Make sure the return of asynchronous methods be either Task (equivalent to void) or Task<TResult> (equivalent to returning TResult).

**Handle Exceptions Properly**

Always handle exceptions that might occur in async methods.

Use try-catch blocks to catch and handle exceptions appropriately.

If an exception is unhandled in an async method, it can crash your application.

**Use async/await for I/O-Bound Operations**

Async and await are best suited for I/O-bound operations, such as file I/O, network requests or database queries. There is no performance benefit in case of async/await with CPU-bound operations.

**Async All the Way**

When you have an asynchronous method, make sure to propagate the asynchrony all the way up the call stack.

**Avoid using Task.Run() in asynchronous method**

Avoid using Task.Run to offload synchronous work to a background thread within an async method.

This can lead to thread pool exhaustion and decreased performance. Instead, use "await".

**Use CancellationToken for Cancellation**

Pass a CancellationToken to your async methods when cancellation is possible. This allows you to cancel long-running tasks gracefully.

Be sure to check for cancellation periodically within your async method.

**Measure and Optimize**

Profile your asynchronous code to identify performance bottlenecks and areas that can be optimized.

Tools like Stopwatch and Visual Studio's Diagnostic Tools can help you identify and address performance issues.