# Asynchronous programming in depth

# Asynchronous Programming Patterns

***Asynchronous Programming Model (APM)***

***Not recommended***

- **`Begin`** and **`End`** methods (for example, **`BeginWrite`** and **`EndWrite`**)
- **`IAsyncResult`**

***Event-based Asynchronous Pattern (EAP)***

***Not recommended***

- Methods with **`Async`** suffix
- One or more events

***Task-based Asynchronous Pattern (TAP)***

***Recommended***

- **`Task`**s and **`async`**/**`await`** keywords.

## Tasks

**Task**s are constructs used to implement what is known as the ***Promise Model of Concurrency***. In short, they offer you a "promise" that work will be completed at a later point, letting you coordinate with the promise with a clean API.

- **Task** represents a single operation which does not return a value.
- **Task<T>** represents a single operation which returns a value of type **T**.

**Task**s expose an API protocol for *monitoring*, *waiting upon* and *accessing the result value* (in the case of **Task<T>**) of a task.

## Asynchronous functions

Language integration, with the **async/await** keywords, provides a higher-level abstraction for using **Task**s.

- Method declaration should be marked with **async** keyword.
- Method should return **Task** or **Task<T>**

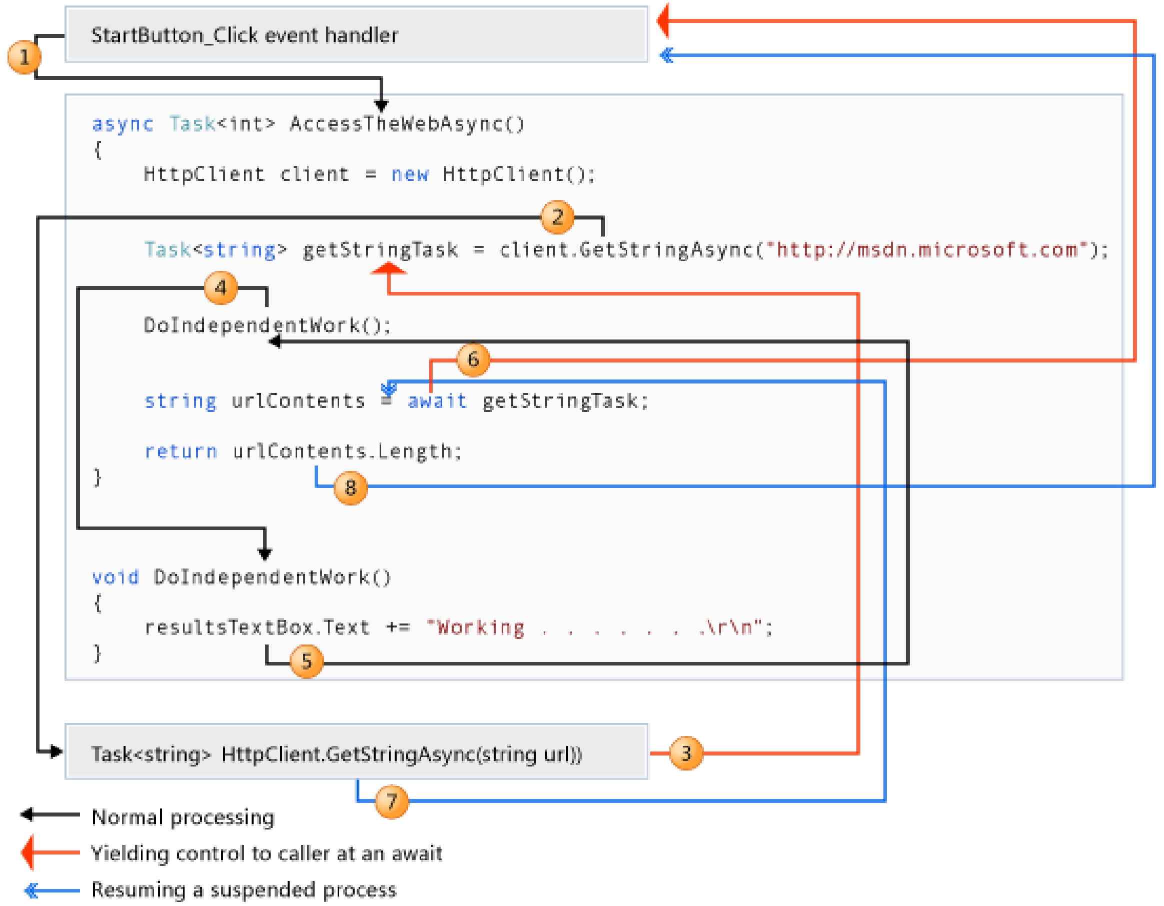
```
async Task<string> DoSomeWork() {
    // ...
    return await sr.ReadToEndAsync();
}
```

Without **async**/**await**:

```
Task.Run(() => ComputeHeavyOp(1))
  .ContinueWith(_ => ComputeHeavyOp(2))
  .ContinueWith(_ => ComputeHeavyOp(3));
```

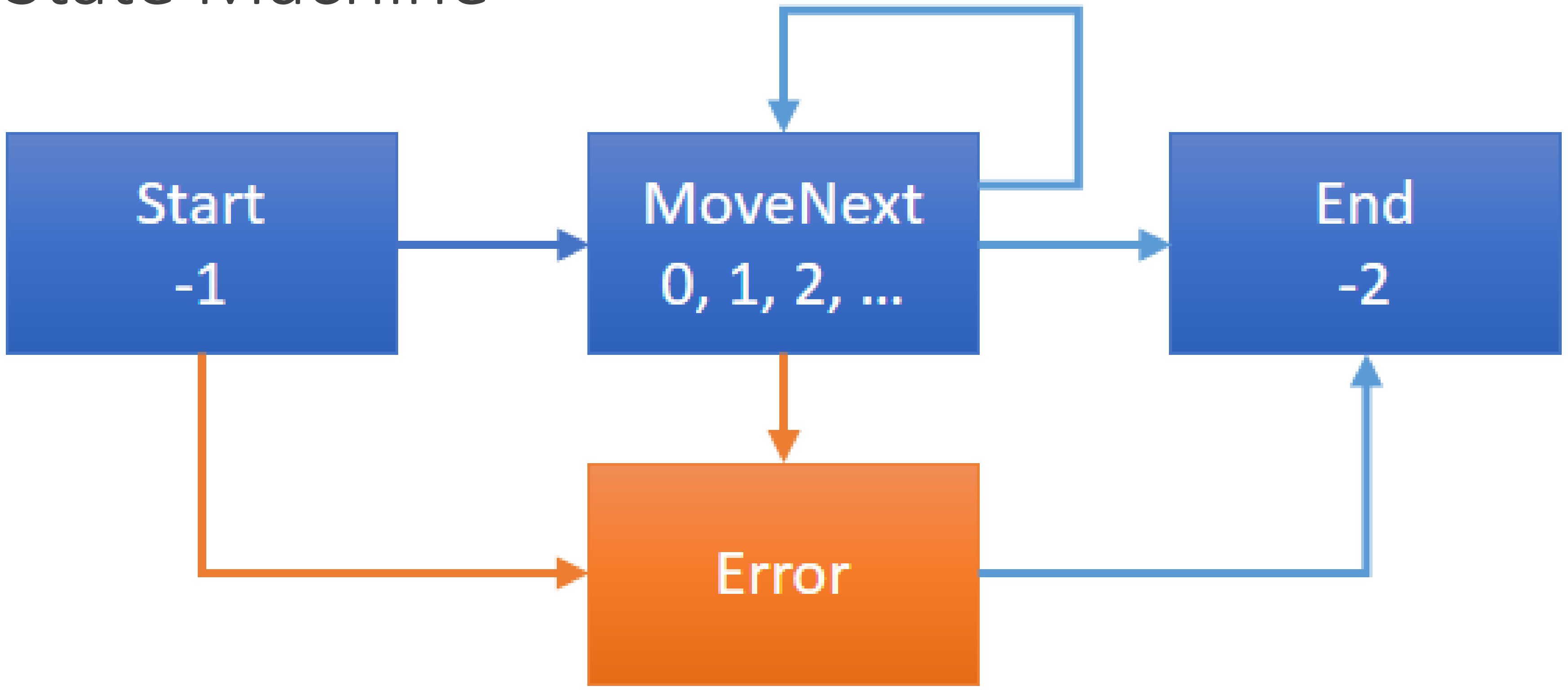
With **async**/**await**:

```
await ComputeHeavyOpAsync(1);
await ComputeHeavyOpAsync(2);
await ComputeHeavyOpAsync(3);
```


StartButton_Click event handler
1
2
3
4
5
6
7
8
Task<string> HttpClient.GetStringAsync(string url))
Normal processing
Yielding control to caller at an await
Resuming a suspended process

```
async Task<int> AccessTheWebAsync()
{
    HttpClient client = new HttpClient();

    Task<string> getStringTask = client.GetStringAsync("http://msdn.microsoft.com");

    DoIndependentWork();

    string urlContents = await getStringTask;

    return urlContents.Length;
}

void DoIndependentWork()
{
    resultsTextBox.Text += "Working . . . . . . .\r\n";
}
```

## State Machine



## Asynchronous functions

C# also supports **void** return type for **async** functions. It can be helpful when you want to create an **async** event handler.

```
async void Form_Load(object sender, EventArgs e) {
    await OnFormLoadAsync(sender, e);
}
```

You can also use **async** lambdas:

```
Task.Run(async () => {
    await DoSomeWorkAsync();
});
```

## Task.WhenAll and Task.WhenAny

**`Task.WhenAll`** waits for all tasks to complete:

```
var tasks = from i in Enumerable.Range(0,100)
    select SomeTask(i);
TResult[] resultsArray = await Task.WhenAll(tasks);
```

**Task.WhenAny** waits for the first task to complete:

```
var tasks = from i in Enumerable.Range(0,100)
    select SomeTask(i);
Task<TResult> firstFinished = await Task.WhenAny(tasks);
```

# async/await limitations

Asynchronous functions have some limitations:

- Constructors, property accessors and event accessors cannot be **`async`**.
- Cannot have any **`out`** or **`ref`** parameters.
- Cannot use **`await`** inside **`unsafe`** block.
- You cannot obtain a **`lock`** before **`await`**, and release it after **`await`**.

## Questions

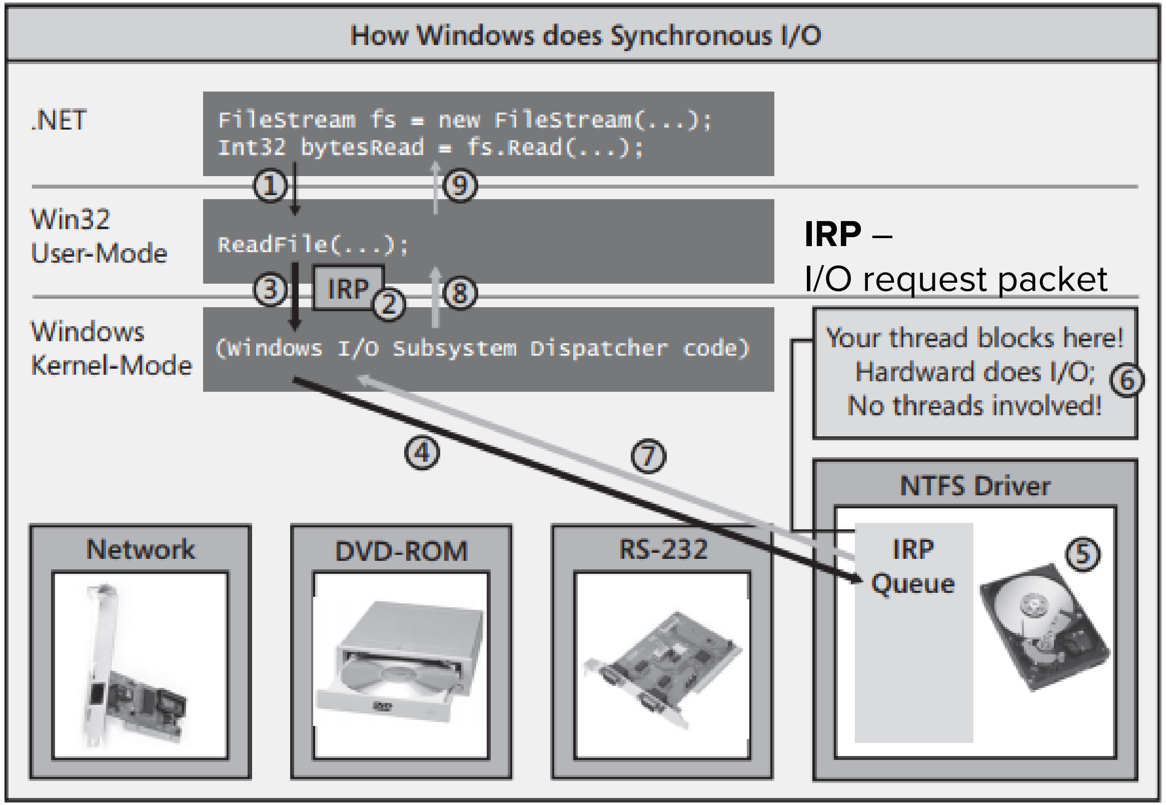
- Does using the “**`async`**” keyword on a method force all invocations of that method to be asynchronous?
- Can I mark any method as “**`async`**”?
- In which cases can we use **`void`** return type in **`async`** functions?
- Is “**`await task;`**” the same thing as “**`task.Wait()`**”?

## Asynchronous operations

There are two major types of asynchronous operations:

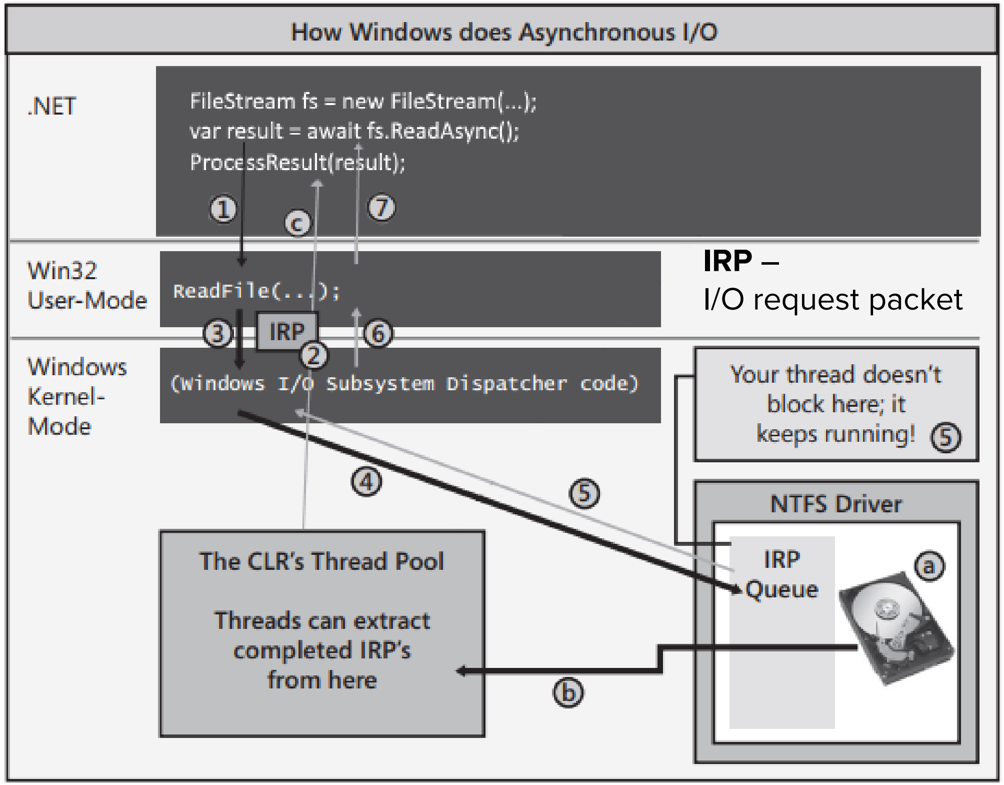
- I/O-bound operations
- CPU-bound operations

# I/O-Bound Operations



**IRP** –
I/O request packet

## I/O-Bound Operations


How Windows does Asynchronous I/O
.NET
FileStream fs = new FileStream(...);
var result = await fs.ReadAsync();
ProcessResult(result);
①
ⓒ
⑦
Win32 User-Mode
ReadFile(...);
③
IRP
②
⑥
Windows Kernel-Mode
(Windows I/O Subsystem Dispatcher code)
Your thread doesn't block here; it keeps running! ⑤
④
⑤
NTFS Driver
IRP Queue
ⓐ
The CLR's Thread Pool
Threads can extract completed IRP's from here
ⓑ

**IRP** –
I/O request packet

I/O result is
awaited with IOCP

# Benefits

Because there are no threads dedicated to blocking on unfinished tasks, the server thread pool can service a much higher volume of requests.

Because I/O-bound work spends virtually no time on the CPU, dedicating an entire CPU thread to perform barely any useful work would be a poor use of resources.

## CPU-Bound Operation

CPU-bound async code is a bit different than I/O-bound async code. Because the work is done on the CPU, there's **no way to get around dedicating a thread** to the computation.

The use of **`async`**/**`await`** provides you with a clean way to interact with a background thread and keep the caller of the async method responsive.

Note that this does not provide any protection for shared data. If you are using shared data, you will still need to **apply an appropriate synchronization strategy**.

# Question

- What are the benefit of using **`async`/`await`** with:
  - I/O-bound operations?
  - CPU-bound operations?

## Awaitable/Awaiter pattern

Requirements to the awaitable types:

- It has a **`GetAwaiter()`** method (instance method or extension method);
- Its **`GetAwaiter()`** method returns an awaiter. An object is an awaiter if:
  - It implements **`INotifyCompletion`** or **`ICriticalNotifyCompletion`** interface;
  - It has an **`IsCompleted`**, which has a getter and returns a Boolean;
  - it has a **`GetResult()`** method, which returns void, or a result.

C# supports both **`GetAwaiter()`** *instance* method and *extension* method.

# Async ASP.NET MVC actions

When a request arrives, a thread from the pool is dispatched to process that request.

If the request is processed synchronously, the thread that processes the request is busy while the request is being processed, and that thread cannot service another request.

An asynchronous request takes the **same amount of time** to process as a synchronous request. However, a **thread is not blocked** from responding to other requests.

Asynchronous requests prevent **request queuing** and **thread pool growth** when there are many concurrent requests that invoke long-running operations.

# Async ASP.NET MVC actions

```
public async Task<ActionResult> GizmosAsync()
{
    ViewBag.SyncOrAsync = "Asynchronous";
    var gizmoService = new GizmoService();
    return View("Gizmos", await gizmoService.GetGizmosAsync());
}
```

Use asynchronous actions for the following conditions:

- The operations are **network-bound** or **I/O-bound** instead of CPU-bound.
- Parallelism is more important than simplicity of code.

## Questions

- How **async** actions helps to increase throughput of the system?