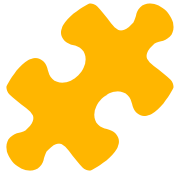
C# Asynchronous Programming Using async and await

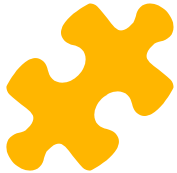
Created by : Umang Dobariya

Synchronous Programming



- Synchronous Nature.
- Line by Line Execution.
- Execution wait for function to return.
- When function takes more time.
- Worst case scenario: Downloading or reading a large file.

Issues with Synchronous Programming



- Client wants UI to be responsive.
- Badly impacts the UI.
- Behaviour leaves end users with bad user experience.

Asynchronous Programming

- Parallel Programming.

Process of executing function:

1. You create the thread.
2. The thread starts executing separately.
3. An asynchronous method call (creation of a thread) will return Immediately.
4. Main and new thread work in parallel.

- **Three Patterns :**
  1. Task based Asynchronous Programming Model
  1. Event based Asynchronous Programming Model
  1. Asynchronous Programming Model

Synchronous

```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");
Egg eggs = FryEggs(2);
Console.WriteLine("eggs are ready");
Bacon bacon = FryBacon(3);
Console.WriteLine("bacon is ready");
Toast toast = ToastBread(2);
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("Juice is ready");
Console.WriteLine("Breakfast is ready!");
```

Asynchronous

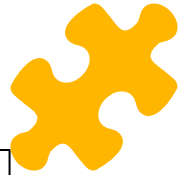
```
Coffee cup = PourCoffee();
Console.WriteLine("coffee is ready");
Task<Egg> eggTask = FryEggs(2);
Task<Bacon> baconTask = FryBacon(3);
Task<Toast> toastTask = ToastBread(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("toast is ready");
Juice oj = PourOJ();
Console.WriteLine("Juice is ready");

Egg eggs = await eggTask;
Console.WriteLine("eggs are ready");
Bacon bacon = await baconTask;
Console.WriteLine("bacon is ready");
Console.WriteLine("Breakfast is ready!");
```

Synchronous Output

```
coffee is ready
eggs are ready
bacon is ready
toast is ready
Juice is ready
Breakfast is ready
Elapsed Time 21.4355
```

Asynchronous Output



```
coffee is ready
toast is ready
Juice is ready
eggs are ready
bacon is ready
Breakfast is ready!
Elapsed Time 16.0543
```

Before .NET 4.5

Asynchronous Delegates

- **Delegate** methods are synchronous when invoked using “normal” syntax.
- Delegate types are automatically equipped with **BeginInvoke**() and **EndInvoke**().

```
SomeDelegate delegate = new SomeDelegate(DoWork);
int result = delegate.Invoke(x);
int result = delegate(x); //Normal syntax
```

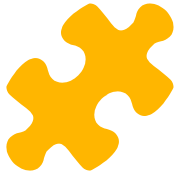
IAsyncResult Interface

- **BeginInvoke** will always return an **IAsyncResult** object.
- **EndInvoke** requires an **IAsyncResult** as its only parameter.

```
SomeDelegate delegate = new SomeDelegate(DoWork);

IAsyncResult ar = delegate.BegingInvoke(x, void, void)
result = delegate.EndInvoke(ar);
```

Async Callbacks



- AsyncCallback will call the specified method when async call is done.

- Useful for avoiding blocking of calling thread

```
  IAsyncResult ar = d.BeginInvoke(x, AsyncCallback(WorkCompleted), void);
}
static void WorkCompleted(IAsyncResult iar)
{
    AsynResult ar = (AsynResult) iar;
    SomeDelegate d = (SomeDelegate) ar.AsyncDelegate;
    Console.WriteLine(d.EndInvoke(iar));
}
```

Using async delegates

```
public delegate string SomeDelegate(string x);
public void Main(string[] args)
{
    SomeDelegate delegate = new SomeDelegate(DoWork)
    IAsyncResult ar = d.BeginInvoke(x, AsyncCallback(WorkCompleted), void);
    Console.WriteLine("Completed");
    Console.ReadLine();
}
static string DoWork(string x)
{
  Thread.Sleep(5_000);
  return "Done with work!";
}
static void WorkCompleted(IAsyncResult iar)
{
    AsynResult ar = (AsynResult) iar;
    SomeDelegate d = (SomeDelegate) ar.AsyncDelegate;
    Console.WriteLine(d.EndInvoke(iar));
}
```

Using async keyword

```
static async Task Main(string[] args)
{
  Console.WriteLine(DoWork());
  Console.WriteLine("Completed");
  Console.ReadLine();
}
static string DoWork()
{
  Thread.Sleep(5_000);
  return "Done with work!";
}
```

Asynchronous Calls with the Async Keyword

Async/Await VS Former Threading Procedures

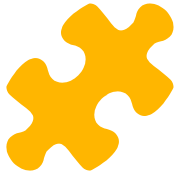
- For **async/await** keywords, compiler generates significant threading code in background for you
  - Mitigate burden of writing complex threading code
  - Uses parts of **System.Threading** and **System.Threading.Tasks** namespaces

Async Keyword

- **async** keyword assures a method, lambda expression, or anonymous method will be invoked asynchronously
- CLR creates new thread of execution

```
static async Task Main(string[] args)
{
  Console.WriteLine(" Fun With Async ===
  //This is to prompt Visual Studio to upgra
  List<int> l = default;
  Console.WriteLine(DoWork());
  Console.WriteLine("Completed");
  Console.ReadLine();
}
static string DoWork()
{
  Thread.Sleep(5_000);
  return "Done with work!";
}
```

Await Keyword



- **await** pauses current thread until task is complete
- Calling thread free to operate as such
- Async function without await => synchronous

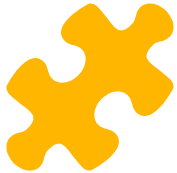
```
static async Task Main(string[] args)
{
  //ommitted for brevity
  string message = await DoWorkAsync();
  Console.WriteLine(message);
  //ommitted for brevity
}
static string DoWork()
{
  Thread.Sleep(5_000);
  return "Done with work!";
}
static async Task<string> DoWorkAsync()
{
  return await Task.Run(() =>
  {
    Thread.Sleep(5_000);
    return "Done with work!";
  });
}
```

Await Keyword (cont.)

- Async functions return **Task<T>**
  - Desired return type is the underlying T
- Await extracts internal return value inside Task object

```
static async Task Main(string[] args)
{
  //ommitted for brevity
  string message = await DoWorkAsync();
  Console.WriteLine(message);
  //ommitted for brevity
}
static string DoWork()
{
  Thread.Sleep(5_000);
  return "Done with work!";
}
static async Task<string> DoWorkAsync()
{
  return await Task.Run(() =>
  {
    Thread.Sleep(5_000);
    return "Done with work!";
  });
}
```

Naming Conventions

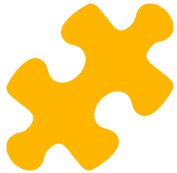


- Provide method names with “**Async**” suffix
- For example,

  string message = DoWork()

```
static async Task<string> DoWork()
{
        return await Task.Run(() =>
        {
                Thread.Sleep(5_000);
                return "Done with work!";
        });
}
```

Async Methods Returning Void



- Use **non-generic Task** class and exclude any return statements

```
static async Task MethodReturningVoidAsync()
{
  await Task.Run(() => { /* Do some work here... */
                         Thread.Sleep(4_000);
                       });
  Console.WriteLine("Void method completed");
}
```

Async Methods with Multiple Awaits

- Multiple **await** statements can be placed in a single async method

```
static async Task MultiAwaits()
{
    await Task.Run(() => { Thread.Sleep(2_000); });
    Console.WriteLine("Done with first task!");
    await Task.Run(() => { Thread.Sleep(2_000); });
    Console.WriteLine("Done with second task!");
    await Task.Run(() => { Thread.Sleep(2_000); });
    Console.WriteLine("Done with third task!");
}
```

Calling Async Methods from Non-Async Methods

- Invoke **async** method without **await**?
  - Call **Result** on method
  - **Result** is a property of **Task**
  - For void async method, call **Wait()** on the Task

```
Console.WriteLine(DoWorkAsync().Result)

MethodReturningVoidAsync().Wait()
```

Await in catch and finally Blocks

- Await calls can be placed in catch and finally blocks
- Method must be async

```
static async Task<string> MethodWithTryCatch()
{
 try
 {
  //Do some work
  return "Hello";
 }
 catch (Exception ex)
 {
  await LogTheErrors();
  throw;
 }
 finally
 {
  await DoMagicCleanUp();
 }
}
```

Generalized Async Return Types

- Before C# 7, only permissible return types were Task, Task<T>, and void
  - C#7 enables more variety in return types, as long as async pattern is followed
  - Example: ValueTask from System.Threading.Tasks.Extensions NuGet package

ValueTask

- Similar to Task<T>
  - Task for value type

```
static async Task<int> CalculateSum(int a, int b)
{
    if (a == 0 && b == 0) { return 0; }
    return await Task.Run(() => a + b);
}
1 reference
static async ValueTask<int> CalculateSum2(int a, int b)
{
    if (a == 0 && b == 0) { return 0; }
    return await Task.Run(() => a + b);
}
```

Local Functions

- From C# 7.0
- Can be useful to check an asynchronous function's input data

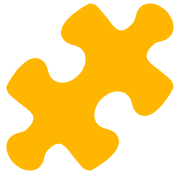
```
static async Task MethodWithProblems(int firstParam, int secondParam)
{
  Console.WriteLine("Enter");
  await Task.Run(() =>
  {
    //Call long running method
    Thread.Sleep(4_000);
    Console.WriteLine("First Complete");
    //Call another long running method that fails because
    //the second parameter is out of range
    Console.WriteLine("Something bad happened");
  });
}
```

Local Functions (cont.)

```
static async Task MethodWithProblemsFixed(int firstParam, int secondParam)
{
  Console.WriteLine("Enter");
  if (secondParam < 0)
  {
    Console.WriteLine("Bad data");
    return;
  }
  actualImplementation();
  async Task actualImplementation()
  {
    await Task.Run(() =>
    {
      //Call long running method
      Thread.Sleep(4_000);
      Console.WriteLine("First Complete");
      //Call another long running method that fails because
      //the second parameter is out of range
      Console.WriteLine("Something bad happened");
    });
  }
}
```

Asynchronous Programming in Javascript

About Asynchronous Programming in Javascript



- Javascript supports asynchronous programming through callbacks and promises.
- Like C# Javascript supports the async-await syntax

Callbacks

- The original way to handle asynchronous situations in Javascript
- A callback is when a function executes after another function finishes
- In Javascript, functions are objects

```
function doHomework(subject, callback) {
  alert('Starting my ${subject} homework.');
  callback();
}

doHomework('math', function() {
 alert('Finished my homework');
});
```

Promises

- Readability and Control
- Promises have four states: fulfilled, rejected,pending and settled
- Helps avoid unintentional chained “callback hell” situations.

```
var promise = new Promise(function(resolve, reject) {
  // do a thing, possibly async, then…

  if (/* everything turned out fine */) {
    resolve("Stuff worked!");
  }
  else {
    reject(Error("It broke"));
  }
});
```

Async/Await syntax in Javascript

- Most Manageable way to handle asynchronous Javascript
- The latest in a series of readability improvements

```
async function f(){

  let promise = new Promise((resolve, reject)=>{
      setTimeout(()=>resolve("done!"),1000);  });

  // wait till the promise resolves (*)
  let result = await promise;

  alert(result);// "done!"
}

f();
```

Summary

Difference between async and await in C# and Javascript

Differences	**C#**	**Javascript**
Underlying	Task	Promise
Use cases	I/O & CPU-bound tasks	I/O tasks
Multi-thread	Use Task.Run	single-threaded; Promise doesn't help with CPU-bound tasks
Asynchronous pattern	Task-based Asynchronous Pattern	Event-based Asynchronous Pattern

Key Points (1/3)

- Methods can be marked with the **async** keyword:
  - **nonblocking** manner.
  - multiple await contexts.
  - run synchronously until we see the **await** keyword.

Key Points (2/3)

- The **await** keyword:
  - the calling thread stop until the awaited task is complete.
  - hide the returned **Task** object from view.

```
public static async void Main()
{
    string message = await DoWorkAsync();
    Console.WriteLine(message);
}
public static async Task<string> DoWorkAsync(){
    return await Task.Run(()=>{return "Done";});
}
```

Key Points (3/3)

- If we have stack variables, using **ValueTask** is more efficient than **Task** object.
- Naming convention of async method.

Thanks!