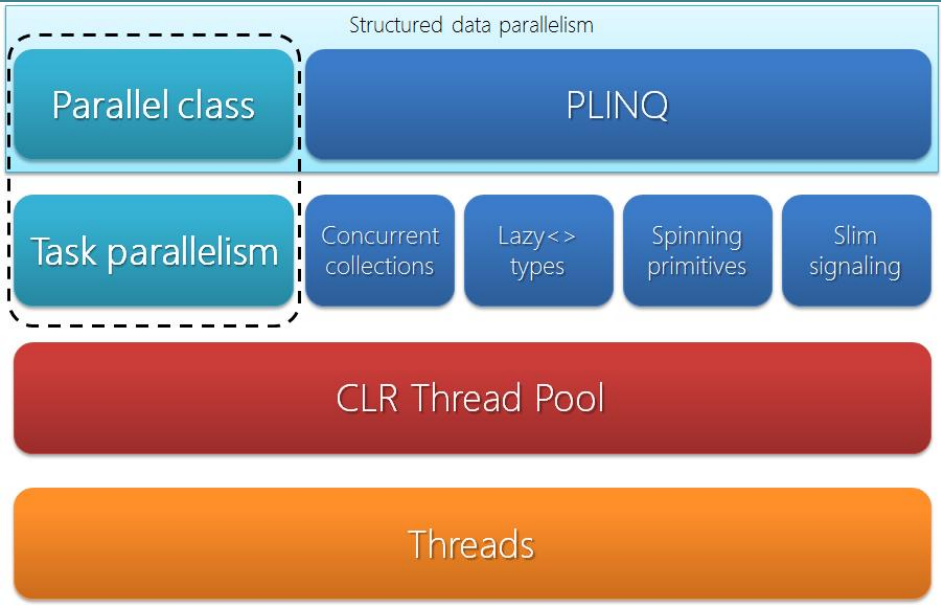


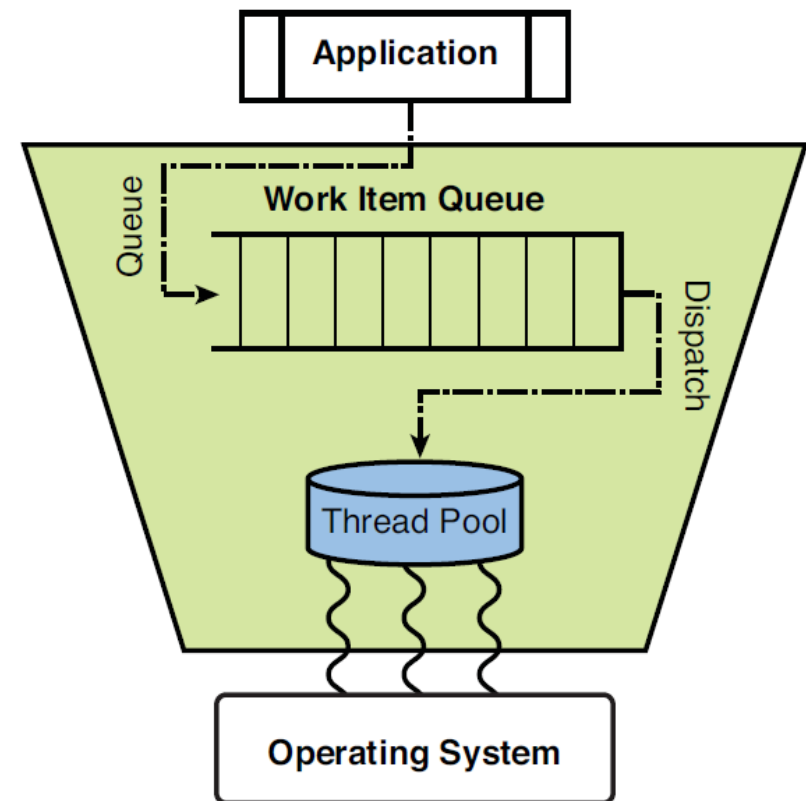
Tasks & Co

Software Entwicklung

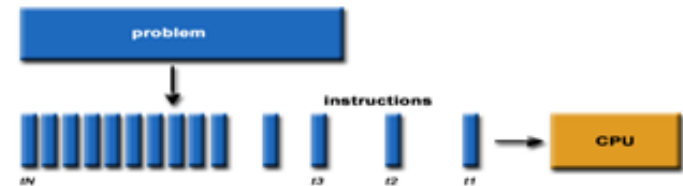


Overview

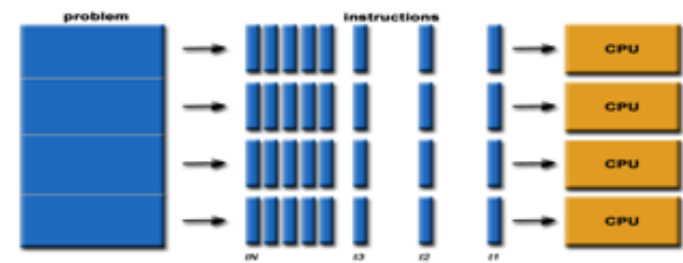
- Concurrent vs Parallel
- Threadpool (TPL)
- Task & Parallel
- Async Await



Traditional Sequential Processing



Parallel Processing

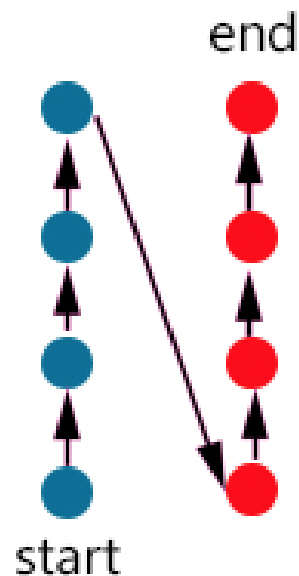


Concurrent vs Parallel?

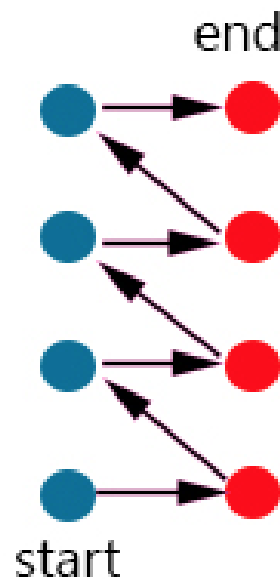
Whats the difference?

Sequential - Concurrent - Parallel

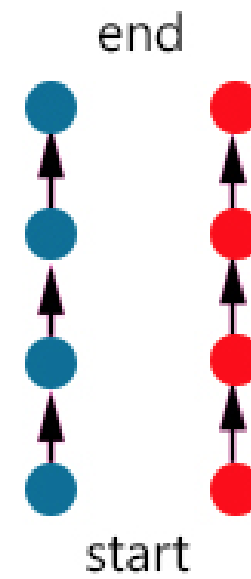
Sequential



Concurrent



Parallel

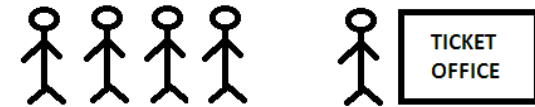


multiple threads which are then
executed on multiple CPUs

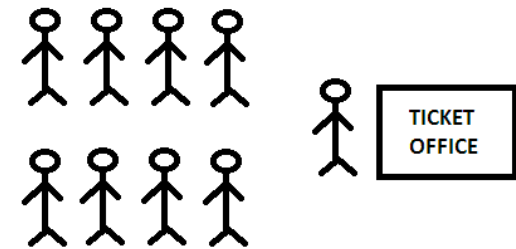
Multithreaded Process

- single processor
 - Processor can switch execution resources between threads -> concurrent execution
- shared-memory multiprocessor
 - each thread in the process can run on a separate processor at the same time -> parallel execution

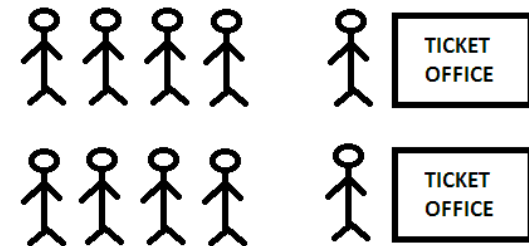
NEITHER PARALLEL NOR CONCURRENT



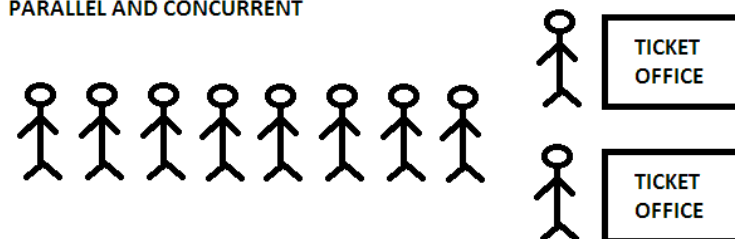
CONCURRENT, NOT PARALLEL

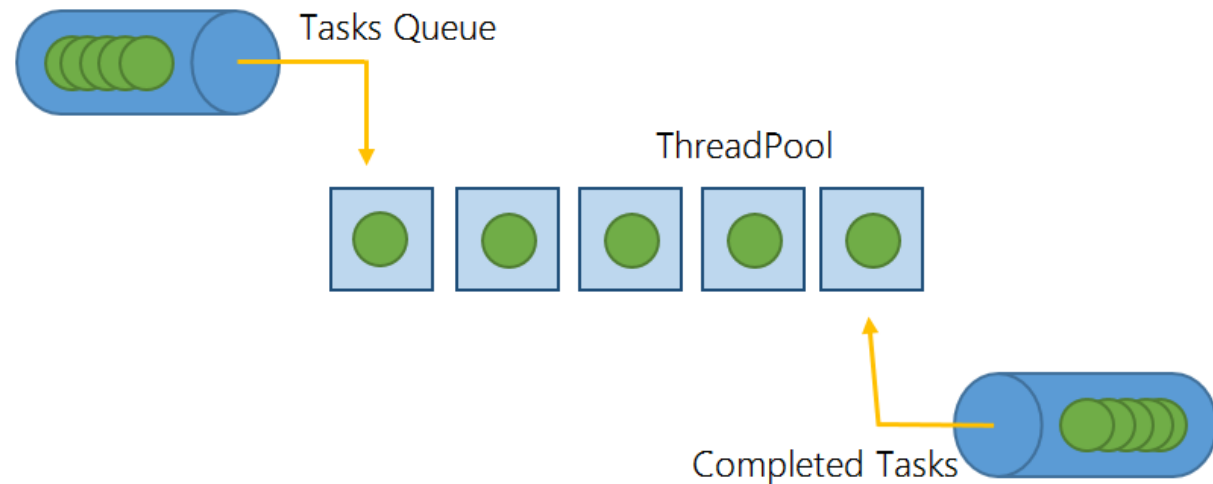


PARALLEL, NOT CONCURRENT



PARALLEL AND CONCURRENT



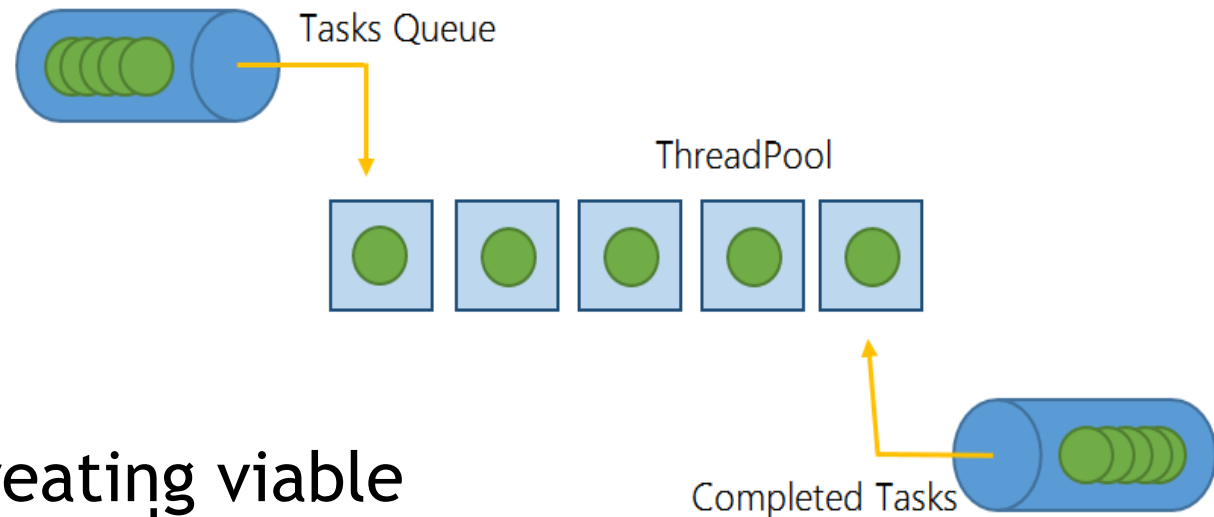


Threadpool

Thread pooling in C# is **the process of creating a collection of threads during the initialization of a multithreaded application**

It enables applications to use threads more efficiently by providing their application with a pool of worker threads that are managed by the system. When a request comes, then it directly goes to the thread pool and checks whether there are any threads available for execution.

What is „Thread-Pooling“



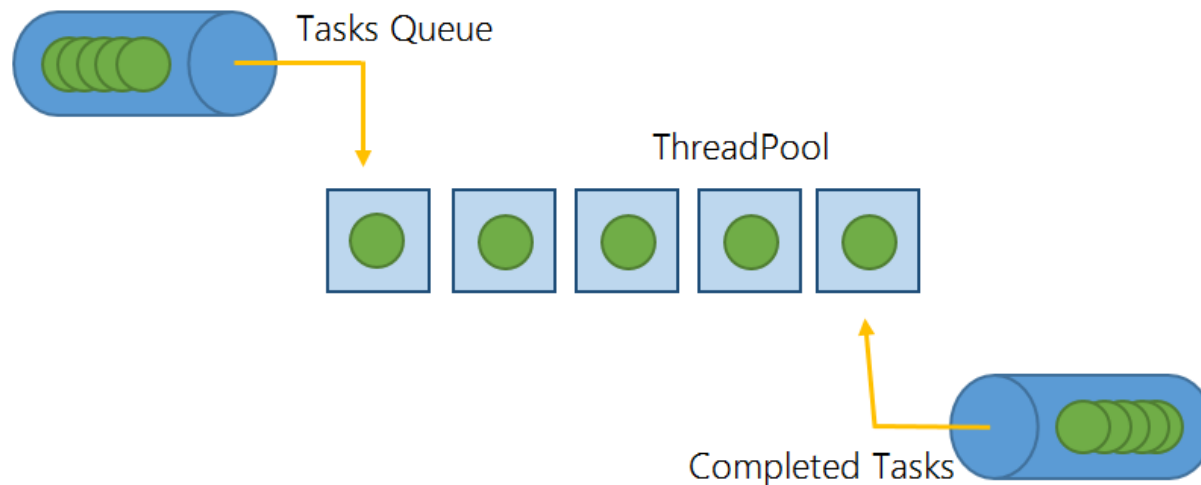
- process of creating viable amount of Threads
- these created Threads are then reused
- every Task gets enqueued

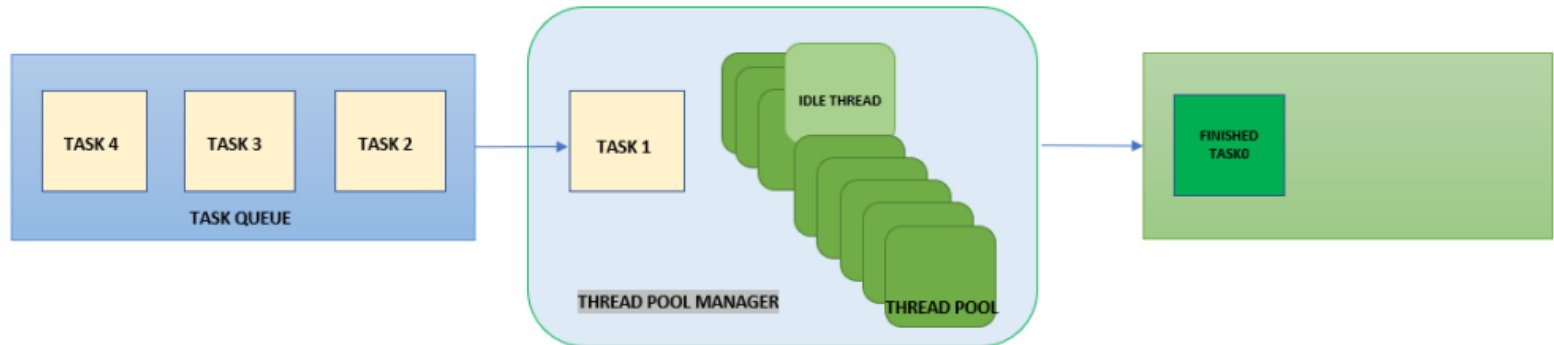
Why use Thread-Pooling

- Creating a new thread
 - takes a few hundred microseconds
 - takes around 1MB of memory and needs to be cleaned up after finishing
- Thread-Pooling
 - only creates Threads once and then reuses them
 - automatically processes the most efficient amount of Tasks simultaneously

Using the ThreadPool

- Ways to enter the Thread Pool
 - Via the Task Parallel Library
 - By calling `ThreadPool.QueueUserWorkItem`
 - Via asynchronous delegates
 - ...





Task & Task<TResult>

Promises the execution in the future → not immediately

Doesn't create his own OS-Thread

TaskScheduler handles it

Control: Finish Return Wait

Implementing Task vs Threads

- Thread

```
static void Main(string[] args)
{
    Thread thread = new Thread(obj => { Thread.Sleep(3000); });
    thread.Start();
}
```

- Task

```
static void Main(string[] args)
{
    Task<int> task = new Task<int>(() =>
    {
        Thread.Sleep(3000);
        return 1;
    });
    task.Start();
}
```

MULTI-THREADING

Die Task-Klasse

- Task can be instantiated by calling a **Task** class constructor, it can be started by calling its **Start()** method.
- Task can be instantiated and started in a single method call by calling the **TaskFactory.StartNew(Action<Object>, Object)** method.
- Task can be instantiated and started in a single method call by calling the static **Task.Run(Action)** method.

<https://www.youtube.com/watch?v=YT-Ror9f6NE>

Instantiating Task - V1

- Version 1 - with constructor

without parameter

```
Action action = () =>
{
    Console.WriteLine(
        "Task={0}, Thread={1}",
        Task.CurrentId,
        Thread.CurrentThread.ManagedThreadId);
};

// Create a task but do not start it.
Task t1 = new Task(action);
```

with parameter

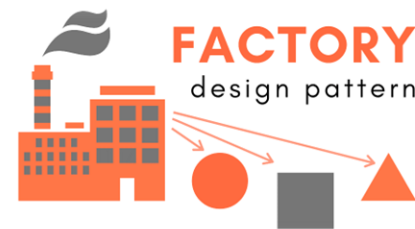
```
Action<object> action = (object obj) =>
{
    Console.WriteLine(
        "Task={0}, obj={1}, Thread={2}",
        Task.CurrentId, obj,
        Thread.CurrentThread.ManagedThreadId);
};

// Create a task but do not start it.
Task t1 = new Task(action, "Hello You!");
```

- Starting: t1.Start();
- Synchronizing: t1.Wait();

Instantiating & Starting Task - V2

- Version 2 - TaskFactory
- Factory is a Design Pattern for creating instances



without parameter

```
Action action = () =>
{
    Console.WriteLine(
        "Task={0}, Thread={1}",
        Task.CurrentId,
        Thread.CurrentThread.ManagedThreadId);
};

// Create a task and start it.
Task t2 = Task.Factory.StartNew(action);
```

with parameter

```
Action<object> action = (object obj) =>
{
    Console.WriteLine(
        "Task={0}, obj={1}, Thread={2}",
        Task.CurrentId, obj,
        Thread.CurrentThread.ManagedThreadId);
};

// Create a task with parameter and start it.
Task t2 = Task.Factory.StartNew(action, "Hello You!");
```

- Synchronizing: t2.Wait();

Instantiating with static Method Run

- Variante 3 - Task.Run(action)
- No explicit parameter possible
- but Lambda Expression takes parameter

```
// Construct and start a task using Task.Run
String taskData = "My Data";

Task t3 = Task.Run( () => {
    Console.WriteLine(
        "Task={0}, obj={1}, Thread={2}",
        Task.CurrentId, taskData,
        Thread.CurrentThread.ManagedThreadId);
});
```

Different Wait-Methods

Two ways for explicitly waiting for a task to complete:

- Calling its **Wait** method optionally with a **timeout**
- Accessing its **Result** property in the case of `Task<TResult>`

Wait on multiple tasks at once via

Task.WaitAll and **Task.WaitAny**

- **Wait()**
 - Waits for one task to continue
- **Wait(<valueInMs>)**
 - Waits the number of ms to continue
- **WaitAny(<nameOfTasksArray>)**
 - Waits for the first task to continue
- **WaitAll(<nameOfTasksArray>)**
 - Waits for all the tasks to continue



Main Task

```
task1.Wait(1000)
```

```
Task.WaitAny( new [] { task1, task2, task3 } )
```

```
Task.WaitAll( new [] { task1, task2, task3 } )
```


Task vs Threads

- Task uses the Threadpool
 - Tasks can return a result
 - Tasks support Cancellation
 - Tasks can wait for a group of tasks
 - Tasks can ContinueWith another Task
 - Tasks can catch Exceptions in the parent method
-
- Tasks maintain the async await concept
 - > next topic

Return Value

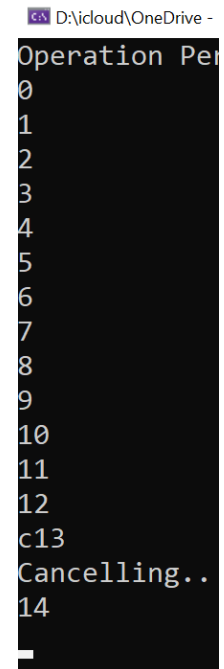
- Task has no return value
- Task<TResult> has a return value
 - Property Result:
contains the result of the calculation

```
static void Main(string[] args)
{
    Task<int> task = new Task<int>(LongRunningTask);
    task.Start();
    Console.WriteLine(task.Result);
}
private static int LongRunningTask()
{
    Thread.Sleep(3000);
    return 1;
}
```

Cancellation

- Tasks support Cancellation, Threads not...

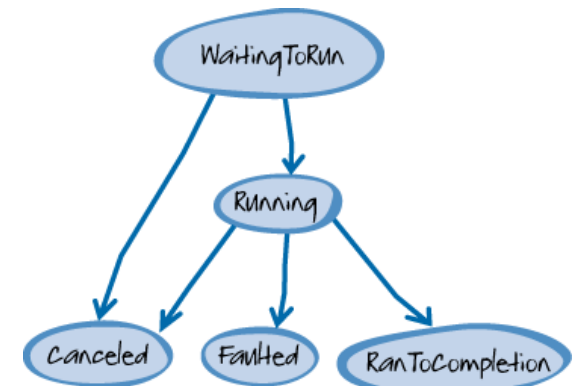
```
static void Main(string[] args) {  
    using (var cts = new CancellationTokenSource()) {  
        Task task = new Task(() => { LongRunningTask(cts.Token); });  
        task.Start();  
        Console.WriteLine("Operation Performing...");  
        if (Console.ReadKey().Key == ConsoleKey.C) {  
            Console.WriteLine("Cancelling..");  
            cts.Cancel();  
        }  
        Console.Read();  
    }  
}  
private static void LongRunningTask(CancellationToken token) {  
    for (int i = 0; i < 100000000; i++)  
        if (token.IsCancellationRequested) {  
            break;  
        } else {  
            Console.WriteLine(i);  
        }  
}
```



```
D:\icloud\OneDrive -  
Operation Per  
0  
1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
c13  
Cancelling..  
14
```

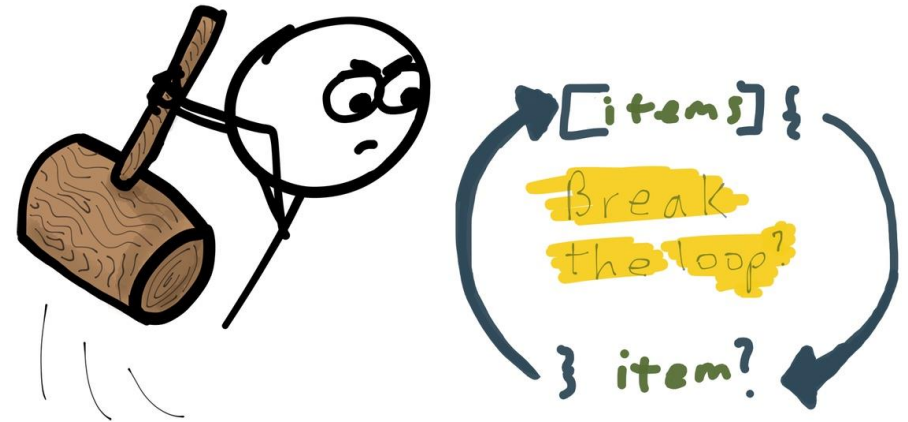
Status Property

- You can ask for the status all the time
 - Canceled
 - Created
 - Faulted
 - RanToCompletion
 - Running
 - WaitingForActivation
 - WaitingForChildrenToComplete
 - WaitingToRun



```
Task t1 = new Task(() => Console.WriteLine("hello"));
Console.WriteLine(t1.Status);
```

Created



Class Parallel

Parallel.Invoke

Parallel.Foreach

Parallel.For

Class Parallel

- **Parallel.Invoke()**

- Creates and runs tasks in one operation
- Has an integrated WaitAll()
- Syntax: public static void Invoke (params Action[] actions);
- Parameters: Methods, Lambda Expressions, anonymous Methods

Parallel.Invoke(params Action[] actions)

Parallel.Invoke(IEnumerable<Action> actions)

Parallel.Invoke(ParallelOptions parallelOptions, params Action[] actions)

Parallel.Invoke(ParallelOptions parallelOptions, IEnumerable<Action> actions)

Use Invoke

- With Methods, Lambda Expressions, anonymous Methods

Output:

Method=gamma, Thread=5
Method=alpha, Thread=1
Method=beta, Thread=8

```
Parallel.Invoke(  
    BasicAction,  
    () => {  
        Console.WriteLine("Method=beta, Thread={0}",  
            Thread.CurrentThread.ManagedThreadId);},  
    delegate () {  
        Console.WriteLine("Method=gamma, Thread={0}",  
            Thread.CurrentThread.ManagedThreadId);});  
  
static void BasicAction() {  
    Console.WriteLine("Method=alpha, Thread={0}",  
        Thread.CurrentThread.ManagedThreadId);  
}
```

Parallel Loops

- **Parallel.For()**

- Iterates numbers in the specified range and makes an operation for all the itmes
- `public static ParallelLoopResult For (int fromInclusive, int toExclusive, Action<int> body)`

- **Parallel.Foreach()**

- Iterates a list and makes an operation with the values
- `public static ParallelLoopResult ForEach<TSource> (IEnumerable<TSource> source, Action<TSource> body)`

Parallel For

6,4,0,1,3,5,2,7,8,9,

- Use the `Parallel.For` to print numbers between 0..9

```
static void Main(string[] args) {  
    int[] numbers = new int[10];  
    for (int i = 0; i < numbers.Length; i++) {  
        numbers[i] = i + 1;  
    }  
    Parallel.For(0, numbers.Length, i => {  
        Console.Write(i + ",");  
    });  
}
```

Parallel Foreach

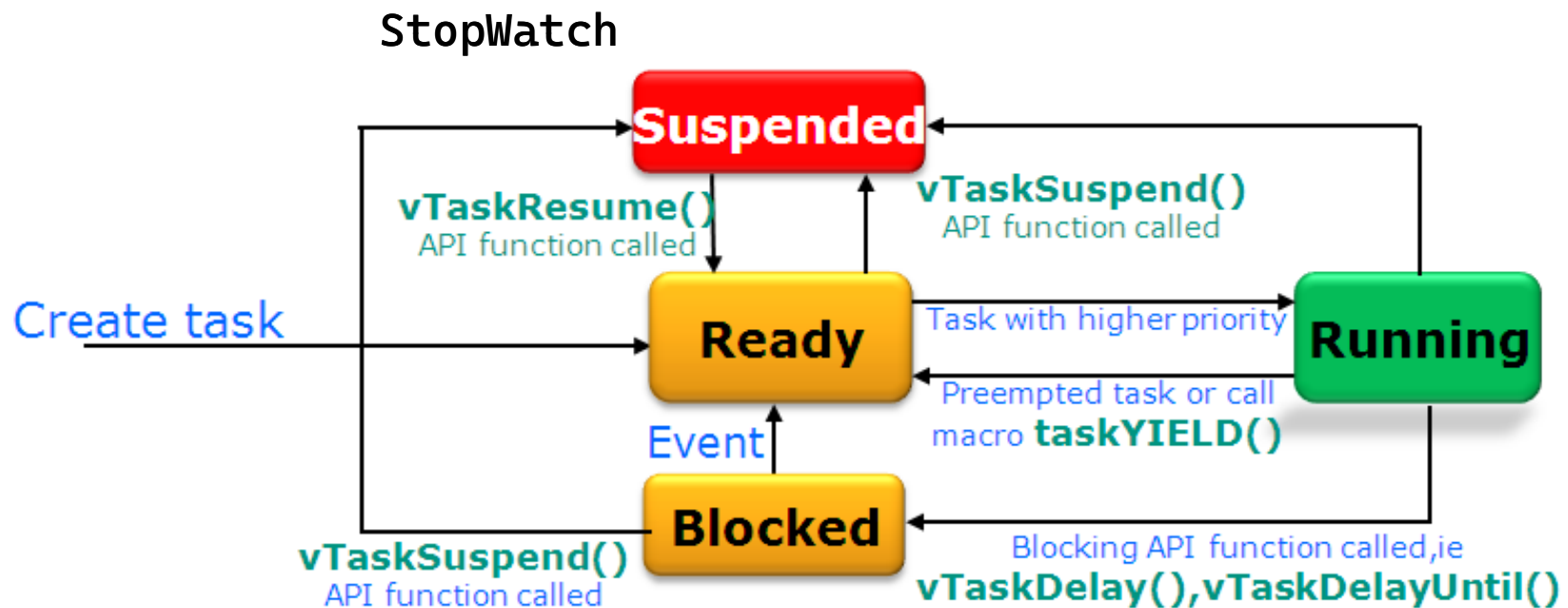
- Test the ForEach with with a simple List<string>
- See which thread does the work

```
internal class Program {  
    static void Main(string[] args) {  
        List<string> actionlist = new List<string> { };  
        for (int i = 0; i < 100; i++)  
            actionlist.Add("Action" + i);  
  
        Parallel.ForEach(actionlist, testaction =>  
        {  
            Console.WriteLine($"{testaction} runs by Thread " +  
                               $"{Thread.CurrentThread.ManagedThreadId}");  
        });  
    }  
}
```

```
Action25 runs by Thread 7  
Action0 runs by Thread 1  
Action50 runs by Thread 6  
Action51 runs by Thread 6  
Action52 runs by Thread 6  
Action53 runs by Thread 6  
Action54 runs by Thread 6  
Action55 runs by Thread 6  
Action56 runs by Thread 6  
Action57 runs by Thread 6  
Action58 runs by Thread 6  
Action59 runs by Thread 6  
Action60 runs by Thread 6  
Action61 runs by Thread 6  
Action62 runs by Thread 6  
Action1 runs by Thread 1  
Action75 runs by Thread 8  
Action26 runs by Thread 7  
Action27 runs by Thread 7  
Action2 runs by Thread 1  
Action76 runs by Thread 8  
Action63 runs by Thread 6  
Action28 runs by Thread 7  
Action3 runs by Thread 1  
Action4 runs by Thread 1  
Action5 runs by Thread 1  
Action6 runs by Thread 1  
Action7 runs by Thread 1  
Action8 runs by Thread 1
```

Summary Class Task & Parallel

- These two classes make up the core of the API for multithreaded programming, referred to as parallel programming.



```
async method() {  
    await ...  
}
```

Async Await

Software Entwicklung





Asynchronous Coding

explained with making breakfast

Implement making breakfast



How to make a breakfast:

1. Pour a cup of coffee.
2. Heat a pan, then fry two eggs.
3. Fry three slices of bacon.
4. Toast two pieces of bread.
5. Add butter and jam to the toast.
6. Pour a glass of orange juice.

If you do each step after the other, it takes about 30 minutes...

Making Breakfast

```
internal class Bacon { }
internal class Coffee { }
internal class Egg { }
internal class Juice { }
internal class Toast { }

class BreakfastSync {
private static Juice PourOJ() {
    Console.WriteLine("Pouring orange juice");
    return new Juice();
}
private static void ApplyJam	Toast toast) =>
    Console.WriteLine("Putting jam on the toast");
private static void ApplyButter	Toast toast) =>
    Console.WriteLine("Putting butter on the toast");
private static Toast ToastBread(int slices) {
    for (int slice = 0; slice < slices; slice++) {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    Task.Delay(3000).Wait();
    Console.WriteLine("Remove toast from toaster");
    return new Toast();
}
```

```
class BreakfastSync {
{...}
private static Bacon FryBacon(int slices) {
    Console.WriteLine($"putting {slices} slices of bacon in the pan");
    Console.WriteLine("cooking first side of bacon...");
    Task.Delay(3000).Wait();
    for (int slice = 0; slice < slices; slice++) {
        Console.WriteLine("flipping a slice of bacon");
    }
    Console.WriteLine("cooking the second side of bacon...");
    Task.Delay(3000).Wait();
    Console.WriteLine("Put bacon on plate");

    return new Bacon();
}
private static Egg FryEggs(int howMany) {
    Console.WriteLine("Warming the egg pan...");
    Task.Delay(3000).Wait();
    Console.WriteLine($"cracking {howMany} eggs");
    Console.WriteLine("cooking the eggs ...");
    Task.Delay(3000).Wait();
    Console.WriteLine("Put eggs on plate");

    return new Egg();
}
private static Coffee PourCoffee() {
    Console.WriteLine("Pouring coffee");
    return new Coffee();
}
}
```

```
internal class Bacon { }
internal class Coffee { }
internal class Egg { }
internal class Juice { }
internal class Toast { }
```


Implementation Synchronioulsy

- synchroniously, working step after step ... every step takes time
- the total amount of time to make breakfast in C# takes 65 milliseconds

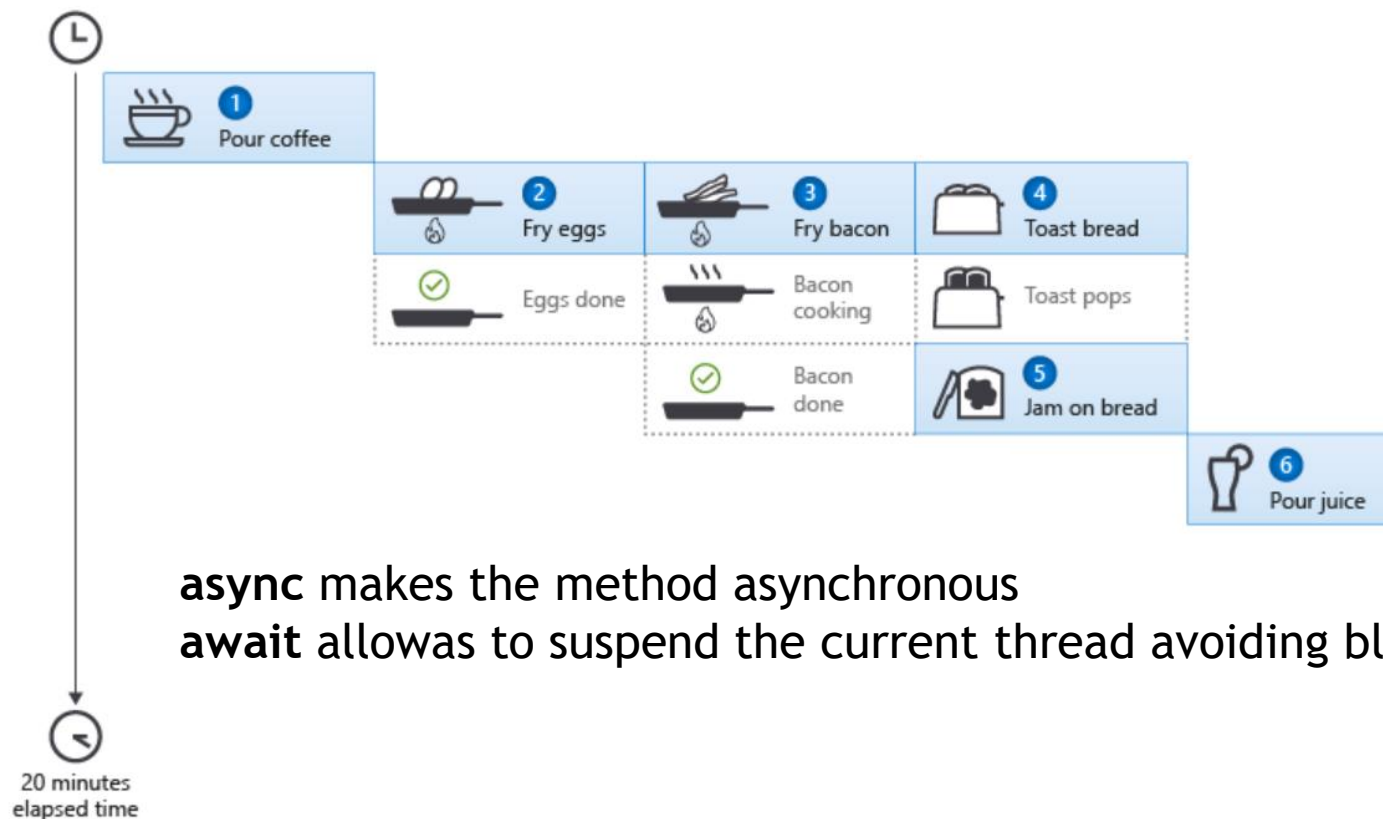
```
class BreakfastSync {  
static void Main(string[] args) {  
    Stopwatch sw = new Stopwatch();  
    sw.Start();  
    Coffee cup = PourCoffee();  
    Console.WriteLine("coffee is ready");  
  
    Egg eggs = FryEggs(2);  
    Console.WriteLine("eggs are ready");  
  
    Bacon bacon = FryBacon(3);  
    Console.WriteLine("bacon is ready");  
  
    Toast toast = ToastBread(2);  
    ApplyButter(toast);  
    ApplyJam(toast);  
    Console.WriteLine("toast is ready");  
  
    Juice oj = PourOJ();  
    Console.WriteLine("oj is ready");  
    sw.Stop();  
    Console.WriteLine($"Breakfast is ready after " +  
        $"{sw.Elapsed.Milliseconds}ms.");  
}
```

```
Pouring coffee  
coffee is ready  
Warming the egg pan...  
cracking 2 eggs  
cooking the eggs ...  
Put eggs on plate  
eggs are ready  
putting 3 slices of bacon in the pan  
cooking first side of bacon...  
flipping a slice of bacon  
flipping a slice of bacon  
flipping a slice of bacon  
cooking the second side of bacon...  
Put bacon on plate  
bacon is ready  
Putting a slice of bread in the toaster  
Putting a slice of bread in the toaster  
Start toasting...  
Remove toast from toaster  
Putting butter on the toast  
Putting jam on the toast  
toast is ready  
Pouring orange juice  
oj is ready  
Breakfast is ready after 128ms.
```

What would you do different in real life?!

Start tasks concurrently

- Fry eggs, fry bacon, toast bread at the same time



`async` makes the method asynchronous

`await` allows to suspend the current thread avoiding blocking it

Eggs with async & await

- Make the method async
- add the await keyword within

```
static async Task Main(string[] args) {  
    Stopwatch sw = new Stopwatch();  
    sw.Start();  
  
    Coffee cup = PourCoffee();  
    Console.WriteLine("coffee is ready");  
  
    Task<Egg> eggsTask = FryEggsAsync(2);  
    Console.WriteLine("eggs are ready");  
  
    ...  
    Egg eggs = await eggsTask;  
  
    sw.Stop();  
    Console.WriteLine($"Breakfast is ready after " +  
        $"{sw.Elapsed.Milliseconds}ms.");  
}
```

```
private static async Task<Egg> FryEggsAsync(int howMany) {  
    Console.WriteLine("Warming the egg pan...");  
    await Task.Delay(3000);  
    Console.WriteLine($"cracking {howMany} eggs");  
    Console.WriteLine("cooking the eggs ...");  
    await Task.Delay(3000);  
    Console.WriteLine("Put eggs on plate");  
  
    return new Egg();  
}
```

```
static async Task Main(string[] args) {
    Stopwatch sw = new Stopwatch();
    sw.Start();
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");
    Task<Egg> eggsTask = FryEggsAsync(2);
    Task<Bacon> baconTask = FryBaconAsync(3);

    ...
    Egg eggs = await eggsTask;
    Console.WriteLine("eggs are ready");
    Bacon bacon = await baconTask;
    Console.WriteLine("bacon is ready");

    sw.Stop();
    Console.WriteLine($"Breakfast is ready after " +
        $"{sw.Elapsed.Milliseconds}ms.");
}

private static async Task<Bacon> FryBaconAsync(int slices) {
    Console.WriteLine($"putting {slices} slices of bacon in the pan");
    Console.WriteLine("cooking first side of bacon...");
    await Task.Delay(3000);
    for (int slice = 0; slice < slices; slice++) {
        Console.WriteLine("flipping a slice of bacon");
    }
    Console.WriteLine("cooking the second side of bacon...");
    await Task.Delay(3000);
    Console.WriteLine("Put bacon on plate");

    return new Bacon();
}
```

Bacon with async & await

Toast with async & await

```
private static async Task<Toast> MakeToastWithAsync(int number) {  
    var toast = await ToastBreadAsync(number);  
    ApplyButter(toast);  
    ApplyJam(toast);  
    return toast;  
}  
  
private static void ApplyJam(Toast toast) =>  
    Console.WriteLine("Putting jam on the toast");  
private static void ApplyButter(Toast toast) =>  
    Console.WriteLine("Putting butter on the toast");  
private static async Task<Toast> ToastBreadAsync(int slices) {  
    for (int slice = 0; slice < slices; slice++) {  
        Console.WriteLine("Putting a slice of bread in the toaster");  
    }  
    Console.WriteLine("Start toasting...");  
    await Task.Delay(3000);  
    Console.WriteLine("Remove toast from toaster");  
    return new Toast();  
}
```

Toast with async & await

```
static async Task Main(string[] args) {  
    Stopwatch sw = new Stopwatch();  
    sw.Start();  
    Coffee cup = PourCoffee();  
    Console.WriteLine("coffee is ready");  
    Task<Egg> eggsTask = FryEggsAsync(2);  
    Task<Bacon> baconTask = FryBaconAsync(3);  
    Task<Toast> toasttask = MakeToastWithAsync(2);  
  
    Egg eggs = await eggsTask;  
    Console.WriteLine("eggs are ready");  
    Bacon bakon = await baconTask;  
    Console.WriteLine("bacon is ready");  
    Toast toast = await toasttask;  
    Juice oj = PourOJ();  
    Console.WriteLine("oj is ready");  
    sw.Stop();  
    Console.WriteLine($"Breakfast is ready after " +  
        $"{sw.Elapsed.Milliseconds}ms.");  
}
```

```
Pouring coffee  
coffee is ready  
Warming the egg pan...  
putting 3 slices of bacon in the pan  
cooking first side of bacon...  
Putting a slice of bread in the toaster  
Putting a slice of bread in the toaster  
Start toasting...  
Remove toast from toaster  
flipping a slice of bacon  
flipping a slice of bacon  
flipping a slice of bacon  
cooking the second side of bacon...  
Putting butter on the toast  
cracking 2 eggs  
Putting jam on the toast  
cooking the eggs ...  
Put eggs on plate  
Put bacon on plate  
eggs are ready  
bacon is ready  
Pouring orange juice  
oj is ready  
Breakfast is ready after 41ms.
```

Await tasks efficiently

- WhenAll
- returns a Task that completes when all the tasks in its argument list have completed

```
await Task.WhenAll(eggsTask, baconTask, toastTask);  
Console.WriteLine("Eggs are ready");  
Console.WriteLine("Bacon is ready");  
Console.WriteLine("Toast is ready");  
Console.WriteLine("Breakfast is ready!");
```

WhenAll

```
static async Task Main(string[] args) {  
    Stopwatch sw = new Stopwatch();  
    sw.Start();  
    Coffee cup = PourCoffee();  
    Console.WriteLine("coffee is ready");  
    Task<Egg> eggsTask = FryEggsAsync(2);  
    Task<Bacon> baconTask = FryBaconAsync(3);  
    Task<Toast> toastTask = MakeToastWithAsync(2);  
  
    await Task.WhenAll(eggsTask, baconTask, toastTask);  
    Console.WriteLine("Eggs are ready");  
    Console.WriteLine("Bacon is ready");  
    Console.WriteLine("Toast is ready");  
    Console.WriteLine("Breakfast is ready!");  
  
    Juice oj = PourOJ();  
    Console.WriteLine("oj is ready");  
    sw.Stop();  
    Console.WriteLine($"Breakfast is ready after " +  
        $"{sw.Elapsed.Milliseconds}ms.");  
}
```

```
Pouring coffee  
coffee is ready  
Warming the egg pan...  
putting 3 slices of bacon in the pan  
cooking first side of bacon...  
Putting a slice of bread in the toaster  
Putting a slice of bread in the toaster  
Start toasting...  
Remove toast from toaster  
cracking 2 eggs  
cooking the eggs ...  
Putting butter on the toast  
flipping a slice of bacon  
flipping a slice of bacon  
flipping a slice of bacon  
cooking the second side of bacon...  
Putting jam on the toast  
Put bacon on plate  
Put eggs on plate  
Eggs are ready  
Bacon is ready  
Toast is ready  
Breakfast is ready!  
Pouring orange juice  
oj is ready  
Breakfast is ready after 26ms.
```


Await tasks efficiently

WhenAny returns a Task<Task> that completes when any of its arguments complete

- Use WhenAny like this:
 - await the returned task, knowing that it has already finished
 - await the first task to finish and then process its result
 - after processing the result from the completed task, remove that completed task from the list of tasks passed to WhenAny

WhenAny

```
static async Task Main(string[] args) {  
    Stopwatch sw = new Stopwatch();  
    sw.Start();  
    Coffee cup = PourCoffee();  
    Console.WriteLine("coffee is ready");  
    Task<Egg> eggsTask = FryEggsAsync(2);  
    Task<Bacon> baconTask = FryBaconAsync(3);  
    Task<Toast> toastTask = MakeToastWithAsync(2);  
  
    var breakfastTasks = new List<Task> {  
        eggsTask, baconTask, toastTask };  
    while (breakfastTasks.Count > 0) {  
        Task finishedTask = await Task.WhenAny(breakfastTasks);  
        if (finishedTask == eggsTask) {  
            Console.WriteLine("Eggs are ready");  
        } else if (finishedTask == baconTask) {  
            Console.WriteLine("Bacon is ready");  
        } else if (finishedTask == toastTask) {  
            Console.WriteLine("Toast is ready");  
        }  
        await finishedTask;  
        breakfastTasks.Remove(finishedTask);  
    }  
    Juice oj = PourOJ();  
    Console.WriteLine("oj is ready");  
    sw.Stop();  
    Console.WriteLine($"Breakfast is ready after " +  
        $"{sw.Elapsed.Milliseconds}ms.");  
}
```

```
Pouring coffee  
coffee is ready  
Warming the egg pan...  
putting 3 slices of bacon in the pan  
cooking first side of bacon...  
Putting a slice of bread in the toaster  
Putting a slice of bread in the toaster  
Start toasting...  
Remove toast from toaster  
cracking 2 eggs  
cooking the eggs ...  
Putting butter on the toast  
flipping a slice of bacon  
flipping a slice of bacon  
flipping a slice of bacon  
cooking the second side of bacon...  
Putting jam on the toast  
Toast is ready  
Put bacon on plate  
Put eggs on plate  
Bacon is ready  
Eggs are ready  
Pouring orange juice  
oj is ready  
Breakfast is ready after 38ms.
```

Async & Await

- are **used to indicate the method that can work asynchronously**
- await is used for waiting the thread until process is done
- async keyword converts a method into an async method, allowing you to use the await keyword in the method's body
- await keyword suspends and control is returned to the caller until awaited task is completed

