# Object-Oriented Programming with C#

Programming, Part IV

By Per Laursen
26-01-2023

# Introduction

Until this point, we have not really worried about the "performance" of applications, i.e. how they behave with regards to the time needed to perform various operations, but also with regards to the "responsiveness" on an application with a user interface, when the application performs time-consuming operations. Such time-consuming operations can in general be divided into two categories:

- **CPU-bound operations**: these are typically operations that involve intense calculations, performed by the Central Processing Unit (CPU).
- **I/O-bound operations**: these are typically operations that involve interaction with some external data source, for storing or retrieving data. Such interaction may involve waiting for a response from the external data source.

Such operations will inevitably be part of certain applications, but why is that necessarily a problem? So far, we have thought of an application as a sequence of operations carried out one after another. If one of these operations takes a long time to complete, we – i.e. the rest of the application – just have to wait for that operation to complete before proceeding.

Suppose we have a CPU-bound operation that takes 10 seconds to complete. We assume that the algorithm as such cannot be changed, so how can we ever make this operation less time-consuming? The only way forward must be if the work involved can be divided into smaller parts, and that each part of the work can be completed simultaneously. For this to be possible, we need to:

- Be able to divide the work into independent parts of comparable size
- Have a number of independent "workers" available, that each can carry out a single part of the work

The first part is highly operation-specific. Some operations can easily be divided into such parts (we will see some examples later), but others have a nature that is inherently sequential.

The second part depends on the available CPU hardware, and to some extent on the computer operating system. With regards to the CPU hardware, the state of CPU hardware is currently that most CPUs from most manufacturers are so-called **multicore** CPUs. A multicore CPU is essentially a collection of independent CPUs, capable

of executing operations in parallel; one operation on each core. Also, modern operating systems are well capable of utilising multicore CPUs, so all we need to figure out is how to divide the work into smaller parts, and how to execute each part of the work in an “optimal” way, depending on the available CPU.

Concerning the I/O-bound operations, the challenges are of a different nature. An application may need to retrieve some data through e.g. a web service, which will involve waiting a reasonable amount of time for a response. If we implemented such an operation in the way we currently know, the application would appear “blocked” while waiting for the response. If this operation was invoked through a GUI, the user would probably experience that the application becomes unresponsive, maybe only displaying a spinning wait cursor. This is not a very user-friendly behavior. Instead, the user should be allowed to perform other operations, while the response is still pending. This can be achieved by turning the *waiting-for-response* operation into a separate operation, which can then be performed in parallel with the main application operation (which is to react promtly to user interaction).

The overall approach to improving the performance for both categories is thus the same; try to divide the operation into parts, which can then be allocated to separate workers. The specific approach is however somewhat different for each category, which we will see in the next two chapters.

## Managing CPU-bound operations

Let's recall what we need to be able to do, in order to improve performance for CPU-bound operations:

- Be able to divide the work into independent parts of comparable size
- Have a number of independent "workers" available, that each can carry out a single part of the work

The latter point is more a prerequisite; what we more specifically need to be able to do is more like:

- Divide the work into independent parts of comparable size
- Allocate the work parts to workers in an optimal way

The first part is application-specific, and is something we as programmers need to consider and implement. In the .NET class library, the **Task** class is available for this purpose. In general, we must then divide the relevant operations into a number of "tasks", that are each represented by a **Task** object. We will go into much more detail concerning the **Task** class in a moment.

The second part is slightly more tricky. In older programming scenarios (older .NET versions, older operating systems), it was also the programmer's responsibility to explicitly allocate such tasks to workers, more specifically to an abstraction called a **thread**. A **thread** can be thought of as a single worker, carrying out instructions one after another. The applications we have seen so far have all been **single-threaded applications**, i.e. only a single worker carries out the operations in the application. When we create applications using multiple **Task** objects, we are effectively creating a number of threads within a single application, and the application becomes a **multi-threaded application**, where several operations can be executed in parallel.

However, this parallel execution of operations is also an abstraction; if you run a multi-threaded application on a single-core CPU – which is indeed possible – the operating system will create an illusion of parallel execution. In reality, each thread is in turn given a bit of time to work on its operation. After a certain amount of time, the operating system will pause (usually called to **suspend**) the thread, and allow another thread to run for a while, and so on.

If you run a multi-threaded application on a single-core processer, you don't really gain anything by creating multiple threads. In fact, the application may even run slower, since creating and managing threads takes a bit of extra time. Conversely, if you run the application on a multi-core CPU, you should usually generate at least as many threads as you have CPU cores, so each core can execute at least one thread. The optimal allocation of tasks to threads – and subsequently threads to CPU cores – is thus highly dependent on the hardware setup, and is definitely not a trivial matter. The good news is that when using the **Task** abstraction, this allocation is entirely delegated to the .NET run-time system and the operating system! This causes the second point in the (short) bullet of responsibilities list to disappear entirely. The only thing we as programmers need to worry about is the division of operations into tasks. Once that is done, the low-level allocation is taken care of by the run-time system. This simplifies development of multi-threaded applications significantly.

**The Task class – creation and invocation**

From this point on, we will thus focus on how to utilise the **Task** class to create well-defined units of work. Suppose we have defined a couple of methods **DoWorkUnitA** and **DoWorkUnitB**. These methods are of the **Action** type, i.e. they do not take any parameters, and do not return any value. We assume that the two methods are independent (we will later on get into more detail about what "independent" actually means), i.e. it makes sense to execute the operations in these two methods in parallel. When can then create a **Task** object for each method:

```
Task taskA = new Task(DoUnitOfWorkA);
Task taskB = new Task(DoUnitOfWorkB);
```

Note that this alone does not cause the methods to be executed! All we have done is to "wrap" each method into a **Task** object. In order to actually execute the methods, you must call the **Start** method on the **Task** objects.

```
taskA.Start();
taskB.Start();
```

This will start execution of **DoWorkUnitA** and **DoWorkUnitB**. Depending on the available hardware, the execution will be allocated appropriately to threads and cores. In case we e.g. have a dual-core CPU, the methods will most likely be executed on separate cores, and thus truly in parallel. Assuming that the tasks are of comparable size, we can then expect the running time of the application as such to be half of what it would be with traditional programming.

It may seem very restrictive to require the methods to be of type **Action**. It is also possible to create a **Task** object with a method of type **Action<Object>** (remember that **Object** is the base class that all classes inherit from). You can then pass a parameter to the method, which is used when the method is executed:

```
Object data = new List<int>();
Task taskC = new Task(DoUnitOfWorkC, data);
taskC.Start();
```

The **DoWorkUnitC** method can then cast the parameter to the appropriate type. You can also combine the creation and start of a **Task** object by using the static method **Run**:

```
Task taskA = Task.Run(() => DoUnitOfWorkA());
```

Both constructions have the same functionality, so choosing one over another is mostly a matter of taste.

Suppose you want to create a task consisting of two method calls. Once the first method call completes, the second method call should be invoked. You could choose to create a new method containing the two method calls, but a more flexible solution is to use the **ContinueWith** method:

```
Task taskAD = taskA.ContinueWith(DoUnitOfWorkD);
```

In order for this to work, **DoWorkUnitD** must take a parameter of type **Task**. This may not seem to add any flexibility, but you can specify a number of options controlling whether or not **DoWorkUnitD** is actually executed. A task should usually run to completion, but it may also be cancelled (see later), or throw an exception. If you want to ensure that **DoWorkUnitD** is only executed if **DoWorkUnitA** completes normally, you can specify this like:

```
Task taskAD = taskA.ContinueWith(DoUnitOfWorkD,
              TaskContinuationOptions.OnlyOnRanToCompletion);
```

You can then call **ContinueWith** yet again on **taskAD**, and thereby piece together a chain of execution, if you have that need.

**The Task class – synchronisation**

Once a task has been started, it will execute in parallel with the main application task (by "main application task", we mean that "task" which is started simply by starting the application itself. This is often also referred to as the **main application thread**, even though the use of the term "thread" is a bit dated now…). The tasks will execute independently, and will not as such have any knowledge about the progression of other tasks. Still, the logic of the application may dictate some coordination between tasks. This is often referred to as **task syncronisation**.

A simple example of task syncronisation is to require that execution cannot proceed beyond a certain point, until a specific task has completed. This is achieved by using the **Wait** method:

```
Task taskA = new Task(DoUnitOfWorkA);
taskA.Start();

// ...do some work

taskA.Wait();
```

It might seem a bit weird that **Wait** is called on **taskA**, since the effect is not that **taskA** should wait for something, but rather that the invoker of **taskA** should wait here until **taskA** completes. An example of a relevant use of **Wait** could be when preparing to open a dialog window. This may require that some sort of long calculation must be done first; that operation could be wrapped into a **Task**. Furthermore, there might be other preparations to do before opening the dialog. If these preparations are independent of the calculation, they could be done in parallel, like:

```
Task taskCalculate = new Task(Calculate);
taskCalculate.Start();

// ... Do other preparations

// All preparations except calculation are done,
// so wait here until it is done
taskCalculate.Wait();

// Now we can open the dialog
dialog.Open(...);
```

The point is that we must be absolutely sure that **taskCalculate** is done, before we open the dialog. All other preparations will be done when we reach the highlighted line of code, so we need to "sit and wait" here until **taskCalculate** has completed.

Note that if **taskCalculate** has completed before we reach the call of **Wait**, the execution will just proceed to the next statement.

What if we have several tasks that need to be completed before proceeding beyond a point? We can then use the static method **WaitAll**:

```
Task taskA = new Task(DoUnitOfWorkA);
Task taskB = new Task(DoUnitOfWorkB);
Task taskC = new Task(DoUnitOfWorkC);
taskA.Start();
taskB.Start();
taskC.Start();

// ...do some work

Task.WaitAll(taskA, taskB, taskC);
```

The **WaitAll** method can take any number of **Task** objects as parameters, and will wait until all tasks have completed. A **WaitAny** method is also available, which also takes a number of **Task** objects as parameters, but will only wait until any one of the tasks are completed.

**The Task class – cancellation**

Suppose we have an application, where we need to peform a calculation on a given set of data. For this particular calculation, two different algorithms exist, and it is not obvious which algorithm will be the fastest one for a given data set. Furthermore, we have a multi-core CPU available, so we simply decide to do both calculations at once, wrapping each calculation into a **Task** object. Our logic would then be:

1. Create and start a **Task** object for each calculation
2. Wait for one of the calculations to complete
3. Once a calculation has completed, we can discard the other calculation

This looks like an obvious case for the **WaitAny** method:

```
Task taskCalcA = new Task(CalcA);
Task taskCalcB = new Task(CalcB);
taskCalcA.Start();
taskCalcB.Start();

Task.WaitAny(taskCalcA, taskCalcB);

// We now have a result
```

The code above takes care of steps 1 and 2, but not really of step 3. How do we "discard" a running task? We must somehow be able to tell the task, that we would like it to stop working, since we don't need it any more. You could imagine that you could just call some method (e.g. called **Stop** or **Cancel**) on the **Task** object, which would simply shut down the task immediately – a sort of *bullet-in-the-head* solution. This is however too crude. You can easily imagine that the task in question could be in a state where some sort of cleaning up is necessary before stopping. The task could e.g. have opened a connection to a database, from which it should disconnect in an orderly manner before stopping. Task cancellation is therefore a cooperative effort between the task itself, and the entity requesting the cancellation.

Cancellation revolves around a so-called **cancellation token**. The entity creating a **Task** object can also create a cancellation token, and provide it as a parameter to the method being invoked by the task. Creating and invoking a **Task** object then becomes a bit more complicated:

```
CancellationTokenSource tokenSource = new CancellationTokenSource();
CancellationToken token = tokenSource.Token;

Task taskCalcA = Task.Run(() => CalcA(token), token);
```

This is somewhat obscure, but the essence is that the **Task** creator must create a **CancellationToken** object, and include it as a parameter to the **Run** method and to the method itself (in this case **CalcA**)! The point of this setup is that the invoker of the task can now "signal" to the task that it should be cancelled. It does this by calling the **Cancel** method on the **CancellationTokenSource** object**,** not the **CancellationToken** object!

```
Task taskCalcA = Task.Run(() => CalcA(token), token);

// .. do some work

tokenSource.Cancel();
```

The **CalcA** method must also be changed. **CalcA** must periodically check the status of its cancellation token, and act accordingly:

```
public void CalcA(CancellationToken token)
{
	while (!token.IsCancellationRequested && /* other conditions */)
	{
		// Keep doing work
	}

	if (token.IsCancellationRequested)
	{
		// Do any operations needed before finishing
	}
}
```

This is just an example; the exact manner in which the token status is checked will vary from method to method. The important point is that the task cannot be "forced" to shut down. The creator of the task can request a cancellation, and the task itself must then honor this request in an appropriate manner.

Note that the same cancellation token can be passed to multiple tasks. If a cancellation is requested by calling **Cancel** on the **CancellationTokenSource** object, all tasks will see this through the status of the cancellation token.

**The Task class – advanced topics**

As you have probably recognised already, things get more complex once you start to divide an application into tasks. At any time, several tasks can be running in parallel in an application, and each task will be in one of several possible states:

- **Created**: Task is created, but not scheduled to run yet.
- **WaitingToRun**: Task is created and scheduled, but not running yet.
- **Running**: Task is running.
- **RanToCompletion**: Task has completed successfully.
- **Cancelled**: Task was cancelled, either before or while running.
- **Faulted**: Task terminated by throwing an exception.

Handling all possible scenarios can become quite complex. One way of managing the complexity can be to use the **ContinueWith** method, where you can specify which method to execute under specific circumstances:

```
taskCalcA.ContinueWith(HandleCalcCancel, TaskContinuationOptions.OnlyOnCanceled);
```

This is likely to be a simpler alternative to a complex **if-else** structure.

When multiple tasks are running, you suddenly also have a scenario where multiple exceptions can be thrown in parallel, and need to be handled somehow. For this purpose, the **AggregateException** class exists, which is essentially a collection of exception objects, which can then be handled individually. You can even rethrow an **AggregateException** containing a subset of the original set of exceptions, if you only handle some of the exceptions in your own exception handler. Proceed carefully…

**The Parallel class**

Suppose your application contains some logic, where you need to perform the same calculation for a large number of values. Furthermore, the calculations can be done independently. That looks like an obvious case for using tasks…and it is! Since such a scenario is fairly common, the .NET class library contains the **Parallel** class, which makes it even easier to divide such an operation into tasks. The original calculation may look like this:

```
for (int i = 0; i < 100; i++)
{
      Calculate(i);
}
```

In order to perform each iteration as a separate task, you need just a little bit of rewriting:

```
Parallel.For(0, 100, Calculate);
```

You can think of this as converting the loop iteration into 100 independent tasks, which are then started and waited on, until the last task has completed. Under the covers, the **Parallel** class will create a number of **Task** objects, but this number may be considerably lower than the number of iterations. This is fine, since we have assumed that the calculations are independent. You should just keep in mind that there is no guarantee about the order in which the tasks are executed. The first task might have **i** set to 0, but the next one might have **i** set to 46, and so on… As we will see in the next chapter, this poses some difficulties when the iterations have dependencies. It is actually possible to use **Parallel.For** even in such cases, but some additional safety measures have to be in place.

## Managing I/O-bound operations

The introduction of the **Task** class enables us to wrap up time-consuming operations in **Task** objects, which can then be executed in parallel with the code creating the task. If the application features a graphical user interface (GUI), it will often be the "GUI code" which creates tasks. By "GUI code" is more specifically meant the code which is executed in the GUI thread. If an application has a GUI, it is only the GUI thread (or task, if you prefer) which can interact with the GUI controls. This provides some additional challenges. Consider the below code, which we assume will run in response to the user clicking on e.g. a **Button** control:

```
private void HandleButtonClick()
{
      DoTaskA();
      DoTaskB();
      statusText.Text = "All done";
}
```

We assume that **statusText** is also a GUI control. This code will execute in the usual manner, i.e. it will not finish before **DoTaskA** and **DoTaskB** are finished. The consequence will be that the GUI becomes unresponsive until the method call has finished. So, if the **Do…** methods are time-consuming, the GUI will be unresponsive for an unacceptably long time. Can we fix this by using tasks? We can try. A first attempt could be this:

```
private void HandleButtonClick()
{
      Task taskAB = new Task(DoTaskA);
      taskAB.ContinueWith(DoTaskB);
      taskAB.Start();
      statusText.Text = "All done";
}
```

The problem with this solution lies in the call of **Start**. Remember that **Start** will invoke the method wrapped inside the **Task** object, and immediately return to the caller. The effect will be that the *"All done"* status text is set before the tasks are done. Maybe we can fix this by adding a call of **Wait** then?

```
private void HandleButtonClick()
{
      Task taskAB = new Task(DoTaskA);
      taskAB.ContinueWith(DoTaskB);
      taskAB.Start();
      taskAB.Wait();
      statusText.Text = "All done";
}
```

This is also a no-go, since we are in fact back to the starting point! Now the call to **HandleButtonClick** will (again) block the GUI, since it will not return before all of the tasks are done.

How about wrapping the GUI interaction itself into a **Task** object? This task could then be set as a continuation to **DoTaskB**, like this (we assume that **UpdateGUI** is a method containing the code for updating the GUI):

```
private void HandleButtonClick()
{
      Task taskAB = new Task(DoTaskA);
      taskAB.ContinueWith(DoTaskB);
      taskAB.ContinueWith(UpdateGUI);
      taskAB.Start();
}
```

It looks promising: the tasks will be executed in correct order – including update of the GUI – and control will return to the caller immediately, making the GUI responsive. Alas, the code will produce an error when executed… Remember the previous claim that the GUI can only be updated from the GUI thread? That's the problem here. By executing the GUI interaction as a task, it will be executed on a different thread, hence the error message. You can actually create a workaround for this, but it is somewhat obscure, and definitely not the recommended approach. Instead, we solve the problem by using two new C# language elements called **async** and **await**.

### Programming with async and await

The high-level definition of the purpose of **async** and **await** is: to enable you to define and call methods that can execute asynchronously. The first mental hurdle here is grasping what is meant by "execute asynchronously". Most important to grasp is that executing code asynchronously does not necessarily mean to execute the code in a separate thread! It rather means that the code can be executed in "chunks", and – very importantly – that the *flow-of-control* returns to the caller of the method, when

such a “chunk” has been executed. This leads to some slightly more detailed definitions of **async** and **await**:

- **async** is a method modifier (like e.g. **public** or **static**), indicating that the method contains code that can be run asynchronously.
- **await** is an operator, which specifies where code will be executed asynchronously.

This is probably still hard to grasp, so let’s see an example. The below code contains a modified version of **HandleButtonClick**, and modified versions of the **Do…** methods:

```
private async void HandleButtonClick()
{
      await DoTaskAUpdated();
      await DoTaskBUpdated();

      statusText.Text = "All done";
}


private Task DoTaskAUpdated()
{
      Task t = Task.Run(() => DoTaskA());
      return t;
}

private Task DoTaskBUpdated()
{
      Task t = Task.Run(() => DoTaskB());
      return t;
}
```

The **Do…Updated** methods wrap the original **Do…** methods into **Task** objects, and return them to the caller. The **await** operators in **HandleButtonClick** are thus applied to a **Task** object in each case. The **await** operator can in general only be applied to an object which is “awaitable”, which is exactly what a **Task** object is. However – and this is the tricky part – the **await** operator does not act like the **Wait** method in the **Task** class. The **Wait** method will suspend the thread on which it is called, and will only resume once the task it was called on has completed. **await** works by “suspending” the method call at the point it has reached, but returns the *flow-of-execution* back to the caller of the method! The thread itself is not suspended.

Let's rephrase that as a sort of statement of intention for both cases. This should be understood as an answer to the question: *"What will happen when you (i.e. either **Wait** or **await**) are reached in a method call?"*

| | |
|---|---|
| **task.Wait()** | *"It does not make sense to proceed beyond this point in the method, until **the task on which I was called** has completed. I will therefore **suspend the thread** I was called on, until that happens. If the thread happens to be the GUI thread, the GUI will become **unresponsive**…"* |
| **await** | *"It does not make sense to proceed beyond this point in the method, until **the task which I'm awaiting** has completed. I will therefore **suspend the method call** until that happens. **I will return the flow-of-execution to the caller**. If I was called from the GUI thread, the GUI thread will therefore still run, and the GUI will be **responsive**.* |

Using **await** is therefore definitely the better approach in this scenario. Still, it is a bit mind-bending to keep track of the *flow-of-execution* here. What happens when the task that is awaited finally completes? The *flow-of-execution* will then jump back into the called method (in this case **HandleButtonClick**), and continue with the next statement. This is what is meant by "asynchronous execution" – the code is executed in "chunks", controlled by use of **await**. In this case, the next statement also contains an **await** operator, so we jump right back to the caller again… However, if we had added some additional statements between the two statements containing **await**, those statements would indeed have been executed before returning to the caller.

It is probably obvious that methods marked as **async** can behave radically different than ordinary methods. A caller of such a method should be aware of its nature, and act accordingly. To help this awareness along, it has become a convention to suffix asynchronous methods with **Async**, so the method above should have been renamed to **HandleButtonClickAsync**. The .NET class library contains several such methods, for instance methods for reading and writing to files.

The asynchronous method in the above example did not return any value. The **await** operators in the **HandleButtonClick** method are thus applied to objects of type **Task**. If an asynchronous method must return a value, things become a bit more complex. Suppose we have a method called **Operation**, which performs some sort of time-consuming operation, and returns a result of type **int**. The definition of a corresponding asynchronous method **OperationAsync** could then be:

```
static async Task<int> OperationAsync()
{
      Task<int> task = Task.Run(() => Operation());
      await task;
      return task.Result;
}
```

First, note that the type of the created **Task** object is **Task<int>**. In general, the type **Task<T>** should be read as: a **Task** object which returns a value of type **T** upon completion. The variable **task** is thus of type **Task<int>**. However, just after having created and started the task, we **await** it… We are thereby returning the *flow-of-execution* to the caller of **OperationAsync**, who may decide what to do next (see later). At some point, the task will have completed, and the *flow-of-execution* will return to **OperationAsync.** The result of the operation can now be extracted from the **Task** object itself, through the **Result** property. This result – which is of type **int** – will then be the return value of the call of **OperationAsync**. This is indeed a bit confusing; we state in the method definition that the return type of **OperationAsync** is **Task<int>**, but it seems that we are returning an **int** value!? In a sense, both are true. Let's see how a caller can invoke **OperationAsync**. This code is legal:

```
Task<int> task = OperationAsync();

// ...do some work

int operationResult = task.Result;
```

This is a two-step operation; the first line contains a call to **OperationAsync**, which will return a **Task<int>** object. But when is this object returned? Recall the code for **OperationAsync**:

```
static async Task<int> OperationAsync()
{
      Task<int> task = Task.Run(() => Operation());
      await task;
      return task.Result;
}
```

When the highlighted **await** statement is reached, the *flow-of-execution* returns to the caller, but it returns in the form of a **Task<int>** object. The caller can then keep a reference to this **Task<int>** object, and proceed with whatever work it makes sense to perform. At some point, the caller will need the value returned by the task, before it makes sense for the caller to proceed further. The last line in the caller's code – where the **Result** property of **task** is referenced – will block the caller until the result is actually ready. When that happens – i.e. when the task has completed – the *flow-*

*of-execution* will return to **OperationAsync,** specifically to the line after the **await** operator. In that line, the result of the task – which is obviously ready now, since the task has just completed – is returned to the caller, meaning that the caller is no longer blocked. The result of the task is then finally available to the caller.

The caller can however also invoke **OperationAsync** like this:

```
int operationResult = await OperationAsync();
```

How is this different from before? Since the caller now also uses **await**, the *flow-of-execution* will now be returned to whoever it was that called the caller! We are no longer blocking the thread until the task finishes, but are instead leaving it up to the caller-of-the-caller what should happen next. Also, the method which the above line belongs to now also becomes an **async** method. Once you start using **async** in your code, you will often experience the *async-all-the-way-up* phenomenon; once one method is made **async**, the caller of that method should also beome **async**, and the caller-of-the-caller should also beome **async**, and so on.

Due to this pervasive nature of using **async**/**await**, it becomes somewhat difficult to add it to existing code. You should carefully consider if your particular application will benefit from using this construction, and then design it into the code from the outset. Using **async**/**await** is not a magic bullet that will make any application run faster, and it definitely makes it harder to develop and test the code.

## Managing concurrent data access

When we have discussed the **Task** concept in the previous chapters, we have not shown any explicit examples of code which can be wrapped into tasks. In theory, we can wrap any sort of code into tasks, and execute them in parallel. Complications do however arise, if the code running in separate tasks tries to access the same data. Consider the below class **PrimeCalc**, which contains a crude method for finding prime numbers up to a specified limit:

```
public class PrimeCalc
{
	private List<int> _primes;

	public PrimeCalc()
	{
		_primes = new List<int>();
	}

	public void FindPrimes(int upper)
	{
		_primes.Clear();
		FindPrimesInInterval(2,upper);
		string text = $"Found {_primes.Count} primes in [2; {upper}]";
		Console.WriteLine(text);
	}

	private void FindPrimesInInterval(int lower, int upper)
	{
		for (int i = lower; i < upper; i++)
		{
			if (IsPrime(i))
			{
				_primes.Add(i);
			}
		}
	}
```

```
        private bool IsPrime(int number)
        {
            if (number < 4) { return true; }

            int limit = Convert.ToInt32(Math.Sqrt(number));
            bool isPrime = true;

            for (int i = 2; i <= limit && isPrime; i++)
            {
                isPrime = number % i != 0;
            }

            return isPrime;
        }
}
```

The **IsPrime** method checks if a given number is a prime number, by trying to divide it with all numbers from 2 to the square root of the number itself. If any of these divisions produce a remainder of zero, the number is not a prime number. More efficient algorithms exist, but this simple algorithm has the advantage that each check is independent of other checks. In the above version, all checks are done sequentially, one after another. A call of **FindPrimes(1000000)** shows that 78,498 primes exist in that interval.

Since the checks are independent, it is fairly easy to split them up into two separate tasks, like this:

```
        public void FindPrimes(int upper)
        {
            _primes.Clear();

            int middle = upper / 2;
            Task t1 = Task.Run(() => FindPrimesInInterval(2, middle));
            Task t2 = Task.Run(() => FindPrimesInInterval(middle + 1, upper));
            t1.Wait();
            t2.Wait();

            string text = $"Found {_primes.Count} primes in [2; {upper}]";
            Console.WriteLine(text);
        }
```

One task checks the lower half of the interval, the other task the higher end. Quite simple. However, when running the application now, something odd happens… The reported numbers of primes is now no longer the same! Also, the number varies if you run the application several times!? A sample of five runs gave these results (you may see different result, if you try to run the code on your own computer):

```
78,377
78,383
78,364
78,384
78,382
```

What is going on here? The problem lies in the data access. The method **FindPrimesInInterval** – which both tasks are executing – contains the call **_primes.Add**, i.e. both tasks try to add data to the list at the same time… Unfortunately, the **Add** operation is not an "atomic" operation (an operation which is always completed in full, or not started at all), so we may sometimes be in the situation that one task starts to add an element, while the other task is in the middle of adding an element. This can have very unpredictable results. The numbers from the sample runs seem to indicate that this happens rarely (all results are within 0.2 % of the correct number), but it must – of course – not happen at all! The fact that it happens rarely makes this error even more devious. It might not show up in any test, but suddenly show itself when the code runs in production (hopefully not in the control system for a nuclear reactor…).

The problem is that the **List** collection is not **thread-safe**. Code is considered thread-safe if it works as expected, even if more than one thread (i.e. task) is executing the code concurrently. In order to fix this problem, we have two options:

- Use some of the so-called **synchronisation primitives** in C# for managing the data access
- Use some of the **thread-safe collection classes** in the .NET library

The first option involves adding some protective code around the data accessing code. One example of such code uses the **lock** keyword.

The **lock** keyword is used like this: **lock**(**lockObject**), where **lockObject** is an object created solely for acting as a locking item. You can think of such an item as an object that can only be "owned" by one thread at a time. The object itself does not need to have any specific properties, so you often simply use an object of type **Object**, i.e. the class from which all other C# classes inherit. If we suppose that an instance field **_lock** of type **Object** is added to the **PrimeCalc** class, we can then modify the code for **FindPrimesInInterval** like this:

```
private void FindPrimesInInterval(int lower, int upper)
{
      for (int i = lower; i < upper; i++)
      {
            if (IsPrime(i))
            {
                  lock (_lock)
                  {
                        _primes.Add(i);
                  }
            }
      }
}
```

The effect of the lock is that only one thread can execute code inside the locked code block. If a thread reaches the **lock** statement while another thread is inside the code block, the thread will be blocked until the lock is released again. Since we would like to minimise the time threads are blocked, locks should contain as little code as possible. In the above example, it is only the **Add** statement which needs to be inside the lock, since it is the only statement modifying the collection. Running the code after this modification always produces the correct result.

The **lock** facility is a simple strategy, and will often be sufficient to ensure orderly and efficient access to data shared between threads. If you need a more sophisticated strategy, the .NET library contains several additional classes for this purpose. They are located in the **System.Threading** namespace, with exotic names like **ManualResetEventSlim**, **CountdownEvent**, **Barrier** and several more. Detailed discussions of these classes are beyond the scope of this text.

The problem discussed here is a fairly common problem, so you would expect that the .NET class library contains ready-made thread-safe collection classes, and indeed it does. They are, however, not a one-to-one reflection of the ordinary collection classes, and have somewhat limited functionality. At the time of writing, these thread-safe collection classes exist:

| | |
|---|---|
| **ConcurrentBag<T>** | Thread-safe unordered collection of elements |
| **ConcurrentDictionary<TKey, TValue>** | Thread-safe collection with dictionary-like functionality |
| **ConcurrentQueue<T>** | Thread-safe collection with queue-like functionality |
| **ConcurrentStack<T>** | Thread-safe collection with stack-life functionality |

These collection classes can also be wrapped into a so-called **BlockingCollection<T>**, which provides further functionality[1].

[1] https://learn.microsoft.com/en-us/dotnet/standard/collections/thread-safe/blockingcollection-overview

In our example, we can simply replace the use of **List<int>** with **ConcurrentBag<int>**, and remove the **lock** statement again. This modification also restores the correctness of the code.

As a final thought on the example, we mention that it could also have been implemented using the **Parallel.For** statement (assuming that we keep using **ConcurrentBag<int>** for storing prime numbers):

```
private void FindPrimesInInterval(int lower, int upper)
{
	Parallel.For(lower, upper, (i) =>
	{
		if (IsPrime(i))
		{
			_primes.Add(i);
		}
	});
}
```

We leave it as an exercise to consider if this implementation could be more efficient than the two-task implementation shown above, even if we only have two CPU cores available…

# Exercises

| **Exercise** | **PRO.4.1** |
|---|---|
| **Project** | CrossTalk |
| **Purpose** | See how the **Task** class can be used to execute some very simple operations in parallel |
| **Description** | The project contains the class **Reciter**, which contains a couple of fairly simple methods. The method **ReciteAllTheWords** executes a recitement of the numbers 1 to 8, in three different languages. Each recitement is done by calling the method **Recite**. |
| **Steps** | 1. Run the application. You will see that the recitements are done sequentially, i.e. one recitement is completed before the next recitement is executed.<br>2. Change the code in the **ReciteAllTheWords** method, such that the recitements are done in parallel. The effect should be that the printouts on the screen are a mix of the three languages (Hint: each recitement should be turned into a task).<br>3. A side-effect of the changes made in step 2 is that the message *"Reciting completed…(press any key to close App)"* is now printed long before the last word is printed. Why does this happen? |

| | |
|---|---|
| **Exercise** | **PRO.4.2** |
| **Project** | NumericalPi |
| **Purpose** | Change the execution of a time-consuming calculation to use **Tasks**, to decrease the absolute running time. |
| **Description** | The project contains the class **PiCalc**, which contains an algorithm for calculating an approximate value of π (the exact value of π can be retrieved from **Math.PI**).<br><br>The method **Iterate** will do this for the specified number of iterations:<br>• Generate a random point within the square [0;1[ x [0;1[ (a.k.a. the unit square)<br>• Count the number of times the point falls within the circle with center at (0,0), and radius 1 (a.k.a. the unit circle).<br><br>The ratio between the parameter **iterations** and the returned number (**inside-UnitCircle**) will approximate π/4. The higher the number of iterations, the closer the ratio will come to π/4. The final estimate is then easy to calculate, as is done in the **Calculate** method.<br><br>Suppose you had to do this calculation manually. You could e.g. draw the square and circle on a piece of paper, and throw a dart at the paper e.g. 100 times. You should then count the number of times the dart has hit within the circle. Say the dart hit within the circle 77 times. Your estimate of π would then be (4.0 * 77) / 100 = 3.08. |
| **Steps** | 1. You invite three friends over to help with your experiment. A total of four persons can now throw darts. How will you utilise this to speed up the experiment?<br>2. See if you can translate your redesigned experiment into a new version of **Calculate**, where you use **Task** objects (Hint: wrap the code corresponding to what one person should do, into a **Task** object, like **Task task1 = Task.Run( () => {…});**)<br>3. Compare the running time of your new version of **Calculate** with the original version. How much faster is your version?<br>4. See if you can figure out how many cores your CPU has. How does this number relate to what you observed in step 3? |

| | |
|---|---|
| **Exercise** | **PRO.4.3** |
| **Project** | SlicesOfPiUI (NB: Unported UWP App!) |
| **Purpose** | See the usefulness of using the **async**/**await** programming model |
| **Description** | In a previous exercise, we saw a an algorithm for calculating an approximate value of π. The algorithm is fairly easy to speed up using tasks, but it still has to run to completion, before a value is available. It could be useful to – at any time during the calculation – be able to:<br>• See how good the currently calculated value of π is.<br>• Stop the calculation, and use the current value as the final result.<br><br>The project contains two approaches to solving this.<br>• A synchronous approach (i.e. no use of tasks or **async**/**await**), where the user can perform a traditional (i.e. uninterruptable) calculation.<br>• An asynchronous approach, where the main calculation loop is wrapped into a task, which is awaited. This makes it possible to interrupt the calculation at any time, and also keeps the application responsive. |
| **Steps** | 1. Start out by experimenting a bit with the application. Note in particular how the application behaves during a synchronous calculation, compared with the behavior during an asynchronous calculation.<br>2. Get an overview of the application. The classes fall in three main categories: View Models, Calculation and Commands, plus a single view.<br>3. Now focus on the class **PiCalcBase**, in particular on the method **Calculate**. Starting from this method, see if you can obtain an understanding of how the calculation algorithm works.<br>4. Take a look at the classes **PiCalcSync** and **PiCalcAsync**, and take note of the differences in the implementations of **CalculateSync** and **CalculateAsync**. What seems to be the most important differences?<br>5. Starting from **CalcViewModelBase**, try to gain an understanding of how the calculation and the UI are linked together. What is it specifically that makes the application "freeze" during a synchronous calculation, while it remains responsive during an asynchronous calculation?<br>6. Can you come up with a way to use synchronous calculations that makes the user experience a bit better (Hint: the title of the C# project ☺)? |

| | |
|---|---|
| **Exercise** | **PRO.4.4** |
| **Project** | ProducerConsumer |
| **Purpose** | Work with a typical producer/consumer setup, involving Tasks and data locking |
| **Description** | The project contains several classes, which participate in a so-called **producer/consumer** scenario. A “producer” produces objects and inserts them into a data structure, while a “consumer” consumes objects by removing them from the same data structure. The production and consumption should be executed in parallel |
| **Steps** | 1. Study all of the classes in the project. Note that this a a fairly large project, and some of the classes are a bit complex, so be prepared to spend some time on this steps.<br>2. Team up with another student, and discuss your understanding of the overall application, and the individual classes. If you disagree on something, you must investigate the code further.<br>3. Try to run the application *as-is*. You will likely see that the reporting looks strange (sort of jumps up and down a bit on the screen). Why do you suppose this happens? Try to fix the problem (Hint: Maybe only one thread should try to print on the screen at any time…)<br>4. Try to change the reporting mode (last parameter in the **Scenario** constructor) to **ReportMode.silent**, and re-run the application. Why is the message *"Press any key to abort the run..."* printed almost immediately after starting the app?<br>5. The above run probably gave the result that all balances were good. Try to change the first five parameters to the **Scenario** constructor to 1000, 500, 1000, 3, 2 (also change the report mode to **silent**). Re-run the app. Do you see any bad balances now? Try to run the app a few times. Do the results change from run to run?<br>6. See if you can figure out how to get rid of the bad balances problem (Hint: What data can be accessed by both the producer and consumer?).<br>7. See if you can extend the application to include several producers and/or consumers. |