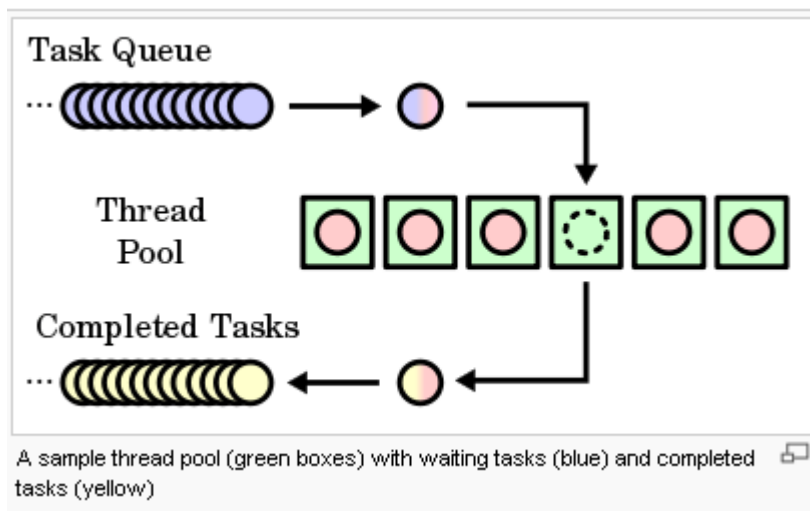


Op Task-gebaseerde asynchrone programmatie – Deel 1

Wat Is een Task In C#?

Een Task-object is een object dat operatie (taak) die moet worden uitgevoerd.

Een Task-object kan asynchroon worden uitgevoerd. Een Task object houdt bij of het werk is beëindigd en geeft ook het resultaat als de operatie bestaat (een return value geeft).



Bron: <https://www.c-sharpcorner.com/article/task-and-thread-in-c-sharp/>

Een task in C# wordt gebruikt voor Task-gebaseerde Asynchrone Programmatie. Het Task object is standaard asynchroon uitgevoerd

Een Task scheduler is verantwoordelijk voor het starten en beheer van een Task. De Task scheduler gebruikt standaard threads van uit een 'thread pool' voor Tasks.

What is een Thread pool in C#?

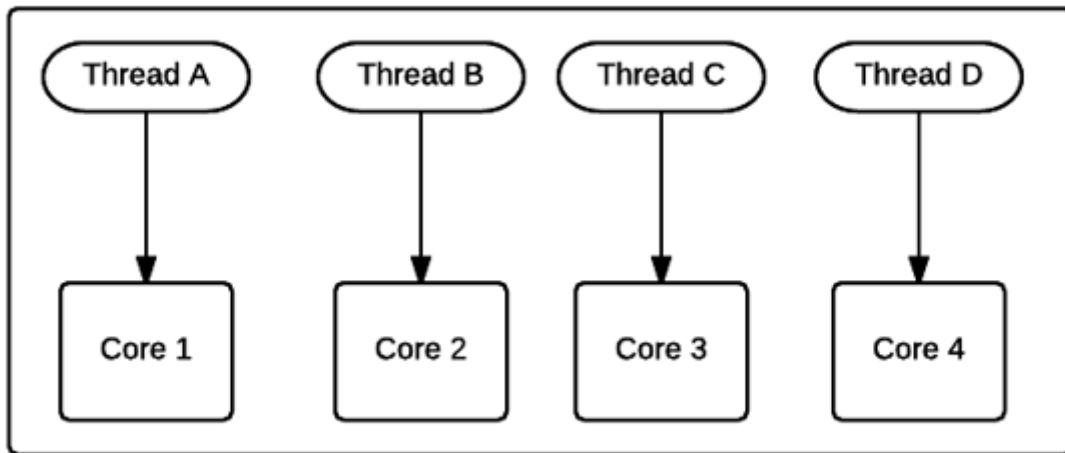
Een **Thread pool in C#** is een collectie van **threads** die gebruikt kunnen worden om werk in de achtergrond uit te voeren. Wanneer een thread zijn werk heeft afgerond, wordt het teruggestuurd in de thread-pool en kan dan later terug worden herbruikt. Dit systeem van herbruiken van threads via een thread-pool vermijdt dat een applicatie steeds nieuwe threads aanmaakt, zodat er minder werkgeheugen wordt ingenomen.

Waarom Tasks gebruiken in C#?

Tasks in C# zorgen dat een C# app meer responsief wordt. Meerdere threads kunnen tegelijkertijd runnen en de thread voor de user interface geraakt niet geblokkeerd wanneer er een langdurig werk (bv laden van gegevens uit database) wordt verricht. Het laden van de gegevens uit de database gebeurt terzelfdertijd in de achtergrond, terwijl de UI responsief blijft (scrollen, lists openklikken, typen in textboxes,... kan de einduser dus bv blijven doen in de UI).

Wat is een Thread?

Een Thread is een kleine reeks van uitvoerbare instructies.



Bron: <https://www.c-sharpcorner.com/article/task-and-thread-in-c-sharp/>

Waarvoor dienen Tasks?

Tasks kunnen gebruikt worden wanneer je iets in parallel wil uitvoeren. Asynchrone programmatie via Tasks gebruiken dmv van de ' **async**' en ' **await**' keywords.

Waarvoor dienen Threads?

Deze zijn nodig wanneer de applicatie verschillende taken terzelfdertijd wil kunnen uitvoeren

Wat is het verschil tussen Task en Thread?

1. De Thread class wordt gebruikt voor creatie en manipulatie van een [thread](#) in Windows. Een [Task](#) is een asynchrone operatie en is een onderdeel van de [Task Parallel Library](#), een aantal APIs om taken asynchroon en in parallel te kunnen uitvoeren.
2. Een Task kan een resultaat teruggeven. Er is geen direct mechanisme om het resultaat van een thread terug te geven.
3. Task ondersteunt annulering dmv "cancellation tokens". Een thread heeft dit niet.
4. Een task kan meerdere operaties hebben die terzelfdertijd gebeuren (runnen). Een thread kan maar één operatie tegelijkertijd doen.
5. Via Tasks is het gemakkelijk om Asynchroon te Programmeren dmv 'async' en 'await' keywords.
6. Een nieuwe Thread() werkt niet standaard met een Thread pool, terwijl Task-objecten standaard de thread pool threads gebruiken.
7. Een Task is een concept op hoger niveau dan een Thread.

Task gebruiken in C#:

De Task classen zijn gedefinieerd in de **System.Threading.Tasks** namespace.

Zet dus eers de lijn **using System.Threading.Tasks** namespace in je code.

De Task class is een enkele operatie die asynchroon op een thread-pool wordt uitgevoerd t.o.v. de Hoofd (Main) thread

Voorbeeld 1 : Task class en Start methode

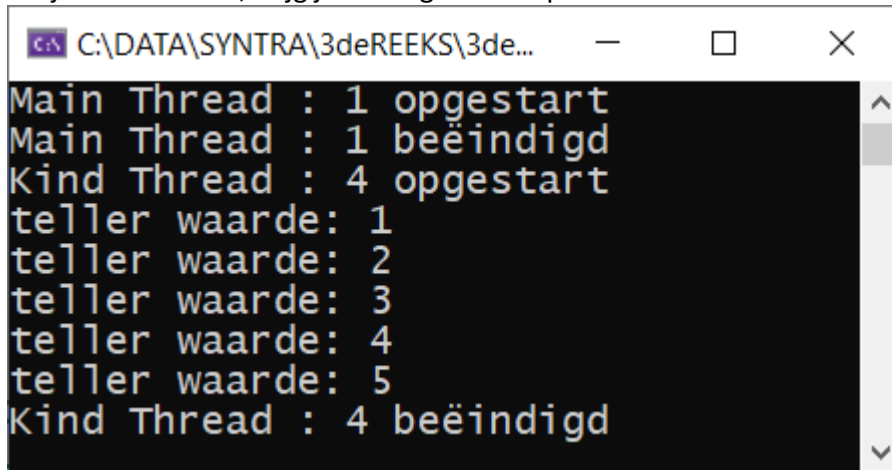
De create van een Task object kan via de Task class gebeuren, het opstarten van uitvoering kan via het aanroepen van de Start methode op het Task object:

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace AsyncProg
{
    class Program
    {
        //Voorbeeld1
        static void Main(string[] args)
        {
            Console.WriteLine($"Main Thread : {Thread.CurrentThread.ManagedThreadId} opgestart");
            Task task1 = new Task(PrintCounter);
            task1.Start();
            Console.WriteLine($"Main Thread : {Thread.CurrentThread.ManagedThreadId} beëindigd");
            Console.ReadKey();
        }
        static void PrintCounter()
        {
            Console.WriteLine($"Kind Thread : {Thread.CurrentThread.ManagedThreadId} opgestart");
            for (int count = 1; count <= 5; count++)
            {
                Console.WriteLine($"teller waarde: {count}");
            }
            Console.WriteLine($"Kind Thread : {Thread.CurrentThread.ManagedThreadId} beëindigd");
        }
    }
}
```

Het task object task1 maakt een nieuw Kind-thread die asynchroon zal worden uitgevoerd in een thread pool.

Als je de code runt, krijg je de volgende output :



```
Main Thread : 1 opgestart
Main Thread : 1 beëindigd
Kind Thread : 4 opgestart
teller waarde: 1
teller waarde: 2
teller waarde: 3
teller waarde: 4
teller waarde: 5
Kind Thread : 4 beëindigd
```

Twee threads worden gebruikt in de applicatie code. De Main thread en de Kind- thread. De 2 threads worden asynchroon uitgevoerd.

Voorbeeld 2: Creatie van task object dmv Factory Property

Een task object aangemaakt dmv van de Factory property zal automatisch opstarten.

```
using System;
using System.Threading;
using System.Threading.Tasks;

namespace AsyncProg
{
    class Program
    {
        //Voorbeeld2
        static void Main(string[] args)
        {
            Console.WriteLine($"Main Thread : {Thread.CurrentThread.ManagedThreadId} opgestart");
            Task task1 = Task.Factory.StartNew(PrintCounter);
            Console.WriteLine($"Main Thread : {Thread.CurrentThread.ManagedThreadId} beëindigd");
            Console.ReadKey();
        }
        static void PrintCounter()
        {
            Console.WriteLine($"Kind Thread : {Thread.CurrentThread.ManagedThreadId} opgestart");
            for (int count = 1; count <= 5; count++)
            {
                Console.WriteLine($"teller waarde: {count}");
            }
            Console.WriteLine($"Kind Thread : {Thread.CurrentThread.ManagedThreadId} beëindigd");
        }
    }
}
```

Het enige verschil met het eerste voorbeeld is dat we de thread creëren en runnen in één enkele statement.

Voorbeeld 3 : Creatie van Task object dmv Run methode

Het is mogelijk om een Task op 3 verschillende manieren te starten. De Task.Run en Task.Factory.StartNew methoden zijn het meest performant voor het creëren en scheduleren van de tasks.

Indien je echter de task creatie en scheduling apart wil doen, dan kan je de task afzonderlijk creëren en daarna de Start methode aanroepen op de task executie te scheduleren voor latere tijd bv.

```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace AsyncProg
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine($"Main Thread : {Thread.CurrentThread.ManagedThreadId} gestart");
            Task task1 = Task.Run(() => { PrintCounter(); });
        }
        static void PrintCounter()
        {
            Console.WriteLine($"Kind Thread : {Thread.CurrentThread.ManagedThreadId} opgestart");
            for (int count = 1; count <= 5; count++)
            {
                Console.WriteLine($"teller waarde: {count}");
            }
            Console.WriteLine($"Kind Thread : {Thread.CurrentThread.ManagedThreadId} beëindigd");
        }
    }
}

```

Task laten wachten via Wait():

Tasks runnen asynchroon op een thread pool. Normaal gezien zal een kind thread blijven werken totdat het is beëindigd, zelfs als de Main thread reeds beëindigd.

Het is mogelijk om de uitvoering van de Main thread te laten wachten totdat al zijn kind-Tasks zijn beëindigd.

Dit is mogelijk via de Wait methode van de Task class. De Wait method zal de uitvoering van de thread(s) blokkeren totdat de task waarop de Wait methode wordt aangeroepen is beëindigd.

In het volgende voorbeeld wordt de Wait() methode op het task1 object aangeroepen om te zorgen dat de uitvoering van het programma(Main thread) wacht totdat task1 afgelopen is.

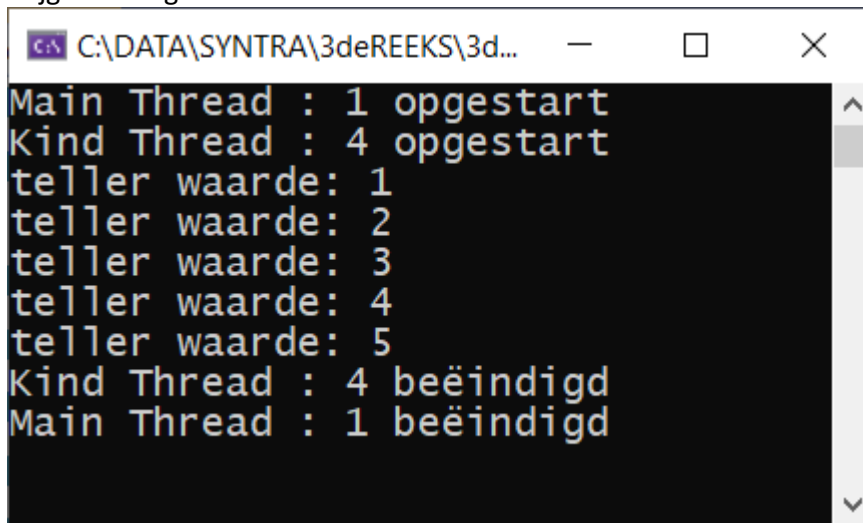
```

using System;
using System.Threading;
using System.Threading.Tasks;

namespace AsyncProg
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine($"Main Thread : {Thread.CurrentThread.ManagedThreadId} opgestart");
            Task task1 = Task.Run(() =>
            {
                PrintCounter();
            });
            task1.Wait();
            Console.WriteLine($"Main Thread : {Thread.CurrentThread.ManagedThreadId} beëindigd");
            Console.ReadKey();
        }
        static void PrintCounter()
        {
            Console.WriteLine($"Kind Thread : {Thread.CurrentThread.ManagedThreadId} opgestart");
            for (int count = 1; count <= 5; count++)
            {
                Console.WriteLine($"teller waarde: {count}");
            }
            Console.WriteLine($"Kind Thread : {Thread.CurrentThread.ManagedThreadId} beëindigd");
        }
    }
}

```

Wanneer je de Wait() methode op een Task gebruikt (hier task1), zal de oproepende thread (de main thread in dit geval) wachten met uitvoering totdat task1 is beëindigd. Run de app en je krijgt het volgende resultaat:



```
C:\DATA\SYNTRA\3deREEKS\3d...
Main Thread : 1 opgestart
Kind Thread : 4 opgestart
teller waarde: 1
teller waarde: 2
teller waarde: 3
teller waarde: 4
teller waarde: 5
Kind Thread : 4 beëindigd
Main Thread : 1 beëindigd
```

C# Task Return Value

Met de **Task<T>** class is het mogelijk om een terugkeerwaarde te krijgen .

Voorbeeld:

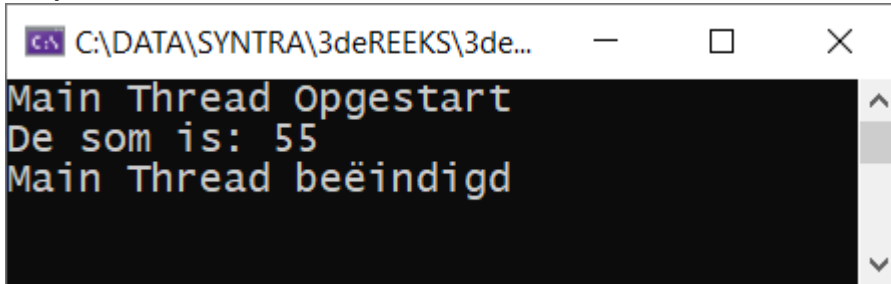
De CalculateSum methode heeft een double als return value:

```
static void Main(string[] args)
{
    Console.WriteLine($"Main Thread Opgestart");
    Task<double> task1 = Task.Run(() =>
    {
        return CalculateSum(10);
    });

    Console.WriteLine($"De som is: {task1.Result}");
    Console.WriteLine($"Main Thread beëindigd");
    Console.ReadKey();
}

static double CalculateSum(int num)
{
    double sum = 0;
    for (int count = 1; count <= num; count++)
    {
        sum += count;
    }
    return sum;
}
```

Output:



```
C:\DATA\SYNTRA\3deREEKS\3de...  
Main Thread Opgestart  
De som is: 55  
Main Thread beëindigd
```

Opmerking: De **Result** property van het Task object **blokkeert** de Main thread totdat de som is berekend.

Voorbeeld 2:

```
static void Main(string[] args)
{
    Console.WriteLine($"Main Thread gestart");
    Task<double> task1 = Task.Run(() =>
    {
        double sum = 0;
        for (int count = 1; count <= 10; count++)
        {
            sum += count;
        }
        return sum;
    });
    Console.WriteLine($"som is: {task1.Result}");
    Console.WriteLine($"Main Thread beëindigd");
    Console.ReadKey();
}
```

Referenties

<https://www.c-sharpcorner.com/article/task-and-thread-in-c-sharp/>