Tasks, Threads, Threading

<https://www.youtube.com/watch?v=gfkuD_eWM5Y>

## **Tasks**

## Vytvoreni :

trida Task ma několik konstruktoru, jeden z parametru je Action delegat, ktereho muzeme napsat pomoci lambdy

Task t = new Task(()=> {

ConsoleWriteLine(“Task starts”);

Thread.Sleep(2000);

ConsoleWriteLine(“Task ends”);

});

**t.Start();** spusti task

**t.Stop()** zastavi task

**Task.Wait(t)** pocka v tomto miste nez Task t dokonci svoji praci

**Vytvoreni se spustenim**

Tast t = Task.**Run**(() =>{

ConsoleWriteLine(“Task starts”);

Thread.Sleep(2000);

ConsoleWriteLine(“Task ends”);

});

## Task.Factory

Pokud nechci vytvaret vice Tasku rucne (t1 = … , t1.Start() t2 = … t2.Start() atd), muzu pouzit Task.Factory.

Task.Factory.StartNew – vytvori a spusti task (stejne jako Task.Run())

Task.Factory.StartNew(() => DoSomething())

## ContinueWith

Pokud chceme, aby se po skonceni tasku pokracovalo jinym delegatem pridame na konec volani .ContinueWith(delegat) . Jde proste o zretezeni volani metod. Po skonceni provadeni prvni metody se zacne vykonavat dalsi.

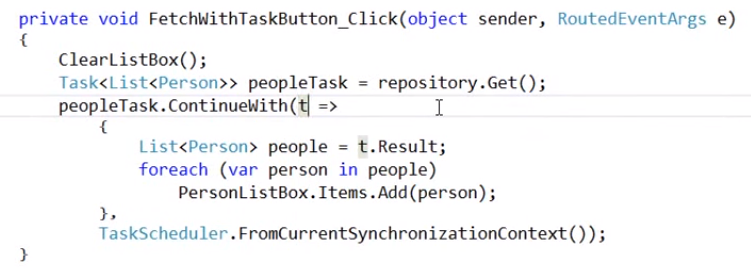
var t1 = Task.Factory.StartNew(() => DoSomething()).ContinueWith((prevTask) => DoSomethingElse());

## TaskScheduler.FromCurrentSynchronisationContext

<https://www.youtube.com/watch?v=0qiB3oW_nd8&list=PLdbkZkVDyKZWWjJ5yUdd_ooLORZgOSHSP.>

Mam Task po jehoz zkonceni chci zavolat novy task. V tomto nasl. Tasku chci predat UIThreadu vysledek ulohy. Dostanu vyjimku, ze nejsem na UIThreadu. K predani slouzi planovac uloh (TaskSheduler)

Parametrem FromCurrentSynchronisationContext rikam, ze mam pouzi stejny planovac uloh, jako je ten ze ktereho jsem vysel. V tomto prikladu je to UIThread protoze se zde reaguje na stlaceni tlacitka.



## Continuation option

Pouzitim tasku mame plnou kontrolu nad bezici operaci. V pripade, ze v bezicim tasku vznikne vyjimka a pouzili jsme ContinueWith, muzeme si pomoci continuation option zvolit za jakych okolnosti bude nasledujici task pokracovat.

Continuation option je enum ve kterem jsou moznosti jako NotOnCanceled, NotOnFaulted, OnlyOnCancelled atd..

Prvnimu tasku muzeme priradit hned nekolik .continueWith casti ve kterych definujeme co se ma stat pokud task projde, spadne, nebo proste co se ma stat vzdy za jakychkoli okolnosti. Kdyz chci tuto funkcionalitu pouzit v await-async muzu to udelat jedine pomoci bloku try-catch-finally

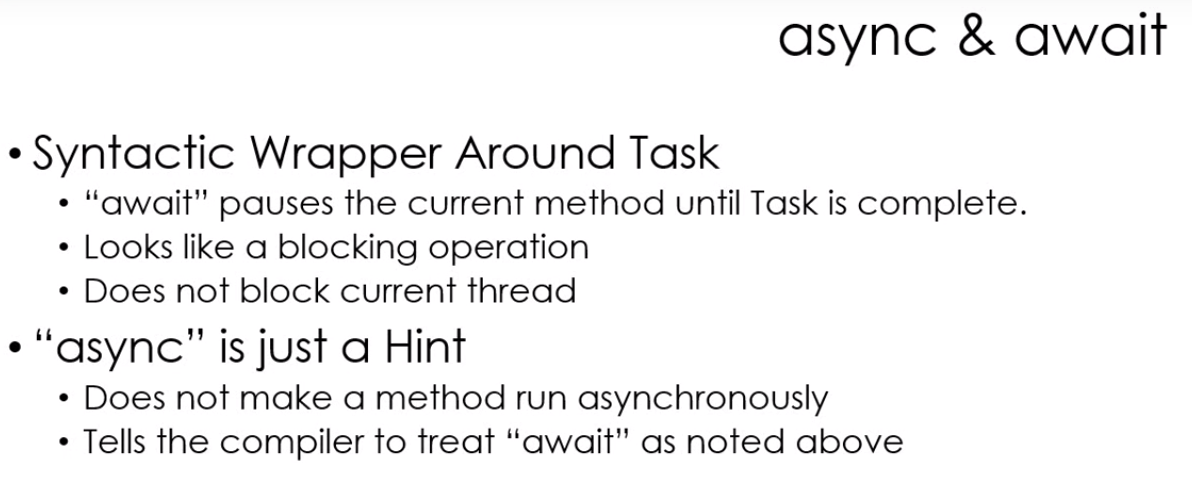
## Async await

Viz take: [Await Async\Await Async.docx](Await%20Async/Await%20Async.docx)

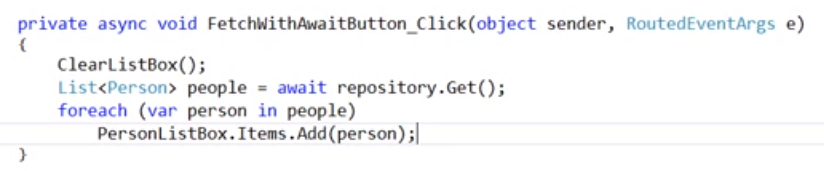
Je jakysi wrapper okolo tasku. “await” jen **zapauzuje** async metodu, dokud v ni bezici task nedobehne. Nezablokuje UI ani current threa. Nespousti metodu asynchronne. To dela az Task.Run(action)

“await” = pockej na Task

Je to nejjednodussi pouziti tasku, ale zaroven neposkytuje zadne moznosti kontroly tasku.



Nasledujici kod provadi to co bylo popsano v TaskScheduler.FromCurrentSynchronisationContext, pomoci async-await.



Pokud chci napodobit ovladani jako pri pouziti continuation option u tasku, musim pouzit try catch finally

## Parallel

Vytvori nekolik paralelnich vlaken naraz. Hlavni duvod pouziti tridy Parallel je, ze praci rozdeli rovnomerne na vsechny jadra procesoru.

var intList = new List<int> {1,2,3,4,5,6,7}

Parallel.ForEach(intLIst, (i)=> Console.WriteLine())

Vystup bude sprehazeny.

Parallel.For(0, 100 (i)=> Console.WriteLine(i))

Vypise cisla od nula do sto pomoci vice vlaken. Vsechny operace za tim, cekaji na dokonceni For loopu

## CancellationTokenSource

Pomuze zastavit pribihajici operaci na tasku. Tasku predam token na kterem muzu zavolat metodu Cancel . Pomoci Task.Factory spustim task v jinem threadu. V nasledujicim kodu muzu zavolat Cancel a probihajici task zastavit.

var source = new **CancellationTokenSource**();

try

{

var t1 = Task.Factory.StartNew(() => DoSomething(source.Token)).ContinueWith((prevTask) => DoSomethingElse(source.Token))

**source.Cancel(); // toto zastavi probihajici Task**

}

catch (Exception e){ cw ex.GetType()}

static void DoSomething(**CancelationToken token**)

{

If(token.IsCancellationRequested)

{

Console.WriteLine(“CancellationRequested ”);

token.throwIfCancellationRequested

}

// zbytek metody

}

# **Threads**

## Vytvoreni

Thread t = new Thread (threadStart);

Thread t = new Thread (parametrizedThreadStart);

## Spusteni

Thread.Start()

TaskFactory (vytvori a spusti vlakna)

## Lock

Pomoci Lockovani muzeme vlakna udelat threadsafe.

Lock neboli zamek, bere jako argument jakykoli objekt.

## AutoResetEvent a ManualResetEvent

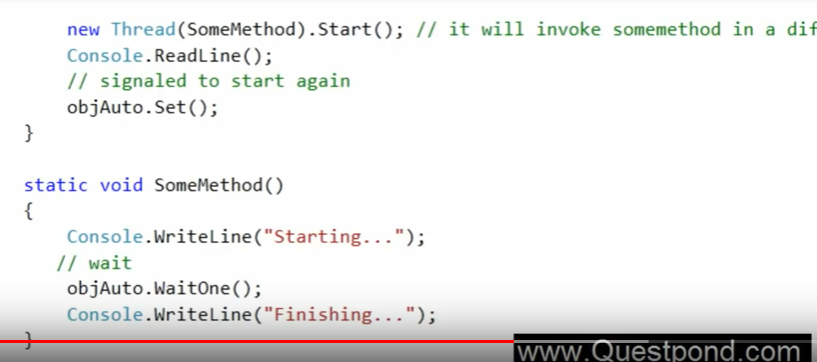
Pokud se vlakno dostane k mistu kde se vola na objektu AutoResetEvent metoda WaitOne(), prepne se do tzv. wait mode a ceka, az jine (typicky hlavni) vlakno na tomto objektu zavola Set();

<https://www.youtube.com/watch?v=xaaRBh07N34&t=302s>

AutoResetEvent objAuto = new AutoResetEvent (false);

objAuto.WaitOne() – vlakno dostane signal, ze si ma dat pauzu.

objAuto.Set(); - pauznute vlakno dostane signal, ze muze pokracovat.



Main thread vytvori vlakno. Vytvorene vlakno vleze do metody SomeMethod, tam se stopne. Main thread mezitim pokracuje a ceka na stisknuti klavesy na konzoli (Console.ReadLine) . Jakmile se klavesa stiskne, main thread zavola na objektu typu AutoResetEvent (na tom ktery stopnul vlakno), metodu Set(), cimz worker thread odblokuje a to muze pokracovat.

**Rozdil mezi AutoResetEvent a ManualResetEvent**

**AutoResetEvent** – (volame metody WaitOne a Set na objektu typu AutoResetEvent) Na vlakne muzeme volat WaitOne kolikrat chceme. Pokazde musime ale vlakno odblokovat pomoci metody Set()

**ManualResetEvent** – (volame metody WaitOne a Set na objektu typu ManualResetEvent) jakmile zavolame metodu Set() jednou, uz se vlakno nezastavi na dalsim volani WaitOne na vlakne.