Programowanie Współbieżne - Tasks

Paweł Paduch

Politechnika Świętokrzyska

10 czerwca 2022

Plan wykładu

1. Wstęp
   - Plan
   - Literatura
   - Podstawowe pojęcia
   - Paterny programowania asynchronicznego
   - Porównanie
2. TAP
   - Nazewnictwo i zwracane typy
   - Inicjowanie
   - Wyjątki
   - Wykonanie
3. TAP
   - Statusy
   - Cancelation
   - Progress
   - Progress - Implementacja
4. Przykłady
   - Startowanie zadań
   - Async i Await
   - Czekanie i kontynuacja
   - Kontynuacja warunkowa
   - Task Schedullers

Literatura

- Paterny programowania asynchronicznego - https://docs.microsoft.com/pl-pl/dotnet/standard/asynchronous-programming-patterns/?view=netframework-4.7.2
- TAP - https://docs.microsoft.com/pl-pl/dotnet/standard/asynchronous-programming-patterns/task-based-asynchronous-pattern-tap?view=netframework-4.7.2

Task

- Klasa *Task* reprezentuje pojedynczą operację nie zwracającą wartości i zwykle wykonującą się asynchronicznie.
- W przypadku operacji zwracającej wartość należy użyć klasy *Task<TResult>*
- Task jest centralną klasą reprezentującą TAP (Task-based Asynchronous Pattern)
- Taski zwykle wykonują się na puli wątków w sposób asynchroniczny, dlatego można używać właściwości *Status* takich jak *IsCanceled*, *IsCompleted* czy *IsFaulted*
- Do określenia zadania jakie ma wykonać *task*, zwykle stosuje się notację *Lambda*

Asynchronous Programming Model (APM)

Asynchronous Programming Model (APM) pattern, zwany też IAsyncResult pattern

- jest starszym modelem wykorzystującym interfejs IAsyncResult do uzyskania asynchronicznego zachowania
- synchorniczne operacje wymagają metod Begin i End np. BeginWrite i EndWrite do zaimplementowania operacji asynchronicznych.
- nie jest już rekomendowany dla nowych projektów aplikacji.

Event-based Asynchronous Pattern (EAP)

Event-based Asynchronous Pattern (EAP)

- został wprowadzony do frameworka .Net 2.0
- model bazujący na zdarzeniach
- dostarcza asynchronicznego zachowania
- wymaga:
  - metod z sufiksem *Async*
  - jednego lub więcej zdarzeń (*event*)
  - delegatów
  - event handlerów
  - Typów pochodzących od *EventArg*
- obecnie nie jest rekomendowanym paternem

Task-based Asynchronous Pattern (TAP)

Task-based Asynchronous Pattern (TAP)

- Wprowadzony w .net framework 4
- Bazuje na *Task* i *Task<TResult>* w przestrzeni nazw *System.Threading.Tasks*
- Używa pojedynczej metody reprezentującej inicjacje i zakończenie operacji asynchronicznej
- Rekomendowany jako patern operacji asynchronicznych
- w C# dwa nowe słowa kluczowe *async* i *await*

Porównanie APM

Dla porównania rozważmy asynchroniczną metodę czytającą określoną ilość danych od pewnego ofsetu do wskazanego bufora. W przypadku APM wystawione były by dwie metody

Listing 1: Przykład APM

```
public class MyClass
{
    public IAsyncResult BeginRead(
        byte [] buffer, int offset, int count,
        AsyncCallback callback, object state);
    public int EndRead(IAsyncResult asyncResult);
}
```

Porównanie EAP

EAP musiałby wystawić następujący zestaw typów i zmiennych

Listing 2: Przykład EAP

```
1 public class MyClass
2 {
3     public void ReadAsync(byte [] buffer, int offset, int count);
4     public event ReadCompletedEventHandler ReadCompleted;
5 }
```

Porównanie TAP

W przypadku TAP wystarczyłaby jedna metoda:

Listing 3: Przykład TAP

```
1 public class MyClass
2 {
3     public int Read(byte [] buffer, int offset, int count);
4 }
```

Nazewnictwo i typy

Asynchroniczne metody w *TAP* zawierają przyrostek *Async* po nazwie operacji oraz zwracają typy typu *await* (*awaitable*) takie jak:

- *Task*,
- *Task<TResult>*,
- *ValueTask*,
- *ValueTask<TResult>*

Przykładowo asynchroniczna metoda *Get* zwracająca *Task<string>* może być nazwana *GetAsync*

Nazewnictwo i typy

- Jeżeli dodajemy asynchroniczne metody *TAP* do klasy, która zawiera już metody *EAP* z przyrostkiem *Async* powinniśmy używać przyrostka *TaskAsync*
- Jeżeli metoda asynchroniczna zaczyna jakąś operację ale nie zwraca powyższych typów *awaitable*, to jej nazwa powinna zaczynać się od *Begin* lub *Start* lub innej nazwy sugerującej że nie zwróci typu *awaitable*
- Metoda *TAP* zwraca albo *System.Threading.Tasks.Task* albo *System.Threading.Tasks.Task* $<$*TResult*$>$ w zależności od tego, czy odpowiednia synchroniczna metoda zwraca *void* czy typ *TResult*.

parametry

- Parametry metody TAP powinny być zgodne z parametrami jej synchronicznego odpowiednika i powinny być dostarczone w tej samej kolejności.
- Parametry *out* i *ref* są wyłączone z tej reguły i nie należy ich stosować.
- Dane zwracane przez *out* lub *ref*, powinny zostać zwrócone jako część *TResult* zwróconego przez *Task* <*TResult*>
- W przypadku zwrotu wielu wartości powinniśmy użyć kolekcji lub bardziej rozbudowanej struktury.
- Należy rozważyć dodanie parametru *CancellationToken*, nawet jeśli synchroniczny odpowiednik metody *TAP* nie oferuje takiego parametru.

Odstępstwa w nazewnictwie

Metody, które służą wyłącznie tworzeniu, manipulowaniu lub łączeniu zadań (gdzie asynchroniczne zamiary metody są jasne w nazwie metody lub w nazwie typu, do którego należy metoda), nie muszą być zgodne z tym wzorem nazewnictwa; takie metody są często określane jako kombinatory. Np. *WaitAll*, *WaitAny*

Inicjowanie operacji asynchronicznej

Metoda asynchroniczna oparta na TAP może synchronicznie wykonywać niewielką część zadnia, na przykład sprawdzać argumenty i inicjować operację asynchroniczną, zanim zwróci wynikowe zadanie. Jednakże cześć synchroniczna powinna być ograniczona do minimum z dwóch powodów.

- jeżeli metoda asynchroniczna jest wołana z wątku UI (interfejsu użytkownika), to może to doprowadzić do zamrożenia go.
- gdy chcemy wystartować wiele metod asynchronicznych, wtedy każda część synchroniczna opóźnia wywołanie kolejnej metody.

W niektórych przypadkach czas inicjacji operacji asynchronicznej może przewyższać samą operację wykonaną synchronicznie, wtedy nie powinniśmy używać asynchroniczności.

Wyjątki w metodzie asynchronicznej

- Metoda asynchroniczna powinna rzucić wyjątek, tylko w odpowiedzi na błąd użycia.
- Wyjątki występujące podczas uruchamiania metody asynchronicznej powinny być przypisane do zwracanego zadania, nawet jeśli metoda asynchroniczna zakończy się synchronicznie przed zwróceniem zadania.
- Zwykle zadanie zawiera co najwyżej jeden wyjątek. Jeśli jednak zadanie reprezentuje wiele operacji (na przykład *WhenAll*), może być z nim związanych wiele wyjątków.

Miejsce wykonania

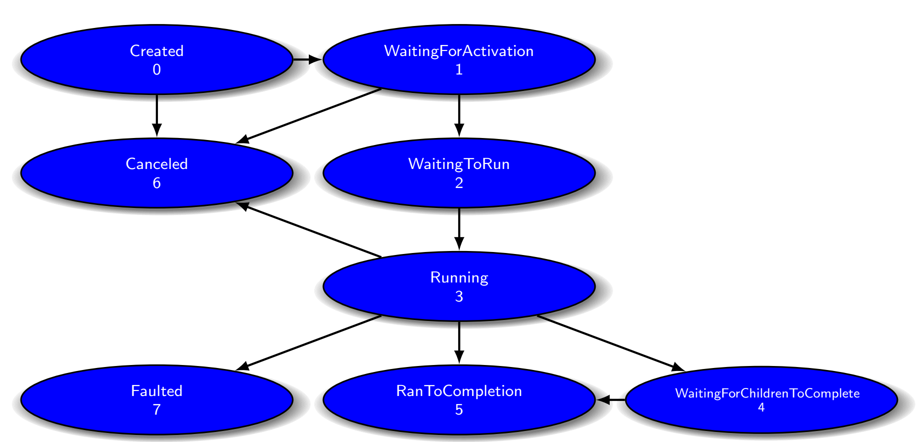
Po zaimplementowaniu metody TAP można określić, gdzie występuje wykonanie asynchroniczne.

- wykonanie na puli wątków
- za pomocą asynchronicznego we/wy (bez wiązania się z wątkiem przez większość wykonywania operacji).
- uruchomić w określonym wątku (np. wątek interfejsu użytkownika).
- lub użyć dowolnej liczby potencjalnych kontekstów
- Metoda TAP nawet nie musi nic wykonywać, wystarcz, że zwróci zadanie reprezentujące wystąpienie jakiegoś warunku w innym miejscu systemu. Np. z informacją, że pojawiły się dane w kolejce.

Program wywołujący

- Program wołający asynchroniczną metodę może zostać zablokowany w oczekiwaniu na wynikowe zadanie albo wywołać dodatkowy kod kontynuacji po zakończeniu operacji asynchronicznej.
- Twórca kodu kontynuacji decyduje gdzie ten kod ma być wykonany. Może być on jawnie utworzony za pomocą metod klasy *Task* np. *ContinueWith* albo niejawnie np. *await*.

Statusy



Created

- Jest to tak zwane zadanie "zimne", bez aktywacji.
- Stan zadania zaraz po utworzeniu przez konstruktor *Task*
- Przejście do innego stanu tylko przy pomocy zawołania metody *Start* lub *RunSynchronously* na instancji zadania.
- Jeżeli metoda TAP tworzy wewnętrznie zadanie za pomocą konstruktora *Task*, to przed zwróceniem jego instancji musi go aktywować.
- Konsumenci metody TAP mogą założyć, że zwrócone zadanie jest aktywne i nie powinni wywoływać już *Start* by go aktywować, ponieważ spowoduje to wyjątek *InvalidOperationException*.

WaitingForActivation

- Stan zadania zaraz po utworzeniu przez metody takie jak *ContinueWith*, *ContinueWhenAll*, *ContinueWhenAny* a także *FromAsync*
- Zadanie nie jest jeszcze zaharmonogramowane i nie będzie dopóki zadania, na które czekają nie zakończą się.
- Zadanie będzie aktywowane i zaharmonogramowane wewnętrznie przez infrastrukturę .NET

WaitingToRun

- Zadanie zaharmonogramowane i czekające na uruchomienie
- Stan początkowy dla zadań utworzonych przez *TaskFactory.StartNew*. Przynajmniej do czasu powrotu z funkcji *StartNew*. Ale może się zdarzyć, że od razu powrócą ze statusem *Running* lub nawet *RanToCompletion*

Running

- Zadanie w trakcie wykonywania

WaitingForChildrenToComplete

- Zadanie wykonało się, ale czeka na wykonanie zadań potomnych

RanToCompletion

- Jeden z 3 finalnych stanów.
- Zadanie dotarło szczęśliwie do końca bez rzucanych wyjątków czy anulowania.

Canceled

- Jeden z 3 finalnych stanów.
- Zadanie w tym stanie jest zakończone za pomocą *Cancel*

Faulted

- Jeden z 3 finalnych stanów.
- Zadanie do tego stanu przechodzi gdy zakończy się nieobsłużonym wyjątkiem
- lub jedno z potomnych zadań zakończy się stanem *Faulted*.

Anulowanie

- W TAP anulowanie jest opcjonalne zarówno dla implementacji metod asynchronicznych jak i dla ich konsumentów.
- Jeżeli operacja pozwala na anulowania wystawia przeciążoną asynchroniczną akceptującą instancję *CancellationToken*.
- Z przyjętą nomenklaturą parametr ten nazywa się *cancellationToken*.

Listing 4: przykład

```
1 public Task ReadAsync(byte [] buffer, int offset, int count,
2                       CancellationToken cancellationToken)
```

Anulowanie

- Operacja asynchroniczna sprawdza token anulowania i może go honorować i anulować operacje.
- Jeżeli spowoduje to przedwczesne zakończenie pracy, metoda TAP zwraca zadanie, które kończy się w stanie *Canceled*.
- Nie ma dostępnego wyniku a wyjątek nie jest rzucany.
- Stan *Canceled* jest uważany za końcowy. (completed) $IsCompleted = true$, wraz ze stanami *Faulted* i *RanToCompletion*.
- Kiedy zadanie zostanie anulowane w stanie *Canceled*, wszelkie kontynuacje zarejestrowane w zadaniu są planowane lub wykonywane chyba, że określono opcję kontynuacji *NotOnCanceled* aby z niej zrezygnować.

Anulowanie

- Każdy kod asynchronicznie oczekujący na anulowane zadanie za pomocą funkcji językowych nadal działa, ale otrzymuje wyjątek *OperationCanceledException* lub jego pochodne.
- Kod blokowany synchronicznie oczekujący na zadanie za pomocą metod takich jak *Wait* i *WaitAll* również nadal działa z wyjątkiem.
- Jeśli token anulowania zażądał anulowania przed wywołaniem metody TAP, która akceptuje ten token, powinna ona zwrócić anulowane zadanie.
- Jeśli jednak anulowanie jest wymagane podczas działania asynchronicznego, operacja asynchroniczna nie musi akceptować żądania anulowania.

Anulowanie

Jak jasno określić co jest anulowalne a co nie?

- W przypadku metod asynchronicznych, które chcą wystawić możliwość anulowania, najlepiej nie wystawiać przeciążenia bez tokena anulowania.
- Wywołujący taką metodę, który nie chce jej anulacji będzie miał możliwość podania *None* zamiast tokena.
- Natomiast gdy chcemy uniemożliwić anulowanie, nie tworzymy przeciążenia akceptującego token anulowania.
- Pomaga to wskazać wywołującemu, czy metoda docelowa jest faktycznie anulowalna.

Anulowanie

- Zwrócone zadanie powinno zakończyć się w stanie *Canceled* tylko wtedy, gdy operacja zakończy się w wyniku żądania anulowania.
- Jeśli zażądano anulowania, ale wynik lub wyjątek jest nadal generowany, zadanie powinno zakończyć się stanem *RanToCompletion* lub *Faulted*.

Progress

Niektóre operacje asynchroniczne korzystają z dostarczania powiadomień o postępie; są one zwykle używane do aktualizacji interfejsu użytkownika z informacjami o postępie operacji asynchronicznej

Zapewnienia interfejs postępu, gdy wywoływana jest metoda asynchroniczna.

Podobnie jak w przypadku anulowania, implementacje TAP powinny zapewniać parametr *IProgress*<*T*> tylko wtedy, gdy API obsługuje powiadomienia o postępie.

Progress

Interfejs postępu obsługuje różne implementacje postępu, określone przez używający go kod. Np.:

- kod używający (konsument) może tylko troszczyć się o najnowszą aktualizacje albo buforować wszystkie elementy.
- można podpiąć akcję obsługującą zdarzenie pojawienia się każdej aktualizacji
- można chcieć kontrolować, czy wywołanie jest kierowane do konkretnego wątku.

Wszystkie te opcje można osiągnąć, wykorzystując inną implementację interfejsu, dostosowaną do konkretnych potrzeb konsumenta.

Progress - przykładowo

Jeśli metoda *ReadAsync* byłaby w stanie raportować postęp pośredni w postaci liczby odczytanych bajtów, wywołanie zwrotne może być interfejsem *IProgress*<*T*>:

Listing 5: przykład

```
1   public Task ReadAsync(byte[] buffer, int offset, int count,
2                         IProgress<long> progress)
```

Progress - przykładowo

Jeśli metoda FindFilesAsync zwraca listę wszystkich plików, które spełniają określony wzorzec wyszukiwania, wywołanie zwrotne postępu może dostarczyć oszacowania procentu ukończonej pracy, jak również bieżącego zestawu częściowych wyników. Może to zrobić albo z krotką:

Listing 6: przykład

```
1   public Task<ReadOnlyCollection<FileInfo>> FindFilesAsync(
2             string pattern,
3             IProgress<Tuple<double,
4             ReadOnlyCollection<List<FileInfo>>>> progress)
5
```

Progress - przykładowo

...albo typem danych specyficznym do API:

Listing 7: przykład

```
1  public Task<ReadOnlyCollection<FileInfo>> FindFilesAsync(
2     string pattern,
3     IProgress<FindFilesProgressInfo> progress)
4
```

W tym drugim przypadku specjalny typ danych jest zwykle uzupełniany o ProgressInfo.

Progress

Jeśli implementacje TAP zapewniają przeciążenia, które akceptują parametr postępu, to muszą one zezwolić argumentowi *progress* na wartość *NULL*, W takim przypadku konsument metody nie jest zainteresowany raportowaniem postępu i nie powinniśmy go zgłaszać. Trzeba w metodach obsługujących sprawdzać czy *progress* != *NULL*

Progress

Implementacje TAP powinny raportować postęp do obiektu Progress<T> synchronicznie, co umożliwi asynchronicznej metodzie szybkie dostarczenie danych o postępie oraz pozwoli konsumentom określić jak i gdzie najlepiej obsłużyć uzyskaną informację o postępie. Na przykład instancja postępu może zdecydować się na przeprowadzenie wywołań zwrotnych. Np. Obsłużyć zdarzenie raportowania.

Progress - Implementacja

.NET Framework 4.5 zapewnia pojedynczą implementację *IProgress<T>: Progress<T>*. Klasa *Progress<T>* jest deklarowana w następujący sposób:

Listing 8: przykład

```
public class Progress<T> : IProgress<T>
{
    public Progress();
    public Progress(Action<T> handler);
    protected virtual void OnReport(T value);
    public event EventHandler<T> ProgressChanged;
}

```

Progress

- Instancja *Progress<T>* dostarcza zdarzenie *ProgressChanged*, które jest wywoływane za każdym razem, gdy asynchroniczna operacja zgłasza aktualizację postępu.
- Zdarzenie *ProgressChanged* jest wywoływane w obiekcie *SynchronizationContext*, który został przechwycony, gdy instancja *Progress<T>* została utworzona.
- Jeśli nie był dostępny żaden kontekst synchronizacji, używany jest kontekst domyślny, który kieruje do puli wątków.
- Do tego zdarzenia można podczepić procedury obsługi (*handlers*). Dla wygody można jedną z nich podać w kontstruktorze *Progress<T>*
- Aktualizacje postępu są wywoływane asynchronicznie by uniknąć opóźnień.

Przeciążenia

Jeśli implementacja TAP korzysta z opcjonalnych parametrów *CancellationToken* i opcjonalnego *IProgress<T>*, może potencjalnie wymagać do czterech przeciążeń:

Listing 9: przykład

```
1 public Task MethodNameAsync(...);
2 public Task MethodNameAsync(..., CancellationToken cancellationToken);
3 public Task MethodNameAsync(..., IProgress<T> progress);
4 public Task MethodNameAsync(...,
5     CancellationToken cancellationToken, IProgress<T> progress);
6
```

Jakkolwiek, wiele implementacji nie obsługuje *CancellationToken* i *IProgress<T>* i wystarczy pierwsza wersja.

Task Start

Tworzymy instancję klasy Task po czym startujemy zadanie. Akcja jest opisana wyrażeniem lambda.

Listing 10: przykład new Task

```
1        var task = new Task(() =>
2            {
3                Console.WriteLine("First task is working...");
4                Thread.Sleep(1000);
5                Console.WriteLine("First task finished");
6            });
7            task.Start();
8            task.Wait();
9
```

Po wywołaniu metody *Start* wątek główny idzie dalej, aby poczekać na zakończenie zadania, należy wywołać *Wait()*

Task Run

Posługujemy się statyczną metodą *Run*. Tworzy ona uruchomiony *Task*, więc nie trzeba wywoływać metody *Start()*

Listing 11: przykład Task.Run

```
1           task = Task.Run(() => {
2                Console.WriteLine("Task Run is working");
3                Thread.Sleep(1000);
4                Console.WriteLine("Task Run finished");
5            });
6            task.Wait();
7
```

Task Factory

Posługujemy się fabryką i statyczną metodą *StartNew* . Tworzy ona uruchomiony *Task*, więc nie trzeba wywoływać metody *Start()*

Listing 12: przykład Task.Factory.Run

```
1            task = Task.Factory.StartNew(() => {
2                  Console.WriteLine("Task from Factory is working");
3                  Thread.Sleep(1000);
4                  Console.WriteLine("Task from Factory finished");
5              });
6              task.Wait();
7
```

Wiele zadań

Tworzymy wiele zadań wykonujących tę samą akcję. CurrentId jest przypisywane w momencie odwołania się do niego a nie przy starcie.

Listing 13: Przykład Mutli Task start

```
1  Action a = () =>
2          {
3              Console.WriteLine($"Task no {Task.CurrentId} has started");
4              Thread.SpinWait(new Random().Next(300000000));
5              Console.WriteLine($"Task no {Task.CurrentId} finished");
6          };
7  List<Task> listaZadan = new List<Task>();
8  for (int i = 0; i < 10; i++)
9          {
10             listaZadan.Add(new Task(a));
11         }
12 listaZadan.ForEach(t => t.Start());
13 listaZadan.ForEach(t => t.Wait());
14
```

Przekazanie parametru

Przekazanie informacji do zadania

Listing 14: Przykład przekazanie parametru

```
1  Task zadanieWyswietlCos = new Task((o) =>
2              {
3                  for (int i = 0; i < 5; i++)
4                  {
5                      Thread.Sleep(200);
6                      Console.WriteLine((string)o+" "+i);
7                  }
8              }, "Ala ma kota");
9              zadanieWyswietlCos.Start();
10             zadanieWyswietlCos.Wait();
```

Zwracanie danych z zadania

Przekazanie informacji od zadania

Listing 15: Przykład zwrócenia wartości

```
1        Task<int> intTask = Task.Run(() =>
2            {
3                Thread.Sleep(1000);
4                return 5;
5            });
6            Console.WriteLine("Waiting for value from task");
7            var result = intTask.Result;
8            Console.WriteLine($"We received value: {result}");
9
```

W linii 7. Czekamy na zakończenie zadania, dopiero wtedy możemy odczytać wynik

Zwykła praca

Zwykła synchroniczna metoda zajmująca przez chwilę procesor.

Listing 16: Przykład metody synchronicznej

```
1          public void ZwyklaPraca(int ile)
2         {
3             Console.WriteLine($"Worker {Name} has begun synchro work....");
4             for (int i = 1; i <= ile; i++)
5             {
6                 Thread.SpinWait(100000000);
7                 Console.WriteLine($"Worker {Name} has produced {i}");
8             }
9             Console.WriteLine($"Worker {Name} finished his work");
10        }
```

Listing 17: Przykład wywołania metody

```
1         Worker pracownik1 = new Worker("Mietek");
2             Console.WriteLine("We order the synchro job to the worker");
3             pracownik1.ZwyklaPraca(10);
4             Console.WriteLine("Synchro job was ordered and done");
```

Praca bez nadzoru

Ta metoda utworzy zadanie, ale będzie to na zasadzie “uruchom i zapomnij”.

Listing 18: Przykład metody tworzącej zadanie

```
1   public void PracaTask(int ile)
2         {   Console.WriteLine($"Worker {Name} has begun synchro work....");
3             Task t = new Task(() =>
4             {
5                 for (int i = 1; i <= ile; i++)
6                 {   Thread.SpinWait(100000000);
7                     Console.WriteLine($"Worker {Name} has produced {i}");
8                 }
9                 Console.WriteLine($"Worker {Name} finished his work");
10            });
11            t.Start();    }
```

Listing 19: Przykład wywołania metody

```
1 Worker pracownik2 = new Worker("Zenon");
2 Console.WriteLine("We order the job to the worker and he create task");
3 pracownik2.PracaTask(10);
4 Console.WriteLine("Job for Zenon was ordered. He started, but we don't know when he finish");
```

Praca o której wiemy

Metoda zwracająca zadanie, zgodnie z zasadami TAP task zwracany musi być już uruchomiony. Jest to metoda asynchroniczna, może się więc nazywać xxxAsync.

Listing 20: Przykład metody tworzącej i zwracającej zadanie

```
1  public Task PracaAsync(int ile)
2          {
3              Console.WriteLine($"Worker {Name} has begun synchro work....");
4              Task t = new Task(() =>
5              {
6                  for (int i = 1; i <= ile; i++)
7                  {
8                      Thread.SpinWait(100000000);
9                      Console.WriteLine($"Worker {Name} has produced {i}");
10                 }
11                 Console.WriteLine($"Worker {Name} finished his work");
12             });
13             t.Start();
14             return t;
15         }
```

Praca o której wiemy

Wywołanie metody zwracającaj zadanie. Ponieważ jest ona awaitable można poczekać na na jej zakończenie.

Listing 21: Przykład wywołania metody tworzącej i zwracającej zadanie

```
1  public Task PracaAsync(int ile)
2 Worker pracownik3 = new Worker("Janusz");
3 Console.WriteLine("We order the job to the worker and he create task");
4 var zadanieJanusza = pracownik3.PracaAsync(10);
5 Console.WriteLine("Job for Janusz was ordered. He returned Task so we are able to wait for it↩
       .");
6 zadanieJanusza.Wait();
```

Czekanie awaitem

Wywołanie metody zwracającaj zadanie. Tym razem czekanie za pomocą await.

Listing 22: Przykład wywołania metody tworzącej i zwracającej zadanie

```
1 Worker pracownik4 = new Worker("Grazyna");
2 Console.WriteLine("We order the job to the worker and he create task");
3 var zadanieGrazyny = pracownik4.PracaAsync(10);
4 Console.WriteLine("Job for Grazyna was ordered. She returned Task so we are able to use await↩
      ");
5 await zadanieGrazyny;
```

Gdy używamy słowa await to metoda w której używamy await musi być async.

async Task Main

Jak w Main zastosować metody asynchroniczne. Należ stworzyć sobie asynchronicznego maina

Listing 23: Przykład metody Main z async

```
1          static void Main(string[] args)
2          {
3              ///Sposób na wywoływanie metod asynchronicznych w Mainie
4              Task.Run(async () =>
5              {
6                  Console.WriteLine("In Main before await");
7                  await MainAsync(args);
8                  Console.WriteLine("I Main after await");
9              }).GetAwaiter().GetResult();
10             Console.WriteLine("Finish!!!");
11             Console.ReadLine();
12         }
13         public static async Task MainAsync(string[] args)
14         ...
15
```

Podwykonawca async await

Przykład metody z wykorzystaniem async await.

Listing 24: Przykład asynchronicznej używającej await

```
1 public async Task PracaZleconaAsync(int ile)
2         {
3             Console.WriteLine($"Worker {Name} has ordered job...");
4             await PracaAsync(ile);
5             Console.WriteLine($"Worker {Name} received ordered job (used await)");
6         }
7
```

Podwykonawca async await

Przykład wywołania metody z wykorzystaniem async await.

Listing 25: Wywołanie metody asynchronicznej używającej await

```
1 Worker pracownik5 = new Worker("Stefan");
2             Console.WriteLine("We are ordering the job to the worker Stefan and he's ordering
        job too");
3             var zadanieStefana = pracownik5.PracaZleconaAsync(10);
4             Console.WriteLine("The job for Stefan was ordered and he he ordered to
       subcontractor. \nStefan returned task so we can wait for it by await");
5             await zadanieStefana;
6
```

Czekamy na zadania

Przykład czekania na wszystkie zadania

Listing 26: Przykład czekania na zadania

```
1              var tasks = new Task[3];
2              for (var i = 0; i < 3; i++)
3              {
4                  tasks[i] = (Task.Run(() =>
5                  {
6                      Console.WriteLine("One of 3 is waiting");
7                      Thread.Sleep(i * 500);
8                      Console.WriteLine("One of 3 finished");
9                  }));
10             }
11             Console.WriteLine("We waiting for all 3");
12             Task.WaitAll(tasks);
13             Console.WriteLine("We got all 3");
14
15             var resultTasks = Task.WhenAll(tasks); //Zwraca zadanie ze wszystkimi
16         // var resultTasks = Task.WhenAll(tasks).Result; //żeby tak było taski muszą coś zwracać
17
```

Złe przekazanie parametru

Nieprawidłowe przekazanie danych do zadania.

Listing 27: Przykład złego przekazania parametru

```
1  var tasks2 = new Task<int>[3];
2  for (var i = 0; i < 3; i++)
3          {
4              tasks2[i] = (Task<int>.Run(() =>
5              {
6                  Thread.Sleep(i * 500);
7                  return i * 2;//Don't do that!!!
8              }));
9          }
10         var resultTasks2 = Task.WhenAll(tasks2);
11         foreach (var item in resultTasks2.Result)
12         {
13             Console.WriteLine($"Task count and return wrong number: {item}");
14         }
15
```

Dostajemy tą samą wartość bo korzystamy ze zmiennej *i*, która na koniec wynosi 3

Dobre przekazanie parametru

Dane przekazane do zadania poprzez parametr.

Listing 28: Przykład prawidłowego przekazania parametru

```
1  var tasks3 = new Task<int>[3];
2          for (var i = 0; i < 3; i++)
3          {
4              tasks3[i] = (new Task<int>((o) =>
5              {
6                  Thread.Sleep((int)o * 500);
7                  return (int)o * 2;
8              }, i));
9          }
10         tasks3.ToList<Task>().ForEach((t) => t.Start());
11
12         var resultTasks3 = Task.WhenAll(tasks3);
13         resultTasks3.Result.ToList().ForEach((i) => Console.Writeine($"Task count return:↩
       {i}"));
14
```

Dostajemy kolejne wartości bo *i* jest przekazywane przez parametr.

ContinueWith

Przykład prostej kontynuacji

Listing 29: Przykład prostej kontynuacji

```
1      Task taskPierwszy = Task.Factory.StartNew(() =>
2              {
3                  Console.WriteLine("First task has started");
4                  Thread.Sleep(1000);
5                  Console.WriteLine("First task has finished");
6              }
7              );
8              Task taskDrugi = taskPierwszy.ContinueWith(ant =>
9              {
10                 Console.WriteLine("Second task has started");
11                 Thread.Sleep(1000);
12                 Console.WriteLine("Second task has finished");
13             }
14             );
15             await taskDrugi;
```

ContinueWith z parametrem

Przykład zadania, które kontynuuje obliczenia po swoim poprzedniku

Listing 30: Przykład kontynuacji obliczeń

```
1 var intTask = Task.Run(() =>
2     {
3         Thread.Sleep(1000);
4         return 12;
5     });
6 var continueTask = intTask.ContinueWith((x) => { Thread.Sleep(1000); return x.Result / 2; });
7 Console.WriteLine($"We have got result = {intTask.Result}");
8 Console.WriteLine($"We have got result2 = {continueTask.Result}");
```

intTask jest zdaniem, które zwraca jakąś liczbę (12), wynik ten jest przekazywany do kolejnego zadania *continueTask* jako rezultat.

ContinueWith z parametrem krótszy zapis

Poprzedni przykład można zapisać jednym poleceniu używając notacji z kropkami.

Listing 31: Przykład kontynuacji obliczeń v2

```
1 int result = Task.Run(() =>
2     {
3         Thread.Sleep(1000);
4         return 12;
5     }.ContinueWith((x) => { Thread.Sleep(1000); return x.Result / 2; })
6     .Result;
7
```

Kontynuacja z wyjątkiem

Listing 32: Kontynuacja z wyjątkiem

```
1  Task<int> zad1ex = Task.Factory.StartNew<int>(() => { throw new Exception("First Task Thrown ↩
       an exception"); });
2  //Task<int> zad2ex = zad1ex.ContinueWith<int>(ant => Console.WriteLine("Exception: " +
       ant.Exception.Message); return 0; );
3  //A safe pattern is to rethrow antecedent exceptions
4  Task<int> zad2ex = zad1ex.ContinueWith<int>(ant => { if (ant.Exception != null) throw ant.↩
       Exception; return 0; });
5  try
6  {
7      Console.WriteLine($"Result zad2ex: {zad2ex.Result}");
8  }
9  catch (AggregateException ae)
10 {
11     Console.WriteLine("AggregateException's message: " + ae.Message);
12     foreach (var ex in ae.InnerExceptions)
13     {
14         Console.WriteLine("I cought exception: " + ex.Message);
15         Console.WriteLine("Inner exception:" + ex.InnerException.Message);
16     }
17 }
```

Różne ścieżki dla wyjątków

Listing 33: Przykład różnych ścieżek

```
1  Task<int> zad1r = Task.Factory.StartNew<int>(() => {
2        //uncomment to simulate error
3        //throw new Exception("The first task Thrown an exception");
4        return 1;
5    });
6  Task<int> taskBlad = zad1r.ContinueWith<int>(ant => { Console.WriteLine("Exception: " + ant.↩
         Exception.Message); return 1; },TaskContinuationOptions.OnlyOnFaulted);
7  Task<int> taskNieBlad = zad1r.ContinueWith<int>(ant => { Console.WriteLine("There are no ↩
         errors") ; return ant.Result+2; }, TaskContinuationOptions.NotOnFaulted);
8    try
9    {
10       if (taskBlad!=null) await taskBlad;
11       if (taskNieBlad != null) await taskNieBlad;
12   }
```

Różne ścieżki dla wyjątków cd

Jeżeli warunek uruchomienia zadania nie zostanie spełniony to takie zadanie jest Anulowane

Listing 34: Przykład różnych ścieżek

```
1   catch (AggregateException ae)
2     {
3         Console.WriteLine("AggregateException message: " + ae.Message);
4         foreach (var ex in ae.InnerExceptions)
5         {
6           Console.WriteLine("I cought exception: " + ex.Message);
7           Console.WriteLine("Inner exception:" + ex.InnerException.Message);
8         }
9     }
10    catch (TaskCanceledException)
11    {
12        Console.WriteLine("Alternative task was canceled");
13    }
```

Kontynuacje i zadania potomne

Przykład zbierania wyjątków od potomków

Listing 35: Potomkowie rzucają wyjątkami

```
1  TaskCreationOptions atp = TaskCreationOptions.AttachedToParent;
2  Task TaskRodzic = Task.Factory.StartNew(() =>
3  {
4      Task.Factory.StartNew(() => { throw new Exception("error1"); }, atp);
5      Task.Factory.StartNew(() => { throw new Exception("error2"); }, atp);
6      Task.Factory.StartNew(() => { throw new Exception("error3"); }, atp);
7  })
8  .ContinueWith(p => {
9      Console.WriteLine("Jako rodzic zlapalem takie cos: " + p.Exception);
10                     throw p.Exception;
11 },
12                     TaskContinuationOptions.OnlyOnFaulted);
13 try
14 {
15     await TaskRodzic;
16 }
```

Kontynuacje i zadania potomne

Listing 36: Zbieramy wyjątki

```
1  catch (AggregateException ae)
2            {
3                Console.WriteLine("AggregateException message: " + ae.Message);
4                foreach (var ex in ae.InnerExceptions)
5                {
6                  Console.WriteLine("I cought exception: " + ex.Message);
7                  Console.WriteLine("Inner exception:" + ex.InnerException.Message);
8                }
9            }
10           catch (Exception ex)
11           {
12               Console.WriteLine("Exception message: ", ex.Message);
13           }
```

Kontynuacja warunkowa

Domyślnie zadanie kontynuowane jest bez warunku, niezależnie czy poprzednik zakończy się normalnie, rzuci wyjątek czy zostanie anulowany, jego następnik zostanie uruchomiony. Można to zmienić za pomocą trzech następujących flag zdefiniowanych w System.Threading.TasksTaskContinuationOptions:

- NotOnRanToCompletion = 0x10000,
- NotOnFaulted = 0x20000,
- NotOnCanceled = 0x40000,

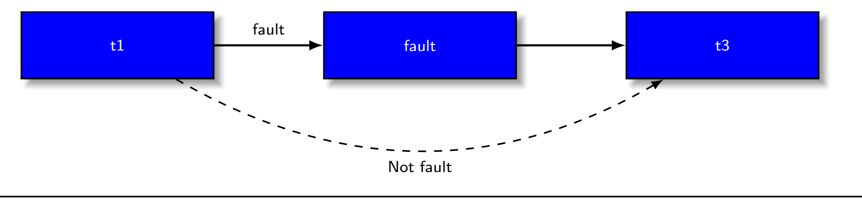
Kontynuacja warunkowa

Flagi można łączyć, a dla wygody zdefioniowano jeszcze dodatkowo:

- OnlyOnRanToCompletion = NotOnFaulted | NotOnCanceled,
- OnlyOnFaulted = NotOnRanToCompletion | NotOnCanceled,
- OnlyOnCanceled = NotOnRanToCompletion | NotOnFaulted

Kontynuacja warunkowa

Zadanie *fault* jest uruchamiane tylko w razie błędu w zadaniu *t1*.
Zadanie *t3* uruchamiane jest bezwarunkowo po *fault* lub *t1*

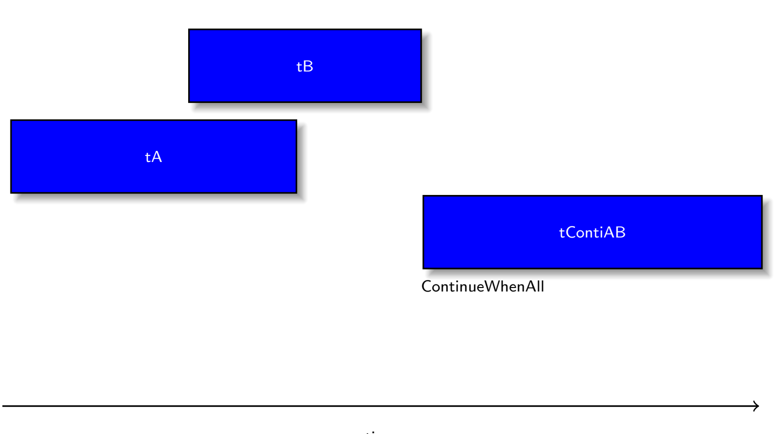


Kontynuacja warunkowa

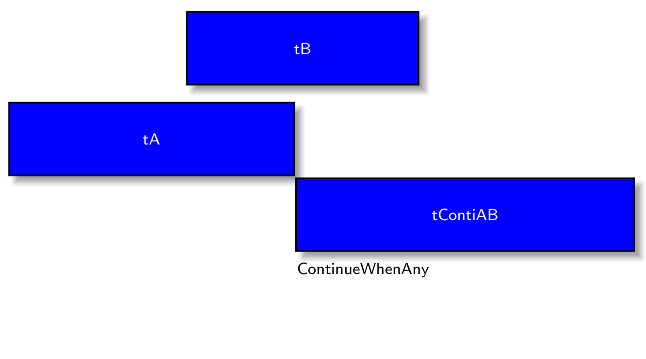
Listing 37: Przykład warunkowej kontynuacji

```
1  Task t1 = Task.Factory.StartNew(() =>
2    {
3        Console.WriteLine("t1 works but it will stop in a while");
4      // throw new Exception("Eny error in t1");
5    }
6  );
7
8  Task fault = t1.ContinueWith(ant => Console.WriteLine("Task run in case of fault of t1"),
9               TaskContinuationOptions.OnlyOnFaulted);
10
11 Task t3 = fault.ContinueWith(ant => Console.WriteLine("t3 run after task fault"));
12 await t3;
```

Kontynuacje z wieloma poprzednikami



Kontynuacje z wieloma poprzednikami



Kontynuacja warunkowa WhenAll

Listing 38: Przykład warunkowej kontynuacji WhenAll

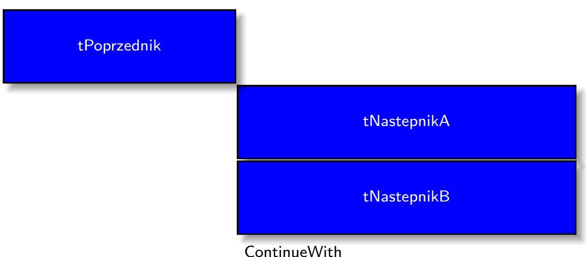
```
1 Task tA = Task.Factory.StartNew(() => Console.Write("A"));
2             Task tB = Task.Factory.StartNew(() => Console.Write("B"));
3             Task tContiAB = Task.Factory.ContinueWhenAll(new Task[] { tA, tB }, tasks =>
4             { Console.WriteLine("\nKontynuacja po A i B"); }
5             );
6             await tContiAB;
```

Kontynuacja z wieloma poprzednikami i wartościami

Listing 39: Przykład kontynuacji WhenAll z wartościami

```
1 Task<int> tRetA = Task.Factory.StartNew(() => { Console.WriteLine("A returns 11"); return 11;↩
       });
2 Task<int> tRetB = Task.Factory.StartNew(() => { Console.WriteLine("B returns 22"); return 22;↩
       });
3 Task<int> tRetAB = Task.Factory.ContinueWhenAll(new Task<int>[] { tRetA, tRetB }, tasks =>
4 { Console.WriteLine($"Continuation after task A returning {tasks[0].Result} i and task B ↩
      returning {tasks[1].Result}");
5     return tasks.Sum(t => t.Result); }
6 );
7 Console.WriteLine("Sum from task A and B: " + tRetAB.Result);
```

Kontynuacje z wieloma następnikami



time

Kontynuacja z wieloma następnikami

Listing 40: Przykład kontynuacji przez wielu następników

```
1  Task tPoprzednik = Task.Factory.StartNew(() => Console.Write("Antecedent"));
2  Task tNastepnikA = tPoprzednik.ContinueWith(ant =>
3  {
4      Console.WriteLine("\nContinuation A after Antecedent");
5  });
6  Task tNastepnikB = tPoprzednik.ContinueWith(ant =>
7  {
8      Console.WriteLine("\nContinuation B after Antecedent");
9  });
10 await tNastepnikA;
11 await tNastepnikB;
```

Planista zadań

- Planista przydziela zadania do wątków
- wszystkie zadania są skojarzone z planistą, który jest reprezentowany przez abstrakcyjną klasę *TaskScheduler*
- Framework dostarcza dwie implementacje
  - Domyślny scheduler, który działa z pulą wątków CLR
  - Synchronization context scheduler, zaprojektowany przede wszystkim, by pomóc w modelu wątków WPF i WinForms, gdzie elementy interfejsu użytkownika dostępne są tylko z wątka, który go stworzył.

Planista zadań

- Przykładowo mamy funkcję długo zwracające dane, np. wywołanie jakiegoś webserwisu bądź metody obliczeniowej.
- Po otrzymaniu danych chcemy wyświetlić te dane w jednej z kontrolek.
- Zrobi to zadanie będące kontynuacją zadania pobierającego dane.
- Zadanie kontynuacji będzie miało wskazany kontekst planisty uzyskany z okienka na, którym dana kontrolka jest osadzona
- Dzięki temu będzie ona mogła być bezpiecznie aktualizowana.

Planista oraz UI

Listing 41: Przykład Schedulera i UI

```
TaskScheduler _uiScheduler;
public MainWindow()
{
    InitializeComponent();
    progress = new Progress<int>(percent =>
    {
        progressBar.Value = percent;

    });
    _uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
}
private string JakasMetodaCoDlugoZwracaDane()
  {
      Thread.Sleep(5000); return "We have it...";
  }
```

Planista oraz UI

Listing 42: Przykład Schedulera i UI

```
TaskScheduler _uiScheduler;
public MainWindow()
{
    InitializeComponent();
    _uiScheduler = TaskScheduler.FromCurrentSynchronizationContext();
}
private string JakasMetodaCoDlugoZwracaDane()
  {
      Thread.Sleep(5000); return "We have it...";
  }

private void GoBT_Click(object sender, RoutedEventArgs e)
  {
    var client = new HttpClient();
    Task.Factory.StartNew<string>(JakasMetodaCoDlugoZwracaDane)
        .ContinueWith(ant =>
        {
            contentViewer.AppendText(ant.Result);
        }, _uiScheduler);
  }
```

Pytania

?

KONIEC

Dziękuję Państwu za uwagę.