C# OO Programmeren

LES 11

JOHAN DONNÉ

Tip: valkuilen bij gelijkheid testen

```
if ( result == 10.0 )
```

```
if (Math.Abs(result – 10.0) < 0.0001 )
```

```
if (((DateTime.Now - previousTime).TotalMilliseconds) == Interval)
```

```
if (((DateTime.Now - previousTime).TotalMilliseconds) >= Interval)
```

Tip: passing parameters by value:

```
public void Run()
{
  int count = 0;
  var myList = new List<int>();
  Adjust(count, myList);
  Console.WriteLine($"count: {count}, myList lengte: {myList.Count}");
}

private void Adjust(int value, List<int> list)
{
  value = 5;
  list.Add(1);
  list = new List<int>();
}
```

count: 0, myList lengte : 1

Bij oproep worden ‘count’ en ‘myList’ gekopieerd naar ‘value’ en ‘list’

Tip: passing parameters by reference:

```
public void Run()
{
  int count = 0;
  var myList = new List<int>();
  Adjust(ref count, ref myList);
  Console.WriteLine($"count: {count}, myList lengte: {myList.Count}");
}

private void Adjust(ref int value, ref List<int> list)
{
  value = 5;
  list.Add(1);
  list = new List<int>();
}
```

count: 5, myList lengte : 0

Bij oproep worden adressen van ‘count’ en ‘myList’ doorgegeven en gebruikt voor ‘value’ en ‘list’ (dat zijn dus symbolische namen voor zelfde variabelen)

Overzicht

Asynchronous programming

Async..Await

C# Long running operations

Probleem:

Userinterface responsief houden tijdens langdurige bewerkingen waarvan resultaat nodig is.
Niet zomaar 'job.Result' gebruiken (blocking)!

Mogelijke oplossing:

- Zorgen dat de UI-thread gewoon kan doorwerken terwijl de langdurige bewerking uitgevoerd wordt.

==> rekenwerk wordt 'asynchroon' uitgevoerd.

C# Synchronous long running operation

```
private void OnStartButtonClick(object sender, EventArgs e)
{
    startButton.Enabled = false;
    resultTextBox.Text = string.Empty;
    int result = worker.CalculateTheAnswer();
    resultTextBox.Text = $"{result}";
    startButton.Enabled = true;
}
```

```
public class Worker
{
    public int CalculateTheAnswer()
    {
        // Simulate long-running operation
        Thread.Sleep(3000);
        return 42;
    }
}
```

C# Asynchronous long running operation

Alternatief 1:

- Task
- Op het einde Event oproepen
- In UI: eventhandler handelt vervolg af.

C# Asynchroon via Task en Event

```
public class Worker
{
    public event Action<int> ResultCalculated;

    public void CalculateTheAnswer()
    {
        Task.Run(() => CalculateIt());
    }

    private void CalculateIt()
    {
        Thread.Sleep(3000);
        ResultCalculated?.Invoke(42);
    }
}
```

C# Asynchroon via Task en Event

```
    private SynchronizationContext context;
    private Worker worker;

    public MainForm()
    {
        InitializeComponent();
        context = SynchronizationContext.Current;
        worker = new Worker();
        worker.ResultCalculated += OnWorkerResultCalculated;
    }

    private void OnWorkerResultCalculated(int result)
    {
        context.Post((r) =>
        {
            resultTextBox.Text = $"{r}";
            startButton.Enabled = true;
        }, result);
    }

    private void StartButtonClick(object sender, EventArgs e)
    {
       resultTextBox.Text = string.Empty;
       startButton.Enabled = false;
       worker.CalculateTheAnswer();
     }
}
```

C# Asynchronous long running operation

Alternatief 2: ‘ContinueWith’ methode van Task

```
Task.ContinueWith( Action<Task<TResult>>,
                                    TaskScheduler sheduler   );
```

==> Na de 1e Task wordt een tweede gestart waarin Action uitgevoerd wordt (in gewenste thread).

==> Ketting van opeenvolgende tasks mogelijk.

C# Asynchroon via Task, ContinueWith en Event

```
public class Worker
{
   public event Action<int> ResultCalculated;

   public void CalculateTheAnswer()
   {
      var job = Task<int>.Run(() => { return CalculateIt(); });
      job.ContinueWith((j) => { ResultCalculated?.Invoke(j.Result); },
                        TaskScheduler.FromCurrentSynchronizationContext());
   }

   private int CalculateIt()
   {
       Thread.Sleep(3000);
       return 42;
   }
}
```

Bij aflopen van 1e task ==> 2e task in thread van UI: event triggeren

C# Asynchronous long running operation

Alternatief 3: ‘async’ & ‘await’ keywords:

```
public async Task<int> DoSomethingAsync()
{
    …
    return 59;
}

int result = await worker.DoSomethingAsync();
```

C# Async Methodes

```
public async Task<int> DoSomethingAsync(){...}

public async Task DoSomethingAsync(){...}

public async void DoSomethingAsync(){...}
```

- ‘async’ is nodig om in een methode ‘await’ te kunnen gebruiken.
- Als returntype ‘Task’ of ‘Task<int> is kan ‘await gebruikt worden bij de oproep.
- Conventie: naam eindigt op ‘Async’.
- ‘async void’ enkel bedoeld voor asynchrone event-handlers.

C# Await

```
int result = await DoSomethingAsync();
```

- Start ‘DoSomethingAsync’ en wacht op resultaat.

- Wachten gebeurt **non-blocking**.
  Geeft de thread vanwaar de oproep gebeurt dus vrij.

- Als task afgelopen is, wordt in de **originele** thread opnieuw verder gewerkt met het resultaat van de task.


```
int result = await DoSomethingAsync();
```

C# Await

‘Await’ moet niet onmiddellijk gebeuren:

```
var job = worker.DoSomethingAsync();
resultTextBox.Text = "waiting";
resultTextBox.Text = $"{await job}";
```

C# async … await voorbeeld

```
public class Worker
{

    public async Task<int> CalculateTheAnswerAsync()
    {
       return await Task<int>.Run(() => { return CalculateIt(); });
    }

    private int CalculateIt()
    {
        Thread.Sleep(2000);
        return 42;
    }

    public async Task<int> DoSomethingAsync()
    {
        await Task.Delay(2000);
        return 2 * 42;
    }
}
```

C# async … await

```
private Worker worker;

public MainForm()
{
   InitializeComponent();
   worker = new Worker();
}

private async void StartButtonClick(object sender, EventArgs e)
{
    resultTextBox.Text = string.Empty;
    startButton.Enabled = false;
    int result1 = await worker.CalculateTheAnswerAsync();
    int result2 = await worker.DoSomethingAsync();
    resultTextBox.Text = $"{result1}, {result2}";
    startButton.Enabled = true;
}
```

C# async … await

Opgelet: ‘async’ methode en ‘.Wait’ of ‘.Result’ kan problemen geven!!!

```
int x = worker.CalculateTheAnswerAsync().Result;
```

‘Result’ blokkeert de oproepende thread.

‘await’ in ‘CalculateTheAnswerAsync’ probeert verder te werken in de oproepende thread (maar die is geblokkeerd).

```
return await Task<int>.Run(() => { return CalculateIt(); }).ConfigureAwait(false);
```

Laat toe dat teruggekeerd wordt in andere thread dan originele.

C# async … await

- ‘async’ is ‘besmettelijk’:

‘await’ kan enkel in een ‘async’ methode. De oproep daarvan gebeurt meestal ook via ‘await’. ==> vaak een ketting van async oproepen.

- Je kan eventhandlers voor bv. Winforms ook async maken:

```
private async void OnStartButtonClick(object sender, EventArgs e)
{
```

- Ook lambda expressions kunnen async zijn:

```
async () => { await DoSomethingAsync()};
```

C# async … await

- ‘async’ methode wordt niet altijd asynchroon uitgevoerd:

```
int x = await worker.Power(2, 1);
int y = await worker.Power(2, 20);

  ...

public async Task<int> Power(int n, int power)
{
  if (power < 0) throw new ArgumentException("Exponent must be positive.");
  if (power == 0) return 1;
  if (power == 1) return n;
  return await Task.Run(() =>
  {
     int result = 1;
     for (int i = 0; i < power; i++) result *= n;
     return result;
  });
}
```

C# async … await - exceptions

- ‘gewone tasks’ waarin een exceptie optreedt :
  1. Task stopt
  2. (enkel) bij ‘.Wait’ of ‘.Result’ wordt exceptie opgegeooid binnen een ‘AggregateException’.

  ==> zonder ‘Wait’ of ‘Result’ geen info over afbreken vanTask

- bij ‘await’:
  1. Task of ‘async’ oproep stopt (net zoals gewone methode).
  2. De originele exception wordt bewaard en opgegooid bij de await (net zoals het bij klassieke ‘synchrone’ code) zou gebeuren.

==> ‘await’ is veel eenvoudiger / eleganter voor exception handling.

C# async … await - Besluit

- ‘async..await’ laat toe om te schrijven op een elegante manier asynchrone code te schrijven.
- Ziet er heel gelijkaardig uit aan ‘gewone’ synchrone code.
- Achter de schermen behoorlijk complex
- Meer en meer bibliotheken hebben async versies van methodes voor langdurige bewerkingen:

```
// call to Web API
var client = new HttpClient();
HttpResponseMessage response = await client.GetAsync("http://www.contoso.com/");

// entity Framework database query
var result = await dbContext.ScoreLevels.ToListAsync();

// File.IO  - .Net core 3.0
string[] lines = await File.ReadAllLinesAsync("config.txt");
```

Bedankt voor de aandacht (en aanwezigheid) !

Suggesties of opmerkingen:
Johan.donne@odisee.be