howest
hogeschool

Errors & Exceptions

Programming Basics

Graduaat in het Programmeren

INHOUD

1 RUNTIME ERRORS

Je hebt nu al geleerd hoe je met statements, methoden en variabelen omgaat in C#.

In dit hoofdstuk leer je hoe je kan omgaan met fouten die kunnen optreden tijdens de uitvoering van een toepassing. Dit soort fouten zijn zogenaamde **runtime errors**, of worden ook wel **Exceptions** genoemd.

We hebben het hier dus niet over **syntactische** fouten die je maakt tijdens het typen van je programmacode, of fouten die je maakt tegen de "spelregels" van C#. Dergelijke fouten worden reeds tijdens het compileren van je code gedetecteerd en verhinderen de uitvoering van je programma (**compile-time** errors).

1.1 RUNTIME ERRORS

Dit hoofdstuk gaat in op fouten die tijdens het uitvoeren van een programma kunnen optreden (runtime-errors). Het is belangrijk dat je in je programma rekening houdt met mogelijke foute handelingen van een gebruiker, mogelijke hardware-matige problemen (netwerkverbindingen vallen uit, bestand is onleesbaar, ...), ...

1.1.1 VOORBEELDSCENARIO

Je maakt bijvoorbeeld een toepassing die twee getallen die door de gebruiker worden ingegeven optelt en de som op het scherm toont.

Wat gebeurt er als de gebruiker letters invult of helemaal niks?

De bedoeling van dit hoofdstuk is dat je dergelijke mogelijke probleemsituaties kan inschatten en softwarematig een oplossing biedt, zonder dat je toepassing "crasht".

Hierbij wordt niet bedoeld dat je elk probleem moet kunnen oplossen: letters kunnen we nu eenmaal niet optellen. Je kan de gebruiker wel op de hoogte stellen van het probleem en informeren hoe dit probleem kan worden opgelost door bijvoorbeeld een bericht te tonen: "Gelieve numerieke waarden in te geven.".

Je kan dus in je programmacode een pad voorzien voor het goede verloop van de toepassing, maar je moet er zeker rekening mee houden dat dit verloop weleens verstoord kan worden.

1.1.2 LOGISCHE FOUTEN

Het is belangrijk in te zien dat we dergelijke onvoorziene omstandigheden in een programma uitzonderingen of exceptions noemen, deze kunnen we gestructureerd in programmacode afhandelen. **Logische fouten** zijn daarentegen verkeerde denkwijzen of verkeerde programma algoritmes: als je de gebruiker belooft twee getallen op te tellen, maar je berekent het verschil, dan zal je programma wellicht niet vastlopen, maar de gebruiker zal ook niet echt gelukkig zijn.

Logische fouten zijn dus geen run-time fouten en om ze te detecteren zijn uitvoerige **tests** van je programma nodig.

2 EXCEPTIES AFHANDELEN

Code repository

De volledige broncode van dit onderdeel is te vinden op

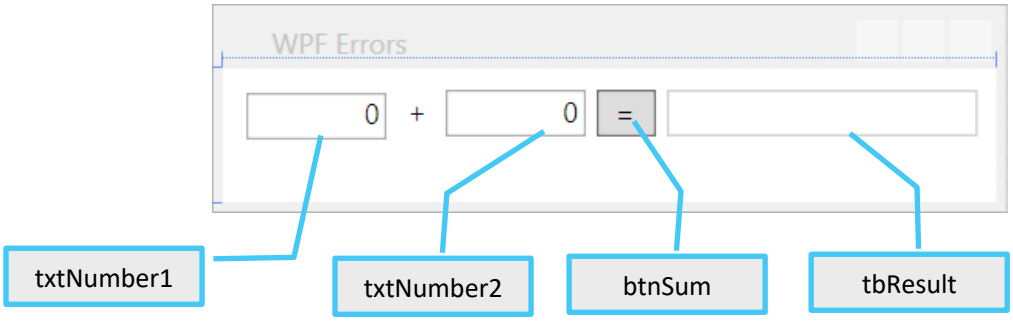
git clone https://github.com/howest-gp-prb/cu-errors-and-exceptions

Startsituatie

https://github.com/howest-gp-prb/cu-errors-and-exceptions-start

Maak een eenvoudige applicatie om het principe van foutafhandeling te leren kennen. Bij het klikken op een knop zet de applicatie twee ingevoerde `string` waarden om naar gehele getallen en toont de som in een `TextBlock` control.

- Maak een WPF `Window` met twee `TextBoxes`, een `Label`, een `Button` en een `TextBlock` control:

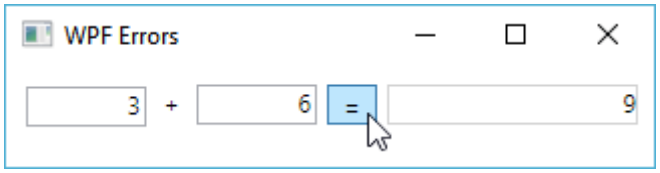


- Genereer een Event Handler methode voor de knop via het Properties venster.
- Wijzig de code van de codebehind als volgt:

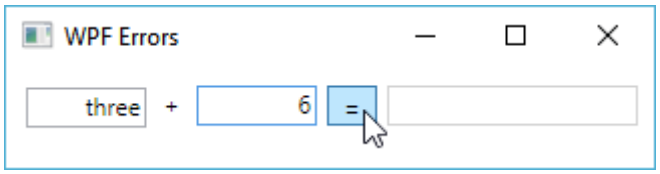
```
private void Sum_Click(object sender, RoutedEventArgs e)
{
    int numberLeft = int.Parse(txtNumber1.Text);  //parse string to int
    int numberRight = int.Parse(txtNumber2.Text); //parse string to int
    int sum = CalculateSum(numberLeft, numberRight);
    tbResult.Text = sum.ToString(); //convert int back to string and set in textblock
}

/// <summary>
/// Calculates the sum of two numbers
/// </summary>
private int CalculateSum(int number1, int number2) {
    return number1 + number2;
}
```

- Compileer en voer de applicatie uit.



Voer een waarde in die niet kan worden omgezet naar een integer. Er doet zich nu een run-time error (van het type `FormatException`) voor en de applicatie **crasht**.




```
1 reference
private void btnSum_Click(object sender, RoutedEventArgs e)
{
    int numberLeft = int.Parse(txtNumber1.Text);  //parse string to int
    int numberRight = int.Parse(txtNumber2.Text); //parse string to int
    int sum = CalculateSum(numberLeft, numberRight);
    tbResult.Text = sum.ToString(); //convert int back to string and set
}

/// <summary>
/// Calculates the sum of two numbers
/// </summary>
1 reference
private int CalculateSum(int number1, int number2)
{
    return number1 + number2;
}
```

Exception Unhandled

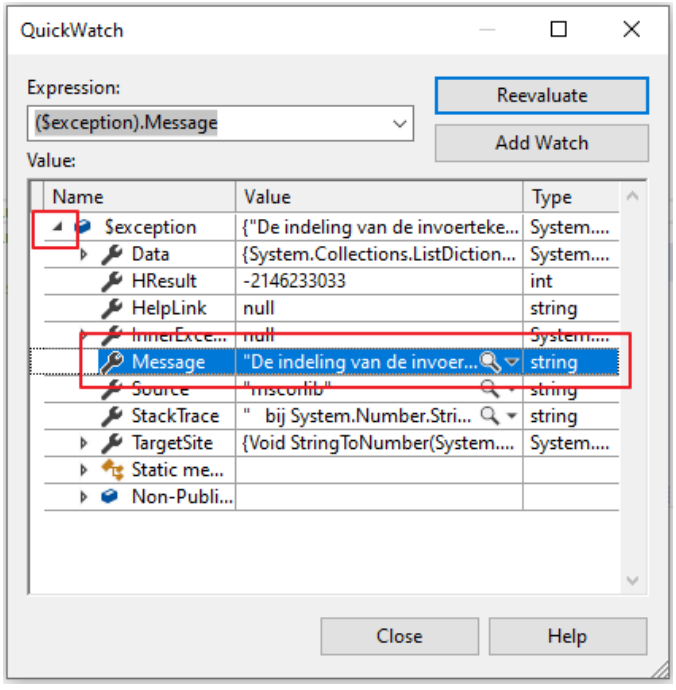
System.FormatException: 'De indeling van de invoertekenreeks is onjuist.'

View Details | Copy Details | Start Live Share session...

Exception Settings

Als programmeur zien wij deze melding verschijnen in Visual Studio, wanneer we de toepassing starten. Visual Studio kijkt achter onze schouder mee bij het werken met de toepassing en grijpt in wanneer het verkeerd loopt.

Om meer details te zien van de Exception klik je in het venster op de link **View Details**:



Om een som te kunnen oplossen moet er gewerkt worden met variabelen van een numeriek type (`int`, `decimal`, `float`, enz.). Daarom moeten de `string` waarden omgezet worden naar een numerieke waarde. In dit geval wordt een `int` verwacht.

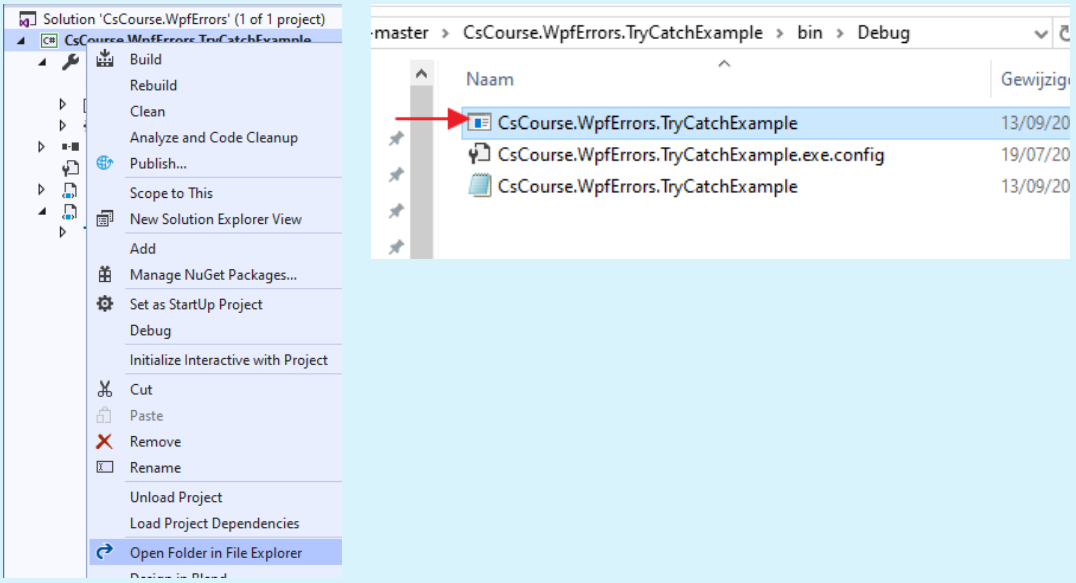
De fout doet zich voor omdat de `string` variabele die de gebruiker heeft ingetikt, “three”, **niet**

kan worden omgezet naar een variabele van het type `int`. Dit levert een `FormatException` met de melding “Input string was not in a correct format.” of op een Nederlands Windowssysteem: “De indeling van de invoertekenreeks is onjuist.”

Wanneer je dit programma verspreidt voor gebruik, dan voeren de gebruikers een .exe-bestand uit, los van Visual Studio. Het uitvoerbare bestand vind je in de /bin/Debug map van je huidige toepassing.

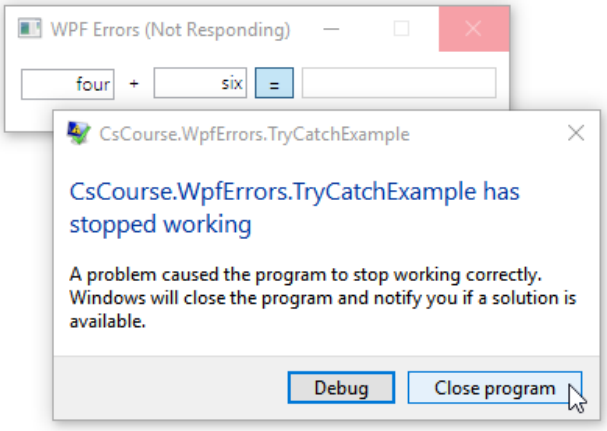
Ga naar de /bin/Debug map van je applicatie en voer de executable uit.

Tip :
klik met de rechtermuisknop op je project en kies in het snelmenu voor “Open Folder in File explorer”



- Geef een ongeldige waarde in voor de numerieke waarde(n) en klik op de knop.

De applicatie crasht en wordt afgesloten na een summiere foutmelding (of zelfs helemaal geen melding en sluit gewoon af) :



Het is voor de gebruiker onduidelijk wat de precieze oorzaak van de fout was. Om meer duidelijkheid te scheppen en te verhinderen dat de applicatie crasht moeten we deze mogelijke fout afhandelen.

2.1 TRY CATCH

C# maakt het eenvoudig om foutafhandelingscode te scheiden van code die de hoofdflow van je programma vormt. Om programma's te maken die zich bewust zijn van fouten doe je twee zaken:

- Schrijf je code binnen een try-blok. Wanneer de code wordt uitgevoerd, wordt gepoogd elk statement na elkaar uit te voeren. Wanneer een fout optreedt, wordt het try-blok verlaten en wordt de fout afgehandeld binnen een catch-blok.
- Schrijf na een try-blok één of meer catch-blokken om mogelijke fouten af te handelen.

Pas dit toe in het voorbeeldprogramma.

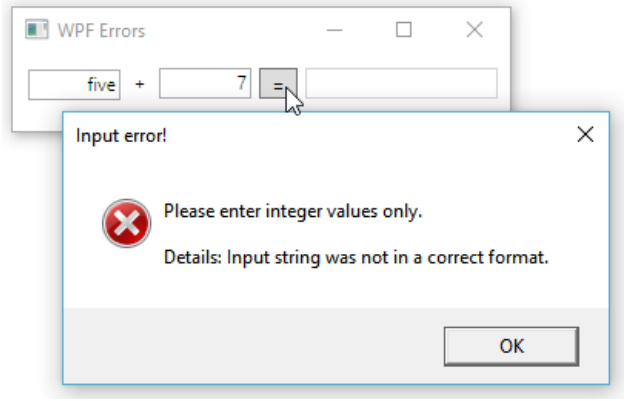
Zet de statements die de fout kunnen veroorzaken in een `try` -blok en de foutafhandelingscode in het bijhorende `catch`-blok daaronder.

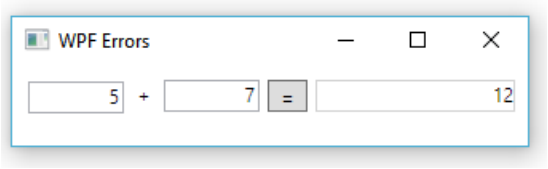
```
try
{
    int numberLeft = int.Parse(txtNumber1.Text);
    int numberRight = int.Parse(txtNumber2.Text); //parse string to int
    int sum = CalculateSum(numberLeft, numberRight);
    tbResult.Text = sum.ToString(); //convert int back to string and set in textblock
}
catch(FormatException fEx)
{
    MessageBox.Show("Please enter integer values only.\n\nDetails: " + fEx.Message,
                    "Input error!", MessageBoxButton.OK, MessageBoxImage.Error);
}
```

Indien er zich nu een `FormatException` voordoet in de instructies die in het `try`-blok staan, dan wordt die fout opgevangen in het `catch`-blok. Er wordt dan een `MessageBox` getoond die de gebruiker inlicht over de precieze oorzaak van de fout.

Omdat je de fout **opvangt** in een `catch`-blok zal de applicatie **niet crashen**. De gebruiker krijgt de gelegenheid om een nieuwe invoer te proberen.

- Voer de applicatie uit en test de foutafhandeling door foutieve invoer in te geven.





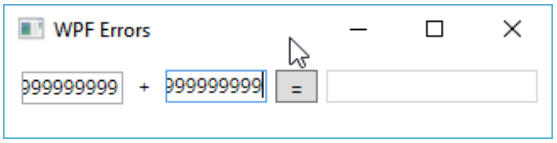
2.2 SOORTEN EXCEPTIONS

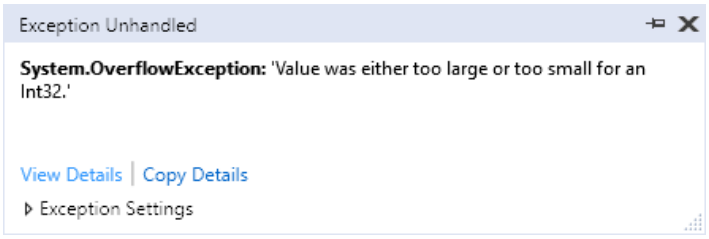
2.2.1 MEERDERE EXCEPTIONS AFHANDELEN

Er bestaan heel wat soorten exceptions. De exception die je zonet hebt afgehandeld was van het type `FormatException` die meestal voorkomt bij het *parsen* van een `string` naar een variabele van een ander type.

Onze applicatie is nog niet helemaal vrij van potentiële crashes. Een variabele van het type int kan nooit groter zijn dan 2.147.483.647 en nooit kleiner dan -2.147.483.648. Indien deze waarden overschreden worden dan krijg je een `OverflowException`.

- Voer de applicatie uit en geef getallen in waarvan de som groter is dan 2.147.483.647
- De applicatie crasht opnieuw, ditmaal met een `OverflowException`.





- Voeg nog een `catch`-blok toe aan de je `try/catch` structuur, dat de `OverflowException` moet opvangen.

```
try
{
    int numberLeft = int.Parse(txtNumber1.Text);
    int numberRight = int.Parse(txtNumber2.Text); //parse string to int
    int sum = CalculateSum(numberLeft, numberRight);
    tbResult.Text = sum.ToString(); //convert int back to string and set in textblock
}
catch(FormatException fEx)
{
    MessageBox.Show("Please enter integer values only.\n\nDetails: " + fEx.Message,
                    "Input error!", MessageBoxButton.OK, MessageBoxImage.Error);
}
catch (OverflowException oEx)
{
    MessageBox.Show("The numbers you entered are too large or to small.\n\nDetails: "
+ oEx.Message,
                    "Input error!", MessageBoxButton.OK, MessageBoxImage.Error);
}
```

- Voer de applicatie nogmaals uit en test opnieuw. De applicatie is nu een stuk robuuster geworden en zal in deze twee specifieke gevallen niet meer crashen.

2.2.2 ALLE EXCEPTIONS AFHANDELEN

Het is onbegonnen werk om een `catch`-blok te schrijven voor elk type fout dat zich kan voordoen. Je gebruikt de specifieke `catch`-blokken enkel om een verschillende foutafhandeling te leveren. Bijvoorbeeld als de weer te geven tekst verschilt of als je enkel een `OverflowException` naar een apart logbestand wenst te registreren.

Om je applicatie te beschermen tegen elke mogelijke fout kan je de joker `Exception` gebruiken in een `catch`-blok.

- Vervang het `catch`-blok voor de `OverflowException` met een algemener catch-blok dat **elke soort** `Exception` kan afhandelen.

```
try
{
    int numberLeft = int.Parse(txtNumber1.Text);
    int numberRight = int.Parse(txtNumber2.Text); //parse string to int
    int sum = CalculateSum(numberLeft, numberRight);
    tbResult.Text = sum.ToString(); //convert int back to string and set in textblock
}
catch(FormatException fEx)
{
    MessageBox.Show("Please enter integer values only.\n\nDetails: " + fEx.Message,
                    "Input error!", MessageBoxButton.OK, MessageBoxImage.Error);
}
catch (Exception ex) //catches all unhandled exceptions (FormatException are already
handled above)
{
    //generic error message is shown
    MessageBox.Show("An error has occurred.\n\nDetails: " + ex.Message,
                    "Input error!", MessageBoxButton.OK, MessageBoxImage.Error);
}
```

Wens je geen gebruik te maken van het Exception object, dat onder meer de foutmelding bevat, dan kan dit nog korter:

```
try
{
    //error prone code goes here
}
catch
{
    //catches all unhandled exceptions here
    MessageBox.Show("An unknown error has occurred!");
}
```

Als je meerdere `catch`-blokken gebruikt, dan moet het algemene `catch`-blok steeds het **laatste** zijn. Op die manier kan je de specifiekere exception soorten eerst afhandelen.

2.3 THROW

Je kan zelf ook Exceptions opgooien, om te voorkomen dat de applicatie een scenario uitvoert dat niet toegestaan is. Zo kan je jezelf of je collega-programmeurs verplichten om een foutafhandeling uit te werken voor dat scenario.

Eerst moet je een nieuwe Exception instantie aanmaken met het new keyword. Vervolgens gooi je de exception instantie op met het `throw` keyword. Bij het maken van een instantie kan je zelf kiezen welk type fout je wenst op te gooien, of je houdt het algemeen met de klasse `Exception`.

Stel dat je wenst te verhinderen dat de `CalculateSum()` methode negatieve getallen optelt, dan kan je hier een controle op doen en, indien een negatief getal wordt gedetecteerd, een fout van het type `ArgumentOutOfRangeException` opgooien.

- Wijzig de code van de `CalculateSum()` methode als volgt:

```
private int CalculateSum(int number1, int number2) {
    //check if either number is negative
    if (number1 < 0 || number2 < 0)
    {
        throw new ArgumentOutOfRangeException("Terms should not be negative");
    }
    //this instruction will only be executed reached if no exception occurred
    return number1 + number2;
}
```

- Voer de applicatie uit en test of je nu nog negatieve getallen kan invoeren. De `Exception` wordt afgehandeld door het algemene `catch`-blok in de Event Handler van de knop.

2.4 FINALLY

Wanneer een fout optreedt in een `try`-blok wordt de flow van een programma veranderd. De instructies na de plaats van de fout worden niet meer uitgevoerd. In plaats daarvan wordt het overeenkomstig `catch` blok uitgevoerd.

Soms is het nodig om code te schrijven die altijd moet uitgevoerd worden, ongeacht het feit of er een fout is opgetreden. Dit kan je bereiken door **na het laatste** `catch` blok (of na het `try` blok, indien je geen foutafhandeling wenst) een `finally` blok te voorzien.

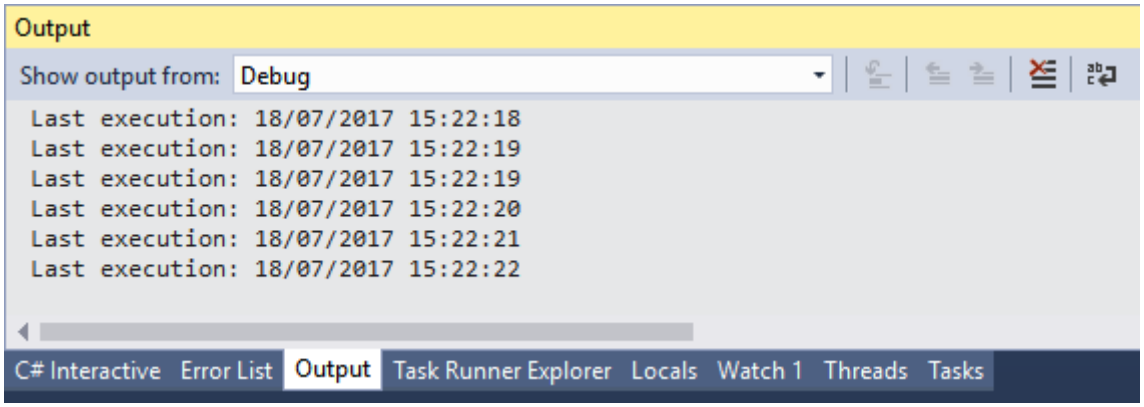
```
try
{
    //code met die mogelijke fouten kan veroorzaken komt hier te staan
}
catch(Exception ex)
{
    //vangt elke exception op
}
finally
{
    //code die steeds moet worden uitgevoerd, of er zich nu een fout voordoet of niet.
}
```

- Voeg een `finally` blok toe onder het laatste `catch` blok dat steeds de tijd weergeeft in het Output venster van Visual Studio. Hiervoor heb je de namespace `System.Diagnostics` nodig als `using` statement.

```
try
{
    int numberLeft = int.Parse(txtNumber1.Text);
    int numberRight = int.Parse(txtNumber2.Text); //parse string to int
    int sum = CalculateSum(numberLeft, numberRight);
    tbResult.Text = sum.ToString(); //convert int back to string and set in textblock
}
catch (FormatException fEx)
{
    MessageBox.Show("Please enter integer values only.\n\nDetails: " + fEx.Message,
```

```
                    "Input error!", MessageBoxButton.OK, MessageBoxImage.Error);
}
catch (Exception ex) //catches all unhandled exceptions (FormatException are already
handled above)
{
    //generic error message is shown
    MessageBox.Show("An error has occurred.\n\nDetails: " + ex.Message,
                    "Input error!", MessageBoxButton.OK, MessageBoxImage.Error);
}
finally
{
    //always write to output window (in Visual Studio)
    Debug.WriteLine("Last execution: {0:dd/MM/yyyy HH:mm:ss}", DateTime.Now);
}
```

- Voer de applicatie uit. Tel zowel geldige als ongeldige getallen op. In het Output venster van Visual Studio zie je nu telkens de tijd van uitvoering verschijnen.



Je kan het Output venster zichtbaar maken tijdens het debuggen van je applicatie via **View → Output**. Het venster staat standaard onderaan in het Visual Studio hoofdvenster.

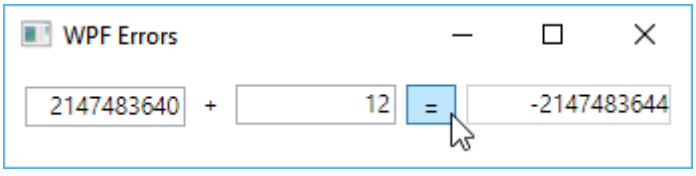
2.5 HET CHECKED KEYWORD EN OVERFLOWS

Zoals eerder al aangegeven, zijn de numerieke datatypes beperkt in grootte. De bovengrens van een `int` is 2147483647 (2^{31} - 1) en de ondergrens is -2147483648 (2^{31}). Binair is dat:

```
 2147483647 = 01111111111111111111111111111111
-2147483648 = 11111111111111111111111111111111
```

Als de bovengrens van een `int` wordt overschreden in een optelling, dan wordt er gewoon verder gerekend vanaf de ondergrens. Dat is zo voor numerieke waarden die negatief kunnen zijn (signed) of steeds positief zijn (unsigned).

- Voer de applicatie uit. Tel de 2147483640 op met 12. Het resultaat hiervan ligt boven de grens van een `int`.



Dit probleem duikt op omdat de .NET runtime toestaat dat de berekening leidt tot een overflow. Deze logische fout wordt toegestaan omdat een controle op overflow voor elke berekening zou leiden tot een gigantisch performantieverlies. Door het ontbreken van een controle noemen we dit een **unchecked** bewerking.

Eigenlijk gaat de .NET CLR er van uit dat je voor berekeningen steeds een realistische keuze in datatype maakt waarbij je aanneemt dat de grens (in de verste verte) niet bereikt zal worden.

Soms kan je dit echter niet voorzien, in dergelijke gevallen kan je een **check** doen op eventuele overflow met het `checked` keyword.

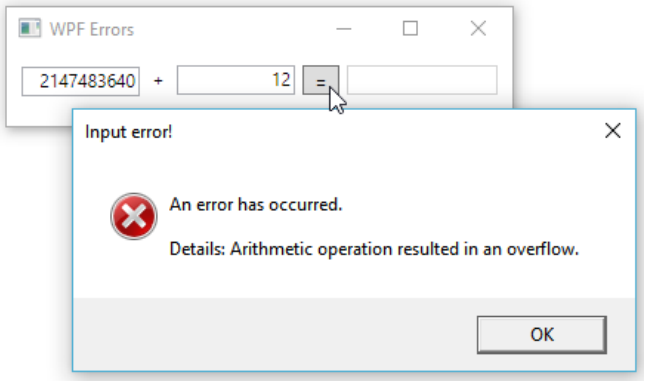
- Wijzig de code van de methode `CalculateSum()`

```
private int CalculateSum(int number1, int number2)
{
    //check if either number is negative
    if (number1 < 0 || number2 < 0)
    {
        throw new ArgumentOutOfRangeException("Terms should not be negative");
    }
    //this instruction will only be executed reached if no exception occurred
    return checked(number1 + number2);
}
```

- Voer de applicatie uit en test dezelfde berekening.

Door het gebruik van het checked keyword zal de runtime deze keer een `OverflowException` genereren, net zoals bij het Parsen.

De fout wordt opgevangen in het laatste catch-blok.



howest.be