

1.0.0

H16: Exception handling

Werken met exceptions

✓ Kennisclip inleiding

Een methode is, in essentie, een stappenplan. Een stappenplan kan niet altijd rekening houden met elke mogelijke situatie. Soms treden er uitzonderlijke situaties op waarin het plan niet meer gevolgd kan worden. Het programmeerconcept dat overeenstemt met zo'n "uitzonderlijke situatie" is de *exception* (Engels voor "uitzondering").

Een uitzonderlijke situatie kan niet altijd verholpen worden door het stappenplan uit te breiden. Code die een deling implementeert kan bijvoorbeeld gewoonweg niet verder als je ze gebruikt om door het getal 0 te delen. Een methode als `File.ReadAllLines` kan zelf niet weten wat ze moet doen als de file die je wil lezen niet bestaat: moet er gebruik gemaakt worden van default data, moet de gebruiker gevraagd worden om de file eerst in te vullen,...? Er is geen eenduidig antwoord. Het hangt af van de **context** waarin de methode gebruikt wordt. Dat wil zeggen: de code die, rechtstreeks of onrechtstreeks, de code heeft opgeroepen waarin de uitzonderlijke situatie is opgetreden.

Exceptions staan toe de verantwoordelijkheid voor het afhandelen van deze uitzonderlijke situatie, dus de exception, te verschuiven naar de context. Zonder het mechanisme te geven, schetst onderstaande code een mogelijke probleemsituatie en mogelijke oplossingen:

```
public void StartProgrammaOp() {  
    // dit kan fout lopen  
    string[] config = File.ReadAllLines(@"C:\configuratiebestand.txt");  
    // StartProgrammaOp weet wat er moet gebeuren als config niet juist is uitgelezen  
    // OPTIE 1, misschien is dit wat je wil  
    Console.WriteLine("Je hebt geen configuratiebestand. Gelieve het in te vullen en dit progr  
    // -1 betekent "iets is misgelopen"  
    Environment.Exit(-1);  
    // OPTIE 2, misschien is dit wat je wil  
    config = ["debug mode", "colorblind mode"];  
    File.WriteAllLines(@"C:\configuratiebestand.txt", config);  
}
```

Misschien is optie 1 beter voor jouw programma, misschien is optie 2 beter voor jouw programma. De auteur van `File.ReadAllLines` kan dat niet voorspellen en kan dus zelf het probleem niet afhandelen. Het probleem moet afgehandeld worden in `StartProgrammaOp`.

Code zonder exception handling

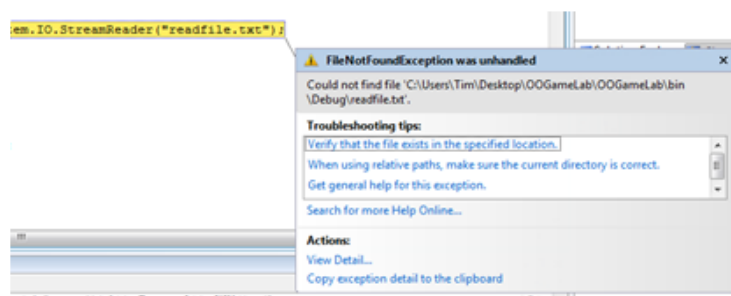
✓ Kennisclip onafgehandelde exceptions

Je zal zelf waarschijnlijk al exceptions zijn tegengekomen in je console programma's. Wanneer je je programma gewoon uitvoert en er plots een hele hoop tekst verschijnt (met onder andere het woord *Exception* in), gevolgd door het prompt afsluiten ervan, dan heb je een exception gegenereerd die je niet hebt afgehandeld.

Niet-afgehandelde exception

Vooraf het eerste zinnetje van zo'n exception is vaak verhelderend. Hier wordt duidelijk aangegeven dat de gezochte file niet bestaat.

Indien je aan het debuggen bent en je krijgt een exception dan zal deze anders getoond worden, maar het gaat wel degelijk om dezelfde fout:



In VS is de foutboodschap iets leesbaarder

Try en Catch

✓ Kennisclip try en catch

Het mechanisme om exceptions af te handelen in C# bestaat uit 2 delen:

- Een `try` blok: dit is de context waarin een mogelijke exception verwacht wordt
- Een of meerdere `catch` -blokken: dit blok zal exceptions die in het bijhorende try-blok voorkomen afhandelen. Met andere woorden: in dit blok staat de code die de uitzondering zo goed mogelijk zal verhelpen.

De syntax is als volgt:

```

try
{
    // context waarin exception mogelijk kan optreden
}
// hieronder zal niet altijd letterlijk Exception staan
// meestal gaat het om specifieke types Exceptions
catch (Exception e)
{
    // code om het probleem zo goed mogelijk af te handelen
}

```

Merk op dat het `catch` -blok meteen na het `try` -blok komt, vergelijkbaar met hoe een `else` -blok meteen na een `if` -blok komt.

Als methode A (zonder geschikt `catch` -blok) methode B oproept en B een exception genereert, moeten we kijken naar de bredere context waarin B is opgeroepen. Dat kan een methode C zijn die A heeft opgeroepen. Een foutmelding zoals in de figuren hoger op deze pagina zie je als er geen context bestaat waarin de exception goed wordt afgehandeld.

Vergelijk deze situatie met een hiërarchisch georganiseerde werkvloer. Veronderstel dat er drie niveaus zijn:

- een directeur
- een departementsmanager
- een bediende

Wat als de bediende tijdens zijn werkdag een belangrijke fout in de boekhouding ontdekt, die hij zelf niet mag rechtzetten? Dan meldt hij dat aan zijn manager. Misschien kan de manager de fout rechtzetten als het gaat om iets dat met zijn eigen departement te maken heeft. Misschien ook niet. Dan moet hij de fout melden aan de directeur, die dan moet beslissen wat er mee moet gebeuren. Als dat niet lukt, kan het bedrijf zware schade oplopen of failliet gaan.

Vervang de directeur door methode C (bijvoorbeeld `Main`), de departementsmanager door methode A (een methode die wordt opgeroepen van uit methode C) en de bediende door methode B. Dan krijg je het mechanisme achter exceptions.

try catch voorbeeld

Voorbeeld try catch

In volgend stukje code kunnen uitzonderingen optreden:

```

string input = Console.ReadLine();
int converted = Convert.ToInt32(input)

```

Een `FormatException` zal optreden wanneer de gebruiker tekst invoert of wanneer een komma-getal wordt ingevoerd. De conversie verwacht dit niet. `Convert.ToInt32()` kan enkel werken met gehele getallen.

We tonen nu hoe we dit met exception handling kunnen opvangen:

```
string input = Console.ReadLine();
try
{
    int converted = Convert.ToInt32(input);
}
catch (Exception e)
{
    Console.WriteLine("Verkeerde invoer!");
}
```

Indien er nu een uitzondering optreedt dan zal de tekst “Verkeerde invoer” getoond worden. Vervolgens gaat het programma verder met de code die mogelijk na het catch-blok staat. Merk ook op dat we het `try`-blok zo klein mogelijk gemaakt hebben door het enkel rond de conversiestap te zetten. Het is een goede gewoonte de context waarin een exception kan optreden zo nauwkeurig mogelijk af te bakenen. Maak je `try`-blokken zo klein als nodig is het gewenste gedrag uit je programma te krijgen, maar niet kleiner.

Meerdere `catch`-blokken



Kennisclip soorten exceptions

`Exception` is een klasse van het .NET framework. Er zijn van deze ouderklasse meerdere exception-classes afgeleid die een specifieke probleemsituatie beschrijven. Enkele veelvoorkomende zijn:

Klasse	Omschrijving
<code>Exception</code>	Basisklasse. Erg breed, dus je gebruikt beter specifiekere exception klassen in je catch blokk
<code>SystemException</code>	Ouderklasse van ingebouwde exceptions. Erg breed, dus je gebruikt beter specifiekere excepti klassen in je catch blokken.
<code>IndexOutOfRangeException</code>	De index is te groot of te klein voor de benaderin van een array.
<code>NullReferenceException</code>	Benadering van een niet-geïnitieerd object. Deze zie je bijvoorbeeld als je een objectmetho oproept van een variabele met waarde <code>null</code> .
<code>ApplicationException</code>	Een ouderklasse voor exceptions die je zelf definieert en die een specifiek soort probleem in jouw applicatie aangeven.

Je kan in het catch blok aangeven welke soort exceptions je wil vangen in dat blok. In het voorbeeld hiervoor stond:

```
catch (Exception e)
{
}
```

Hiermee vangen we dus **alle** Exceptions op, daar alle Exceptions van de klasse `Exception` afgeleid zijn en dus ook zelf een `Exception` zijn. Dit is een vorm van **polymorfisme**: één type data kan meerdere concrete vormen aannemen.

We kunnen nu echter ook specifieke exceptions opvangen. Wanneer je meerdere `catch` -blokken hebt, wordt het eerste eerst toegepast indien mogelijk. Daarom moet je eerst `catch` -blokken voor specifiekere exceptions voor blokken voor algemenere exceptions plaatsen. Stel bijvoorbeeld dat we weten dat de `FormatException` kan voorkomen en we willen daar iets mee doen. Volgende code toont hoe dit kan:

```

try
{
    //...
}
catch (FormatException e)
{
    Console.WriteLine("Verkeerd invoerformaat");
}
catch (Exception e)
{
    Console.WriteLine("Exception opgetreden");
    // hier kunnen we waarschijnlijk niet meer veel aan de fout doen
    // eventueel kunnen we ze wel naar een log schrijven, maar oplossen zal niet gaan
}

```

Indien een `FormatException` optreedt dan zal het eerste catch-blok uitgevoerd worden, anders het tweede. Het tweede blok zal niet uitgevoerd worden indien een `FormatException` optreedt.

Welke exceptions worden gegooid?

De MSDN bibliotheek is de manier om te weten te komen welke exceptions een methode mogelijk kan gooien. Gaan we bijvoorbeeld naar [de pagina van de ReadAllLines methode van de File klasse](#), dan zien we onder "Exceptions" een aantal scenario's waarin het kan foutlopen en hoe deze gesignaleerd worden.

De stack (trace)

✓ Kennisclip stack (trace)

Herinner je uit [het hoofdstuk rond geheugenbeheer](#) dat elke methode-oproep data op de stack plaatst, het "snelle programmeergeheugen". Dus als methode A methode B oproept en methode B roept methode C op, krijg je een stack die er als volgt uitziet:

informatie over methode C
informatie over methode B
informatie over methode A

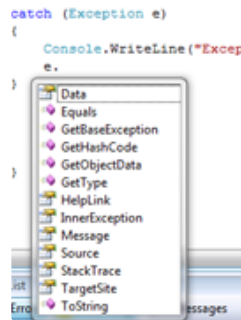
Hier is een belangrijke link met exceptions: de methodes die op een gegeven moment op stack worden opgevolgd, vormen de context waarin een exception kan optreden. Als er een exception optreedt in C is het aan B om die af te handelen. Lukt dat niet, dan is het aan A. Wanneer er een exception optreedt, krijg je een gedeeltelijke weergave van de stack te zien. Deze weergave heet de **stack trace**. Ze is een erg nuttig hulpmiddel als je probeert op te sporen waar een exception precies vandaan komt.

Werken met de exception parameter

✓ Kennisclip exception parameter

De Exceptions die worden 'gegooid' door het programma zijn objecten van de Exception-klasse. Deze klasse bevat standaard een aantal interessante properties en methoden, die je kan oproepen in je code.

Bovenaan de declaratie van het catch-blok geef je aan hoe het exception object in het blok zal heten. Je kent de exception dus toe aan een variabele. In de vorige voorbeelden was dit altijd `e`.



IntelliSense toont de verschillende methodes en properties

Omdat alle exception van Exception afgeleid zijn bevatten ze allemaal minstens:

Element	Omschrijving
Message	Foutmelding in relatief eenvoudige taal
StackTrace	De weergave van de stack die je vertelt hoe de exception is ontstaan.
TargetSite	Methode die de exception heeft gegenereerd. Dit is de onmiddellijke context waarin de exception is opgetreden. Ze staat ook bovenaan de stack trace.
ToString()	Geeft het type van de exception, Message en StackTrace terug als string.

We kunnen via deze parameter meer informatie uit de opgeworpen uitzondering uitlezen en bijvoorbeeld aan de gebruiker tonen:

```
catch (Exception e)
{
    Console.WriteLine("Exception opgetreden");
    Console.WriteLine($"Message: {e.Message}");
    Console.WriteLine($"Targetsites: {e.TargetSite}");
    Console.WriteLine($"StackTrace: {e.StackTrace}");
}
```

Opgelet: vanuit security standpunt is het zelden aangeraden om Exception informatie zomaar naar de gebruiker te sturen. Mogelijk bevat de informatie gevoelige informatie en zou deze door kwaadwillige gebruikers kunnen misbruikt worden!

Finally


Sommige zaken moeten sowieso gebeuren in je programma, of er nu een fout is opgetreden of niet. Een voorbeeld: je opent een databaseverbinding via C# om zo bepaalde data uit de database te lezen. Het blijkt dat de uitgevoerde query geen resultaat oplevert. Dit leidt tot een exception, omdat je programma verwacht dat de opgevraagde data aanwezig is in het systeem. **Of deze fout zich nu voordoet of niet**, achteraf moet de databaseconnectie gesloten worden.


Dit kan met het woordje `finally`. `finally` duidt een block aan dat sowieso wordt uitgevoerd. Als er geen exception is opgetreden, wordt dit block uitgevoerd na het `try` block. Als er wel een is opgetreden, na het `catch` block.

Volgend voorbeeld toont dit aan. Probeer het uit op je eigen machine:

```
try {
    // probeer met en zonder deze regel in commentaar
    throw new Exception("Boo!");
}
catch (Exception e) {
    System.Console.WriteLine("Hooray!");
}
finally {
    System.Console.WriteLine("Phew!");
}
```

Een `finally` block voert bijna altijd uit. **De enige situatie waarin het niet uitvoert, is als je programma stopt terwijl de try of bijbehorende catch nog niet volledig is afgewerkt.** Dit kan bijvoorbeeld zijn omwille van een oproep van de methode `Environment.Exit` of omdat je catch block zelf een exception oplevert die niet wordt afgehandeld en die zo ernstig is dat het controlemechanisme van C# in de war raakt.

 Het is moeilijk op voorhand duidelijk te maken welke exceptions ernstig genoeg zijn om het controlemechanisme van C# in de war te brengen. Volgens [de officiële documentatie](#) is het in de meeste situaties ook niet erg belangrijk wat je programma doet nadat het gecrasht is.

 "Maar de code hierboven werkt ook zonder `finally` !" In dit geval wel. Maar `finally` is "krachtiger" dan code die gewoon achter alle `catch` blokken staat. `finally` voert altijd uit, tenzij het programma volledig afsluit. Zelfs na een `return` of na een handler op hoger niveau.

Zelf uitzonderingen maken

✓ Kennisclip voor deze inhoud

Zelf exceptions opwerpen

Je kan ook in je eigen code uitzonderingen genereren, zodat deze elders opgevangen worden. Je kan hierbij zelf exceptions maken of gewoon gebruik maken van een bestaande `Exception`-klasse.

Een voorbeeld:

```
static int DoeIets(int getal)
{
    if (getal == 0) {
        // DivideByZeroException is ingebouwd
        throw new DivideByZeroException("Getal is 0. Dit is niet voorzien.");
    }
    else {
        return 100 / getal;
    }
}

static void Main(string[] args)
{
    try
    {
        Console.WriteLine(DoeIets(0));
    }
    catch(DivideByZeroException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

"Getal is 0. Dit is niet voorzien." is dus de boodschap die we toevoegen aan onze exception. Ze wordt "opgelost" door de boodschap gewoon te tonen. In een complexer programma zou je bijvoorbeeld de waarde van de input kunnen aanpassen en dan opnieuw de methode aanroepen.

Een eigen exception ontwerpen

Je kan ook eigen klassen afleiden van `Exception` zodat je eigen uitzonderingen kan maken en gooien in je programma. Je maakt hiervoor gewoon een nieuwe klasse aan die je laat overerven van de `ApplicationException`-klasse. Een voorbeeld:

```

class MyException: ApplicationException
{
    public override string ToString()
    {
        return "Dit is een voorbeeld.";
    }
}

```

Om deze exception nu zelf op te gooien gebruiken we het keyword `throw`. In volgende voorbeeld gooien we onze eigen exception op een bepaald punt in de code en vangen deze dan op:

```

static void Main(string[] args)
{
    try
    {
        TimsMethod();
    }

    catch (ApplicationException e)
    {
        Console.WriteLine(e.ToString());
    }
}

static public void TimsMethod()
{
    // deze methode doet niets interessants
    // ze gooit gewoon een Exception als voorbeeld
    MyException exp = new MyException();
    throw exp;
}

```

⚠ Overdrijf niet met eigen Exceptions. Op [de pagina van SystemException](#) vind je, onder "Derived", een heleboel kant-en-klare exceptions voor allerlei situaties.

⚠ Technisch gezien kan je ook rechtstreeks erven van `Exception` en `SystemException`, maar in de documentatie staat uitdrukkelijk dat je eigen klassen best afleidt van `ApplicationException`.

Wanneer exceptions en handling gebruiken

✓ Kennisclip voor deze inhoud

In het begin kan het onduidelijk zijn wanneer je problemen best afhandelt met klassieke conditionele code (met andere woorden, `if` en verwanten) en wanneer met exceptions. Je vertrekt best van uit twee vragen:

1. Welke code kan het probleem tijdig vaststellen?
2. Welke code kan het probleem oplossen?

Als de code die het probleem kan vaststellen het probleem ook kan oplossen, heb je geen exception handling nodig. Volgende code bevat twee problemen die de code wel kan vaststellen en één probleem dat ze niet tijdig kan vaststellen:

```
public static void WensGelukkigeVerjaardag() {  
    bool gewenst = false;  
    byte leeftijd;  
    while(!gewenst) {  
        try {  
            leeftijd = Convert.ToByte(Console.ReadLine());  
        }  
        catch (FormatException e) {  
            Console.WriteLine("Dat was geen (geheel) getal tussen 0 en 255.");  
            continue;  
        }  
        if (leeftijd < 1) {  
            Console.WriteLine("Een nulde verjaardag? Dat heet een geboorte.");  
        }  
        else if (leeftijd > 125) {  
            Console.WriteLine("Sorry, dat geloof ik niet.");  
        }  
        else {  
            Console.WriteLine($"Gelukkige {leeftijd}e verjaardag!");  
            gewenst = true;  
        }  
    }  
}
```

De nulde verjaardag en de verjaardagen vanaf 126 kan onze code tijdig zelf detecteren. Dat is gewoon een vergelijking met een getal. Een probleem met het formaat doet zich pas voor wanneer de conversie plaatsvindt. Dan is het al te laat. Dan kunnen we alleen het probleem nog oplossen.

(i) Kan je een fout goed oplossen met `1±` ? Doe dat dan. Maar gebruik een exception als de code die het probleem tijdig kan vaststellen niet dezelfde is die het probleem kan oplossen.

Labo

h16-weekdagen-zonder-exception-handling

Functionele analyse

Je krijgt code die een exception oplevert, maar je kan deze deze oplossen zonder exception handling.

Technische analyse

Maak eerst een klasse ExceptionHandling met een methode ToonSubmenu zodat je je oefeningen rond exception handling kan demonstreren. Voeg dan volgende methode toe:

```
private static void DemonstreerFoutafhandelingWeekdagenZonderException()
{
    string[] arr = new string[5];
    arr[0] = "Vrijdag";
    arr[1] = "Maandag";
    arr[2] = "Dinsdag";
    arr[3] = "Woensdag";
    arr[4] = "Donderdag";

    for (int i = 0; i <= 5; i++)
    {
        Console.WriteLine(arr[i].ToString());
    }
}
```

Verbeter zelf de fout.

Voorbeeldinteractie (na fix)

```
Vrijdag
Maandag
Dinsdag
Woensdag
Donderdag
```

h16-weekdagen-met-exception-handling

Start terug vanaf de code van eerder. Noem ze ditmaal

`DemonstreerFoutafhandelingWeekdagenMetException`. Los nu het probleem op, enkel en alleen door exception handling toe te voegen op de juiste plaats. De voorbeeldinteractie blijft identiek dezelfde.

h16-overflow-zonder-exception-handling

Functionele analyse

Je krijgt opnieuw code die een exception oplevert, maar je kan deze deze oplossen zonder exception handling.

Technische analyse

Start vanaf volgende code:

```
private static void DemonstreerFoutafhandelingOverflowZonderException()
{
    int num1, num2;
    byte resultaat;
    num1 = 30;
    num2 = 60;
    resultaat = Convert.ToByte(num1 * num2);
    Console.WriteLine("{0} x {1} = {2}", num1, num2, resultaat);
}
```

Spoor zelf de fout op en pas de code aan zodat ze hetzelfde doet, zonder gebruik te maken van exception handling. Gebruik eventueel de debugger.

Voorbeeldinteractie

```
3 x 60 = 1800
```

h16-overflow-met-exception-handling

Start terug vanaf de code van eerder. Los nu het probleem op door te vermelden wat er is misgelopen met behulp van exception handling. Noem je methode nu

`DemonstreerFoutAfhandelingOverflowMetException`.

Voorbeeldinteractie

```
Het getal is te groot om te converteren naar het gewenste formaat.
```

h16-juiste-index

Functionele analyse

Schrijf een programma dat een array maakt met drie willekeurige gehele getallen in en de gebruiker toestaat om een getal naar keuze te tonen, tot hij klaar is.

Technische analyse

Eerst maak je de array aan. Daarna start je een bepaald soort lus op. Kijk hiervoor in de voorbeeldinteractie welke stappen zich steeds herhalen. Als je programma werkt wanneer de gebruiker zich netjes aan de regels houdt, voeg je exception handling toe om rekening te houden met verkeerde indexwaarden. Op andere soorten exceptions wordt niet voorzien. Noem de methode hiervoor `DemonstreerKeuzeElement`.



Omdat dit een oefening op het basisgebruik is, wijken we hier af van [onze richtlijnen over wanneer je exceptions moet gebruiken](#).

Voorbeeldinteractie

```
Geef de index van het getal dat je wil zien
2
Het getal is 9
Wil je doorgaan?
ja
Geef de index van het getal dat je wil zien
17
Die index hebben we niet!
Wil je doorgaan?
ja
Geef de index van het getal dat je wil zien
-1
Die index hebben we niet!
Wil je doorgaan?
ja
Geef de index van het getal dat je wil zien
0
Het getal is 4
Wil je doorgaan?
ja
Geef de index van het getal dat je wil zien
fozejioefzio
Unhandled exception: System.FormatException: Input string was not in a correct format.
   at System.Number.ThrowOverflowOrFormatException(ParsingStatus status, TypeCode type)
   at System.Number.ParseInt32(ReadOnlySpan`1 value, NumberStyles styles, NumberFormatInfo info)
   at System.Convert.ToInt32(String value)
   at IndividueleOefeningen.ExceptionHandling.DemonstreerKeuzeElement() in /home/vincent/Projects/ooprogrammeren/IndividueleOefeningen/IndividueleOefeningen/ExceptionHandling.cs:line 123
   at IndividueleOefeningen.ExceptionHandling.ToonSubMenu() in /home/vincent/Projects/ooprogrammeren/IndividueleOefeningen/IndividueleOefeningen/ExceptionHandling.cs:line 29
   at IndividueleOefeningen.Program.Main(String[] args) in /home/vincent/Projects/ooprogrammeren/IndividueleOefeningen/IndividueleOefeningen/Program.cs:line 30
```

h16-juiste-index-extra-voorzichtig

Functionele analyse

Bij de vorige oefening zijn er nog randsituaties mogelijk. Handel de meest waarschijnlijke op een specifieke manier af en voorzie een algemeen vangnet.

Technische analyse

Test voorgaande code uit met tekst in plaats van een getal. Test voorgaande code uit met een enorm groot getal. Test voorgaande code uit met een kommagetal. Test voorgaande code uit door meteen op enter te duwen. Test voorgaande code uit met een dollarteken in plaats van een getal. Onthoud de soorten exceptions.

Voorzie vervolgens exception handling om uit te leggen wat er is misgelopen zonder het programma te laten crashen, zoals je in de vorige oefening ook hebt aangegeven dat een bepaalde index niet geldig was.

Voorzie ook code om om het even welk type exception af te handelen.

h16-leeftijd-kat

Functionele analyse

Schrijf een klasse `Kat`. Deze klasse encapsuleert onze domeinkennis over katten en zorgt ervoor dat we geen onrealistische katten kunnen voorstellen in een softwaresysteem.

Technische analyse

Maak de klasse. Deze beschikt over een property `Leeftijd`, met een publieke getter, maar geen publieke setter. De leeftijd wordt meegegeven bij constructie en wordt ingesteld, maar als hij hoger is dan 25, moet de code die het `Kat`-object heeft proberen aanmaken een `ArgumentException` afhandelen, met de boodschap: "Deze kat is te oud!".

Pas ook onderstaande code (die je in een methode `DemonstreerLeeftijdKat` van `ExceptionHandler` mag plaatsen) aan zodat deze boodschap wordt geprint, maar je programma niet crasht:

```
Kat kat = new Kat(27);
```


Voorbeeldinteractie

```
Deze kat is te oud!
```

h16-leeftijd-katten

Functionele analyse

Schrijf code die op willekeurige wijze een lijst met katten aanmaakt. Dit kan mis lopen. Hoe dan ook moet je code netjes achter zich opkuisen door in alle gevallen deze lijst terug leeg te maken wanneer het werk gedaan is.

 Dit is een nogal vreemd voorbeeld, maar we hebben in deze cursus niet gezien hoe je met databaseconnecties, streams, e.d. werkt en dat zijn het soort zaken die je typisch opkuisst in alle mogelijke scenario's.

Technische analyse

- Maak een methode `DemonstreerLeeftijdKatMetResourceCleanup`
- Maak in deze methode een lijst met katten
- Voeg twintig katten met een willekeurige leeftijd van 0 tot 30 toe aan deze lijst
- Als dit zonder problemen verloopt, toon je: "De volledige lijst met katten is aangemaakt!"
- Als er ergens een probleem optreedt omwille van een ongeldige leeftijd, toon je: "Het is niet gelukt :-(
- In beide gevallen zorg je dat de methode eindigt door te lijst terug leeg te maken met de methode `Clear`
 - Doe dit op zo'n manier dat dit **altijd** gebeurt, ook als er een andere exception dan de `ArgumentException` optreedt en deze exception op een hoger niveau wordt opgevangen

Voorbeeldinteractie

```
De volledige lijst met katten is aangemaakt!
```

OF

```
Het is niet gelukt :-(
```

h16-filehelper

Functionele analyse

We willen een *utility* methode schrijven om makkelijk files te lezen.

Technische analyse

Schrijf een methode `FileHelper`. Deze vraagt eerst om een pad naar een file en probeert deze file te lezen. Als dit lukt, geeft ze heel de inhoud van de file terug als string. Als de file niet bestaat, geeft ze nog altijd een string terug (geen exception!) met de waarde: "File kon niet gevonden worden." In het geval van andere problemen met input/output, geeft ze ook een string terug, met waarde: "File bestaat, maar kon niet gelezen worden. Mogelijk heb je geen toegangsrechten." In nog algemenere problemen geeft ze een string terug met waarde: "Er is iets misgelopen. Neem een screenshot van wat je aan het doen was en contacteer de helpdesk."

```
Welke file wil je lezen?  
> C:\Users\vincent\TODO.txt  
naar de winkel gaan  
met de hond wandelen  
backups maken
```

OF

```
Welke file wil je lezen?  
> C:\Users\vincent\FILEDIENIETBESTAAT.txt  
File kon niet gevonden worden
```

OF

```
Welke file wil je lezen?  
> C:\Windows\beschermdedefile.txt  
File bestaat, maar kon niet gelezen worden. Mogelijk heb je geen toegangsrechten.
```

h16-leeftijd-kat-custom

Functionele analyse

We doen een uitbreiding op h16-leeftijd-kat. We zouden graag makkelijk in detail kunnen uitleggen aan de gebruiker waarom het is misgelopen. We doen dit hier in de eerste plaats door een custom exception type te voorzien.

Technische analyse

- Maak een kopie van je klasse `Kat`. Noem deze `KatMetCustomException`.
- Maak een klasse `KatLeeftijdException`. Deze erft van `ArgumentException`.
 - Ze heeft drie read-only properties, waarvan je zelf het juiste type zou moeten kunnen bepalen:
 - `MeegegevenWaarde`
 - `LaagstMogelijkeWaarde`
 - `HoogstMogelijkeWaarde`
 - Ze heeft een constructor die (enkel) waarden voor deze drie properties als parameters heeft.
 - Wanneer de leeftijd van een `KatMetCustomException` wordt ingesteld, wordt een exception van dit type in plaats van een `ArgumentException` gegooid. Hierbij vul je de argumenten in op basis van de rest van je code.
- Maak ook een variatie op je eerdere demonstratiemethode. Noem deze `DemonstreerLeeftijdKatMetCustomException`.

Voorbeeldinteractie

Als dit in je code staat:

```
KatMetCustomException kat = new KatMetCustomException(37);
```

37 is geen geldige leeftijd. De laagst mogelijke leeftijd is 0 jaar, de hoogst mogelijke lee

Dit bericht mag niet "hardgecodeerd zijn". Elk getal moet uit de exception gehaald worden.

Schooladmin project: geen dubbele data

Maak in je SchoolAdmin project een klasse `DuplicateDataException`. Deze heeft twee properties, `Waarde1` en `Waarde2`, beide van type `System.Object`. Ze heeft ook een constructor die een message en de twee waarden als parameter heeft.

Schooladmin project: geen dubbele cursusnamen

Wanneer je een nieuwe cursus aanmaakt, wordt deze vanzelf geregistreerd in het systeem. Pas je code aan zodat geen twee cursussen met dezelfde naam kan registreren. Meerbepaald: zorg dat een poging om een cursus aan te maken afgebroken wordt door middel van een `DuplicateDataException` vooraleer de teller van alle cursussen wordt verhoogd. De boodschap die je meegeeft is: "Nieuwe cursus heeft dezelfde naam als een bestaande cursus." Voor de eerste waarde geef je de nieuwe cursus, voor de tweede geef je de bestaande cursus.

Zorg er ook voor dat je keuzemenu niet crasht wanneer deze fout zich voordoet, maar de boodschap van de exception toont en het ID van de bestaande cursus waarmee de nieuwe cursus zou overlappen. Dit kan je doen door `Waarde2` te casten.

Schooladmin project: geen lege waarden voor VakInschrijving

Het is niet logisch een inschrijving te hebben zonder student of zonder vak. Zorg ervoor dat een VakInschrijving niet kan aangemaakt worden zonder een (of beide) van deze elementen. Gebruik hiervoor een `ArgumentException`. Breid bij wijze van demonstratie je keuzemenu om een student of een vak toe te voegen uit met een optie met nummer 0 om de waarde `null` te gebruiken. (Dit zou je in het echt niet toevoegen aan je systeem zelf, maar je zou aparte testcode schrijven die dit doet.) Zorg ook dat het niet toegelaten is een student twee keer in te schrijven voor hetzelfde vak. Ook dat levert een `ArgumentException`. Zorg dat het keuzemenu niet crasht wanneer je deze optie kiest, maar gewoon de boodschap van de exception toont.

Schooladmin project: beperkt aantal inschrijvingen per vak

Er mogen niet meer dan 20 lopende inschrijvingen per cursus zijn. Zorg ervoor dat er een `CapaciteitOverschredenException` (met enkel de message als parameter) optreedt wanneer je iemand probeert in te schrijven voor een cursus waarvoor al 20 inschrijvingen (zonder toegekend resultaat) bestaan. Zorg ervoor dat je keuzemenu hierop voorzien is en de message toont, zonder te crashen.

h16-gedeeltelijke-afhandeling

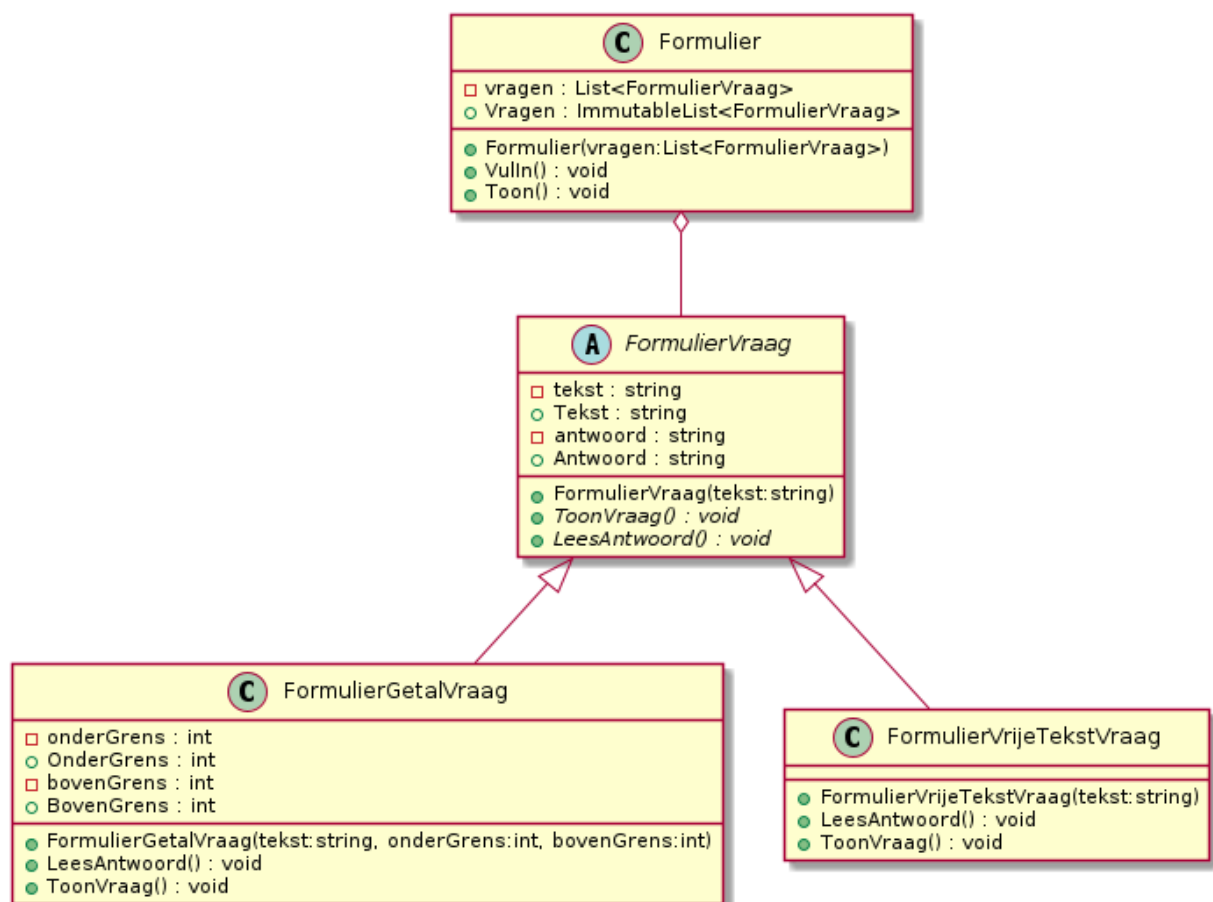
Functionele analyse

We schrijven flexibele formuliercode, die we ook zullen gebruiken om de duidelijkheid van onze formulieren te verbeteren. Een formulier logt ongeldige antwoorden op vragen vooraleer het de fout signaleert aan een hoger niveau.

! Dit is een uitdagende, maar leerrijke en realistische oefening.

Technische analyse

We vertrekken vanaf dit klassendiagram:



Klassendiagram formulieren

Je krijgt ook volgende demonstratiecode om in het submenu van je klasse `ExceptionHandler` te plaatsen:

```

private static void DemonstreerFormulieren() {
    var vraag1 = new FormulierGetalVraag("Hoe oud ben je?", 18, 130);
    var vraag2 = new FormulierVrijeTekstVraag("Hoe ziet jouw ideale dag eruit?");
    var vraag3 = new FormulierGetalVraag("Hoe veel personen heb je ten laste?", 0, 10);
    var vraag4 = new FormulierVrijeTekstVraag("Wie is je idool?");
    Formulier f1 = new Formulier(new List<FormulierVraag>{ vraag1, vraag2 });
    Formulier f2 = new Formulier(new List<FormulierVraag>{ vraag3, vraag4 });
    try {

        f1.VulIn();
        f1.Toon();
    }
    catch (Exception) {
        System.Console.WriteLine("We zullen dit formulier weggooien.");
        f1 = null;
    }
    try {
        f2.VulIn();
        f2.Toon();
    }
    catch (Exception) {
        System.Console.WriteLine("We zullen dit formulier weggooien.");
        f2 = null;
    }
}

```

De werking van elke klasse is als volgt:

- **FormulierVraag:**
 - Dit stelt één vraag op één formulier voor, inclusief het antwoord dat eventueel al is gegeven op deze vraag.
 - De tekst is de vraag waarop een antwoord verwacht wordt. Deze mag nooit leeg of null zijn.
 - Het antwoord is het antwoord dat de invuller gegeven heeft op deze vraag, in tekstformaat. Dit moet initieel null zijn maar mag later nooit meer naar null gewijzigd worden.
 - Het presenteren van een vraag en het inlezen van een antwoord hangt af van het vraagtype, omdat elk vraagtype eigen instructies heeft (bv. antwoorden in tekst of met een reeks cijfers,...)
- **FormulierGetalVraag:**
 - Dit stelt een vraag voor waarbij een getal wordt verwacht.
 - De ondergrens is het kleinste getal dat mag worden ingegeven, de bovengrens is het grootste getal dat mag worden ingegeven.
 - Als een vraag van dit type wordt aangemaakt met een ondergrens die groter is dan de bovengrens, krijgen we een `ArgumentException`.
 - Bij het inlezen van een antwoord wordt de ingetypte tekst geconverteerd naar een getal. Als dit getal tussen de ondergrens en bovengrens ligt, wordt het antwoord (het getal, voorgesteld als string) opgeslagen. Anders wordt gesignaleerd dat het antwoord tussen deze twee getallen moet liggen en wordt er opnieuw tekst ingelezen, tot er een antwoord verkregen is (of er een exception optreedt).
 - Tonen van een vraag gaat als volgt:
 - Eerst wordt de vraagtekst geprint.
 - Daaronder wordt toegevoegd: "Dit is een getal tussen ... en ..." (met daar de grenzen ingevuld)
- **FormulierVrijeTekstVraag:**
 - Alle tekst is geldig als antwoord
 - Tonen van een vraag: de vraagtekst wordt getoond. Daaronder wordt getoond: "Sluit af met ENTER."
- **Formulier:**
 - Bij constructie wordt er een lijst met `FormulierVraag`-objecten meegegeven.
 - Deze vragen worden opgeslagen en na aanmaak van het formulier kan de lijst met vragen niet meer gewijzigd worden.
 - Een formulier invullen betekent dat we één voor één elke vraag in het formulier tonen en het antwoord inlezen.
 - Dit kan fout lopen. Als er iets fout loopt (wat dan ook), tonen we een bericht "Onverwachte fout wordt naar schijf weggeschreven." en staan we vervolgens toe dat de fout naar een hoger niveau van de programmacode gaat.
 - Een formulier tonen betekent dat we voor elke vraag in het formulier de tekst van de vraag en het opgeslagen antwoord tonen.

Voorbeeldinteractie

Hoe oud ben je?
Dit is een getal tussen 18 en 130
23
Hoe ziet jouw ideale dag eruit?
Sluit af met ENTER.
Koekjes bakken, thee drinken.
Vraag: Hoe oud ben je?
Antwoord: 23
Vraag: Hoe ziet jouw ideale dag eruit?
Antwoord: Koekjes bakken, thee drinken.
Hoe veel personen heb je ten laste?
Dit is een getal tussen 0 en 10
20
Je antwoord moet tussen 0 en 10 liggen.
15
Je antwoord moet tussen 0 en 10 liggen.
1
Wie is je idool?
Sluit af met ENTER.
Billy Gibbons
Vraag: Hoe veel personen heb je ten laste?
Antwoord: 1
Vraag: Wie is je idool?
Antwoord: Billy Gibbons