HO
GENT

Exception handling

Ludwig Stroobant

Table of Contents

1. Doelstellingen

Na het studeren en maken van de oefeningen van dit hoofdstuk ben je in staat om volgende zaken te herkennen, toe te lichten, te definiëren, toe te passen en te implementeren:

- Excepties afhandelen
- Propagatie van een exceptie
- Excepties gooien
- Excepties opvangen
- `using` statement
- Excepties declareren

2. Inleiding tot 'exception handling'

De Engelse term 'exception' (uitzondering) is een verkorting van de uitdrukking 'exceptional event' (uitzonderlijke gebeurtenis).

Een exception is een uitzonderlijke gebeurtenis, die kan optreden bij het uitvoeren van een applicatie en die de normale voortgang van de applicatie onderbreekt.

Via exception handling kan die uitzonderlijke gebeurtenis opgevangen worden. Zo kunnen robuuste programma's ontwikkeld worden die kunnen omgaan met probleemsituaties. Het programma blijft stabiel draaien ofwel wordt het programma correct en gebruiksvriendelijk beëindigd.

3. Wat gebeurt er zonder exception handling?

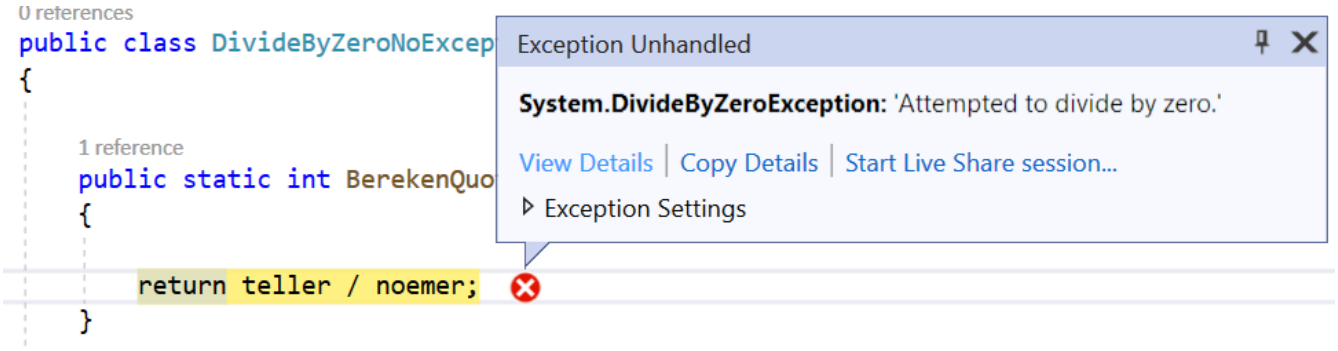
Een exception is een uitzonderlijke gebeurtenis, die kan optreden bij het uitvoeren van een applicatie en die de normale voortgang van de applicatie onderbreekt.

Een voorbeeld van zo een uitzonderlijke gebeurtenis is het uitvoeren van een *deling door nul*. Indien de exception niet wordt afgehandeld zal de applicatie "crashen".

```
public class DivideByZeroNoExceptionHandling
{

    public static int BerekenQuotient(int teller, int noemer)
    {
        return teller / noemer;
    }

    public static void Main(string[] args)
    {
        Console.Write("Geef een integere waarde voor de teller: ");
        int teller = Int32.Parse(Console.ReadLine());

        Console.Write("Geef een integere waarde voor de noemer: ");
        int noemer = Int32.Parse(Console.ReadLine());

        int quotient = BerekenQuotient(teller, noemer);
        Console.Write($"\nResultaat: {teller} / {noemer} = {quotient}\n");
    }
}
```

Indien er in bovenstaand voorbeeld als **noemer** een waarde wordt ingegeven verschillend van nul, zal het programma normaal verlopen.

Wordt echter de waarde nul (= '0') ingegeven, dan zal het programma bruusk onderbroken worden en niet tot het einde lopen.



Voer je het programma uit vanaf de command line krijg je de output:

```
Geef een integere waarde voor de teller: 10
Geef een integere waarde voor de noemer: 0
Unhandled exception. System.DivideByZeroException: Attempted to divide by zero.
   at Cui.DivideByZeroNoExceptionHandling.BerekenQuotient(Int32 teller, Int32 noemer) in C:\...\Programmeren
   Gevorderd\Excepties\repo\Exceptions\Cui\DivideByZeroNoExceptionHandling.cs:line 12
   at Cui.DivideByZeroNoExceptionHandling.Main(String[] args) in C:\...\Programmeren
   Gevorderd\Excepties\repo\Exceptions\Cui\DivideByZeroNoExceptionHandling.cs:line 23
```

Er wordt een "stack trace" afgedrukt met extra informatie:

- Soort fout wordt aangegeven (`DivideByZeroException`)
- Er wordt ook aangegeven waar de fout zich voordoet in de code (paarse kleur)

De stack trace toont welke methodes werden aangeroepen en waar het precies fout liep.

- Weergave aanroep methode: klasse, methode, bestandsnaam, regelnummer.
- Als je het chronologisch in de tijd wil volgen, lees je van onder naar boven.
- De hoogste vermelding in het exceptie gedeelte van de uitvoer noemen we het **throw point** (*DivideByZeroNoExceptionHandling.cs:line 12*). Daar is de fout ontstaan.

In bovenstaand voorbeeld kunnen we nog een tweede exception triggeren: er wordt van de gebruiker verwacht dat hij een geheel getal ingeeft. Wat gebeurt er als hij voor de waarde van de noemer de tekst "hallo" ingeeft?

```
Geef een integere waarde voor de teller: 10
Geef een integere waarde voor de noemer: hallo
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.Int32.Parse(String s)
   at Cui.DivideByZeroNoExceptionHandling.Main(String[] args) in
   C:\Users\lstr508\Documents\Archive\Onderwijs\HoGent\gitHub\Cursussen\Programmeren
   Gevorderd\Excepties\repo\Exceptions\Cui\DivideByZeroNoExceptionHandling.cs:line 21
```

- We wensen een geheel getal in te lezen, maar er wordt tekst ingegeven.
- Ook deze fout onderbreekt het programma bruusk.

Indien het optreden van een exception niet wordt afgehandeld, dan stopt het programma bruusk. We kunnen dit oplossen door deze exceptions "af te handelen" zodat het programma robuust wordt. Op die manier krijgen we een stabiele applicatie die onder controle blijft tot het einde.

4. Exceptions

4.1. Exception object gooien

Als er tijdens het uitvoeren van een methode of constructor een uitzonderlijke gebeurtenis optreedt kan er een exception object aangemaakt en gegooid worden. Het normale verloop van het programma wordt zo onderbroken:

```
1     if (String.IsNullOrWhitespace(mijnString)) {
2         throw new ArgumentException();
3     }
```

- de code maakt een instantie van een exception object aan (`new ArgumentException()`) en levert dit object aan het runtime systeem.
  - dit object is het *exception object*
- het exception object bevat informatie over de exceptie, inclusief:
  - zijn type (welke exceptie is er opgetreden) (`ArgumentException`)
  - de toestand waarin het programma zich bevond toen de exception optrad.

De creatie van een exception object en het doorgeven van dit object aan het runtime systeem noemt men **throwing an exception** (het gooien van een exceptie). Vandaar het keyword `throw`.

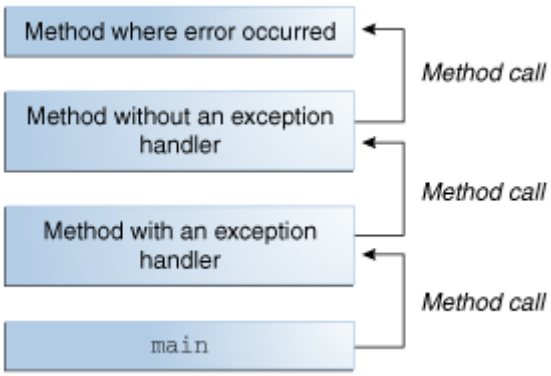
Indien de exceptie wordt gegooid vanuit een methode, zal deze methode geen return waarde teruggeven.

Indien de exceptie wordt gegooid vanuit een constructor (direct of indirect), wordt de constructie van het object onderbroken. Resultaat is dat de creatie van het object niet afgerond wordt: **er wordt geen object gecreëerd!**.

4.2. De gegooide exception verwerken

Nadat een methode een exception object wierp zal het runtime systeem *iets* proberen zoeken dat de exceptie kan afhandelen.

De mogelijkheden van dat 'iets' om de exceptie af te handelen wordt gezocht in de lijst van methoden die aangeroepen werden om tot de methode te komen waar de exceptie optrad. Deze lijst van methoden noemen we de **call stack**.



Indien er in de call stack niets gevonden wordt om de exceptie af te handelen zal het programma bruusk eindigen en is het o.a. deze **call stack** die via de error stream op het scherm wordt geplaatst.

4.3. Op zoek doorheen de 'call stack'

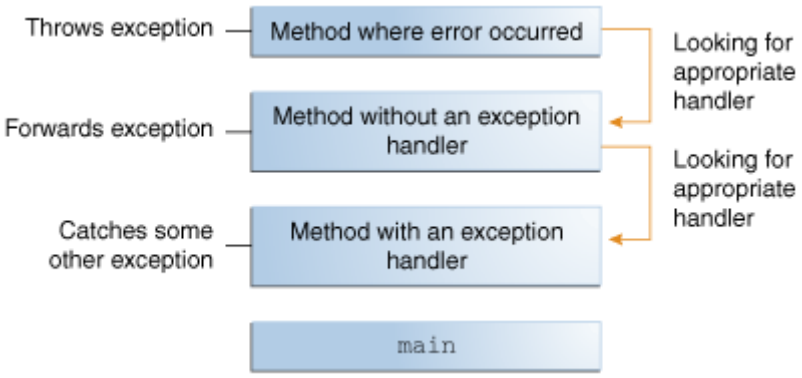
Het runtime systeem zal doorheen de methoden in de call stack op zoek gaan naar een stukje code die de exceptie kan behandelen.

De code die de exceptie zal afhandelen noemt de **exception handler**.

Eerst wordt naar deze exception handler gezocht in de methode/constructor waar de exceptie optrad, waarna verder wordt gezocht doorheen de call stack in de omgekeerde volgorde waarin de methodes opgeroepen werden.

Wanneer een gepaste exception handler gevonden wordt zal het runtime systeem het exception

object doorgeven als *argument* aan deze exception handler.



Een exception handler is pas geschikt om een exception af te handelen als het type van het exception object overeenstemt met het type dat de exception handler kan afhandelen: het type van de ***exception parameter***.

De exception handler die de exceptie afhandelt vangt het exception object op (**catching an exception**).

Als het runtime systeem geen geschikte exception handler vindt tijdens het doorzoeken van de call stack, zal het runtime systeem stopgezet worden (met als gevolg dat de applicatie stopt!) Dit moet vermeden worden.

5. Voordelen van exception handling

Het gebruik van exceptions heeft zijn voordelen t.o.v. traditionele error-management technieken.

5.1. Error-Handling afzonderen van "gewone" code

Excepties laten ons toe om het afhandelen van fouten gescheiden te houden van het normale verloop van een applicatie. In traditionele software leidde het vaststellen van fouten en het afhandelen hiervan meestal tot moeilijk leesbare (spaghetti) code. Neem bijvoorbeeld volgende pseudo code:

```
readFile {
    open the file;
    determine its size;
    allocate that much memory;
    read the file into memory;
    close the file;
}
```

Dit lijkt een eenvoudige methode, volgende fouten kunnen echter optreden:

- Wat als het bestand niet kan geopend worden?

- Wat als de lengte van het bestand niet bepaald kan worden?
- Wat als er onvoldoende vrij geheugen is om te alloceren?
- Wat als een fout optreedt tijdens het lezen van het bestand?
- Wat als het bestand niet gesloten kan worden?

Om al deze gevallen te dekken moet de methode readFile extra code krijgen om de error te detecteren en af te handelen. Traditioneel zou dit error-management er als volgt kunnen uitzien:

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

Er werd zoveel code toegevoegd dat de originele zeven lijnen code verloren gaan in de toegevoegde error-management code. Ook de logische flow van de code is verloren wat ze moeilijk leesbaar maakt.

Het gebruik van exception handling laat toe om de logische flow van de code te behouden en de fouten elders af te handelen. Door gebruik te maken van exception handling zou de originele code er als volgt uitzien:

```
readFile {
    try {
        open the file;
        determine its size;
        allocate that much memory;
        read the file into memory;
        close the file;
    } catch (fileOpenFailed) {
       doSomething;
    } catch (sizeDeterminationFailed) {
        doSomething;
    } catch (memoryAllocationFailed) {
        doSomething;
    } catch (readFailed) {
        doSomething;
    } catch (fileCloseFailed) {
        doSomething;
    }
}
```

De logische flow van het programma blijft bestaan en is leesbaar. De error handling loopt gescheiden van de normale flow.

5.2. Propagatie van een exception

Een tweede voordeel van exeptions is de mogelijkheid om de exceptie te propageren doorheen de stack.

Exception handling laat toe om een exceptie op een hoger niveau af te handelen.

Vaak is het eenvoudiger om een fout af te handelen op een hoger niveau (bv in de user interface). Het propageren van een exception tot een hogere niveau, waar de afhandeling logischer of eenvoudiger is, leidt op zijn beurt tot een hogere *maintenance* factor.

6. Hoe een exceptie gooien

Voor een exceptie kan opgevangen en afgehandeld worden, moet ze eerst gegooid worden. Eender welke code kan een exceptie gooien: eigen geschreven code of code uit een andere library geschreven door iemand anders zoals de libraries die meekomen met het platform.

Als je eigen code een exception kan gooien, vermeld dit dan zeker in de documentatie!

6.1. Het `throw` statement

Gebruik het `throw` statement om een exceptie te gooien. Dit statement verwacht één argument: een object dat gegooid kan worden. Een object dat gegooid kan worden is een instantie van een subklasse van de klasse Exception.

```
1     throw someExceptionObject;
```

De volgende methode `pop` is terug te vinden in een veelgebruikt stack object. De methode verwijdert een element van de top van de stack en geeft dit element terug.

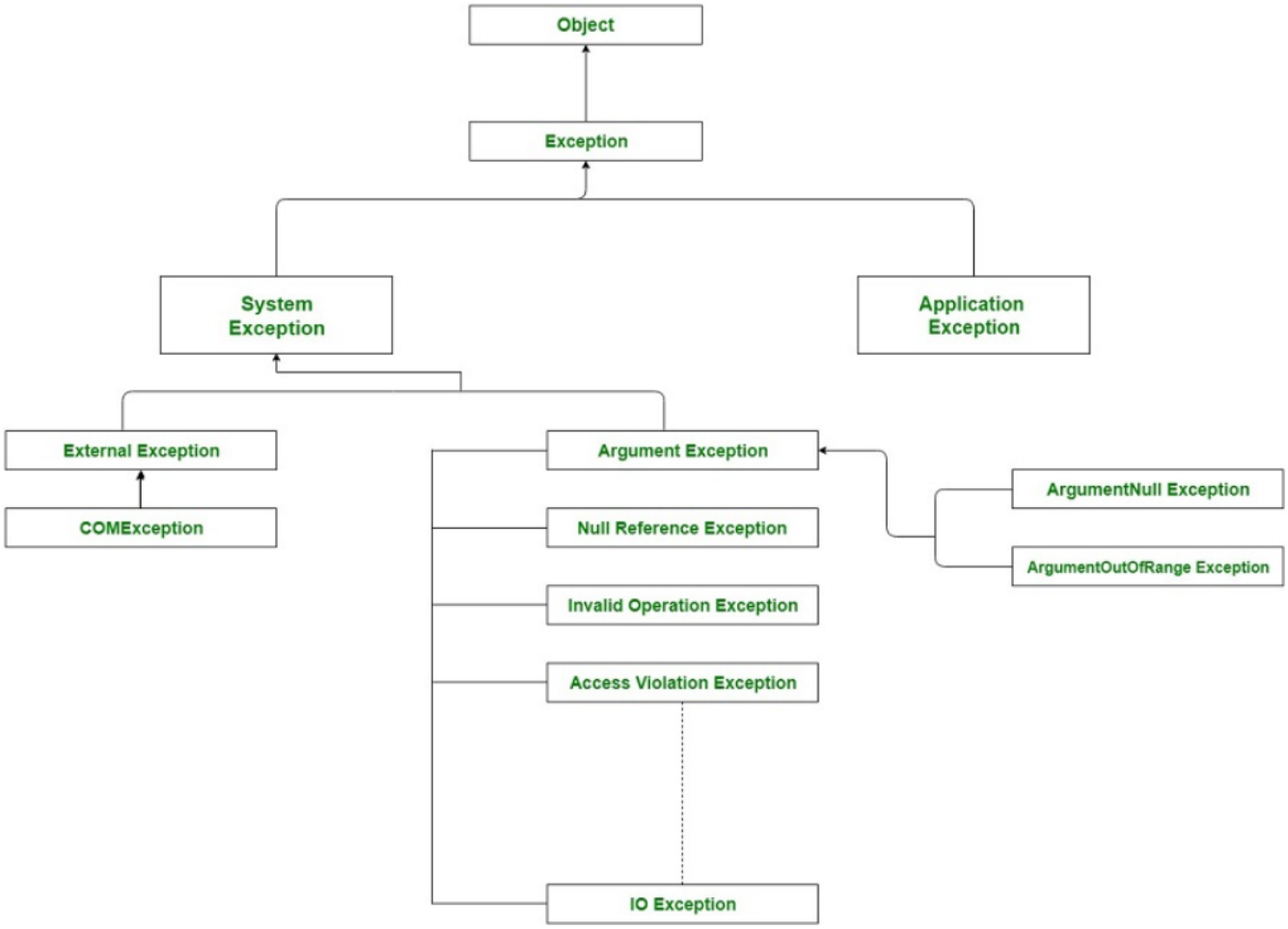
```
1  public Object pop() {
2      Object obj;
3
4      if (size == 0) { ①
5          throw new EmptyStackException();
6      }
7
8      obj = objectAt(size - 1);
9      setObjectAt(size - 1, null);
10     size--;
11     return obj;
12 }
```

① De pop methode gaat na of er elementen zijn op de stack. Als deze leeg blijkt te zijn (als size == 0), dan wordt een instantie van EmptyStackExceptions aangemaakt (`new EmptyStackException()`) en dit object wordt gegooid.

6.2. Exception klasse en subklassen

Je kan zelf een klasse hierarchy voor excepties opgebouwen om specifieke problemen op te vangen. Je kan dus zelf een exception klasse aanmaken en hiervan een object gooien. Zo kan je eenvoudig onderscheid maken tussen verschillende gebeurtenissen.

Onderstaande figuur illustreert de klasse hierarchie van de klasse Exception en enkele belangrijke subklassen.



Let op:

> You should derive custom exceptions from the Exception class rather than the ApplicationException class. You should not throw an ApplicationException exception in your code, and you should not catch an ApplicationException exception unless you intend to re-throw the original exception.
>
> — Microsoft API for the ApplicationException Class

7. Opvangen en afhandelen van exceptions (catching & handling)

Een applicatie kan excepties gebruiken om aan te geven dat een bijzondere gebeurtenis of error zich voordeed.

Om een exception te gooien maak je gebruik van het *throw* statement en voorzie je dit statement van een exception object dat informatie bevat van de specifieke error die optrad.

Gebruik je code geschreven door iemand anders, ga dan zeker in de documentatie na of er exceptions kunnen gegooid worden door die code. Die exceptions handel je best af!

Een applicatie kan exceptions opvangen gebruik makende van een `try/catch` blok, eventueel samen met een `finally` blok.

- Het `try` blok identificeert een blok code waarin een exception kan optreden. Een exception kan enkel afgehandeld worden indien ze optreedt binnen een try blok.
- Het `catch` blok identificeert een blok code, de **exception handler**, die een specifieke exception (= het type van de exception parameter) kan afhandelen.
- Het `finally` blok identificeert een blok code, waarvan gegarandeerd is dat het zal uitgevoerd worden (onafhankelijk of er een error optreedt of niet). Dit is de geschikte plaats om resources af te sluiten en alles uit het try blok op te ruimen.

7.1. Het `try` blok

In een eerste stap om een exception handler op te zetten dient de code die een exception kan gooien in een try-blok geplaatst te worden.

```
try {
    code ①
}
catch and finally blocks ... ②
```

① Dit segment omvat geldige code die een exceptie kan gooien.

② Het catch en finally blok worden later uitgelegd.

Als een exceptie wordt gegooid binnen een try blok, kan deze afgehandeld worden door een exception handler van het juiste type. Om een exception handler te koppelen aan het try-blok koppel je aan dit blok een catch-blok.

7.2. Het `catch` blok

Exception handlers worden aan een try-blok gekoppeld door één of meer catch blokken direct na het try blok te plaatsen. Tussen de try-catch kan geen andere code voorkomen.

```
try {

} catch (ExceptionType name) { ①
 // error handling code ②
} catch (OtherExceptionType name) {
 // error handling code ②
}
```

① Elke catch declaratie is een **exception handler** die het type exceptie kan afhandelen vermeld in zijn parameterlijst (de exception parameter types). In het voorbeeld declareert het eerste catch blok dat deze exception handler een gegooid exception object van het type `ExceptionType` kan afhandelen. Dit type moet een subklasse zijn van de klasse Exception. De handler verwijst ernaar d.m.v. de naam van de klasse.

② Dit catch blok bevat code die uitgevoerd wordt als de exception handler wordt aangeroepen. Het runtime systeem zal deze exception handler aanroepen als het de eerste is in de call stack wiens exception parameter type overeenkomt met de exceptie die gegooid werd. Het systeem vergelijkt de types en vindt een geschikte kandidaat als het gegooide exception object kan toegekend worden aan het argument van de exception handler (denk aan het `is` keyword).

Let op voor polymorfisme! De eerste exception handler waarvan het type van de exceptie parameter overeenkomt met het type van het gegooide exceptie object zal uitgevoerd worden. Let hierbij op de "IS EEN" relatie tussen verschillende klassen.

Een exception handler is in staat meer te doen dan enkel een foutboodschap tonen of de applicatie te stoppen. Een error kan hersteld worden, de gebruiker kan een boodschap krijgen of de exceptie kan ingevoegd worden in een exceptie ketting om deze te propageren naar een hoger niveau.

> In general, you should only catch those exceptions that you know how to recover from. Therefore, you should always specify an object argument derived from System.Exception
>
> — C# reference - try/catch

7.3. Het `finally` blok

Het finally-blok wordt **altijd** uitgevoerd als het try blok afloopt, al dan niet door een exceptie.

Dit zorgt ervoor dat het finally blok altijd wordt uitgevoerd, ook al treedt er een onverwachte exception op.

Het runtime systeem voert altijd de statements uit in het finally-blok onafhankelijk van wat er in het try-blok gebeurt. Dit is de ideale plaats om alles mooi op te ruimen.

```
    finally {
        if (out != null) {
            System.out.println("Closing PrintWriter");
            out.close(); ①
        } else {
            System.out.println("PrintWriter not open");
        }
    }
```

① Het finally blok wordt uitgevoerd en zorgt ervoor dat het PrintWriter object `out` wordt gesloten.

Het finally blok is, op het `using` statement na, het meest geschikt om het lekken van resources te voorkomen. Aangezien het finally blok altijd wordt uitgevoerd kan je resources hier manueel vrijgeven en ben je zeker dat deze code ook wordt uitgevoerd. Als programmeur kan je het vrijgeven van resources in een finally blok wel "vergeten" te implementeren. Het `using` statement past hier een mouw aan.

Een resource is een gedeelde component. Een voorbeeld hiervan is een gedeelde netwerkpoort op een computer: er is maar één ethernetpoort, die gedeeld wordt door alle applicaties. Indien een connectie wordt opgezet via de netwerkpoort, dan krijgt een applicatie een combinatie van het IP adres en een poort nummer. Wordt bij een fout deze resource niet opnieuw vrijgegeven, dan kan later dezelfde combinatie IP adres en poort nummer niet meer opnieuw gebruikt worden. Je wil dus 100% zeker zijn dat de resource, ook in geval van een fout, vrijgegeven wordt!

Overweeg om het `using` statement te gebruiken dat automatisch resources vrijgeeft als ze niet meer nodige zijn.

8. Het `using` statement

Het `using` statement is een statement dat één of meerdere resource objecten instantieert. Een resource is een object dat **moet** gesloten worden nadat een applicatie afsluit of nadat het niet meer nodig is.

Het `using` statement garandeert dat resources objecten afgesloten worden als het `using` statemnet blok direct of indirect verlaten wordt. Elk object dat de interface `IDisposable` implementeert kunnen in het `using` statement gebruikt worden.

> The C# using statement defines a boundary for the object outside of which, the object is automatically destroyed. The using statement in C# is exited when the end of the "using" statement block or the execution exits the "using" statement block indirectly, for example - an exception is thrown.
>
> — www.c-sharpcorner.com/article/the-using-statement-in-C-Sharp

Onderstaand voorbeeld leest een eerste lijn uit een bestand. Een instantie BufferedReader wordt gebruikt om de data in te lezen. BufferedReader is een resource die gesloten moet worden als de applicatie eindigt:

```
1 public static String readFirstLineFromFile(String path) throws IOException {
2     try {
3         using (BufferedReader br = new BufferedReader(new FileReader(Paths.
  get(path)))) ①
4         {
5             return br.readLine();
6         }
7     } ②
8 }
```

① De resource gedeclareerd en geïnstantieerd in het `using` statement is de BufferedReader. Dit gebeurt tussen de haakjes, onmiddellijk na het `using` keyword. De klasse BufferedReader implementeert de interface `IDisposable`. De variabele br is een lokale variabele binnen het using-blok.

② Aangezien de instantie van BufferedReader gedeclareerd wordt in het `using` statement zal het gesloten worden ongeacht het try statement normaal eindigt of er een exception optreedt (bv. omdat de methode `br.readLine` een exceptie gooit). Hiervoor hoef je zelf geen actie meer te ondernemen.

Analoog zou code zonder `using` er als volgt uitzien. Het finally blok kan echter per ongeluk vergeten worden:

```
1 static String readFirstLineFromFileWithFinallyblok(String path) {
2     BufferedReader br = new BufferedReader(new FileReader(path));
3     try {
4         return br.readLine();
5     } finally {
6         if (br != null) br.close();
7     }
8 }
```

9. Soorten Excepties

9.1. Groeperen en differentiëren van exception types

Elke exceptie die binnen een applicatie gegooid kan worden is een instantie van een klasse. Het groeperen of categoriseren van exceptions kan dus eenvoudig op basis van een klasse hiërarchie.

Elke exception klasse erft direct of indirect van de klasse `Exception`, dewelke zelf een subklasse is van Object. Zo ontstaat er een overervingshiërarchie. Je kan deze boomstructuur zelf ook nog uitbreiden met eigen ontworpen Exception-klassen. Het mechanisme om fouten af te handelen werkt voor alle Exception-objecten

9.2. Zelf een exception klasse definiëren

Een eigen exception klasse zal altijd erven van de bestaande Exception klasse.

Het is de gewoonte om de naam van die nieuwe klasse te laten eindigen op Exception, vb EmailException

Een exception klasse heeft typisch 4 of minstens 2 constructoren. In de body van deze constructoren roepen we via base() de overeenkomstige constructor van de superklasse aan:

```
public class EmailException : Exception
{
    public EmailException() :
        base("Algemene foutboodschap")
    {
    }
    public EmailException(string message) :
        base(message)
    {
    }
    public EmailException(string message, Exception inner) :
        base(message, inner)
    {
    }
    protected EmailException(
      System.Runtime.Serialization.SerializationInfo info,
      System.Runtime.Serialization.StreamingContext context) :
        base(info, context)
    {
    }
}
```

Via een snippet kan je zelf eenvoudig het geraamte van een nieuwe exception klasse genereren.

10. Excepties in een ketting

Een ketting van excepties ontstaat als een exceptie die opgevangen wordt resulteert in een nieuwe exceptie. De eerste is oorzaak van de tweede.

Soms is het handig om te weten welke exceptie een andere exceptie tot gevolg heeft. Dit kan bereikt worden door een ketting van excepties te maken. Bij het gooien van een exceptie kan de exceptie die de oorzaak was als argument meegegeven worden. Je vangt een exceptie op, voegt wat informatie toe en gooit de originele exceptie samen met de toegevoegde informatie verder door.

```
1 try {
2
3 } catch (IOException e) {
4     throw new SampleException("Other IOException", e); ①
5 }
```

① Als een `IOException` opgevangen wordt, wordt een nieuwe `SampleException` instantie gemaakt met als oorzaak de originele exceptie. Deze ketting van excepties wordt gegooid om op een hoger niveau afgehandeld te worden.

11. Toegang tot de *Stack Trace* informatie

De 'stack trace', vervat in een Exception object, bevat informatie over de executie geschiedenis van een thread. Deze omvat o.a. de namen van klassen en methoden die werden aangeroepen tot het punt waar de exceptie optrad.

Zo een stack trace is zeer handig bij het debuggen, waar zeker voordeel uit gehaald kan worden indien een exceptie wordt gegooid (waar trad de exceptie op!). De stack trace die bij een exceptie hoort geeft o.a. aan waar (in welke file en op welke regel) in de source de exceptie precies optrad.

```
1 catch (Exception cause) {
2     Console.Write(cause.StackTrace); ①
3 }
```

① Toont aan hoe je een stack trace kan opvragen o.b.v. een exceptie object.

12. Test jezelf

- Is onderstaande snippet geldige code?

```
try {

} finally {

}
```

- Welke exceptie types zal volgende exception handler opvangen: geen enkele, exact één of meerdere (zo ja, welke dan…)?

```
catch (Exception e) {

}
```

- Waarom implementeer je deze exception handler beter niet? Wat loopt er mis? Zal deze code compileren?

```
try {

} catch (Exception e) {

} catch (ArithmeticException a) {

}
```