HO
GENT

OOSDII

Exception handling

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
- Try-with-resources statement
- Soorten excepties
- 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".

```java
public class DivideByZeroNoExceptionHandling {

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

       public static void main(String[] args)
       {
          Scanner scanner = new Scanner(System.in);

          System.out.print("Geef een integere waarde voor de teller: ");
          int teller = scanner.nextInt();
          System.out.print("Geef een integere waarde voor de noemer: ");
          int noemer = scanner.nextInt();

          int quotient = berekenQuotient(teller, noemer);
          System.out.printf(
             "%nResultaat: %d / %d = %d%n", teller, noemer, quotient);

          scanner.close();
       }

    }
```

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.

```
run:
Geef een integere waarde voor de teller: 12
Geef een integere waarde voor de noemer: 0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at main.DivideByZeroNoExceptionHandling.berekenQuotient(DivideByZeroNoExceptionHandling.java:9)
        at main.DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:21)
Java Result: 1
BUILD SUCCESSFUL (total time: 4 seconds)
```

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

- Soort fout wordt aangegeven (gele kleur)
- Er wordt ook aangegeven waar de fout zich voordoet in de code (oranje 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.java:9*). 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?

```
run:
Geef een integere waarde voor de teller: 12
Geef een integere waarde voor de noemer: hallo
Exception in thread "main" java.util.InputMismatchException
        at java.util.Scanner.throwFor(Scanner.java:864)
        at java.util.Scanner.next(Scanner.java:1485)
        at java.util.Scanner.nextInt(Scanner.java:2117)
        at java.util.Scanner.nextInt(Scanner.java:2076)
        at main.DivideByZeroNoExceptionHandling.main(DivideByZeroNoExceptionHandling.java:19)
Java Result: 1
BUILD SUCCESSFUL (total time: 6 seconds)
```

- Via het scanner-object willen we een geheel getal inlezen, maar er wordt tekst ingegeven.
- Ook deze fout onderbreekt het programma bruusk.

In Eclipse kan je klikken op “bestandsnaam:regelnummer” en dan spring je naar die plaats in de code.

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 (mijnString == null || mijnString.isBlank()) {
2         throw new IllegalArgumentException();
3     }
```

- de methode of constructor maakt een instantie van een exception object een 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)

- 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`.

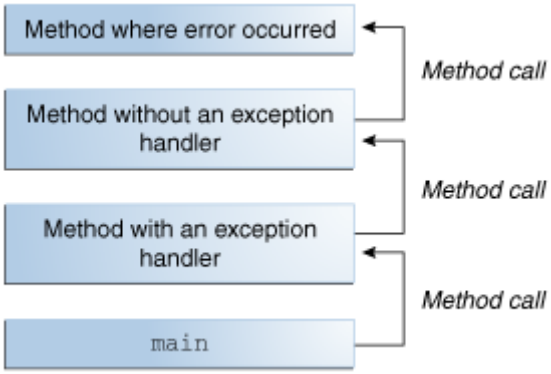
Indien de exceptie wordt gegooid vanuit een methode, zal deze methode geen return waarde teruggeven.

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 zal geen object gecreëerd zijn.

4.2. De gegooide exception verwerken

Nadat een methode een exception object wierp zal het runtime systeem *iets* proberen zoeken die 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 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.

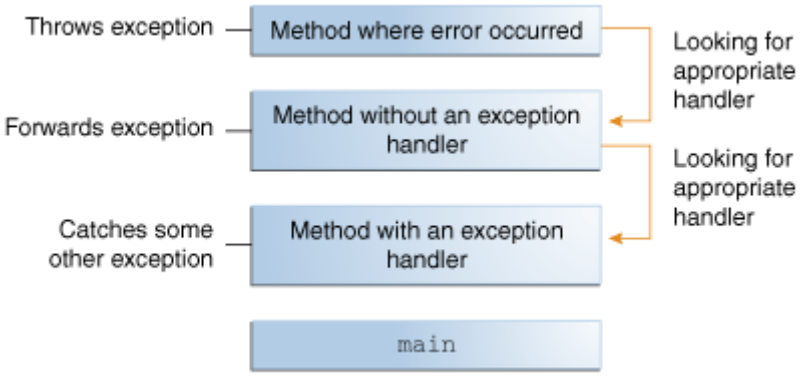
4.3. Op zoek doorheen de 'call stack'

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 code 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 gegeven 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 een 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.

Voor een bepaalde soort exceptions (*checked exceptions*) zal de compiler zelfs nagaan of deze wordt afgehandeld. Indien niet krijg je bij het compileren een foutboodschap. De compiler gaat ervan uit dat het zinloos is om een fout te detecteren en te gooien zonder ze achteraf op te vangen en af te handelen.

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 package geschreven door iemand anders zoals de packages die meekomen met het Java platform.

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

6.1. Het `throw` statement

Iedere methode gebruikt 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 Throwable.

```
1 throw someThrowableObject;
```

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 }
```

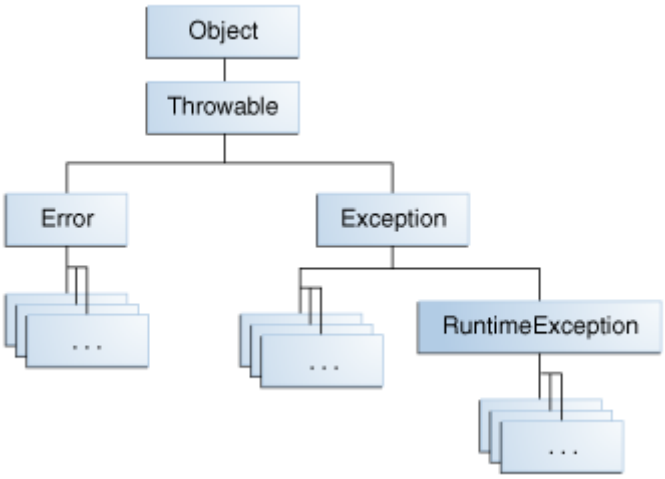
① De pop methode gaat na of er elementen zijn op de stack. Als deze leeg blijkt te zijn (als size == 0), dan wordt een instantie van EmptyStackExceptions aangemaakt (`new EmptyStackException`) en deze wordt gegooid.

Merk op dat de declaratie van de pop methode geen *throws* clausule bevat. EmptyStackException is geen checked exception, dus pop is niet verplicht om te declareren dat deze exceptie kan gegooid worden.

6.2. Throwable klasse en subklassen

Een eigen klasse hierarchy voor excepties kan opgebouwd worden, 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 Throwable en enkele belangrijke subklassen.



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 je best afhandelt!

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 bloks ... 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 Throwable. De handler verwijst ernaar d.m.v. de naam van de klasse.

② Het 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.

Let op voor polymorfisme! De eerste exception handler waarvan het type van de excpetie 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.

7.2.1. Eén Exception handler voor meerdere types

Een try-blok kan aan meerdere catch-blokken gekoppeld worden en één enkel catch-blok kan meerdere exceptie types afhandelen!

Deze laatste mogelijkheid laat toe om code te hergebruiken of om programmeurs te weerhouden een te algemeen type exception op te vangen.

```
catch (IOException|SQLException ex) { ①
    logger.log(ex);
    throw ex;
}
```

① In de catch clausule worden verschillende type exceptions vermeld, gescheiden door een verticale lijn (`|`)

7.3. The 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 de PrintWriter `out` wordt gesloten.

Het finally blok is, op het try-with-resources 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 try-with-resources 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.

Overweeg om het try-with-resources statement te gebruiken, die automatisch resources vrijgeeft als ze niet meer nodige zijn.

8. The try-with-resources Statement

Het try-with-resources statement is een try statement dat één of meerdere resources declareert. Een resource is een object dat **moet** gesloten worden nadat een applicatie afsluit of nadat het niet meer nodig is.

Het try-with-resources statement garandeert dat resources afgesloten worden op het einde van het try statement. Elk object dat de interface `java.lang.AutoCloseable` implementeert, inclusief alle objecten die de interface `java.io.Closeable` implementeren, kunnen in het try-with-resources statement gebruikt worden.

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             BufferedReader br = new BufferedReader(new FileReader(Paths.get(path)))
  ①
4         ) {
5         return br.readLine();
6     } ②
7 }
```

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

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

Analoog zou code zonder try-with-resources er als volgt uitzien. Het finally blok kan per ongeluk vergeten worden:

```
static String readFirstLineFromFileWithFinallyblok(String path)
                                                    throws IOException {
    BufferedReader br = new BufferedReader(new FileReader(path));
    try {
        return br.readLine();
    } finally {
        if (br != null) br.close();
    }
}
```

Eén of meer resources mogen gedeclareerd worden in het try-with-resources statement.

```
    try (
        ZipFile zf = new ZipFile(zipFileName); ①
        BufferedWriter writer = BufferedWriter(outputFilePath, charset)
    ) {

    }
}
```

① Het try-with-resources statement bevat twee declaraties, gescheiden door een ';': ZipFile en BufferedWriter Als het codeblok er net onder eindigt, normaal of met een exceptie, zal de close methode van zowel BufferWriter als ZipFile automatisch aangeroepen worden.

De close methode van resources worden aangeroepen in omgekeerde volgorde van hun declaratie/creatie!

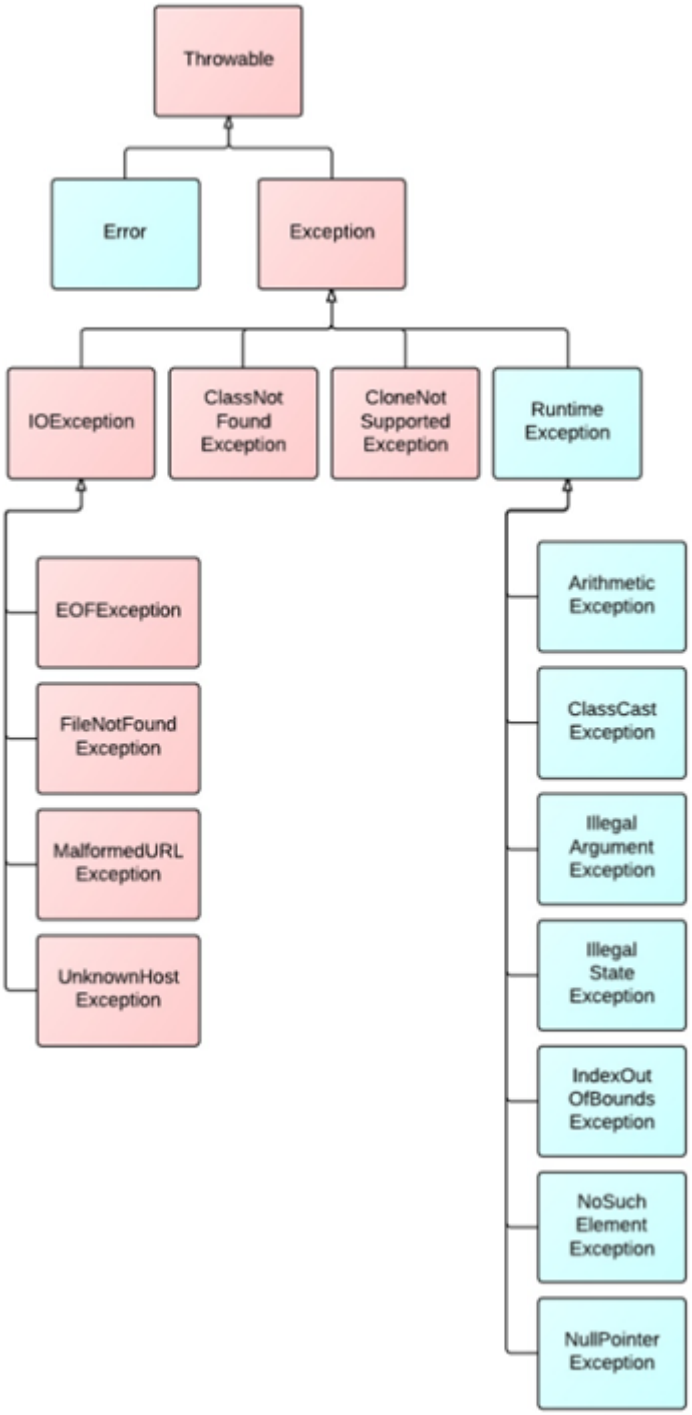
Een try-with-resources statement kan net zoals een gewone try-catch clausule een of meer catch blokken en een finally blok hebben. Elke catch of finally blok wordt uitgevoerd nadat de gedeclareerde resources werden gesloten.

9. Soorten Excepties

9.1. Groeperen en differentiëren van error 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 Java Exception klasse erft direct of indirect van de klasse Exception, dewelke zelf een subklasse is van Throwable. 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 Throwable-objecten



Deze hiërarchie legt ook drie soorten Throwable objecten vast, op basis van de subklassen: Errors, Checked exceptions en Unchecked exceptions.

9.2. Errors

Dit zijn events waarop de applicatie geen vat geeft, waarop niet kan geanticipeerd worden en waarvan de applicatie zich ook niet kan herstellen.

9.3. Runtime exceptions

Dit zijn events binnen de applicatie, waarop meestal niet kan geanticipeerd worden en waarvan de applicatie zich niet kan herstellen. Meestal aangeduid als software bugs, logische fouten of het

onjuist gebruik van een API.

Runtime exceptions hoeven niet omringd te worden door een try-catch clausule, een methode hoeft ook niet aan te geven dat het een RuntimeException kan gooien.

Errors en runtime exceptions zijn gekend als **unchecked exceptions**.

Voorbeelden zijn ArrayIndexOutOfBoundsException, ArithmeticException…

Ook unchecked exceptions kunnen afgehandeld worden.

9.4. Checked exceptions

Op sommige uitzonderlijke gebeurtenissen in een applicatie kan geanticipeerd worden. Deze gebeurtenissen zouden moeten opgevangen worden, waarbij de gebruiker een boodschap kan ontvangen.

Om te garanderen dat een exception wordt opgevangen, worden *checked* exceptions gebruikt.

Een *checked exception* **moet** omringd worden door een try clausulel en **moet** een geschikte catch clausule bevatten. Indien dit niet het geval is moet de methode aangeven in zijn declaratie dat het deze exception kan gooien (*throws*). De verantwoordelijkheid voor het afhandelen van de exception wordt zo op een hoger niveau gebracht.

Alle exceptions zijn checked exceptions, behalve deze van het type Error en RuntimeException of hun subklassen.

9.5. Checked vs Unchecked exceptions

De compiler maakt wel degelijk een verschil tussen deze twee types Exception: een checked Exception moet afgehandeld worden of moet expliciet gepropageerd worden (catch-or-declare requirement). Bij compilatie wordt dit gecontroleerd: de compiler waarschuwt je dus dat een mogelijke fout niet wordt afgehandeld (→ je zegt dat er een fout kan optreden, maar je handelt ze niet af…)

Bij een unchecked exception treedt deze controle tijdens compilatie niet op!

9.6. Specifiëren dat een methode een Exception gooit

Soms is het handig in code om exceptions direct op te vangen, binnen dezelfde methode. Vaak is het echter handiger om exceptions op te vangen op een hoger niveau, hogerop de call stack.

Als een methode zelf niet de **checked exception** opvangt die mogelijk kunnen gegooid worden, dan moet deze methode aangeven in zijn declaratie dat het deze exception zou kunnen gooien! Indien dit niet gedeclareerd wordt, treedt een compilatie fout op.

Om te specifiëren dat een methode een checked exception kan gooien, maar deze zelf niet afhandelt, dient een throws clausule toegevoegd te worden aan de declaratie van de methode. Deze clausule omvat het `throws` keyword, gevolgd door een lijst van excepties gescheiden door een komma.

```
1 public void writeList() throws IOException, IndexOutOfBoundsException { ①
```

① De clausule volgt na de naam en argumentenlijst en voor de eerst accolade die de scope van de methode start.

10. Zelf een Exception klasse definiëren

Een eigen Exception klasse zal altijd erven van een bestaande Exception klasse.

- Als het de bedoeling is dat het programma zich kan herstellen na de Exception, maak dan een checked Exception (m.a.w. erf van Exception, maar niet van RuntimeException)
- Als het de bedoeling is dat de Exception kan genegeerd worden, maak dan een unchecked Exception en erf van RuntimeException.
- 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 super() de overeenkomstige constructor van de superklasse aan:

```java
package exceptions;

public class EmailException extends RuntimeException{

    public EmailException(){
        //super();  --> wordt impliciet aangeroepen
        // OF
        // algemene foutmelding
        super("Er loopt iets fout met het e-mailadres.");
    }

    public EmailException(String message){
        super(message);
    }

    public EmailException(String message, Throwable cause){
        super(message,cause);
    }

    public EmailException(Throwable cause){
        super(cause);
    }

}
```

11. 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.

```
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.

```java
public class UsingChainedExceptions
{
   public static void main(String[] args)
   {
      try
      {
         method1();
      }
      catch (Exception exception) // exceptions thrown from method1
      {
         exception.printStackTrace();
      }
   }

   // call method2; throw exceptions back to main
   public static void method1() throws Exception
   {
      try
      {
         method2();
      }
      catch (Exception exception) // exception thrown from method2
      {
         throw new Exception("Exception thrown in method1", exception);
      }
   } // end method method1
```

```
java.lang.Exception: Exception thrown in method1
        at main.UsingChainedExceptions.method1(UsingChainedExceptions.java:26)
        at main.UsingChainedExceptions.main(UsingChainedExceptions.java:9)
Caused by: java.lang.Exception: Exception thrown in method2
        at main.UsingChainedExceptions.method2(UsingChainedExceptions.java:39)
        at main.UsingChainedExceptions.method1(UsingChainedExceptions.java:22)
        ... 1 more
Caused by: java.lang.Exception: Exception thrown in method3
        at main.UsingChainedExceptions.method3(UsingChainedExceptions.java:46)
        at main.UsingChainedExceptions.method2(UsingChainedExceptions.java:35)
        ... 2 more
```

Via de methode getCause() uit de klasse Throwable kan je terugvinden welke subklasse van Throwable initieel de Exception veroorzaakt heeft.

12. Toegang tot de *Stack Trace* informatie

Een '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 handige 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     StackTraceElement elements[] = cause.getStackTrace(); ①
3     for (int i = 0, n = elements.length; i < n; i++) {
4         System.err.println(elements[i].getFileName()
5             + ":" + elements[i].getLineNumber()
6             + ">> "
7             + elements[i].getMethodName() + "()");
8     }
9 }
```

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

13. 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) {

}
```