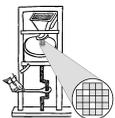


Kapitel 15

Exceptions





Exceptions

In der Abarbeitung eines Methodenaufrufs kann die Methode immer potentiell auf Probleme stoßen, mit denen sie selbst nicht umzugehen weiß.

Exceptions geben einer Methode die Möglichkeit,

- den Methodenaufruf in einem solchen Fall umgehend, aber dennoch **kontrolliert** zu beenden
- und das Problem damit an die **aufrufende** Methode zu delegieren.

Idee dahinter:

- Vielleicht weiß ja die aufrufende Methode besser mit dem Problem umzugehen.
- Soll die sich eben damit herumschlagen.





Beispiel: Einfacher Taschenrechner

Betrachte ein Programm zur Auswertung beliebig komplexer mathematischer Ausdrücke mit den vier Grundrechenarten und verschiedenen Arten von Klammern.

Beispiel:

- $3 + 2 * (6-4) / (7-5)$

Eine interaktive Methode `readExpression` in diesem Programm sei dafür zuständig, einen solchen Ausdruck von der Tastatur einzulesen und den Wert des Ausdrucks auf dem Bildschirm zu präsentieren.

Die Methode `readExpression` ruft dann zweckmäßigerweise eine andere, separate Methode `parseExpression` auf, die

- diesen Ausdruck (z.B. als `String`-Objekt) als Parameter erhält,
- den Wert des darin gespeicherten Ausdrucks berechnet und
- das Ergebnis an `readExpression` zwecks Präsentation auf dem Bildschirm zurückreicht.



Fehlerbehandlung

Der eingegebene mathematische **Ausdruck** kann auch **fehlerhaft** sein.

- Öffnende und schließende Klammern passen im eingegebenen Ausdruck vielleicht nicht zusammen.
- An irgendeiner Stelle wird in einem kleinen Teilausdruck des Gesamtausdrucks durch Null geteilt.
- usw.

Bei der Berechnung des Wertes des Ausdrucks in `parseExpression` fallen solche Fehler automatisch auf.

- Es ist also zweckmäßig, wenn die Suche nach solchen Fehlern nicht von `readExpression`, sondern quasi nebenher von `parseExpression` erledigt wird.

Nur:

- Was soll `parseExpression` mit einem gefundenen Fehler machen?
- Verschiedene Aufrufe könnten verschiedene Fehlerbehandlungen erfordern
- Aber derselbe Fehler-Typ kann an verschiedenen Stellen im Programm auftreten und immer gleich behandelt werden





Fehlerbehandlung

Die Methode `parseExpression` ist ja eigentlich nur ein "blinder Rechenknecht" und im Grunde für die Behandlung von Benutzerfehlern inkompetent.

- Kompetent dafür ist eher die benutzerorientierte Methode `readExpression`

Zum Beispiel könnte `readExpression` dem Benutzer

- eine informative Fehlermeldung (genaue Fehlerstelle, Art des Fehlers) geben,
- ein Fenster zur unmittelbaren Nachkorrektur des Ausdrucks aufmachen
- konkrete Vorschläge zur Korrektur machen.

Welche dieser Optionen in einem konkreten Anwendungskontext sinnvoll wäre, kann nur `readExpression`, nicht aber `parseExpression` wissen.





Fehlerbehandlung

Die Methode `parseExpression` berechnet den Ausdruck und achtet dabei nebenher auf Fehler.

- Im Fehlerfalle stellt sie ihre weitere Arbeit ein und reicht eine Fehlerdiagnostik (genaue Fehlerstelle, Art des Fehlers o. ä.) zurück an `readExpression`.

Die Methode `readExpression` behandelt nun den Fehler bspw. auf eine der auf der letzten Folie angedeuteten Arten.

- Erhält `readExpression` durch Korrekturen des Benutzers einen neuen, korrigierten Ausdruck, könnte `parseExpression` zum Beispiel damit erneut aufgerufen werden.
- Bis der Benutzer es fertig bringt, den gewünschten Ausdruck korrekt hinzuschreiben (oder entnervt aufgibt).





Fehlerbehandlung in Java

Bei der Programmierung von `readExpression` und `parseExpression` ist in den Quelltexten beider Methoden "vereinbart" worden, dass

- `parseExpression` in gewissen Fällen **eine *Exception* wirft**,
- die dann von `readExpression` **gefangen werden muss**.

Wenn `parseExpression` auf einen Fehler im mathematischen Ausdruck stößt, dann

- **wirft** `parseExpression` **eine *Exception***,
- was zugleich bedeutet, dass die Abarbeitung von `parseExpression` **sofort** (und ohne Rückgabewert) beendet wird.

Die aufrufende Methode `readExpression` soll darauf mit einer (wie auch immer gearteten) Fehlerbehandlung reagieren.



Syntaktische Umsetzung

```
public double parseExpression ( String ausdruck )
    throws Exception
{
    double wertDesAusdrucks;

    ...

    if ( divisor == 0 )
    {
        Exception exc
            = new Exception( "Fehler im Ausdruck!" );
        throw exc;
    }

    ...

    return wertDesAusdrucks;
}
```

hier wird vereinbart, daß
parseExpression
eine Exception vom Typ
Exception werfen kann

Hier wird das Exception-
Objekt definiert..

... das hier geworfen
wird, d.h. der Fehler
wird an die aufrufende
Methode gemeldet.



return und throw

Solange kein Fehler auftritt, wird in der Variablen `wertDesAusdrucks` nach und nach der mathematische Wert des Strings `ausdruck` zusammengebastelt.

- Falls in diesem ganzen Arbeitsgang **kein Fehler** aufgetreten ist, wird das Endergebnis dann mit **return** ausgegeben.
- Falls doch **ein Fehler** entdeckt wird, wird die Abarbeitung von `parseExpression` durch das **throw**-Konstrukt sofort beendet.

Beispiel:

Im Beispiel auf der letzten Folie wurde nur eine einzige Stelle exemplarisch gezeigt, an der ein möglicher Fehler abgetestet wird:

- ob der zweite Operand (der *Divisor*) einer Division gleich Null ist.
- Falls so eine Division durch Null auftritt, wird durch `throw` ein neu kreierte Objekt des Typs `Exception` als Exception geworfen.



return und throw

In gewisser Weise ist `throw` daher so etwas wie eine Variation von `return`:

- Der Methodenaufruf wird beendet,
- und es wird ein Objekt an die aufrufende Methode zurückgereicht.

Allerdings hat `throw` für die aufrufende Methode `readExpression` völlig andere syntaktische und semantische Konsequenzen als `return`.

- Dazu mehr auf den nächsten Folien.

Kleiner weiterer Unterschied:

- Nicht jedes `return` muss ein Objekt zurückliefern.
→ `return` liefert genau dann ein Objekt zurück, wenn die Methode nicht `void` ist.
- Durch `throw` muss hingegen grundsätzlich immer eine Exception geworfen werden.





Beispiel (Fs.)

```
public void readExpression ( ... )
{
    String ausdruck;
    double wertDesAusdrucks;
    ...

    try { ←
        wertDesAusdrucks = parseExpression( ausdruck );
        System.out.println ( wertDesAusdrucks );
    }

    catch ( Exception exc ) {
        System.out.println ( exc.getMessage() );
    }

    ...
}
```

`try` gibt an, dass hier ein Block von Anweisungen folgt, der möglicherweise zu einer Exception führt.

normale Ausführung, d.h. `parseExpression` wird ohne Fehler beendet

`catch` fängt eine Exception auf (das erzeugte Objekt wird im Argument übergeben) und führt im darauffolgenden Block die Fehlerbehandlung durch.



try

Eine Exception, die von einer aufgerufenen Methode wie `parseExpression` **geworfen** wird, muss von einer sie aufrufenden Methode wie `readExpression` mit einem solchen `try-catch`-Konstrukt behandelt ("gefangen") werden.

- Der Aufruf der potentiell werfenden Methode muss im `try`-Block stehen.

Wenn durch `parseExpression` **keine Exception** geworfen wird, dann wird der `try`-Block normal zu Ende abgearbeitet (also `wertDesAusdrucks` ausgegeben)

- der `catch`-Block wird übersprungen,
- d.h. nach Beendigung des `try`-Blocks wird mit der nächsten Anweisung, die regulär nach dem `catch`-Block folgt, fortgefahren.



catch

Wenn durch `parseExpression` **eine Exception** geworfen wird, wird nicht nur der Aufruf von `parseExpression`, sondern auch der ganze `try`-Block in `readExpression` sofort beendet.

→ Die Ausgabe des Werts von `wertDesAusdrucks` findet nicht statt (würde ohnehin keinen sinnvollen Wert ausgeben).

Statt dessen wird die Abarbeitung des `catch`-Blocks begonnen.

→ Der `catch`-Block ist für die Anweisungen zur Fehlerbehandlung da.

- Der `catch`-Block besitzt einen Kopf, in dem der Programmierer von `readExpression` für das geworfene Exception-Objekt (genauer: dem Verweis darauf) einen Identifier als Namen vergibt.

→ Im Beispiel hat der Programmierer von `readExpression` den Namen `exc` gewählt.

Nach Abarbeitung des `catch`-Blocks geht es normal mit der nachfolgenden Anweisung weiter.





Bemerkungen zur Syntax

Nach `try` bzw. `catch` müssen geschweifte Klammern kommen, auch wenn nur eine einzelne Anweisung folgt.

Frage:

- Das `try-catch`-Konstrukt wurde eigentlich ja nur angewandt, um die Exception von `parseExpression` abzufangen.
- Wieso steht dann die Schreibausgabe von `wertDesAusdrucks` ebenfalls im `try`-Block?

Antwort:

- Diese Schreibausgabe macht nur Sinn, wenn keine Exception geworfen wurde.
- Sie sollte daher an einer Stelle stehen, die nach Wurf einer Exception nicht erreicht wird.
 - also nicht nach dem `try/catch`-Block





Klasse (`java.lang.`) `Exception`

Ein Objekt der Klasse `Exception` besitzt eine `String`-Variable als **Datenkomponente**.

- Das `Exception`-Objekt dient damit praktisch als **Bote**, der diesen `String` als Botschaft **von der aufgerufenen zur aufrufenden Methode** trägt.
- In der Methode `parseExpression` wird die Botschaft dem Boten schon bei seiner Erzeugung mitgegeben:

```
new Exception( "Fehler im Ausdruck!" )
```

Die Klasse `Exception` hat eine Methode `getMessage`, die diese Botschaft zurückliefert.

- Mit dieser Methode kann dann in `parseExpression` auf die Botschaft von `exc` zugegriffen werden.



Unterklassen von `Exception`

Wie immer, kann man von der Basisklasse `Exception` auch andere Klassen ableiten.

- Ein Objekt einer solchen abgeleiteten Klasse kann also wie üblich anstelle eines Objekts der Basisklasse (z.B. nach `throw`) verwendet werden.

Es gibt vordefinierte Klassen, die

- für spezifische Anwendungsfälle konzipiert waren
- in der Regel weitere Möglichkeiten enthalten, Botschaften in spezifischerer Form als nur durch eine einfache Zeichenkette zu kodieren.

Man kann natürlich auch neue Unterklassen von `Exception` selbst definieren





Mögliche Lösung im Beispiel

Definiere eine eigene Klasse `FehlerInAusdruckException` als Erweiterung von `Exception`.

Als Datenkomponenten werden zum Beispiel eingerichtet:

- Eine Zahl zur Lokalisierung des genauen Fehlerpunktes als Index in der Zeichenkette.
- Eine weitere Zahl als Fehlerdiagnostik (z.B. 0=fehlende schließende Klammer, 1=schließende Klammer zuviel, 2=...)
- Falls nicht nur ein, sondern alle Fehler im Ausdruck zugleich zurückgeliefert werden sollen, bieten sich bspw. zwei Arrays von solchen Zahlen als Botschaft an.

Mit einer solchen von `parseExpression` mittels `Exception` empfangenen Botschaft kann `readExpression` recht informative Fehlermeldungen an den Benutzer ausgeben.





Beispiel

```
public class FehlerInAusdruckException extends Exception {
    private int fehlerStelle;
    private int fehlerArt;

    public FehlerInAusdruckException( int fehlerStelle,
                                      int fehlerArt ) {
        this.fehlerStelle = fehlerStelle;
        this.fehlerArt = fehlerArt;
    }

    public String getMessage () {
        String fehlerBeschreibung;
        if ( fehlerArt == 0 )
            fehlerBeschreibung = "Schliessende Klammer fehlt";
        else if ( fehlerArt == 1 )
            fehlerBeschreibung = "Schliessende Klammer zuviel";
        else ...

        return fehlerBeschreibung + " an Index "
            + fehlerStelle + "!";
    }
}
```

Interne Datenkomponenten

Konstruktor für die Exception

Implementierung von `getMessage`
zur Fehlerausgabe



Exceptions weiterreichen

Bisher wurde ein `try-catch`-Konstrukt für die von `parseExpression` potentiell geworfenen Exceptions eingerichtet.

Alternativ könnte `readExpression` aber auch

- diese Exceptions **nicht selbst abfangen** und bearbeiten,
- sondern durch ein eigenes `throws` im Methodenkopf **an die nächsthöhere Methode weiterreichen**.

Wenn nun `parseExpression` eine Exception wirft,

- dann wird nicht nur die Bearbeitung von `parseExpression` sofort abgebrochen,
- sondern auch die von `readExpression` und
- die Exception wird an die Methode, die `readExpression` aufgerufen hat, weitergereicht



Beispiel

```
public void readExpression ( ... )
    throws Exception
{
    String ausdruck;
    double wertDesAusdrucks;
    ...

    wertDesAusdrucks = parseExpression(ausdruck);
    System.out.println ( wertDesAusdrucks );
    ...
}
```

readExpression muß
mit throws Exception
deklariert werden...

...da parseExpression eine
Exception wirft, und diese
nicht mit einem try-catch-
Konstrukt aufgefangen wird.



Exceptions müssen irgendwo gefangen werden

Wenn man eine vordefinierte Methode überschreiben muß, dann wird erwartet, dass die Signatur der Startmethode exakt so ist wie erwartet.

- Insbesondere wird eigentlich immer eine leere `throws`-Liste erwartet.

Konsequenz:

- Exceptions können *nicht* beliebig lange mit `throws` in der Aufrufhierarchie weiter hochgereicht werden.
- Spätestens die überschriebene Methode muss die Exception fangen.
- Versucht diese Methode statt dessen, die Exception mittels `throws` weiterzureichen, entspricht die Signatur nicht mehr der Erwartung
- Der Compiler steigt sofort mit einer entsprechenden Fehlermeldung aus



Ausnahme: `main`

Die Einstiegsmethode `main` für die JVM kann auch mit einer `throws`-Liste deklariert werden

- In diesem Fall werden Exceptions von der `main`-Methode an den Interpreter weitergereicht und der produziert eine Fehlermeldung

Eine weitere Ausnahme stellen `RuntimeExceptions` dar (kommen später)



Ungefangene Exceptions

```
public class testExceptions {  
  
    public static void test () throws Exception  
    {  
        Exception e = new Exception("Fehler!");  
        throw e;  
    }  
  
    public static void main (String[] args){  
        test();  
    }  
}
```

Aufruf einer Methode, die eine Exception wirft, ohne daß die Exception gefangen oder weitergegeben wird

Fehler wird beim **Compilieren** erkannt:

```
$ javac testExceptions.java  
testExceptions.java:8: unreported exception  
java.lang.Exception; must be caught or declared to be thrown  
    test();  
    ^  
1 error
```





Ungefangene Exceptions

```
public class testExceptions {  
  
    public static void test () throws Exception  
    {  
        Exception e = new Exception("Fehler!");  
        throw e;  
    }  
  
    public static void main (String[] args)  
        throws Exception {  
        test();  
    }  
}
```

Hier wird deklariert, daß `main` eine Exception an die JVM weiterreichen kann

Ungefangene Exception wird beim **Interpretieren** gemeldet:

```
$ java testExceptions  
java.lang.Exception: Fehler!  
    at testExceptions.test(testExceptions.java:4)  
    at testExceptions.main(testExceptions.java:9)  
Exception in thread "main"
```





Mehrere Exceptions

Der Java-Compiler verlangt, daß jede Methode, die mit `throws` deklariert wird, nur verwendet werden kann

- in `try`-Block eines `try-catch` Konstrukts
- in einer Methode, die eine Exception derselben Klasse (oder einer Überklasse) wirft

Eine Methode kann daher auch mehrere Exceptions werfen!

- auch Exceptions verschiedener Typen
- Diese verschiedenen Typen werden in einer einzigen `throws`-Klausel (durch Kommas voneinander getrennt) aufgelistet.



Beispiel

```
class MyException1 extends Exception { ... }
class MyException2 extends Exception { ... }

...

void test ( int a, int b )
    throws MyException1, MyException2
{
    if ( a < b )
        throw new MyException1 (...);
    else
        throw new MyException2 (...);
}
```



Beispiel: Aufruf

```
try { test(2,3); }
    catch ( MyException1 exc ) ←
        { System.out.println ( "2 < 3" ); }
    catch ( MyException2 exc ) ←
        { System.out.println ( "2 >= 3" ); }
```

Erläuterungen:

Für jeden potentiell geworfenen **Exception-Typ** muss eine entsprechende `catch`-Klausel eingefügt werden.

Beachte:

- Es kommt dabei allein darauf an, welche **Exception-Typen** in der `throws`-Klausel deklariert sind.
- Selbst wenn (wie im Beispiel oben) klar ist, dass ein bestimmter **Exception-Typ** nie geworfen werden kann, muss die zugehörige `catch`-Klausel vorhanden sein.



Beispiel: Alternativer Aufruf

```
try { test(2,3); }
catch ( Exception exc ) { ←
    System.out.println( "2 < 3 oder 2 >= 3" );
}
```

MyException1 und MyException2 sind beides Unterklassen von Exception.

Erläuterungen:

- Es muss nicht unbedingt genau die Exception-Klasse gefangen werden, die geworfen wurde.
- Es kann auch eine beliebige Exception-Klasse gefangen werden, von der die geworfene Exception-Klasse direkt oder indirekt abgeleitet wurde.
 - Wie das Beispiel oben zeigt, kann sich dadurch die Zahl der notwendigen catch-Klausel verringern.
 - Im Extremfall, wenn Typ Exception selbst gefangen wird, reicht (wie im Beispiel oben) auch eine einzige catch-Klausel



Runtime Exceptions

Gewisse Typen von Exceptions müssen nicht abgefangen oder weitergereicht werden.

→ `java.lang.RuntimeException` (abgeleitet von `Exception` und alle direkt oder indirekt davon abgeleiteten Klassen).

Grund:

- Arithmetische Operationen, Arrayzugriffe usw. kommen in vielen Java-Quelltexten häufig vor.
- Müssten alle Exceptions, die sich aus Fehlern mit diesen Operationen ergeben, abgefangen werden, würde man die eigentliche Programmlogik hinter den vielen `trys` und `catchs` kaum noch finden.
- Alle diese Exceptions in der Aufrufhierarchie weiter hochreichen wäre auch keine Lösung (wohin am Ende?).





Beispiel 1

```
int i = 1;  
int j = 0;  
System.out.println ( i/j );
```

Ergebnis:

- Beim Kompilieren mit `javac` gibt es trotz der Division durch Null keinen Fehler.
- Beim Lauf des Programms gibt es Absturz mit Fehlermeldung:

```
java.lang.ArithmeticException: / by zero
```

Die Divisionsoperation kann also ebenfalls eine Exception werfen (wie eine Methode mit einer `throws`-Klausel)

- Der Compiler hat den Code aber augenscheinlich akzeptiert und übersetzt, obwohl die Exception weder gefangen noch weitergereicht wurde.



Beispiel 2

```
int[] A = new int [100];  
int i = 100;  
System.out.println ( A[i] );
```

Ergebnis:

- Praktisch identisch mit dem Ergebnis von der vorherigen Folie.
- Nur die Meldung beim Absturz lautet nun anders:

```
java.lang.ArrayIndexOutOfBoundsException: 100
```

Bemerkung:

Die Zusatzinformationen "100" in der Fehlermeldung oben und "/ by zero" auf der vorherigen Folie sind Beispiele dafür, dass in spezifischen Exception-Klassen weitere, spezifischere Zusatzinformationen enthalten sein können.



Runtime Exceptions

Ausnahmsweise wurde hier deshalb einmal beim Entwurf von Java die Entscheidung contra Ablaufsicherheit getroffen.

- Nur durch diese Sicherheitslücke sind Programmabstürze in Java überhaupt möglich.
- Die Grundidee dabei ist, daß Runtime Exception meistens auf Programmier-Fehler sind, deren Behandlung innerhalb des Programms ohnehin nicht sinnvoll ist.

