Exceptions genauer betrachtet

Exception bedeutet übersetzt "Ausnahme". Es handelt sich bei einer Exception also um ein Ereignis, das nur im Ausnahmefall eintreten sollte. Eine Methode meldet mit einer Exception, das sie ihre Arbeit nicht regulär ausführen konnte, sondern dass ein Fehler stattgefunden hat. Diese Art von Fehlermeldung führt allerdings immer dazu, dass die normale Programmausführung abgebrochen wird. Wird diese Fehlermeldung nicht bearbeitet, dann führt eine Exception zum Programmabbruch.

Es gibt prinzipiell zwei Arten von Exceptions:

- **checked Exceptions**: Wenn eine Methode eine checked Exception werfen kann, dann überprüft der **Compiler**, ob der Aufrufer die Exception auch in irgendeiner Form bearbeitet. Ist dies nicht der Fall, dann lässt sich das Programm gar nicht erst kompilieren.
- **unchecked Exceptions**: Diese werden auch Runtime-Exceptions genannt. Wenn eine Methode eine unchecked Exception werfen kann, dann kann man die Exception auch ignorieren, aber nur, wenn man sich des Risikos bewusst ist.
  Der Compiler lässt es zu, dass eine Methode mit einer unchecked Exception aufgerufen wird, ohne dass man sie berücksichtigt. Sehr oft wird mit einer unchecked Exception auf einen Programmierfehler aufmerksam gemacht und Programmierfehler sollten vermieden werden anstatt dass man das Symptom (die Exception) bearbeitet.
  Die häufigsten unchecked Exceptions spiegeln das wider:
  - `NullPointerException`: eine Referenzvariable wird für einen Methodenaufruf verwendet, obwohl diese Referenzvariable noch nicht auf ein Objekt verweist (dann ist nämlich der Wert null in der Referenzvariablen gespeichert).
  - `StringIndexOutOfBoundsException`: hier wird auf den Index eines Zeichens in einem String zugegriffen, der nicht existiert, z.B. ein Index von -1 oder ein Index von 10 in einem Text, der nur aus 10 Zeichen besteht.

In seltenen Fällen können auch Operationen unchecked Exceptions auslösen, z.B. wird bei einer Division durch 0 die `ArithmeticException` ausgelöst.

Fehlerhafte Eingaben

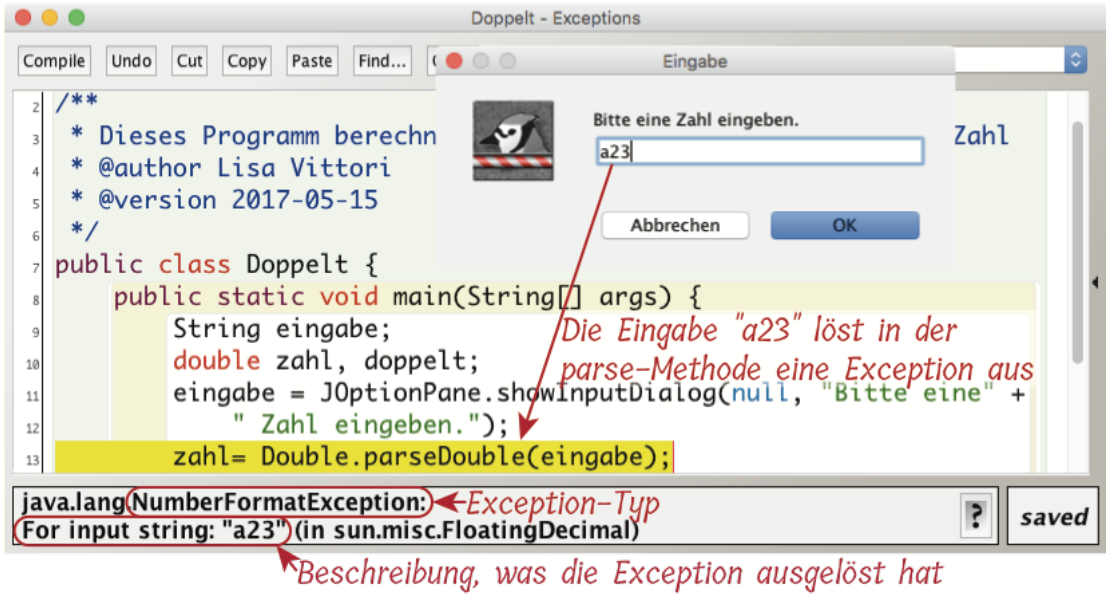
Ein Fall, wo Exceptions nur schwer vermieden werden können, sind fehlerhafte Eingaben. Diese verursachen beim umwandeln in andere Datentypen je nach verwendeter Eingabe entweder die `NumberFormatException` oder die `InputMismatchException` – beide unchecked Exceptions:

```
import javax.swing.JOptionPane;
/** ... */
public class DoppeltScanner {
    public static void main(String[] args) {
        String eingabe;
        double zahl, doppelt;
        eingabe = JOptionPane.showInputDialog(null, "Bitte eine" +
            " Zahl eingeben.");
        zahl= Double.parseDouble(eingabe);
        doppelt = zahl * 2;
        System.out.println("Das Doppelte von " + zahl + " ist " +
            doppelt);
    }
}
```

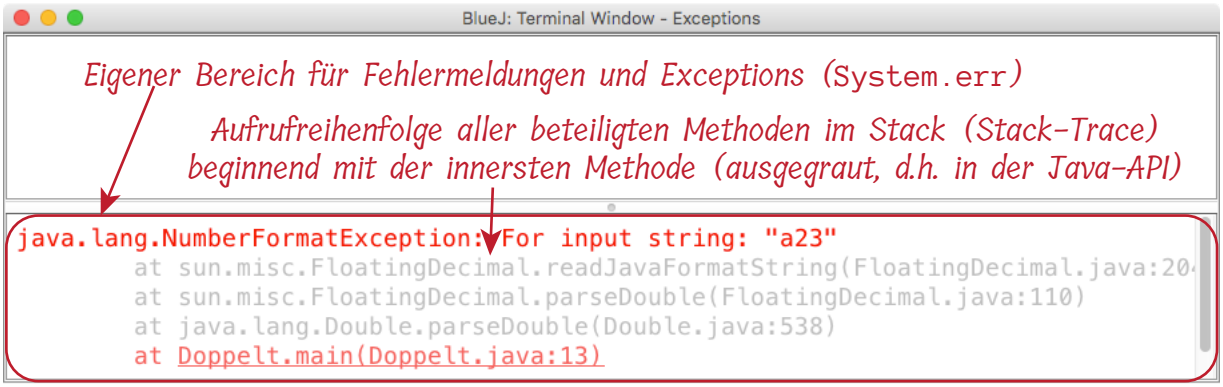
Dieser Methodenaufruf ist „gefährlich", er kann die NumberFormatException auslösen.

Obwohl in der markierten Methode eine Exception auftreten könnte, lässt sich das Programm ohne Weiteres kompilieren – das Merkmal einer unchecked Exception. Wenn alles richtig eingegeben wurde tritt die Exception auch nicht auf. Erst wenn eine Eingabe gewählt wird, die sich nicht in eine double-Zahl parsen lässt, wird die `NumberFormatException` geworfen. Das Programm bricht an dieser Stelle ab.

In BlueJ sieht diese geworfene Exception (die `NumberFormatException`) dann so aus:



Die Code-Zeile, bei der das Programm abgebrochen wurde wird markiert und die Exception-Art wird in der Statusleiste angegeben. Auch im BlueJ-Ausgabe-Fenster wird die Exception angezeigt:



Dabei wird auch ein **Stack-Trace** in einem eigenen Ausgabebereich für Fehler (`System.err`) angezeigt. Dieser zeigt alle Methoden an, die beim Werfen der Exception beteiligt waren. Die oberste Methode ist dabei jene Methode, die ursprünglich die Exception geworfen hat. Die nachfolgenden Methoden haben diese Exception dann einfach solange weitergegeben, bis sie in der `main`-Methode einen Programmabbruch ausgelöst hat. Diese anderern Methoden sind grau dargestellt, da sie Methoden der Java-API sind, deren Quellcode hier nicht zur Verfügung steht.

Man kann also anhand dieses Stack-Traces erkennen, dass unsere `main`-Methode in der Klasse `Doppelt` die Methode `parseDouble` der Klasse `Double` aufgerufen hat. Diese hat wiederum die Methode `parseDouble` in der Klasse `FloatingDecimal` aufgerufen, die wiederum die Methode `readJavaFormatString` in der gleichen Klasse aufgerufen hat. Die letzte Methode ist jene Methode in der die Exception eigentlich aufgetreten ist. Falls man hier einen Programmierfehler sucht, sollte man dazu in der letzten Methode vor den Java-API-Methoden beginnen. Der Stack-Trace bleibt dabei solange in der Anzeige bis er über die Optionen (Befehl Clear: `Strg+K`) gelöscht wird.

Was ist eigentlich eine Exception?

Eine Exception ist ein Objekt, das durch einen speziellen Mechanismus in Java "geworfen" werden kann und damit den üblichen Programmablauf umgeht. Man kann sich eine Exception daher auch als eine Art Kapsel vorstellen, in der alle Informationen über die Exception gespeichert sind – ähnlich wie bei einem Pokeball in dem bereits ein Pokemon drin ist.

Die Fehlerart wird über den Typ des Objekts bestimmt. Eine `NumberFormatException` ist also ein Objekt vom Typ `NumberFormatException`. Dieses wird von der Methode geworfen in der Erwartung, dass der Aufrufer etwas mit diesem Exception-Objekt macht. Wenn dieses nicht berücksichtigt wird, folgt ein Programmabbruch. Dies ist wieder ähnlich zu den Pokebällen: wird der Pokeball (inkl. Pokemon) geworfen, wird das Exception-mon frei und das Programm stürzt ab.

Nachdem es sich hierbei um ein Objekt handelt, können mit Hilfe einer Referenvariablen auf dieses Objekt auch Objektmethoden aufgerfuen werden. So haben zum Beispiel alle Exceptions die Methode `getMessage()`, die die im Objekt gespeicherte Fehlermeldung als String zurückgibt.

Auffangen von Exceptions

Dies ist die einzige Möglichkeit, bei der ein Exception-Objekt keinen Programmabbruch verursacht. Dazu muss die Methode, die die Exception verursachen kann, – die "gefährliche" Methode – in einen mit `try` überwachten Block gegeben werden. Zusätzlich müssen auch jene Programm-Anweisungen, die von der korrekten Ausführung der problematischen Methode abhängen und unter Umständen weitere Methoden, deren Exceptions aufgefangen werden sollen, in den `try`-Block geschrieben werden.

```
import javax.swing.JOptionPane;
/** ... */
public class DoppeltAbgesichert{
    public static void main(String[] args) {
        String eingabe, nachricht;
        double zahl, doppelt;
        eingabe = JOptionPane.showInputDialog(null, "Bitte eine" +
            " Zahl eingeben.");
        try{
            zahl= Double.parseDouble(eingabe);
            doppelt = zahl * 2;
            System.out.println("Das Doppelte von " + zahl + " ist " +
                doppelt);
        }
        catch (NumberFormatException numberEx){
            nachricht = numberEx.getMessage();
            System.err.println("Ungültige Eingabe!" + nachricht);
        }
    }
}
```

Versuche die folgenden Anweisungen auszuführen und überwache sie auf auftretende Fehler

Mögliche Exception hier

Diese Anweisungen sind von der Anweisung darüber abhängig

Falls im überwachten try-Block die nebenstehende Exception auftritt, fang sie auf.

Variablenname

es wird hier eine Referenzvariable für ein Objekt vom Typ `NumberFormatException` deklariert.

Hier wird eine Objekt-Methode des `NumberFormatException`-Objekts aufgerufen, die die im Objekt gespeicherte Fehlermeldung zurückgibt.

Ausgabe einer eigenen Fehlermeldung zusammen mit der Fehlermeldung der Exception auf dem Ausgabebereich für Fehlermeldungen

Auf den `try`-Block folgt ein oder mehrere catch-Blöcke. Diese müssen für jede Exception, die sie fangen wollen, ein passendes "Lasso" zur Verfügung stellen – eine passende Referenzvariable. Es wird also am Beginn des jeweiligen `catch`-Blocks eine Referenzvariable deklariert, die als Datentyp jenen Exception-Typ hat, der mit diesem `catch`-Block gefangen werden soll.

Der Aufbau eines `try`-Blocks mit dazugehörigem `catch`-Blöcken sieht also folgendermaßen aus:

```
try {

    Anweisungen im überwachten Block

} catch ( Exception1 variablenname1 ) {

    Anweisungen, die ausgeführt werden sollen, wenn im
    überwachten Bereich Exception1 ausgelöst wird

} catch ( Exception2 variablenname2 ) {

    Anweisungen, die ausgeführt werden sollen, wenn im
    überwachten Bereich Exception2 ausgelöst wird

} ...
```

Achtung: das Überwachen von Methoden auf Exceptions mit Hilfe eines `try`-Blocks verbraucht Arbeitsspeicher und Prozessorressourcen und soll daher wirklich nur für jene Anweisungen verwendet werden, die entsprechend den obigen Kriterien.

1. eine Exception auslösen können, die wir auffangen wollen

   oder

2. von einer Anweisung abhängig sind, die eine Exception auslösen kann, die wir auffangen wollen.

Programmablauf in einer try-catch-Anweisungen

Der Programmablauf in einer try-catch-Anweisung sieht jenachdem, ob eine Exception auftritt oder nicht, ganz unterschiedlich aus.

Beim Auftreten der `NumberFormatException` wird die Ausführung des try-Blocks abgebrochen:

```
try {
    zahl = Double.parseDouble(eingabe);
    doppelt = zahl * 2;
    System.out.println("Das Doppelte von " + zahl + " ist " +
        doppelt);
}
catch (NumberFormatException numberEx) {
    nachricht = numberEx.getMessage();
    System.err.println("Ungültige Eingabe!" + nachricht);
}
```

1. Die `NumberFormatException` unterbricht „normalen" Programmablauf: alle weiteren Anweisungen im `try`-Block werden **nicht** ausgeführt
2. Die NF-Exception wird vom `catch` und der Variablen `numberEx` gefangen
3. Alles innerhalb **dieses** `catch`-Blocks wird ausgeführt
4. Das Programm wird nach **dem letzten** zum obigen `try` gehörenden `catch`-Block weiter ausgeführt. Andere `catch`-Blöcke werden übersprungen.

Wenn die Exception nicht auftritt, dann wird der gesamte `try`-Block ausgeführt.

```
try {
    1. Wenn keine Exception auftritt werden alle Anweisungen
                innerhalb des try-Blocks ausgeführt
    zahl = Double.parseDouble(eingabe);
    doppelt = zahl * 2;
    System.out.println("Das Doppelte von " + zahl + " ist "
        + doppelt);
}
catch (NumberFormatException numberEx) {
    nachricht = numberEx.getMessage();
    System.err.println("Ungültige Eingabe!" + nachricht);
}
```

2. Anschließend werden alle catch-Blöcke übersprungen und das Programm wird danach fortgesetzt

Eine Exception, die nicht aufgefangen wird, bricht das ganze Programm nach wie ab:

```
try {
    1. Eine nicht in einem catch-Block stehende Exception unterbricht „normalen“
Programmablauf: alle weiteren Anweisungen im try-Block werden nicht ausgeführt
    zahl = Double.parseDouble(eingabe);
    doppelt = zahl * 2;
    System.out.println("Das Doppelte von " + zahl + " ist " +
        doppelt);
}
catch (NumberFormatException numberEx) {
    nachricht = numberEx.getMessage();
    System.err.println("Ungültige Eingabe!" + nachricht);
}
```

2. Diese Exception wird nicht gefangen und das Programm wird abgebrochen

Generisches Abfangen von Exceptions

In manchen Beispielen sieht man anstelle eines speziellen Datentyps für die Exception-Variable im catch (wie z.B. `NumberFormatException`) nur `Exception`. Dies veranlasst Java dazu **alle** Exceptions aufzufangen, die auftreten könnten. Dies ist aus zwei Gründen wenig optimal[1]:

1. RuntimeExceptions werden vor allem auch dazu verwendet Programmierfehler zu melden. Dadurch dass diese Exceptions ebenfalls abgefangen werden, könnten so Programmierfehler länger verborgen bleiben.
2. Das Auffangen von Exceptions soll es ermöglichen, den Fehler in irgendeiner Form zu korrigieren (z.B. durch das Wiederholen der Eingabe). Dabei muss man aber Berücksichtigen um welchen Fehler es sich handelt und das kann nur sicher gestellt werden, wenn man die spezielle Exception abfängt.

[1] Vergleiche dazu auch den Artikel in Stackoverflow: https://stackoverflow.com/questions/2416316/why-is-the-catchexception-almost-always-a-bad-idea

Oft geben aber erfahrenere Programmierer in kurzen Programmbeispielen nur diese wenig optimale Variante an, da sie kürzer ist. Sie gehen dann aber davon aus, dass diejenigen, die dieses Beispiel verwenden, die passenden Exceptions abfangen.

Habe ich es verstanden?

- Ich kann die zwei prinzipiellen Exception-Arten bennenen und ihre Unterschiede erklären.
- Ich kann erklären, warum unchecked Exceptions oft nicht bearbeitet werden sollen.
- Ich kann erklären was ein Stack-Trace ist und inwiefern er bei einer Exception eine Rolle spielt.
- Ich kenne den Ausgabebereich für Fehler und kann eigene Meldungen dort ausgeben.
- Ich kann eine Exception mit einer try-catch-Anweisung auffangen
- Ich kann den Programm-Ablauf in verschiedenen Szenarien mit einer try-catch-Anweisung erklären.
- Ich kann die Problematik das Auffangen des generischen Exception-Typs erklären

Erweiterter Inhalt: finally

Nach den `catch`-Blöcken oder auch **anstelle** der `catch`-Blöcke kann zu einem `try`-Block auch ein `finally`-Block angegeben werden. Mit `finally` kann die `try-catch`-Anweisung so aussehen:

```
try {
    Anweisungen im überwachten Block
} catch ( Exception1 variablenname1 ) {
    Anweisungen, die ausgeführt werden sollen, wenn im
    überwachten Bereich Exception1 ausgelöst wird
} catch ( Exception2 variablenname2 ) {
    Anweisungen, die ausgeführt werden sollen, wenn im
    überwachten Bereich Exception2 ausgelöst wird
} ...
finally {
    Anweisungen, die immer ausgeführt werden sollen, wenn
    aus irgendeinem Grund der try-Block verlassen wird, z.B.
    auch wenn eine nicht abgefangene Exception auftritt oder
    wenn jemand unabsichtlich ein return in ein try hinein
    geschrieben hat.
}
```

Im finally kann man daher Codezeilen schreiben, wo auf jeden Fall versucht werden soll, diese Auszuführen. Dies kann z.B. noch ein Sichern der Daten sein oder ein Benachrichtigen des Servers, dass das Programm nun beendet wird.

Habe ich es verstanden?

- Ich kann zu einem try-Block noch einen finally Block schreiben
- Ich kann erklären unter welchen Umständen der finally-Block ausgeführt wird
- Ich kann erklären, wozu man einen finally-Block verwenden kann.