Schülerakademie Schelklingen 2008

Kurs 4.2 - eXtreme Programming

Exceptions und Logging in Java

Manuel Eberl

7.8.2008

1 Wozu Exceptions?

Exceptions, zu Deutsch: "Ausnahmen", sind ein Konzept in vielen Programmiersprachen, unter anderem Delphi, C++, Java, C#, VisualBasic.NET, das es erlaubt, auf Fehler, die in einem Programm auftreten zu reagieren. Bemerkt ein Programmteil einen Fehler, zum Beispiel beim Versuch, eine nicht existente Datei zu laden, oder wenn eine Netzwerkverbindung von der Gegenseite unerwartet beendet wird, so löst er eine Exception aus. Diese Exception wird durch die Hierarchie des Programms nach oben weitergereicht, bis sie einen Programmteil erreicht, der sie verarbeiten kann, und von ihm abgefangen wird.

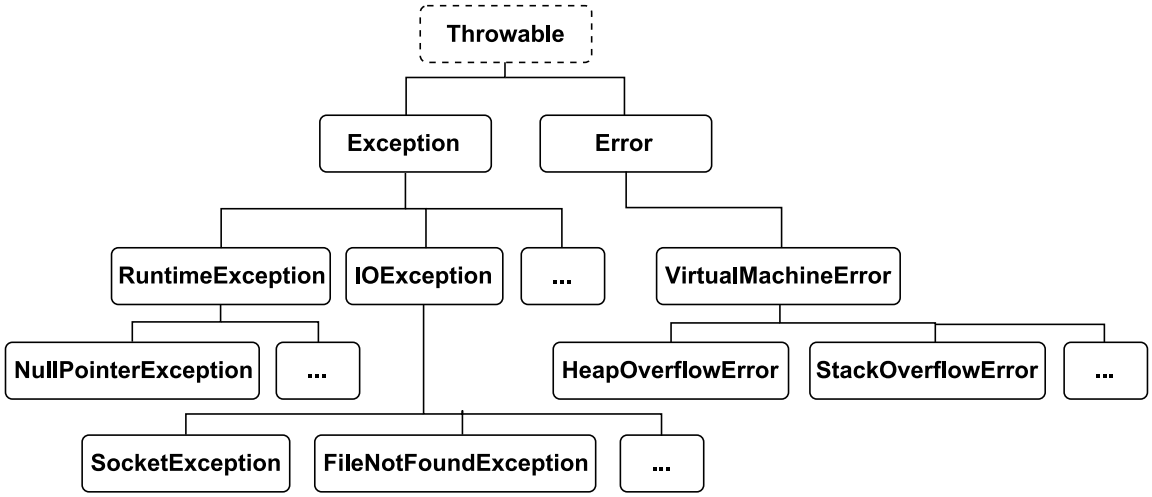
Ein Beispiel wäre ein Textbearbeitungsprogramm, bei dem der Benutzer mithilfe einer Benutzeroberfläche einen Dateinamen zum Laden einer Datei eingibt. Der Benutzer gibt nun eine Datei ein, die nicht existiert. Die Benutzeroberfläche ruft daraufhin eine Funktion auf, die die Datei laden soll. Da sie aber nicht existiert, muss diese Ladefunktion eine Fehlerbehandlung auslösen, da sie aber nicht weiß, wofür die Datei überhaupt gebraucht wird, kann sie diese Behandlung nicht selbst durchführen. Darum löst sie eine Exception aus, die sie mit Informationen über den Grund des Fehlers versieht und reicht sie an die aufrufende Benutzeroberfläche weiter, die sie dann behandeln kann, indem sie zum Beispiel eine entsprechende Fehlermeldung anzeigt.

Exceptions erlauben es dem Programm, Fehlerbehandlungen bis zu der Ebene weiterzuleiten, die dafür geeignet ist. Ohne sie müsste jede Funktion eine Fehlerbehandlung durchführen, wofür sie auch sämtliche relevanten Informationen zur Reaktion auf den Fehler erhalten müsste. Dies macht den Programmcode unübersichtlich und unflexibel.

2 Exceptions und Errors in Java

2.1 Throwables

Throwable ist ein Interface, von dem sich alle Objekte ableiten, die ausgelöst und abgefangen werden können. Die einzigen Klassen der Java-Bibliothek, die Throwable implementieren, sind Exception und Error. Ausgehend von Throwable und diesen beiden Klassen sowie RuntimeException, einer von Exception abgeleiteten Klasse, erstreckt sich eine komplexe Vererbungshierarchie von Exceptions und Errors:



2.2 Checked und unchecked Exceptions

In Java wird ein besonderes Prinzip von Exceptions verwendet: Es gibt "checked" und "unchecked" Exceptions. Java prüft bereits bei der Kompilierung des Programmcodes, ob alle "checked" Exceptions, die in einer Funktion auftreten können, auch gefangen oder (per 'throws') an die aufrufende Funktion weitergeleitet werden. Ist dies nicht der Fall, wird die Kompilierung mit einen Fehler abgebrochen. Unchecked Exceptions werden normalerweise, weil sie nicht gefangen werden, durch die gesamte Aufrufhierarchie nach oben gereicht, bis sie von der Java-Umgebung abgefangen werden und dadurch das Programm abgebrochen wird.

2.3 Syntax in Bezug auf Throwables in Java

2.3.1 Auslösen eines Throwables

throw obj; (wobei obj eine Instanz eines Throwables ist)

Meistens wird die folgende Form benutzt:
throw new IllegalArgumentException("Ungültiger Parameter: "+param); 'IllegalArgumentException' ist hierbei der Typ des Fehlers, dieser Typ wird ausgelöst, wenn eine Funktion einen ungültigen Wert übergeben bekommen hat. '"Ungültiger Parameter. "+param' ist die Fehlermeldung, die möglichst aussagekräftig sein sollte, um sofort auf den Grund des Fehlers schließen zu können. Exceptions, die nicht von RuntimeException abgeleitet sind, sind checked, Errors und RuntimeExceptions sind unchecked. Dies liegt daran, dass Exceptions in einem Programm auftreten und verarbeiten werden können und das Programm danach weiterlaufen kann, während RuntimeExceptions meist Programmierfehler und Errors fatale Fehler sind. Beide sollten nicht abgefangen werden, erstere sollten behoben werden, letztere lassen sich nicht vermeiden und erzwingen einen Programmabbruch.

2.3.2 Behandeln von Throwables

try-catch-Block:
Ein try-catch-Block fängt alle Exceptions ab, die in den catch-Blöcken angegeben werden. catch (Exception1 e) fängt dabei alle Exceptions ab, die vom Typ Exception1 sind oder von ihm ableiten. Könnte eine Exception theoretisch von mehreren catch-Blöcken gefangen werden, z.B. wenn eine FileNotFoundException geworfen wurde und zwei catch-Blöcke für FileNotFoundException und IOException definiert sind, so wird immer der erste passende catch-Block ausgeführt.

```
try {
    Programmcode der Exceptions vom Typ Exception1, Exception2 auslösen kann
} catch (Exception1 e) {
```

```
    Fehlerbehandlung für Exception1
} catch (Exception2 e) {
    Fehlerbehandlung für Exception2
}
```

Ein finally-Block wird immer ausgeführt, egal ob der Code im try-Block eine Exception auslöst oder nicht. Dies ist zum Beispiel nützlich, wenn im try-Block eine Ressource, etwa eine Datei, angefordert wird, die nach der Bearbeitung wieder für andere Prozesse freigegeben werden sollte.

try-finally-Block:

```
try {
    Programmcode, der Exceptions auslösen kann
} finally {
    Programmcode, der immer ausgeführt wird.
        (Egal ob der Code im try-Block Exceptions auslöst oder nicht)
}
```

try-catch und try-finally lassen sich auch kombinieren. In diesem Fall wird der finally-Block einfach an den letzten catch-Block angehängt. Wird in einem der catch-Blöcke eine Exception ausgelöst und an die aufrufende Funktion weitergereicht, so wird der finally-Block trotzdem noch ausgeführt, bevor dies geschieht.

2.3.3 Weiterreichen von Exceptions

Falls eine Funktion nicht die erforderlichen Informationen besitzt, um einen Fehler zu verarbeiten (etwa weil sie nur etwas tun soll, beispielweise eine Datei öffnen, aber nicht warum und daher auch nicht, was geschehen soll, wenn dies fehlschlägt), so kann sie einen auftretenden Fehler an die aufrufende Funktion weiterreichen. Damit dies vom Compiler zugelassen wird, muss die Exception in die throws-Deklaration der Funktion aufgenommen werden. Die Syntax dafür lautet:

`public void function() throws Exception1, Exception2` Code, der Exception1 und Exception2 verursachen kann, diese aber nicht abfängt

Errors und RuntimeExceptions sind unchecked und sollten sowieso nicht gefangen werden, deshalb müssen sie auch nicht in die throws-Deklaration mit aufgenommen werden.

2.4 Exceptions

"Normale" Exceptions, das heißt, Typen die von Exception (nicht aber von RuntimeException oder einem ihrer Untertypen) abgeleitet sind, werden in Java für Fehler benutzt, mit denen in einer normalen Ausführung des Programms gerechnet werden muss, ein gutes Beispiel ist zum Beispiel das oben genannte Problem einer nicht gefundenen Datei (FileNotFoundException) oder einer unerwartet geschlossenen Netzwerkverbindung (IOException).

2.5 Errors

In Java gibt es außer Exceptions noch Errors, die sich ebenso wie Exceptions auslösen und fangen lassen. Errors jedoch sind Fehler, die abnormale Zustände repräsentieren, die aus einer Fehlfunktion der Java-Umgebung resultieren, und deshalb nicht gefangen werden sollten. In den meisten Fällen handelt es sich dabei um fatale Fehler, bei denen die Programmausführung sofort beendet werden muss. Ein Beispiel hierfür sind der StackOverflowError und der HeapOverflowError, die von der Java-Umgebung augelöst werden, wenn kein freier Speicher mehr vorhanden ist, was meistens auf Programmierfehler zurückzuführen ist, oder der NoClassDefFoundError, der ausgelöst wird, wenn eine Klasse benötigt wird, aber im kompilierten Programmcode nicht mehr gefunden werden kann.

2.6 RuntimeExceptions

RuntimeExceptions ähnelns Errors darin, dass sie nicht gefangen werden sollten, da sie im normalen Programmablauf nicht auftreten sollten. Anderes als bei Errors arbeitet die Java-Umgebung hier jedoch korrekt, der Fehler liegt im Programmcode. Beispiele für RuntimeExceptions sind die NullPointerException, die ausgelöst wird, wenn versucht wird, auf ein nicht instantiiertes Objekt zuzugreifen, oder die ArithmeticException, die zum Beispiel ausgelöst wird, wenn versucht wird durch 0 zu teilen. RuntimeExceptions sollten vom Programmierer behoben, nicht gefangen werden.

2.7 Verschachteln von Exceptions

Es ist oft der Fall, dass beim Auslösen einer Exception eine in der Aufrufhierarchie höher stehende Funktion mehr Informationen zum Grund dieser Exception hat und diese Information auch ihren aufrufenden Funktionen zur Verfügung stellen möchte. In diesem Fall kann die ausgelöste Exception abgefangen werden und in eine andere, neue Exception "verpackt" werden. Dieses "Verpacken" kann beliebig oft verschachtelt werden. Eine Funktion, die weit oben in der Aufrufhierarchie steht und genügend Informationen besitzt, um eine Fehlerbehandlung durchzuführen, kann dann diese Verschachtelung, falls nötig, aufdröseln und die Gründe für die Exception für die Behandlung verwenden.

Eine häufig verwendete Anwendung dieser Verschachtelung ist, eine gefangene Exception in eine RuntimeException zu verpacken und diese weiterzuwerfen. Dies kann bei Exceptions nützlich sein, die den Programmablauf so empfinglich stören, dass das Programm nicht mehr weiterarbeiten kann, was die auslösende Funktion jedoch nicht weiß.

So kann zum Beispiel eine nicht gefundene Datei, deren Pfad vom Benutzer eingegeben wurde, ein erwarteter und behandelbarer Fehler sein, während eine vom Programm benötigte Datei, die nicht gefunden werden kann, den sofortigen Abbruch des Programmes nach sich zieht. Die Funktion, die die Datei lädt, kann dies nicht wissen und löst deshalb eine FileNotFoundException aus. Im ersten Fall würde diese nun behandelt werden und eine Fehlermeldung nach sich ziehen, während im zweiten Fall ein Verpacken in eine RuntimeException die beste Behandlungsmöglichkeit wäre, da sie das Programm sofort abbricht und dabei trotzdem verwertbare Informationen zur Ursache dese Fehlers zur Verfügung stellt.

2.8 Vor- und Nachteile des Exceptionmodells in Java

Einer der großen Vorteile von checked Exceptions ist es, dass der Programmierer nicht vergessen kann, bestimmte Exceptions zu behandeln, da der Programmcode ohne eine entsprechende Behandlung nicht kompiliert. Dadurch erhält das Programm eine geringere Fehleranfälligkeit, sofern die Fehlerbehandlung richtig implementiert wird.

Ein oft angeführter Nachteil ist, dass die Flexibilität des Codes durch checked Exceptions verringert wird, da beim Hinzufügen einer Exception zu einer Funktion Code, der diese benutzt, ohne ein Hinzufügen einer Fehlerbehandlung nicht mehr kompiliert. Jedoch sollte dieser Code sowieso geändert werden, wenn zu der Funktion eine neue Exception hinzugefügt wird, da diese ohne Behandlung im schlimmsten Fall zu einem Programmabsturz führen kann. Eine checked Exception verhindert einen solchen Fehler sogar, indem sie den Programmierer, der die Funktion benutzt, zwingt, auf die neue Exception zu reagieren und das Programm somit stabil zu halten, während ohne eine checked Exception der Programmierer diese Änderung möglicherweise nicht bemerken würde.

3 Logging

Um Fehler im Programmablauf nachvollziehen zu können, kann es nützlich sein, bestimmte Aktionen des Programms oder möglicherweise fehlerhafte Werte aufzuzeichnen. Hierfür eignen sich sogenannte Logger. Sie erleichtern derartige Aufzeichnungen, indem sie Funktionen zur Verfügung stellen, um bestimmte Mitteilungen leicht entsprechend formatiert auf der Konsole oder in eine Datei ausgeben zu können. Die Vorteile von Loggern sind dabei:

- Leichte Formatierung (z.B. hinzufügen der Zeit)
- Flexibles Ausgabeziel (Konsole, Datei...)
- Filterung (Verschiedene Detailstufen von Informationen wählbar, je nach Einstellung werden Nachrichten ab einem bestimmten Detailgrad verworfen)

Detaillierte und präzise Ausgaben durch Logger an geeigneten Stellen im Programm können die Fehlersuche deutlich vereinfachen und ermöglichen es auch unerfahrenen Benutzern, bei der Behebung eines aufgetretenen Fehlers zu helfen, indem sie die Log-Datei an den Programmierer schicken, der mit deren Hilfe dann unter Umständen die Fehlerquelle nachvollziehen kann.