

Skript Informatik B

Objektorientierte Programmierung in Java

Sommersemester 2011

- Teil 4 -

Inhalt

0 Einleitung

1 Grundlegende objektorientierte Konzepte (Fundamental Concepts)

2 Grundlagen der Software-Entwicklung

3 Wichtige objektorientierte Konzepte (Major Concepts)

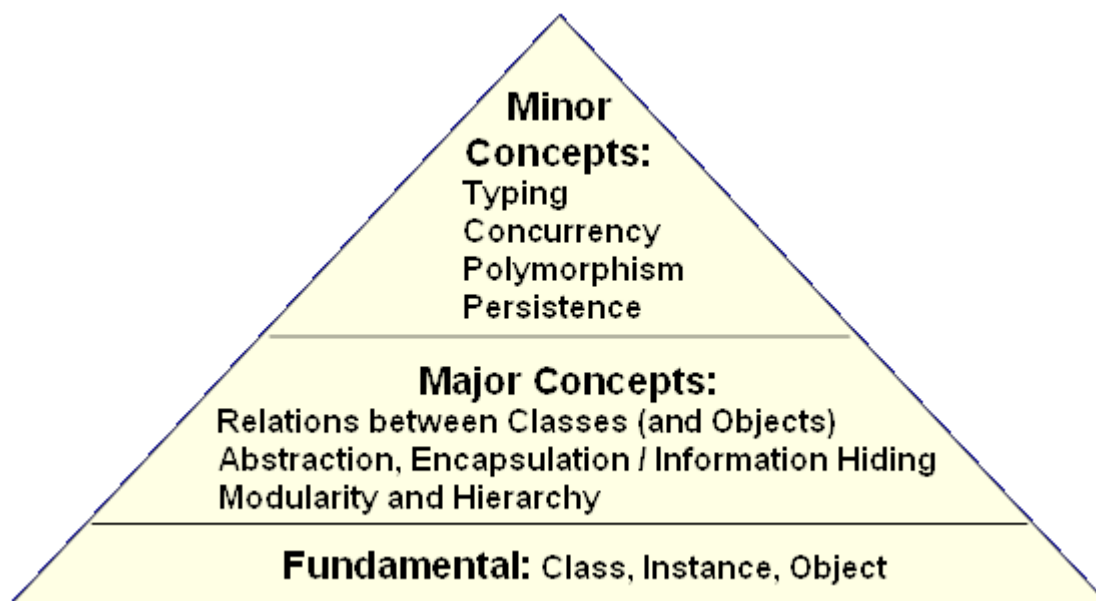
4 Fehlerbehandlung

... wird schrittweise erweitert

Kapitel 4:

Fehlerbehandlung

- 4.1 Grundlagen der Fehlerbehandlung
- 4.2 Zusicherungen (Assertions)
- 4.3 Ausnahmen (Exceptions)
- 4.4 Fehler (Errors)
- 4.5 Zusicherungen und andere Erweiterungen in UML
- 4.6 Laufzeitstapel (Stack Trace)



[Boo94]

Vorbemerkung:

Die Fehlerbehandlung wird im Übersichtsbild objektorientierter Konzepte nicht genannt. Das bedeutet, dass die Fehlerbehandlung kein typisches Konzept der Objektorientierung ist bzw. nicht originär an die Objektorientierung gebunden ist. Es wäre grundsätzlich falsch, daraus den Schluss zu ziehen, dass eine Fehlerbehandlung in der Objektorientierung nicht wichtig ist. Vielmehr ist eine Fehlerbehandlung auch in anderen Paradigmen zentral. Sie zeichnet das objektorientierte Paradigma also nicht speziell gegenüber anderen Paradigmen aus. An Java werden wir sehen, wie eine typische Fehlerbehandlung in einer objektorientierten Welt realisiert sein kann.

In der Objektorientierung unterscheidet sich die Art und Weise der Fehlerbehandlung dennoch zu der in anderen Paradigmen.

4.1 Grundlagen der Fehlerbehandlung

Fehlerbehandlung geschieht auf sehr unterschiedliche Weise.

Wir betrachten in den folgenden Seiten nur die Fehlerbehandlung, die in der Programmiersprache verankert ist, d.h. diejenige, die der Programmierer in sein Programm einbaut.

Andere Arten der Fehlerbehandlung außerhalb der Programmiersprache sind beispielsweise

- Debugging,
- Testen (auf der Basis einfacher aber systematischer Ein-/Ausgaben oder mit umfangreichen Testumgebungen),
- Verifikation (Beweis der Korrektheit bestimmter Programmeigenschaften).

Noch besser als eine Fehlerbehandlung ist natürlich die Fehlervermeidung (z.B. durch bestimmte Entwicklungsrichtlinien). Da eine Vermeidung aber niemals vollständig sein kann, kann sie die Fehlerbehandlung nicht ersetzen.

Fehlerklassen

1. Programmierfehler: Das Programm macht nicht was es soll.
2. Ausnahmesituation: Das Programm gerät in eine nicht vorhersehbare Situation zur Laufzeit.

Wie kann man auf Laufzeitfehler reagieren?

- Rückgabe eines Fehlercodes (z.B. 0, NULL, -1), Beispiel: `if (error) return -1;`

Laufzeitfehler werden häufig auf diese Art und Weise behandelt.

Sie ist jedoch nicht (immer) ideal.

Probleme mit dieser Lösung:

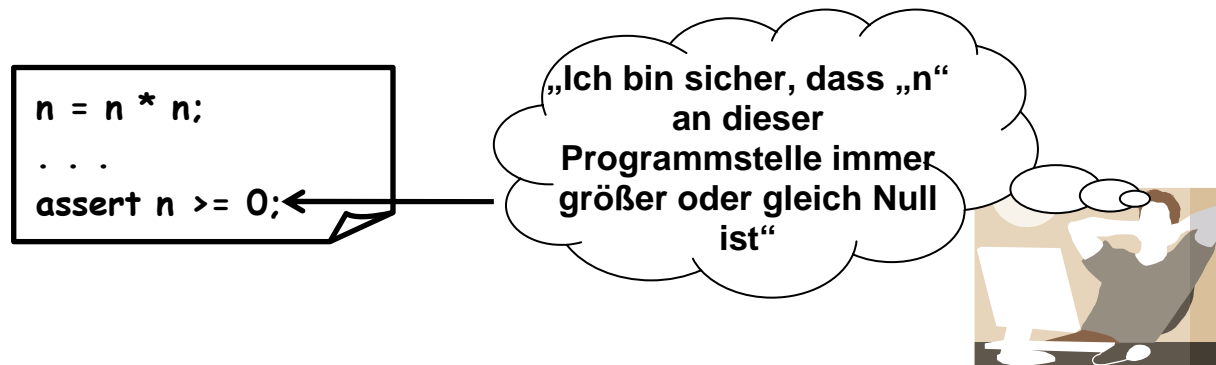
- Der Rückgabewert ist in seiner Bedeutung überladen.
 - Der Rückgabewert wird eventuell nicht beachtet.
 - Der Programmfluss wird durch Abfragen des Funktionsergebnisses unterbrochen.
- Einsatz von *Assertions* und *Exceptions* (zum Teil als Sprachkonstrukt in den Programmiersprachen vorhanden).

4.2 Zusicherungen (engl. Assertions)

Assertion = Zusicherung, Behauptung

- Ist die Zusicherung (Assertion) nicht erfüllt, dann hat das Programm einen Fehler.
- Bei einem Zusicherungsfehler, sollte das Programm abgebrochen werden.
- In Java: Die JVM überwacht vom Entwickler angegebene Assertions beim Programmablauf (seit Java 1.4).

Bei nicht erfüllten Zusicherungen wird ein Fehler geworfen: Fehlertyp **Error**.



Einsatz: Sicherstellung der Korrektheit des Programms (zur Laufzeit)

- Assertions sollen nur dann scheitern, wenn Fehler in der Programmierung vorliegen.
- Mit Assertions sollten keine von äußeren Einflüssen abhängigen Bedingungen zugesichert werden (z.B. Parameterwerte, Eingaben).
- Assertions können zur Absicherung von Vorbedingungen, Nachbedingungen, Invarianten eingesetzt werden: Garantie eines korrekten Ablaufs (z.B. innerhalb einer Methode).

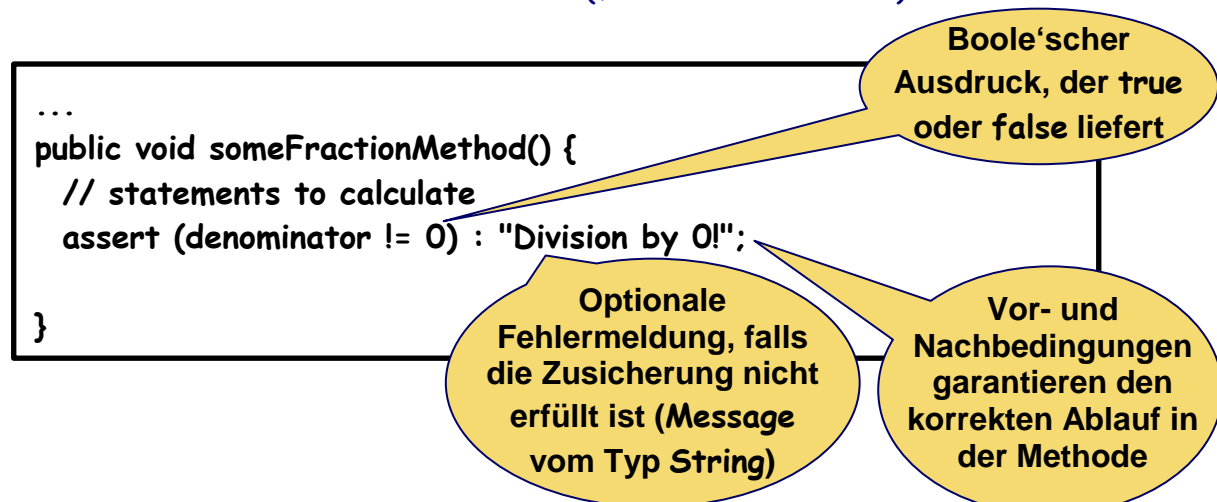
Beispiel einer Assertion in Java:

- Syntax, um eine Zusicherung in Java auszudrücken: `assert expression [:Message];`
- Bei nicht erfüllter Assertion: Programmabbruch und Fehlermeldung

Exception in thread main java.lang.AssertionError:

Message

at classname.methodname(filename:linenumber)



- Explizite Aktivierung von Assertions beim Start der JVM durch den Parameter: **-ea** (enable assertions)

Generell für ein Programm: **java -ea**

Für einzelne Klassen: **java -ea: Klassenname**

Für ein Paket: **java -ea: packagename...** (Bemerkung: „...“ gehört ebenfalls zur Syntax)

Standard: Aus Gründen der Performance sind Assertions in Java nicht aktiviert.

- Deaktivierung der Assertion-Überprüfung analog zur Aktivierung aber mit Parameter **-da**
- Mehrere Kommandozeilenparameter können kombiniert werden (dies erlaubt eine möglichst genaue Einstellung der Assertion-Berücksichtigung).


4.3 Ausnahmen (engl. Exceptions)

- Exception = Ausnahme, Ausnahmesituation
- Eine Ausnahme schließt den Fehlerfall als mögliches Ergebnis ein.
- Bestimmte Programmezustände (meist Fehlerzustände) werden an eine andere Programmebene zur Weiterbehandlung weitergegeben.
- Behandlungsalternativen im Programm, wenn eine Ausnahme auftritt:
 1. definierte Abarbeitung,
 2. (explizites) Ignorieren,
 3. Anzeigen (z.B. durch eine Bildschirmausgabe).

Beispiel für Exceptions an einem Programmbeispiel in Java: Datei öffnen

```
...  
try {  
    f.createNewFile();  
} catch (IOException e) {  
    System.err.println("Error");  
} ...
```

„Hier könnte eine kritische Situation bzw. ein Fehlerfall auftreten“



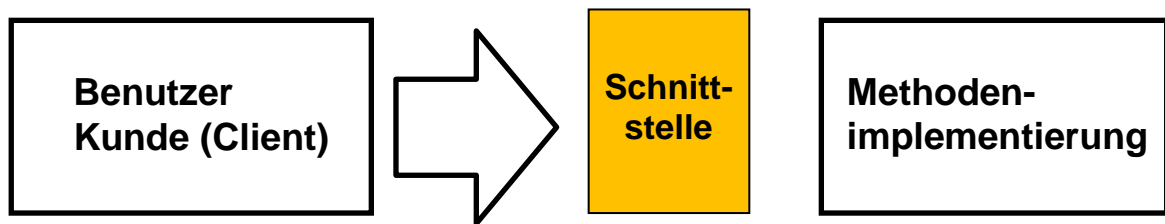
Konzepte der Exception-Behandlung in Java:

- Java unterstützt eine strukturierte Ausnahmebehandlung (*Structured Exception Handling* SEH): Man spricht von strukturierter Ausnahmebehandlung, dann, wenn der Code zur Ausnahmebehandlung von “normalem” Code getrennt ist. In Java enthält der **try**-Block den „normalen“ Code während der **catch**-Block den Code zur Ausnahmebehandlung implementiert.
- Java realisiert *Checked Exceptions*: Man spricht von *Checked Exceptions*, wenn der Compiler prüft, ob alle Ausnahmen, die jemals geworfen werden könnten, auch wirklich behandelt werden. In diesem Fall ist also eine explizite Angabe notwendig, wenn eine

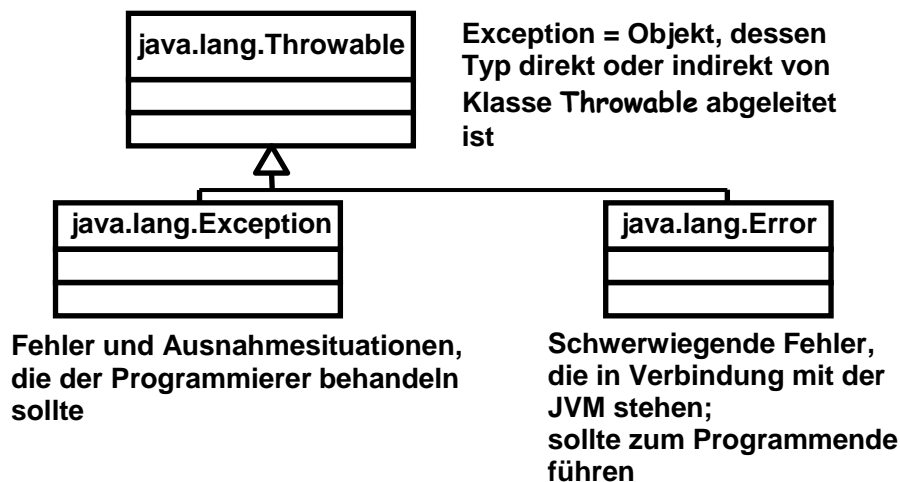
Exception nicht aufgefangen bzw. behandelt werden soll.

Problem für Checked Exceptions: Erweiterung der Implementierung um Exceptions. Der Vorteil einer Schnittstelle ist, dass wir die Implementierung hinter der Schnittstelle austauschen können, ohne dass der Client davon betroffen ist. Sein Code muss nicht geändert werden.

Wenn wir die Implementierung ändern, wirft die Implementierung eventuell auch neue Exceptions, die wir nach dem Konzept der Checked Exceptions im Client behandeln müssen. Die Änderung der Implementierung erfordert durch diese neuen Ausnahmen – trotz Schnittstellenvereinbarung – also doch wieder eine Anpassung im Client. Dies ist der Preis, den wir für die zusätzliche Sicherheit durch die Checked Exceptions bezahlen müssen.

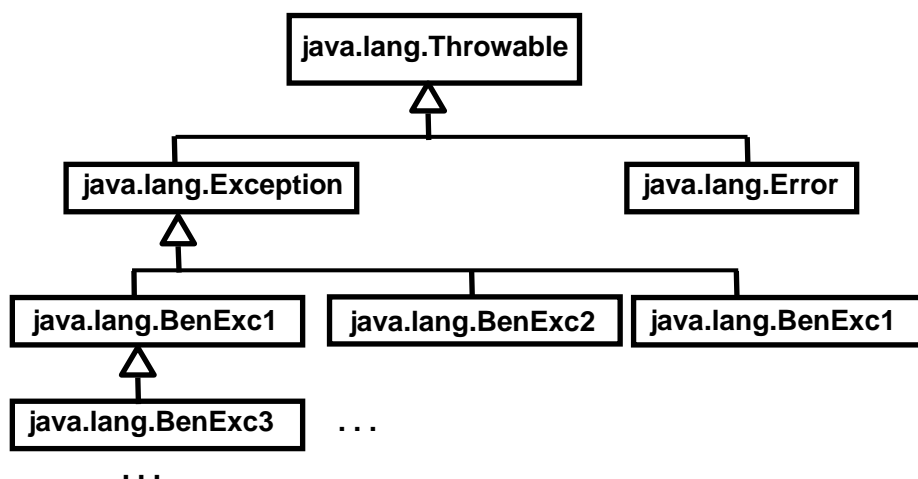
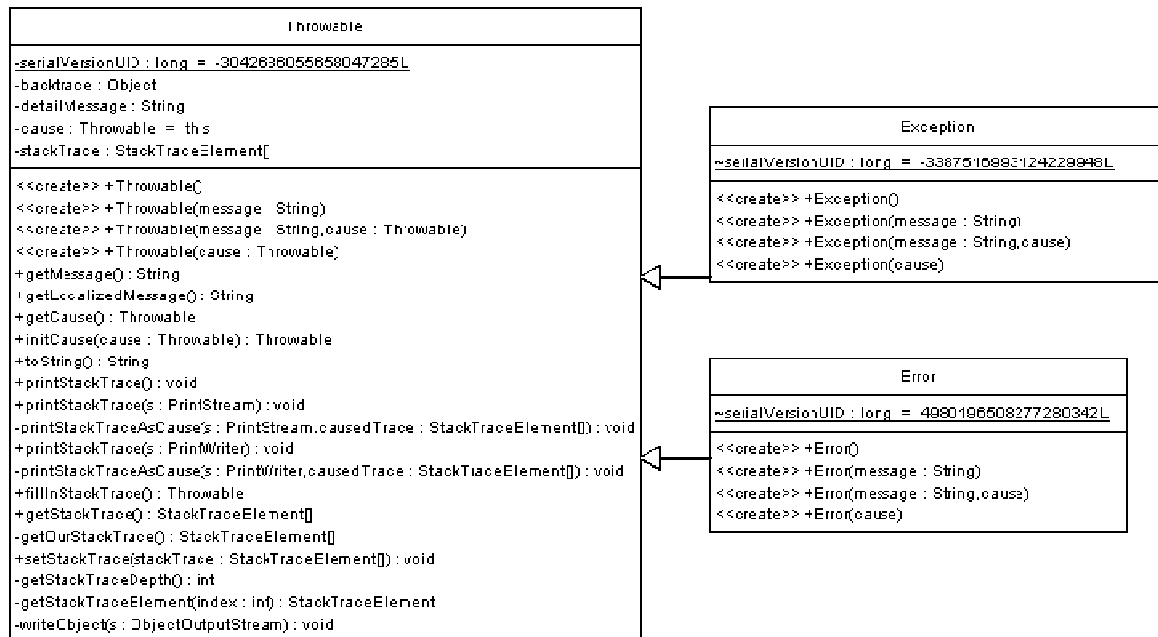


4.3.1 Exception-Behandlung in Java: Klassenhierarchie



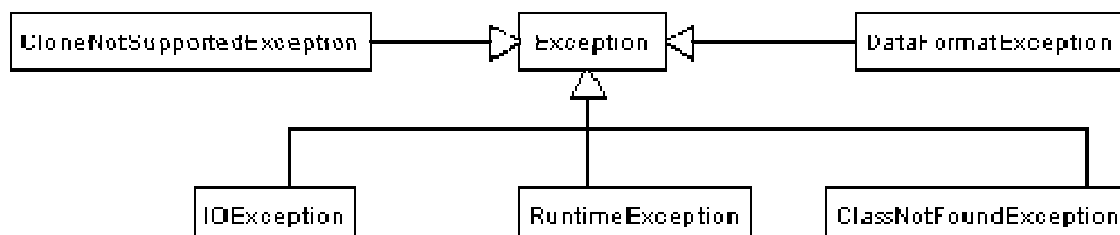
Eine Exception in Java ist letztlich nichts anderes als ein Objekt vom Typ `Throwable` bzw. vom Typ `Exception`.

Während Objekte vom Typ `Exception` vom Programmierer behandelt werden sollten, gilt dies nicht für den Typ `Error`. Wird ein Objekt vom Typ `Error` „geworfen“, sollte das Programm beendet werden, da es einen schwerwiegenden Fehler enthält.



Eine Java-Entwicklerin kann eigene **Exception**-Typen bilden, indem sie Subklassen zum Typ `java.lang.Exception` aufbaut.

Beispiele für Subklassen zu Exception in Java:



Exceptions und ihre Unterklassen sollten vom Programmierer behandelt werden, wenn sie im Programm (als Objekte) geworfen werden können.

4.3.2 Fehlerbehandlung in Java (catch)

Beispiel für eine Fehlerbehandlung in Java:

...	
import java.io.IOException;	
...	
MyFile mf =	
new MyFile("exception3", "NewFile");	Beginn (try)
try {	Überwacher
if(mf.createFile())	Programmblock
System.out.println(„File created“);	Ende (catch)
} catch (IOException e) {	Programmblock, der im Fehlerfall ausgeführt wird, um den Fehler abzufangen (engl. to catch) und zu behandeln
System.err.println(„File could not be created“);	
}	
...	



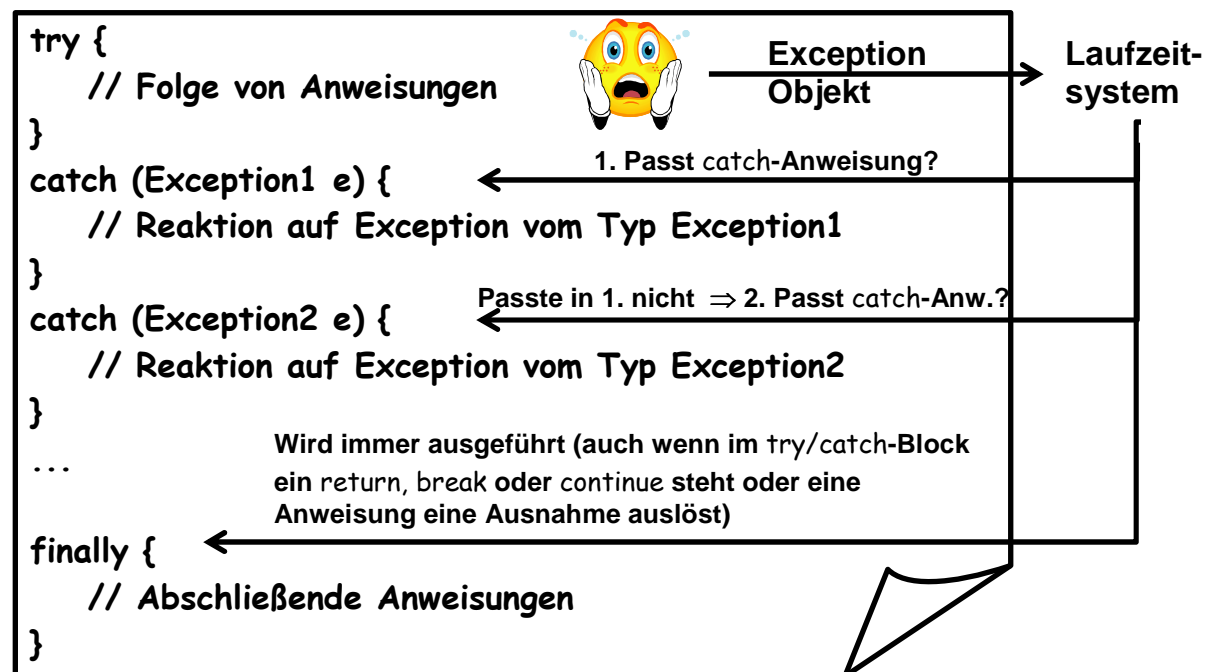
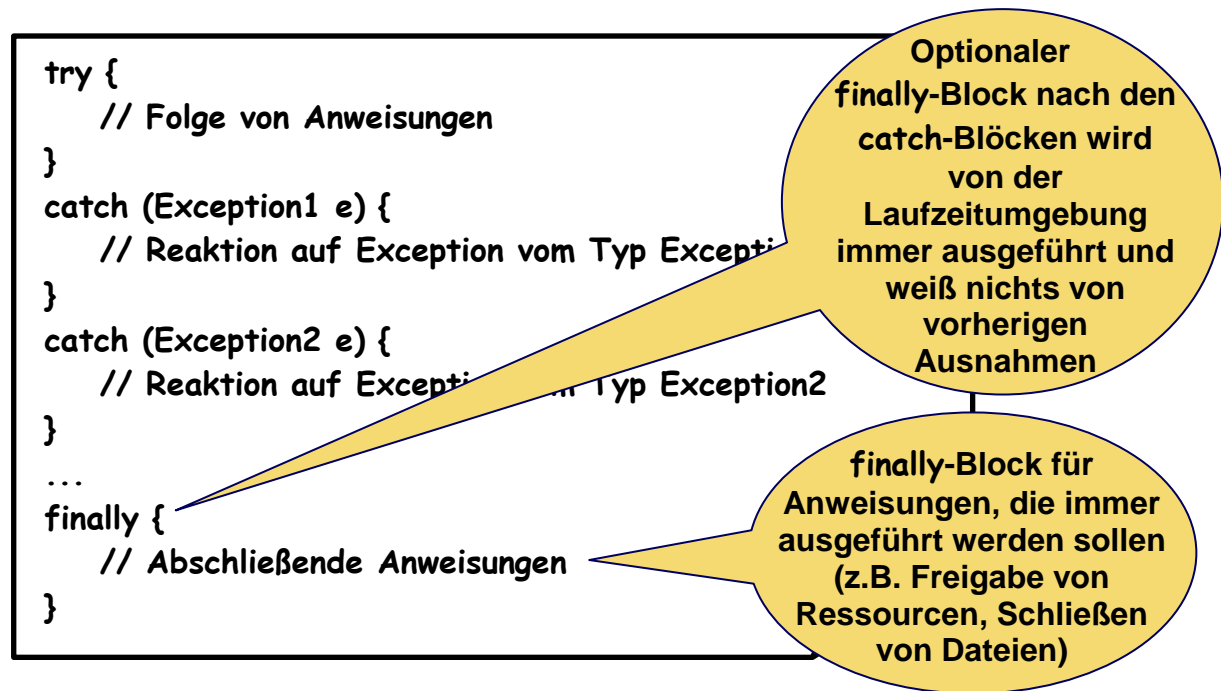
Ein besonders ausgezeichnetes Programmstück (try-catch-Block) überwacht mögliche Fehler und ruft gegebenenfalls speziellen Programmcode zur Behandlung auf.

Allgemeiner Aufbau einer Exception-Behandlung in Java:

try {	
// Folge von Anweisungen	Keine Unterbrechung des Programmflusses durch Abfrage von Rückgabewerten (if), die einen möglichen Fehler anzeigen
}	
catch (Exception1 e) {	
// Reaktion auf Exception vom Typ Exception1	
}	
catch (Exception2 e) {	
// Reaktion auf Exception vom Typ Exception2	
}	
...	
finally {	
// Abschließende Anweisungen	
}	

Keine Unterbrechung des Programmflusses durch Abfrage von Rückgabewerten (if), die einen möglichen Fehler anzeigen

Es können mehrere catch-Blöcke angegeben werden, die unterschiedliche Fehlertypen abfangen



Ablauf im Fall einer Exception in Java:

- In einem Methodenaufruf innerhalb eines try-Blocks wird ein Exception-Objekt erzeugt.
- Die Abarbeitung der Programmzeilen wird sofort unterbrochen, und das Laufzeitsystem steuert selbstständig die erste catch-Klausel an.
- Wenn die erste catch-Anweisung nicht zur Art des aufgetretenen Fehlers passt, werden der Reihe nach alle übrigen catch-Klauseln untersucht. Die erste übereinstimmende Klausel wird ausgewählt.

- Achtung: Es wird nur die erste passende **catch**-Klausel ausgeführt! Nachfolgende werden anschließend nicht weiter betrachtet.
 - Die spezielleren **catch**-Anweisungen müssen daher vor den allgemeineren stehen (z.B. **catch: IOException** vor **catch: Exception**).
 - Fehlerarten, die unterschiedlich behandelt werden müssen, verdienen immer getrennte **catch**-Klauseln.
- Die Laufzeitumgebung führt die Anweisungen im (optionalen) **finally**-Block immer aus - egal, ob eine Ausnahme auftrat, oder die Anweisungen im **try-catch**-Block wie erwartet abgearbeitet wurden.
 - Nach der Fehlerbehandlung ist es nicht möglich, an der Stelle fortzufahren, an der der Fehler auftrat. Es geht vom **catch** -Block aus weiter.



Aus [JavaInsel09]

Besonderheiten mit dem Duo **return** und **finally**:

Ein Phänomen in der Ausnahmebehandlung von Java ist eine **return**-Anweisung innerhalb eines **finally**-Blocks.

Verschwinden des Rückgabewerts:

- Ein **return** im **finally**-Block „überschreibt“ den Rückgabewert eines **returns** im **try-catch**-Block.

Verschwinden von Exceptions:

- Ist im **finally**-Block eine **return**-Anweisung vorhanden, wird eine im **catch**-Block ausgelöste Exception nicht zum Aufrufer weitergeleitet.
- Es wird lediglich der Rückgabewert zurückgeliefert.

4.3.3 Werfen von Exceptions in Java (throw/s)

Ausnahmen können auf zwei Arten geworfen werden:

- (1) Ausnahmen können an den Aufrufer weitergereicht werden.
- (2) Es können eigene (neue) Exceptions ausgelöst werden.



Fall 1: Weiterreichen von Ausnahmen

- Die Methode zeigt an, dass sie eine bestimmte Exception (im folgenden Java-Beispiel: `IOException`) nicht selbst behandelt, sondern diese unter Umständen an die aufrufende Methode weitergibt.
- Im Fehlerfall (d.h. die Exception tritt auf) wird die Methode abgebrochen und die Exception wird an den Aufrufer weitergegeben.

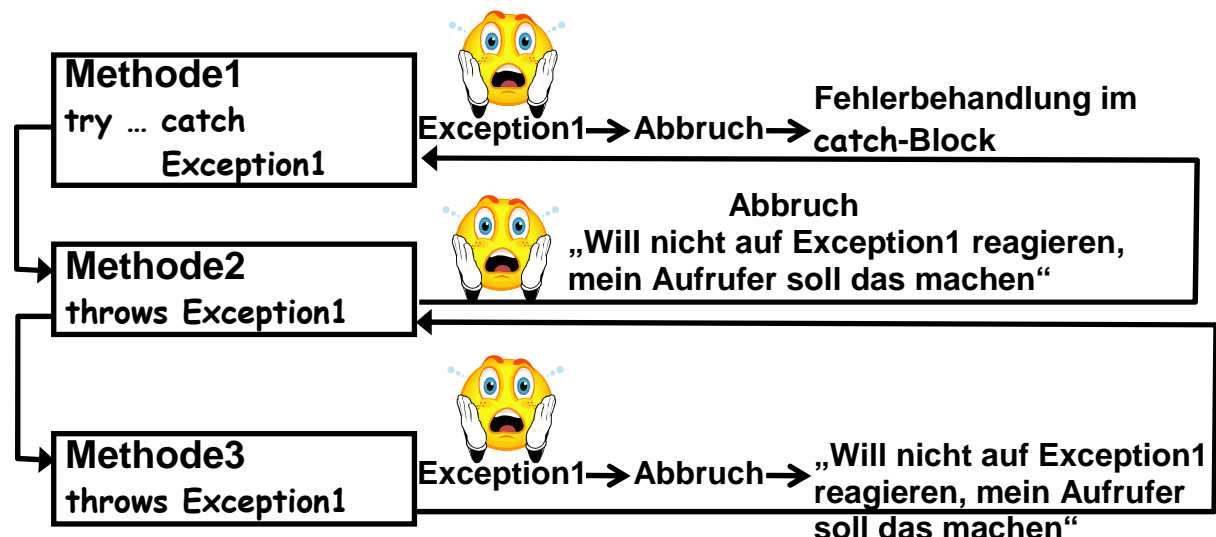
```

/**
 * Creates a File based on the Parameter Values
 * @param path File Path
 * @param name File Name
 * @return true if successful else false
 * @throws IOException in case of I/O Error
 */
public boolean createFile(String path, String name)
                        throws IOException {
    return (new File(path+File.separator+name).createNewFile());
}

```

**throws im
Methodenkopf: Die
Ausnahme soll an
den Aufrufer
weitergereicht
werden.**

Ablauf:

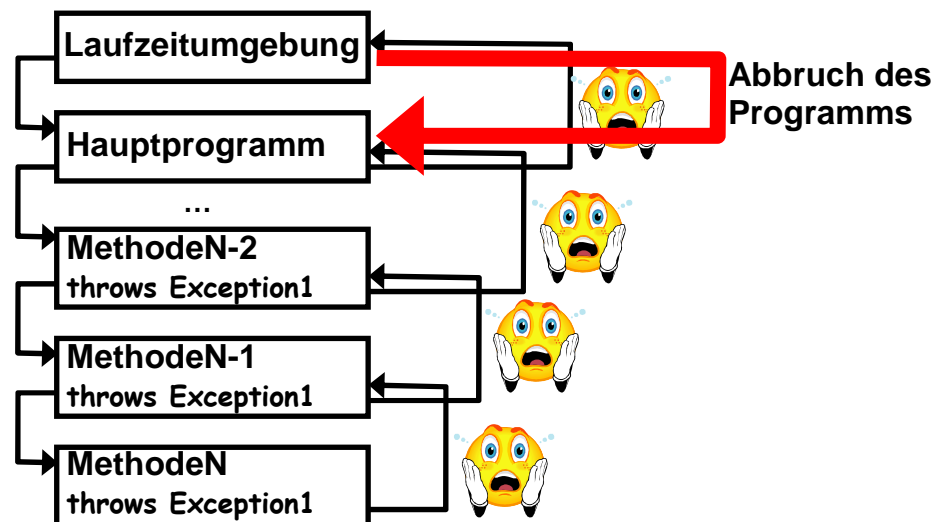


Methode1 ruft Methode2, die wiederum Methode3 ruft.

In Methode3 tritt eine Exception auf. Da Methode3 im Methodenkopf via `throws` anzeigt,

dass eine solche Exception (**Exception1**) nicht in der Methode behandelt wird, sondern an den Aufrufer weitergeleitet werden soll, erreicht die Exception Methode2.

In der aufrufenden Methode2 kommt **Exception1** an und wird dort ebenfalls an den Aufrufer weitergereicht. Methode1 fängt schließlich Exceptions vom Typ **Exception1** im **catch**-Block.



Im Beispiel oben ist dargestellt, wie die Laufzeitumgebung in Java die Klasse mit **main()** unseres Programms startet. Von dort aus werden nach und nach weitere Methoden gerufen. MethodeN wirft schließlich eine Ausnahme, die aufgrund der **throws**-Anweisung im Methodenkopf an die Aufrufermethode MethodeN-1 weitergereicht wird. Diese verfährt ebenso. Wenn das Hauptprogramm ebenfalls keine Fehlerbehandlung durchführt, landet die Ausnahme schließlich bei der Laufzeitumgebung. Diese bricht daraufhin das Programm ab.

Wo sollte eine auftretende Exception behandelt werden?

Die Antwort liegt im Verantwortlichkeitsprinzip: Eine Exception sollte dort behandelt werden, wo sie logisch zugehörig ist. Es ist demnach eine Entwurfsentscheidung.

Fehlerbehandlung und Substitutionsprinzip:

Ausnahmen sind Informationen, die aus einer Methode bzw. Klasse herauskommen.

Sie können daher in dieser Eigenschaft mit Rückgabewerten verglichen werden.

Wir haben vorn bereits gesehen, dass kovariante Rückgabetypen das Substitutionsprinzip erfüllen. Demnach erfüllt auch eine Kovarianz bei den Ausnahmen das Substitutionsprinzip.

In Java: Für **throws** bei überschriebenen Methoden gilt Kovarianz:

Überschriebene Methoden in einer Unterklasse dürfen nicht mehr Ausnahmen in der **throws**-Klausel deklarieren als schon bei der **throws**-Klausel der Oberklasse aufgeführt sind.

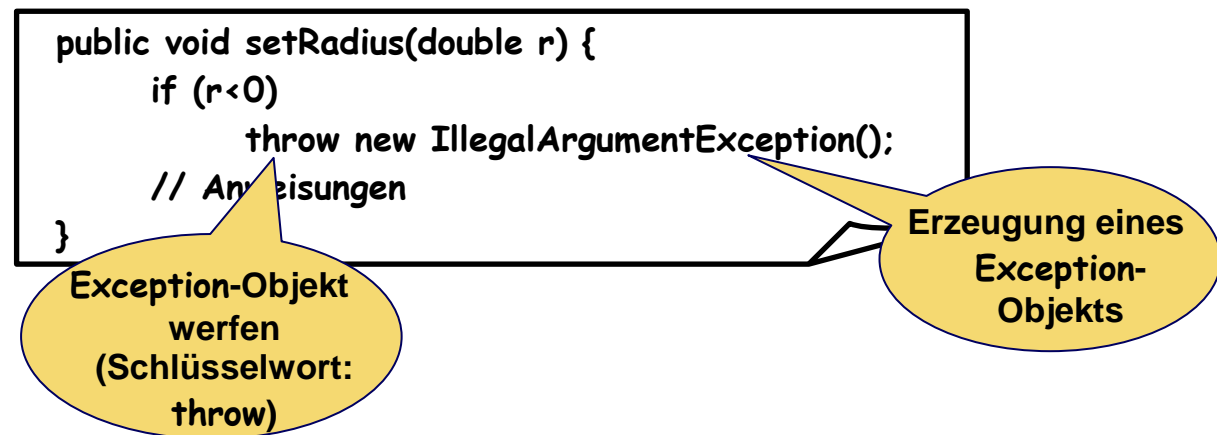
Das würde ansonsten gegen das Substitutionsprinzip verstoßen.

Eine Methode der Unterklasse kann damit:

- dieselben Ausnahmen wie die Oberklasse auslösen,
- Ausnahmen spezialisieren,
- Ausnahmen weglassen.

Fall 2: Eigene (neue) Exception auslösen

Beispiel für eine Methode in Java, die selbst eine Ausnahme auslösen soll:



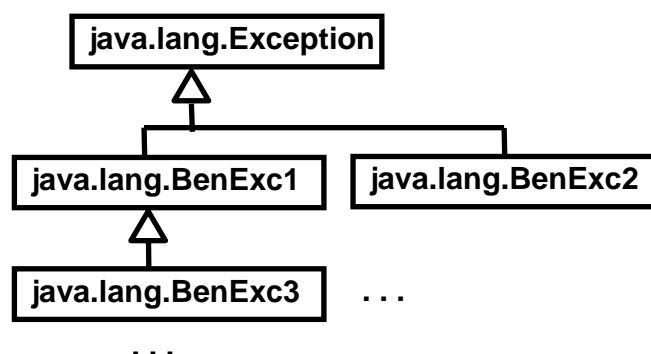
Beim Erzeugen eines `Exception`-Objekts kann auch eine Fehlermeldung angegeben werden (Custom Constructor), z.B.:

`throw new IllegalArgumentException("Fehlermeldung");`

In Java gibt es bereits den Exception-Typ `IllegalArgumentException`. Er kann daher direkt verwendet werden. Gelegentlich ist kein passender Exception-Typ vorhanden. In diesen Fällen kann ein eigener Exception-Typ aufgebaut werden.

Eigene Exception-Typen in Java aufbauen:

- Zur klaren Unterscheidung der verschiedenen Ausnahmesituationen sollten verschiedene, jeweils passende Exception-Typen eingesetzt werden.
- Realisierung: Direkte oder indirekte Unterklasse von Klasse `Exception`.



- In der neuen Unterklasse sollten zwei Konstruktoren implementiert werden:
 1. ohne Parameter,
 2. mit formalem Parameter vom Typ `String`.
- Man sollte nicht zu inflationär mit den Exception-Hierarchien umgehen (d.h. möglichst die schon vorhandenen Standard-Ausnahmen nutzen).

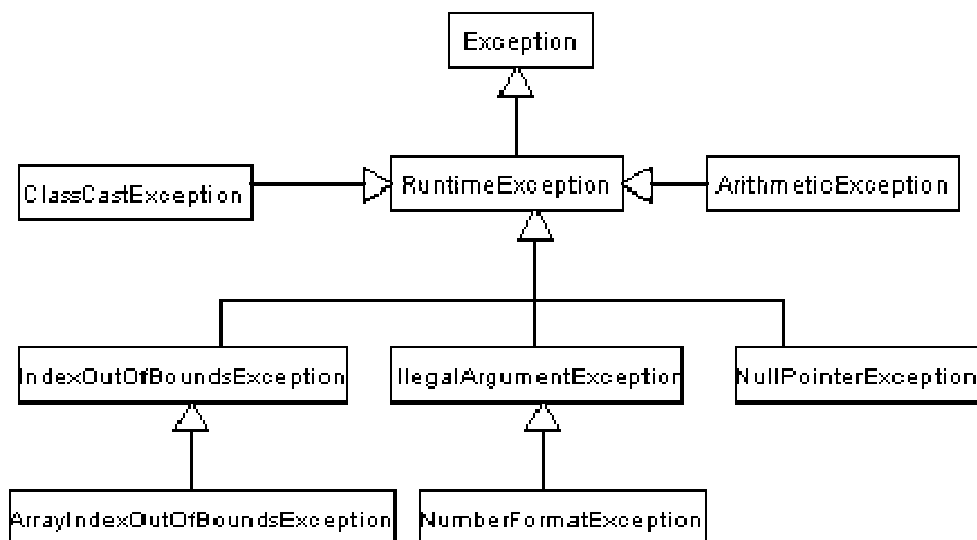
4.3.4 Besondere Exception in Java: RuntimeException

- Eine Ausnahme vom Typ **RuntimeException** beschreibt Fehler, die vom Programmierer behandelt werden können, aber nicht müssen:
 - **RuntimeExceptions** müssen nicht aufgefangen / behandelt werden.
Im Gegensatz zu den übrigen Exception-Typen werden sie dadurch auch *Unchecked Exceptions* genannt (versus *Checked Exceptions*).
 - Eine **RuntimeException** muss nicht in der **throws**-Klausel angegeben werden (kann aber zur Dokumentation).
- Einsatz: Für Fehlerarten, die an potenziell vielen Programmstellen auftreten (z.B. Division durch Null, ungültige Indexwerte beim Zugriff auf Array-Elemente).
- Im Programm sollte man **RuntimeExceptions** nicht generell abfangen, sodass das Programm dadurch grundsätzlich trotz Fehler laufen würde.
- Der Name **RuntimeException** ist seltsam gewählt, da alle Ausnahmen immer zur Laufzeit erzeugt, ausgelöst und behandelt werden.

Beispiele von häufig vorkommenden **RuntimeExceptions**:

Unterklasse von RuntimeException	Was den Fehler auslöst
ArithmeticException	Division durch Null.
ArrayIndexOutOfBoundsException	Indexgrenzen missachtet.
ClassCastException	Die Typanpassung ist zur Laufzeit nicht möglich.
IllegalArgumentException	Eine häufig verwendete Ausnahme, mit der Methoden falsche Argumente melden.
NullPointerException	Falscher Zugriff auf eine Referenz mit dem Wert null .

Klassenhierarchie in Java (Auszug): **RuntimeException**



4.4 Fehler (engl. Errors)

Fehler, die von der Klasse `java.lang.Error` abgeleitet sind, sind schwere Fehler, die in der Regel mit der JVM in Verbindung stehen.

- Beispiele: `AssertionError`, `AWTError`, `ThreadDeath`, `VirtualMachineError`, `InternalError`, `OutOfMemoryError`, `StackOverflowError`, `UnknownError`
- Es ist möglich, ein `Error`-Objekt mit `try-catch` aufzufangen, da `Error`-Klassen Subklassen von `Throwable` sind und sich daher genauso verhalten.
- Das Auffangen ist in der Regel nicht sinnvoll, denn wenn die JVM einen Fehler anzeigt, bleibt offen, wie darauf sinnvoll zu reagieren ist. Das Programm sollte abgebrochen werden.
- Es gilt die starke Empfehlung: Man sollte keine Unterklassen von `Error` bauen.

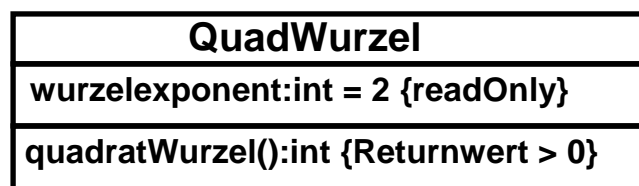
4.5 Zusicherungen u. andere Erweiterungen in UML

Um zusätzliche Informationen wie beispielsweise Zusicherungen und sonstige Einschränkungen in UML darzustellen, sind in erster Linie drei Wege möglich. Auf diese Weise können Ergänzungen dargestellt werden, die in UML nicht speziell spezifiziert sind.

1. Nutzung von Property Strings (Eigenschaftswerte)
2. Einsatz von Stereotypen (z.B. <<interface>>)
3. Nutzung von Annotationen bzw. Kommentaren

Eigenschaftswerte:

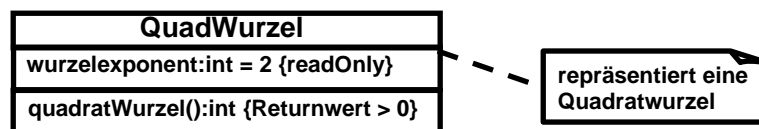
- Beispiel einer Einschränkung als Eigenschaftswert bzw. *Property String* (= Zusicherung, die die Operation erfüllen muss):
Die Operation sichert zu, dass ihr Ergebniswert größer als Null ist.



- Beispiele für andere in UML vordefinierte Eigenschaftswerte:
 {query}: drückt aus, dass die Operation keine Daten des Systems ändert.
 {ordered}: drückt aus, dass die Werte des Ergebnisparameters (z.B. vom Typ **Array**) geordnet sind.
- Für Eigenschaftswerte an Methoden und Attributen gilt das gleiche Darstellungsprinzip (z.B. {readOnly}, {ordered} bzw. wenn beide erfüllt sein sollen durch Kommatrennung: {readOnly, ordered}).

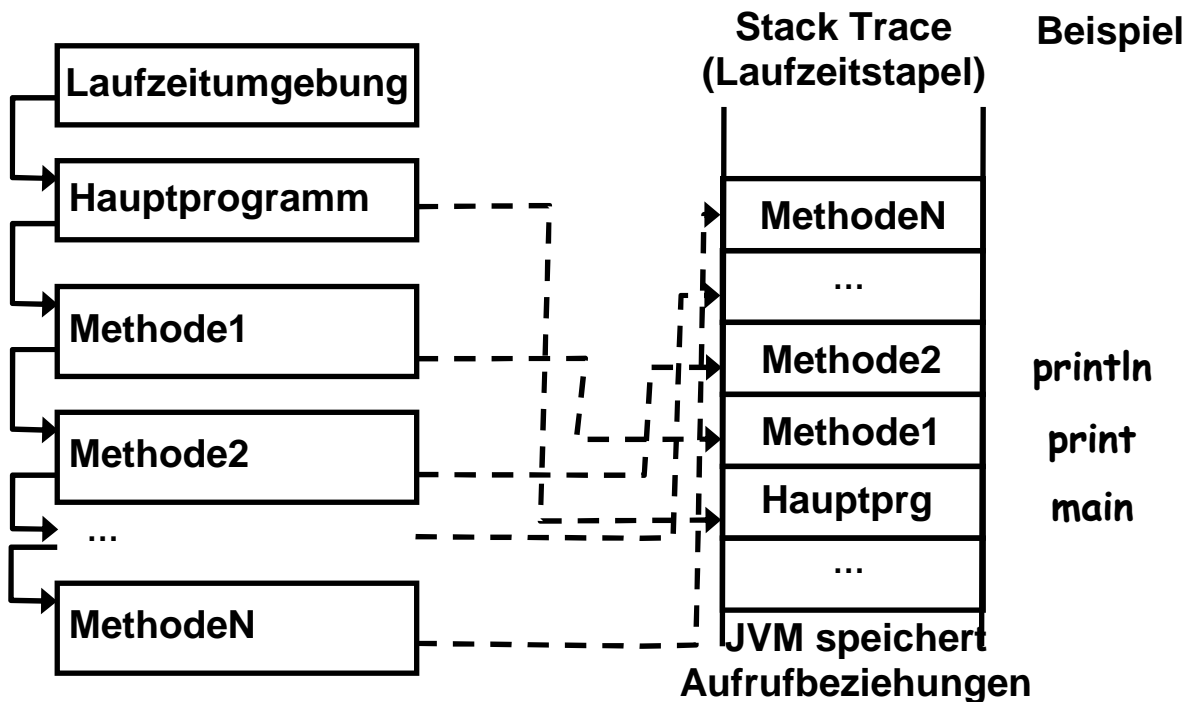
Annotation / Notiz / Kommentar:

Annotationen werden in erster Linie zur Dokumentation verwendet. Sie haben keine semantische Wirkung (sind also nur Zusatzbeschreibungen).



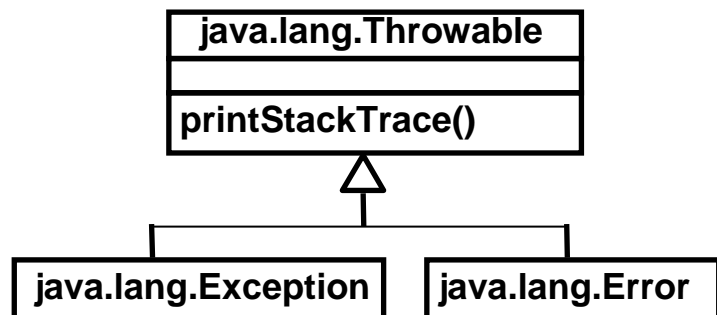
4.6 Laufzeitstapel (engl. Stack Trace)

Der *Stack Trace* speichert und protokolliert die Aufrufgeschichte während des Programmverlaufs.



Aus der Aufrufgeschichte können Schlüsse über den Programmablauf (z.B. zur Analyse eines Fehlers) mit Hilfe der Klasse **Throwable** abgeleitet werden.

Throwable erlaubt mit ihren Instanzmethoden den Zugriff auf den aktuellen Stack Trace (z.B. mit den Informationen: Dateiname, Methodenname, Programmzeile).



Betrachtung des Stack Trace über zwei Möglichkeiten:

- Bei einem Fehler wegen einer Ausnahme: Befragung des **Exception**-Objekts
- Falls keine Exception vorhanden ist:

```
StackTraceElement[] trace =
    new Throwable().getStackTrace();
```

