Seminarbericht zum Vortrag

„Exception Handling in Java“

im Rahmen der Lehrveranstaltung:
„Seminar aus Softwareentwicklung“

Wintersemester 2003/04

Autor: Wiener Norbert
MatrNr.: 9956158
SKZ: 880

Inhaltsverzeichnis:

1. EINLEITUNG: 3
2. AUSNAHMEBEHANDLUNG: 3
2.1 Typen von Ausnahmen: 4
2.2 Auslösen von Exception: 4
2.3 Konstrukte des Exception Handlings: 5
2.5 Die throws Klausel 6
2.6 Erstellen eigener Exception-Klassen: 7
2.7 Die Hierarchie der Ausnahme-Klassen: 8
2.8 Exception Table 10
3. INTERNE ABLÄUFE BEI EINER EXCEPTION: 10
3.1 Beispiel mit throw-Anweisung: 11
3.2 Interne Abläufe bei einer Exception mit finally: 14
3.3 Beispiel mit finally-Anweisung: 15
4. LITERATUR: 17

1. Einleitung:

Da Programme zunehmend komplizierter werden, wird es immer schwieriger, sie stabil zu erhalten. Traditionelle Programmiersprachen, wie zum Beispiel C, verlassen sich sehr stark darauf, dass IF-Statements abnorme Zustände entdecken, GOTO-Statements zu den Fehlerbehandlungsroutinen verzweigen und kryptische Rückgabewerte diese abnormen Zustände an die aufrufenden Methoden zurückliefern. Dadurch wird der Programmablauf unter einem Netz aus Statements zur Ausnahmeerkennung und Ausnahmebehandlung begraben.

Java bietet einen eleganten Weg an, um Programme zu erstellen, die sowohl robust als auch klar sind. Dazu verwendet Java einen Mechanismus zur Ausnahmebehandlung, der dem von C++ ähnelt.

2. Ausnahmebehandlung:

Exceptionhandling sollte aus mehreren Gründen verwendet werden:

- Saubere Trennung zwischen Programmlogik und Fehlerbehandlung. Das erhöht die Lesbarkeit des Codes und die Wartungsfähigkeit des Programms erheblich.

- Programme robuster machen gegen verschiedene Ausnahmesituationen (falsche Benutzereingabe, Gerätefehler, Programmierfehler z.B. Division durch 0).

- Fehlerbehandlung durch Compiler überprüfbar machen. Speziell die explizit geworfenen Exception müssen im Programm abgefangen werden. Bei Missachtung dieser Vorgabe kommt es beim Compilieren zu einer Fehlermeldung.

- Höhere Performance der Programmlogik. Jede zusätzliche Fehlerbehandlung (zusätzlicher Catch-Block) führt zu keiner Verlängerung der Programmlogik. Würde aber die Fehlerbehandlung mit Hilfe von IF-ELSE-Statements in der Programmlogik durchgeführt, so würde das immer zu einer Erhöhung der Programmlaufzeit führen.

2.1 Typen von Ausnahmen:

- Kontrollausnahmen: z.B. bei IO-Probleme, Anwenderfehler.
- Laufzeitfehler: z.B. Division durch 0, Grenzüberschreitung in Arrays.
- Programmfehler: z.B. OutOfMemoryError, IntenalError.

2.2 Auslösen von Exception:

Exception können auf zwei verschiedene Arten ausgelöst werden:

1. **Mittels THROW-Anweisung (= Explizite Ausnahme):**

   Es gibt oft Situationen, dass an einer bestimmten Programmstelle Blockaden auftreten (z.B. Zugriff auf Server, Öffnen einer Datei) oder die Programmlogik erkennt eine falsche Benutzereingabe.
   Hier bietet Java eine throw-Anweisung an, mit der der Programmierer eine Exception werfen kann. Die Syntax sieht folgendermaßen aus: „**throw exceptionObject**" .
   Üblicherweise wird das exceptionObject nach der throw-Klausel neu angelegt und mit der entsprechenden Fehlermeldung initialisiert, wie das Beispiel unten zeigt:

   ```
   setPassword(String pw) {
     ......
     if(pw.length() < 5)  throw new Exception(„mind. 5 Zeichen");
     .......
   }
   ```

   Diese explizit geworfene Ausnahme muss entweder noch in der selben Methode oder, bei entsprechender Änderung des Methodenkopfes, in der aufrufenden Methode abgefangen werden. (dazu später mehr)

2. **Implizite Ausnahmen werden von der JVM ausgelöst:**
   Erkannt werden diese Fehler aufgrund ungültiger Operationen (z.B. Division durch 0 oder Zugriff über ein null-Pointer). Weitere Ursachen für implizite Ausnahmen sind fatale Errors, die das Fortfahren der Programmausführung unmöglich machen. InternalError, VirtualMachineError usw. könnten jederzeit auftreten,

daher ist eine zwingende Fehlerbehandlung bei den impliziten Ausnahmen nicht sinnvoll und wird von Java nicht verlangt. Java sieht nur bei den expliziten Ausnahmen eine zwingende Ausnahmebehandlung vor.

Bei diesem Beispiel wird eine impliziete Ausnahme von der JVM ausgelöst, falls die Methode mit j=0 aufgerufen wird.

```
int intDiv (int i, int j) {
    return i/j;
}
```

Hier kann der Programmierer selbst entscheiden, ob er eine Ausnahmebehandlung macht. Sinnvollerweise lässt man sie weg, wenn die Programmlogik in der aufrufenden Methode eine 0-Zuweisung der Variable j ausschließen kann.

2.3 Konstrukte des Exception Handlings:

Die Struktur einer Ausnahmebehandlung beginnt mit dem Schlüsselwort try gefolgt von einem geschützten Anweisungsblock. Gleich anschließend folgen eine oder mehrere Catch-Blöcke. Der finally-Block kann noch optional hinzugefügt werden.

```
try{
   Anweisungen im geschützten Bereich
} catch ( ExceptionType1 exceptionVar1){
   exceptionHandler1
} catch ( ExceptionType2 exceptionVar2){
   exceptionHandler2
} finally {
   finallyBlock
}
```

Frühe Versionen des JDK (vor 1.0.2) benötigten keine geschweiften Klammern im Rumpf eines try-catch-finally-Konstrukts, falls der Rumpf des Programms aus einem einzigen Statement gestand. Seit JDK 1.0.2 sind geschweifte Klammern jedoch immer notwendig.

Alle im try-Block befindlichen Anweisungen leiten im Fehlerfall eine Suche nach einer passenden Ausnahmebehandlung ein. Dabei beginnt die Suche unmittelbar nach dem try-Block mit dem ersten Catch-Block. Ein Exceptionhandler gilt aus gefunden, wenn das geworfene ExceptionObjekt dem ExceptionTyp des jeweiligen Catch-Blocks entspricht, bzw. von diesem abgeleitet ist. Falls ein finally-Block vorhanden ist, wird dieser immer ausgeführt. Wenn kein geeigneter catch-Block gefunden wurde, setzt die Suche in der aufrufenden Methode (Vatermethode) fort.
Die Anordnung der catch-Blöcke ist entscheidend und sollte so erfolgen, dass zuerst die speziellen Ausnahmefehler und unten die allgemeinen Fehlertypen folgen. Wird diese Regel missachtet, erscheint beim Compilieren eine Meldung.

2.5 Die throws Klausel

Wenn eine explizite Exception geworfen wird, kann sie im einfachsten Fall in der selben Methode behandelt werden. Soll aber die Ausnahmebehandlung in der anderen Methode (z.B. in der aufrufenden Methode) durchgeführt werden, ist dies im Methodenkopf entsprechend zu deklarieren. Die ungeprüften Ausnahmen, also jene Fehler, die von der JVM ausgelöst werden, sind von der Deklarationspflicht ausgenommen.
Eine throws-Klausel besteht aus dem Schlüsselwort throws und eine Liste von Typen, die alle Throwable oder Subtypen von Throwable sein müssen.

Hier ein kleines Beispiel dazu:

```
static setPassword(String pw)  throws Exception {
    if(pw.length() < 5){
      throw new Exception(„Fehler“);
    }
    .......
}
```

```
..........
Try{
    setPassword(pw);
}catch(Exception e){
    Ausnahme behandeln!
}
.........
```

Das Weiterleiten der Ausnahmebehandlung kommt in der Praxis sehr oft vor. Vor allem in den Bibliotheksklassen werden die Fehlerbehandlungen an die aufrufenden Methoden delegiert, weil diese Methoden von den Anwendungsprogrammierern implementiert werden.
Wenn beim Zugriff auf einen Server die UnknownHostException geworfen wird, so könnte eine falsche Adresse über die Benutzereingabe die Fehlerursache sein. Es kann aber auch eine Unterbrechung der LAN-Verbindung die Exception auslösen. Daher ist es auch sinnvoll, die Fehlerbehandlung dem Anwendungsentwicker zu überlassen, weil nur er über mögliche Fehlerursachen bescheid weiß.

2.6 Erstellen eigener Exception-Klassen:

In Java besteht auch die Möglichkeit, zu den bereits vordefinierten Ausnahmeklassen, eigene Exception - Klassen zu erstellen. Zu Entscheiden, wann die Definition einer neuen Exception sinnvoller ist als die Anwendung einer bereits existierenden, ist meist schwieriger als das Schreiben selbst.
Die Definition eigener Exceptions unterscheidet sich nur wenig von der Deklaration anderer Java-Klassen. Um nun eine neue Exception zu implementieren, muss man lediglich von einer Klasse der Exception - Hierarchie ableiten. Dies kann direkt von Throwable erfolgen, sollte allerdings entweder von Exception oder einer ihrer Subklassen geschehen. Damit werden die neuen Ausnahmeklassen ebenfalls als Fehlerobjekte für benutzerdefinierte Ausnahmen gekennzeichnet. Abzuraten ist hier eine Ableitung von der Error-Klasse, da diese auf schwere Ausnahmefehlern in der JVM verweisen.
Es empfiehlt sich, die Referenz der Exception-Hierarchie samt ihrer Methoden zu studieren, um eine Übersicht der enthalten Operationen vor Augen zu haben. Die Throwable-Klasse definiert unter anderem einen parameterlosen Konstruktor und einen mit einem Stringparameter für die Fehlerbeschreibung.

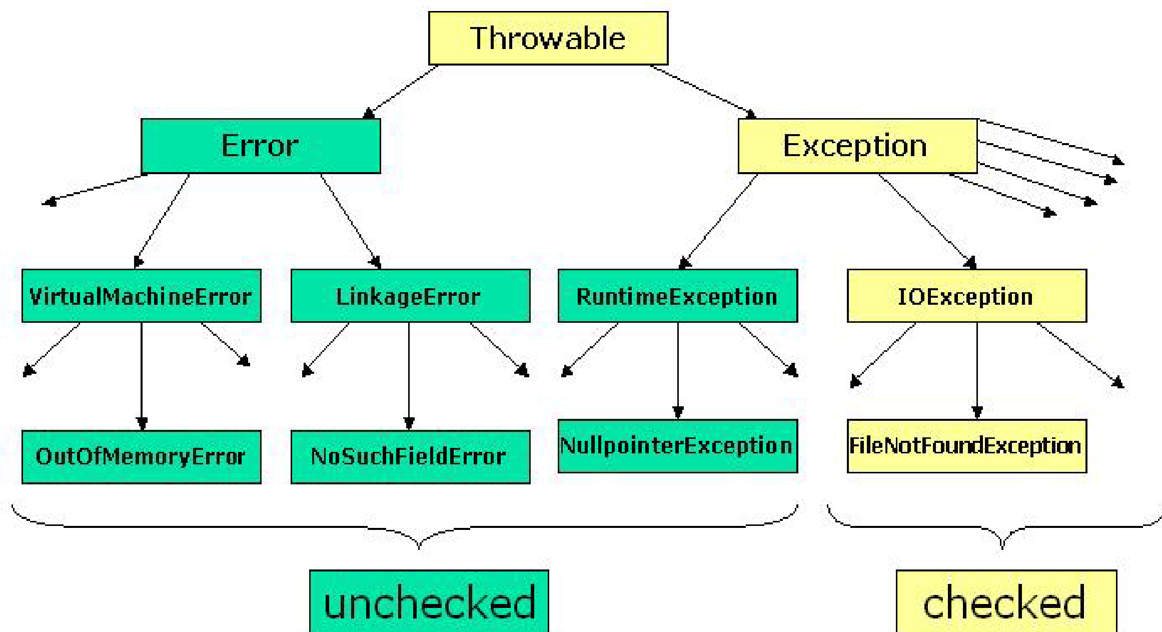
Das folgende Beispiel definiert eine neue Ausnahme. Diese leiten wir gleich von Exception ab und überschreiben sowohl den Standardkonstruktor ohne Parameter als auch den mit Fehlerbeschreibung.

```
class MyException extends Exception{
   public MyException() {
   }
   public MyException(String msg) {
      super(msg);
   }
}
```

2.7 Die Hierarchie der Ausnahme-Klassen:

Der Ursprung aller Klassen von Ausnahmen ist die Klasse *Throwable*, die eine unmittelbare Unterklasse der Klasse *Object* ist.

Hier ein kleiner Auszug der Exception-Hierarchie:



In der Klasse Throwable befinden sich einige Methoden für die Fehlerausgabe (getMessage(), toSting(), printStackTrace() usw.).
Die Klasse Throwable hat zwei unmittelbare Unterklassen, nämlich die Klasse Error und die Klasse Exception. Die Unterklassen der Klasse Exception sind an der Namensendung Exception erkennbar. Die Unterklassen der Klasse Error haben dagegen die Namensendung Error.

Ausnahmen des Typs Error sind normalerweise katastrophal und nicht vernünftig zu behandeln. In den meisten Fällen werden die Applikationen mit solchen Error-Fehlern nicht alleine fertig und werden sie deswegen auch nicht behandeln. Vielmehr obliegt die Abarbeitung solcher Katastrophen der Systemprogrammierung, denn dafür ist es notwendig, in die Innereien des Laufzeitsystems und der Architektur der Sprache hinabzusteigen.

Exception-Klassen:

Die Unterklassen der Klasse Exception gelten dagegen grundsätzlich als behebbar. Alle Unterklassen der Klasse Exception mit Ausnahme der Runtime-Klassen sind sogenannte CHECKED EXCEPTION. Diese geprüften Ausnahmen werden explizit geworfen und müssen einer Ausnahmebehandlung unterzogen werden. Oftmals werden die Exceptions in den Klassenbibliotheken ausgelöst (z. B. FileInputStream(..)), welche dann der Anwendungsprogrammierer in einer Applikation behandeln muss. Bei Missachtung dieser Regel würde beim Compilieren des Programms eine entsprechende Fehlermeldung erscheinen.

RuntimeException:

Laufzeitfehler können in nahezu jeder Methode auftreten, und der Versuch, sie alle abzufangen und zu behandeln, kann aufwendiger als der mögliche Nutzen sein. Java verlangt daher nicht, dass RuntimeException abgefangen werden. Es steht den Programmieren jedoch frei, dies zu tun. Das gleiche gilt für Ausnahmen vom Error-Typ.
Error und RuntimeException werden daher auch als ungeprüft (unckecked Exception) bezeichnet. Das bedeutet, der Compiler überwacht ihre Behandlung nicht, aber alle anderen Ausnahmen, auch Ausnahmen, die wir selbst von Throwable ableiten, werden von Compiler überwacht.

Hier einige wichtige Unterklassen der RuntimeException:

ArithmeticException: Für außergewöhnliche arithmetische Zustände, wie zum Beispiel die Division durch Null.

IndexOutOfBoundsException: Diese Exception wird ausgelöst, wenn der Index außerhalb des Wertebereichs liegt.

NullPointerException: weist auf einen Zugriff auf ein Feld oder einer Methode eines nicht existierenden Objekts hin.

StringIndexOutOfBoundsException: Tritt auf, bei einem Zugriff auf ein Zeichen in einem String oder StringBuffer mit einem Index, der kleiner als Null oder größer gleich der Länge des Strings ist.

2.8 Exception Table

In Java wird beim Compilieren eines Programms für jede Methode eine Exception-Table angelegt, in der alle Exceptionhandler der jeweiligen Methode eingetragen werden. Jede Zeile in der Tabelle entspricht einem Exceptionhandler (Catch-Block) und besteht aus vier Teilinformationen.
Die ersten 2 Einträge geben den zu überprüfenden Bereich (try-Block) an, wobei die End-Adresse nicht mehr hinzugezählt wird. (also im unterem Beispiel von 0 bis 3).
Der Type-Eintrag repräsentiert die Ausnahmeklasse, welcher der Exceptionhendler unterstützt. Der „hander" verweist auf den Anfang der entsprechenden Exceptionhandler (=Catch-Block).

Hier ein kleines Beispiel mit 2 Exceptionhandler:

```
void m() {
        try {
                a();
        } catch (ExO1 e) {
                h1();
        } catch (ExO2 e) {
                h2();
        }
}
```

start	end	handler	type
0	4	5	ExO1
0	4	10	ExO2

```
 0: aload 0
 1: invokevirtual <a()>
 4: return
 5: pop
 6: invokevirtual <h1()>
 9: return
10: pop
11: invokevirtual <h2()>
14: return
```

3. Interne Abläufe bei einer Exception:

Wenn in Java eine Methode aufgerufen wird, so wird ein neuer FrameStack erzeugt, der sämtliche Daten unter anderem auch die Referenzadresse der neu erzeugten Exception-Table enthält.
Das bedeutet, dass zu jeder Methode (Frame) eine eigene Exception-Table zugewiesen wird.
Jeder Catch-Block in der Methode erzeugt jeweils einen Eintrag in der Exception-Table. Die Reihenfolge der Einträge in der Exception Tabelle entspricht der Anordnung der Catch-Blöcke in der Methode.

Beim Wurf einer Ausnahme werden folgende Schritte durchgeführt. Zuerst wird ein Exception-Objekt auf dem Heap angelegt. Anschließend wird die zugeordnete Exception-Tabelle selektiert und die Suche nach dem geeigneten Exceptionhandler wird eingeleitet.
Da die Suche von oben nach unten erfolgt, müssen oben die allgemeineren Ausnahmefehler und unten die speziellen Ausnahmefehler aufgelistet sein. Ein gültiger Exceptionhandler muss folgende Bedingungen erfüllen:

- die Adresse, an der die Exception geworfen wurden, muss innerhalb des Try-Blocks liegen, also zw. Start- und End-Adresse.

- das erzeugte Exception-Objekt muss mit dem ExceptionTyp in der Exception-Table kompatibel sein. (d.h. der Typ des Exception-Objetes muss gleich oder eine Unterklasse des ExceptionTyp in der Exception-Table sein.)

Nachdem ein geeigneter Exceptionhandler gefunden wurde, leert die JVM den Operanden-Stack und gibt die Referenzadresse des Exception-Objektes auf dem leeren Operanden-Stack. In weiter Folge wird der Programcounter auf die Handler-Adresse gesetzt und die Programmabarbeitung setzt somit an dieser Stelle fort. (= Beginn des entsprechenden Exceptionhandler).

Enthält die Exception-Table keinen passenden Eintrag, so wird die Suche nach der geeigneten Ausnahmebehandlung in der Exception-Table der aufrufenden Methode fortgesetzt, dabei werden in der JVM folgende Aktionen ausgeführt:
Der Programcounter wird auf die Rücksprungadresse des aktuellen Frames gesetzt (= Adresse der aufrufenden Methode) und der aktuelle Frame wird entfernt. Nun beginnt die Suche erneut bis der passende Exceptionhandler gefunden wird, oder das Programm bricht nach erfolgloser Suche in der Main-Methode ab.

3.1 Beispiel mit throw-Anweisung:

Das unten angeführte Beispiel zeigt die Vorgänge auf der Java Bytecode Ebene. Die Methode „handle" enthält einen Methodenaufruf („load") und zwei Exceptionhandler.
In der load-Methode könnte eine explizite Exception (falls from < to ist) oder eine implizite Exception (falls ein IO-Fehler bei System.in.read() auftritt) geworfen werden. Mittels throws-Anweisung wird die Fehlerbehandlung an die aufrufende Methode weiter delegiert.

```
void load (int from, int to)
        throws IOException, NoSuchElementException {
  if (from > to){
     throw new NoSuchElementException();
  }
  int data = System.in.read();
  …
}
```

Hier der Java-Bytecode von der oberen Methode:

```
 0: iload 1
 1: iload 2
 2: if_icmple +11
 5: new <NoSuchElementException>
 8: dup
 9: invokespecial <NoSuch....<init>>
12: athrow
13: getstatic <System.in>
16: invokevirtual <InputStream.read>
19: istore 3
20: …
```

Die ersten 3 Befehle werten die if-Bedingung aus, dabei werden die Methodenparameter auf den Operanden-Stack gelegt und miteinander verglichen. Bei „false" der If-Bedingung setzt das Programm an der Stelle 13 fort.

Fall 1, wenn die If-Bedingung erfüllt ist:
Bei Erfüllung der If-Bedingung wird eine explizite Exception geworfen. Dazu sind 4 Anweisungen erforderlich.
Mit new<NoSuchElementException> legt die JVM ein Objekt am Heap an und gibt die Referenzadresse auf den Operandenstack. Mit der dup-Anweisung wird die Referenzadresse auf den Operandenstack dupliziert. Die eigentliche Inizialisierung (Konstrukteraufruf) des ExceptionObjektes erfolgt mit der invokespecial<NoSuchElementException<init>> Anweisung.
Ab dieser Stelle ist das Exception-Objekt vollständig inizialisiert, und die Such nach dem passenden Exceptionhandler wird mit „athrow" eingeleitet. Aufgrund der leeren Exception-Table dieser Methode, wird die Suche in der Exception-Table der aufrufenden Methode (handle) fortgesetzt.

Fall 2, wenn die If-Bedingung nicht erfüllt ist:
Dann setzt das Programm an der Stelle 16 fort, von wo aus die JVM eine implizite Exception auslöst, falls irgendwelche IO-Fehler auftreten.
Die JVM führt dabei folgende Anweisungen eigenständig durch:

- Anlegen des Exception-Objektes am Heap,
- Initialisierung des Objektes
- Werfen der Exception (athrow)

An dieser Stelle beginnt jetzt die Suche nach dem entsprechenden Exceptionhandler, und auch hier wird die Suche an die aufrufenden Methode (=Vatermethode) weiter geleitet.

Hier der Code der aufrufenden Methode:

```
void handle (int x, int y){
    try{
        load(x, y);
    }catch (IOException ioe){
        System.err.print(ioe);
    }catch (NoSuchElementException nsee){
        System.err.print("No data");
    }
}
```

Hier der entsprechende Java Bytecode:

```
0: aload 0
1: iload 1
2: iload 2
3: invokevirtual <Ex.load>
6: return
7: astore 3
8: getstatic <System.err>
11: aload 3
12: invokevirtual <PrintStream.print>
15: return
16: pop
17: getstatic <System.err>
20: ldc "No data"
22: invokevirtual <PrintStream.print>
25: return
```

Diese Methode besitzt einen try-Block, von wo aus die load-Methode aufgerufen wird, und 2 Catch-Blöcke für die Fehlerbehandlung. Wenn die load-Methode ordnungsgemäß ausgeführt wird, so endet auch die handle-Methode an der Stelle 6 mit der return-Anweisung.
Tritt jedoch ein Fehler in der load-Methode auf, so setzt die Suche, wie bereits oben erwähnt, in dieser Methode fort.

Hier die Exception-Table der handle-Methode:

start	end	handler	type
0	**6**	**7**	IOException
0	**6**	**16**	NoSuchElementException
try		**catch**	

Fall 1:
Bei einer NoSuchElementException wird die JVM in der zweite Zeile der Exception-Table fündig. Der Programcounter bekommt die Adresse 16 zugewiesen, wo zu Beginn ein pop-Befehl ausgeführt wird, dass die Referenzadresse des ExceptionObjektes vom OperandenStack entfernt. Der Grund dafür ist, weil in dem catch-Block kein Zugriff auf das ExceptionObjekt erfolgt. Am Ende wird die Methode mit der return-Anweisung verlassen.

Fall 2:
Bei einer IOException erfüllt die erste Zeile in der dazugehörigen Exception-Table alle 2 Bedingungen (Exception-Objekt muss mit dem Type kompatibel sein und der Fehler muss im try-Block aufgetreten sein, also zw. 0 und exklusiv 6). Die Fehlerbehandlung beginnt daher an der Stelle 7 und endet mit der return-Anweisung an der Stelle 15.

3.2 Interne Abläufe bei einer Exception mit finally:

Der finally-Block wird auf jeden Fall durchlaufen, unabhängig von einer Exception im try-Block.

Das folgende Beispiel führt eine finally-Anweisung aus, und zeigt die internen Sprünge mittels jsr-Befehl auf der Java Bytecode Ebene.

An der Stelle 0 wird ein beliebiger Befehl ausgeführt (try-Block).
Der nächste jsr-Befehl führt zwei Schritte durch. Als erstes legt er die Adresse des darunter liegenden Befehls auf den Operanden-Stack (Adresse von return = 4) und springt anschließend an die Stelle 5. Mit astore_2 wird die auf dem Operanden-Stack gespeicherte return-Adresse

in die lokale Variable 2 gespeichert. An der Stelle 6 kann wiederum ein beliebiger Befehl ausgeführt werden und der letzte Befehl (ret 2) führt einen Sprungbefehl auf jene Stelle, deren Adresse in der lokalen Variable 2 liegt (=4). Der return-Befehl beendet die Methode.

```
void m() {
        try {
           ….
        } finally {
           …
        }
}
```

```
0       …
1       jsr 5
4       return
5       astore_2
6       …
7       ret 2
```

3.3 Beispiel mit finally-Anweisung:

Folgendes Beispiel soll den internen Ablauf einer Exception mit zusätzlichem finally-Block veranschaulichen:

```
void tryCatchFinally() {
        try {
                tryItOut();
        } catch (TestExc e) {
                handleExc(e);
        } finally {
                wrapItUp();
        }
}
```

Folgende Annahmen werden getroffen:

Die Methode „tryItOut“ wirft eine Exception (TestExc), welche im catch-Block behandelt wird, während dessen tritt in der Methode „handleExc(e)“ eine neue implizite Exception auf, die nicht in dieser Methode behandelt wird. Auch in der tryCatchFinally() gibt es keinen Execptionhandler für die implizite Exception. In diesem Fall muss gewährleistet sein, dass der finally-Block durchlaufen wird, bevor die Suche in der aufrufenden Methode fortgesetzt wird.

Hier der entsprechende Java Bytecode:

```
Method void tryCatchFinally()
  0     aload_0
  1     invokevirtual <Ex. tryItOut>
  4     goto 16
  7     astore_3
  8     aload_0
  9     aload_3
 10     invokevirtual <Ex. handleExc>
 13     goto 16
 16     jsr 26
 19     return
 20     astore_1
 21     jsr 26
 24     aload_1
 25     athrow
 26     astore_2
 27     aload_0
 28     invokevirtual <Ex. wrapItUp>
 31     ret 2
```

start	end	handler	type
0	4	7	handleExc
0	16	20	Throwable

TryItOut wirf eine Exception ohne entsprechenden Exceptionhandler. Die Suche wird in der „TryCatchFinally" fortgesetzt, und die JVM findet auch den richtigen Eintrag in der Exception-Table. Die Ausnahmebehadlung beginnt an der Stelle 7, von wo aus die Referenzadressen dieser Methode und die Adresse des ExceptionObjekts in die richtige Reihenfolge auf den OperandenStack gelegt werden. Anschließend wird die Methode „handleExc" aufgerufen.
Mit der Annahme, dass auch in der aufgerufenen Methode (handleExc) eine implizite Exception (z.B. Division durch 0) auftritt, die nicht abgefangen wird, muss die Suche wieder hier fortgesetzt werden. Die zweite Zeile in der ExceptionTable unterbricht die Suche und setzt die Programmausführung an der Stelle 20 fort. Bevor der finally-Block ausgeführt wird, erfolgt eine Zwischenspeicherung der Referenzadresse des ExceptionObjekts. Der jsr-Befehl funktioniert hier nach dem selbem

Schema wie oben bereits erläutert. Nach Abarbeitung des finally-Blocks, setzt die Programmausführung an der Stelle 24 fort. Von dort aus wird die zwischengespeicherte Referenzadresse auf dem OperandenStack gelegt. Die vorher abgefangene implizite Exception, welche einer Ausnahmebehandlung noch nicht unterzogen wurde, wird an dieser Stelle mittels „athrow" erneut geworden.
Nachdem kein passender Exceptionhandler gefunden wird und der finally-Block bereits durchlaufen wurde, geht die Suche an die nächste aufrufende Methode weiter.

4. Literatur:

- Bill Venners : Inside the Java 2 Virtual Machine, McGraw Hill, 1999
- Frank Yellin, Tim Lindholm: The Java Virtual Machine Specification, Addison-Wesley 1997
- http://java.sun.com/docs/books/jls/second_edition/html/jTOC.doc.html