Weiterführende Konzepte des Programmierens mit Java
Exceptions

HOCHSCHULE DER MEDIEN

Prof. Dr. Peter Thies
Prof. Dr. Christian Rathke
Hochschule der Medien (HdM)

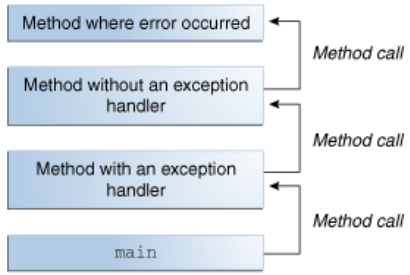
{thies|rathke}@hdm-stuttgart.de
http://www.hdm-stuttgart.de/{thies|rathke}

Exceptions

- *Exception* bedeutet "außergewöhnliches Ereignis", das bei Programmausführung auftritt und den normalen Ablauf des Programms unterbricht.
- Wenn eine solches Ereignis auftritt, erzeugt die gerade ausgeführte Methode ein sog. *Ausnahmeobjekt (exception object)* und übergibt es dem Laufzeitsystem.
- Das Ausnahmeobjekt enthält Informationen über die Ausnahmesituation, wie z.B.
  - den Ausnahmetyp und
  - den Programmzustand zum Zeitpunkt des Entstehens der Ausnahmesituation
- Erzeugen und Übergeben des Objekts wird als "*eine Ausnahme werfen" (throwing an exception)* bezeichnet.
- Das Laufzeitsystem versucht entlang der Aufrufkette der Methoden eine Methode zu finden, die mit dem Ausnahmeobjekt "umgehen" kann.

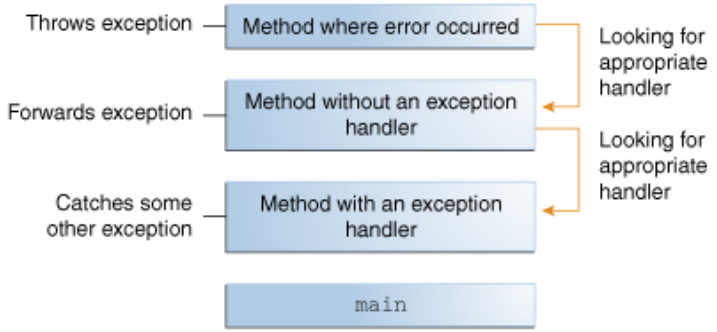
Die Aufrufkette von Methoden

- Suche nach einem Code-Block, der die Ausnahme behandeln kann: *Exception Handler*
- Übergabe des Ausnahmeobjekts an den Handler. Dies wird "die Ausnahme abfangen" (*catch the exception)* genannt.



Suchen nach einem passenden Exception Handler

- Falls kein passender Exception Handler gefunden wird, terminiert das Programm.



Die Catch-oder-Angabe-Anforderung

- Jeder Java-Code, der Ausnahmen werfen kann, muss eingeschlossen sein von
  a) einer try-Kontrollflussanweisung zum Auffangen der Ausnahme, oder
  b) einer Methode, die angibt, dass sie die Ausnahme (weiter-) werfen kann.
- Dies nennt man die *„Catch-or-Specify"-Anforderung*.
- Diese gilt jedoch nur für die erste der drei möglichen Ausnahmetypen:

Ausnahmetypen

1. Kontrollierte Ausnahmen (*checked exceptions),* z.B. das Fehlen einer zu öffnenden Datei. Diese müssen die „Catch-or-Specify"-Anforderung erfüllen.
2. Fehler (*errors)*: diese sind nicht der „Catch-or-Specify"-Anforderung unterworfen (Klasse **`Error`** und deren Subklassen), z.B. wenn von einer geöffneten Datei nicht gelesen werden kann, weil ein Hardware-Problem aufgetreten ist. Anwendungen können diese Ausnahme abfangen, müssen es aber nicht.
3. Laufzeitfehler (*runtime exceptions):* auch diese sind nicht der „Catch-or-Specify"- Anforderung unterworfen (Klasse **`RuntimeException`** und deren Subklassen), z.B. Zugriff auf ein nicht existierendes Feldelement.

- Die Fälle 2 und 3 nennt man auch "nicht kontrollierte Ausnahmen" (*unchecked exceptions)*

Das Werfen und Behandeln von Ausnahmen

Beispiel:

```
//Note: This class won't compile by design!
import java.io.*;
import java.util.List;
import java.util.ArrayList;

public class ListOfNumbers {

    private List<Integer> list;
    private static final int SIZE = 10;

    public ListOfNumbers () {
        list = new ArrayList<Integer>(SIZE);
        for (int i = 0; i < SIZE; i++) {
            list.add(new Integer(i));
        }
    }

    public void writeList() {
        PrintWriter out = new PrintWriter(
                              new FileWriter("OutFile.txt"));

        for (int i = 0; i < SIZE; i++) {
            out.println("Value at: " + i + " = " +
                        list.get(i));
        }

        out.close();
    }
}
```

Werfen einer (checked) IOException, falls die Datei nicht geöffnet werden kann.

Werfen einer (unchecked) ArrayIndexOutOfBoundsException, falls das Argument zu klein oder zu groß ist.

Der Try-Block

- Umschließen des Codes, der Ausnahmen werfen kann.

```
try {
    code
}
catch and finally blocks . . .
```

```
private List list;
private static final int SIZE = 10;

PrintWriter out = null;

try {
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = "
                    + list.get(i));
    }
}
catch and finally blocks . . .
```

Catch-Blöcke

Zurodnung von Exception Handlers zu einem Try-Block durch einen oder mehrere Catch-Blöcke:

```
try {

} catch (ExceptionType name) {

} catch (ExceptionType name) {

}
```

Ein Catch-Block enthält Code, der ausgeführt wird, wenn der betreffende Exception-Handler aufgerufen wird. Dies geschieht, wenn die geworfene Exception vom angegebenen Typ ist.

```
try {

} catch (IndexOutOfBoundsException e) {
    System.err.println("IndexOutOfBoundsException: "
                       + e.getMessage());
} catch (IOException e) {
    System.err.println("Caught IOException: "
                       + e.getMessage());
}
```


Catch-Blöcke (fortg.)

- Ausnahmebehandlung kann auch darin bestehen,
  - den Fehler zu beheben,
  - den Nutzer zu befragen oder
  - den Fehler zur nächsten Ausnahmebehandlung weiter zu reichen.
- ab Java SE7 kann mehr als ein Ausnahmetyp mit einer einzigen Ausnahmebehandlung verarbeitet werden:

```
catch (IOException|SQLException ex) {
    logger.log(ex);
    throw ex;
}
```


Der Finally-Block

- Der Finally-Block wird *immer* ausgeführt, nachdem der try-Block beendet ist, unabhängig davon, ob eine Ausnahme aufgetreten ist, sogar wenn er mit „return“, „continue“ oder „break“ beendet wurde.
- Im Beispiel sollte der geöffnete Stream auf jeden Fall geschlossen werden:

```
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close();
    } else {
        System.out.println("PrintWriter not open");
    }
}
```


Try-with-resources

- Try-Statement, das innerhalb eines runden Klammerpaars eine oder mehrere sog. *Resources* deklariert.
- Eine *Resource* ist ein Objekt, das nach Verwendung geschlossen werden muss.
- Jedes Objekt, das `java.lang.AutoCloseable` implementiert kann als Resource verwendet werden, z.B. Instanzen der Klasse `java.io.Closeable`.

```
static String readFirstLineFromFile(String path) throws IOException {
    try (BufferedReader br =
                   new BufferedReader(new FileReader(path))) {
        return br.readLine();
    }
}
```

- `BufferedReader` implementiert java.lang.AutoCloseable und deklariert daher eine Resource, die am Ende geschlossen wird.

Vollständiges Beispiel

Je nachdem, ob Ausnahmen geworfen werden, werden unterschiedliche Programmteile ausgeführt.

```
public void writeList() {
    PrintWriter out = null;

    try {
        System.out.println("Entering try statement");
        out = new PrintWriter(
                        new FileWriter("OutFile.txt"));
            for (int i = 0; i < SIZE; i++)
                out.println("Value at: " + i + " = "
                            + list.get(i));

    } catch (IndexOutOfBoundsException e) {
         System.err.println("Caught "
                    + "IndexOutOfBoundsException: "
                    +   e.getMessage());

    } catch (IOException e) {
         System.err.println("Caught IOException: "
                            +  e.getMessage());

    } finally {
         if (out != null) {
             System.out.println("Closing PrintWriter");
             out.close();

         }
         else {
             System.out.println("PrintWriter not open");
         }
     }
}
```


Angabe von Exceptions, die von einer Methode geworfen werden können, als Alternative zum Try-Block

- Anstelle des Abfangens von Ausnahmen, können diese auch durch Methoden "weitergeworfen" werden.

Da dies keine kontrollierte Ausnahme ist, könnte dies auch weggelassen werden.

```
// Note: This method won't compile by design!
public void writeList() throws IOException,
                                ArrayIndexOutOfBoundsException {
    PrintWriter out =
              new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++) {
        out.println("Value at: " + i + " = "
                    + vector.elementAt(i));
    }
    out.close();
}
```


Das Werfen von Ausnahmen

- Das Werfen von Ausnahmen erfolgt grundsätzlich mit dem Operator **`throw`**.
- Alle Ausnahme-Klassen sind Erweiterungen (Abkömmlinge, (in)direkte Subklassen) der Klasse **`Throwable`**.
- Es können damit eigene Ausnahme-Klassen erzeugt werden.

Die Throw-Anweisung

- Einziges Mittel, um Ausnahmen anzuzeigen.
- Einziges Argument ist ein "werfbares" Objekt
- Werfbare Objekte sind Instanzen einer Subklasse von **`java.lang.Throwable`**
- Beispiel:

Die throws-Angabe kann hier fehlen, da die Erzeugte Ausnahme ein nicht-kontrollierte Ausnahme (unchecked Exception) ist

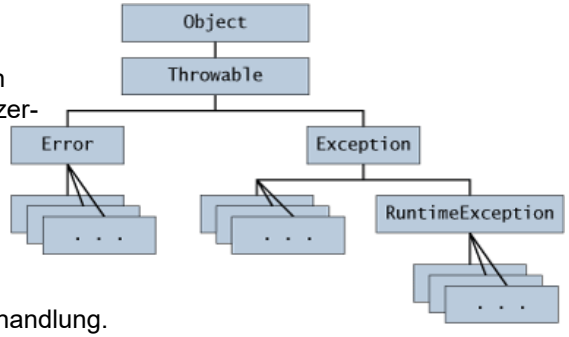
Teil des Packages java.util

```
public Object pop() {
    Object obj;

    if (size == 0) {
        throw new EmptyStackException();
    }

    obj = objectAt(size - 1);
    setObjectAt(size - 1, null);
    size--;
    return obj;
}
```

Die Klasse `Throwable` und ihre Subklassen

Die Klasse **`Error`**

- Fehler in der JVM
- Normalweise werden solche Fehler in Nutzer-programmen weder geworfen noch abgefangen.



Die Klasse **`Exception`**

- Übliche Klasse für die Ausnahmebehandlung.
- Wird im Java-API häufig erweitert.
- Die Klasse **`RuntimeException`** wird speziell dazu verwendet, nicht korrektes Verwenden einer API anzuzeigen (z.B. **`NullPointerException`**).

Verkettete Ausnahmen

- Entstehen durch Werfen von Ausnahmen in Catch-Blöcken
- Spezielle Unterstützung durch folgende Methoden und Konstruktoren von **`Throwable`**:

```
Throwable getCause()
Throwable initCause(Throwable)
Throwable(String, Throwable)
Throwable(Throwable)
```

Ergibt den Grund für die aktuelle Ausnahme

Grund der aktuellen Ausnahme

Ergibt die aktuelle Ausnahme

```
try {

} catch (IOException e) {
    throw new SampleException("Other IOException", e);
}
```


Zugriff auf die Methoden-Aufrufkette

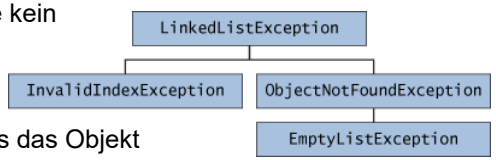
- Beispiel für eine Formatierung der Informationen, die von der Methode **`getStackTrace`** zur Verfügung gestellt werden.
- Ergebnis dieser Methode ist eine Feld bestehend aus den Gliedern der Aufrufkette.
- Jedes Kettenglied ist vom Typ **`StackTraceElement`**:

```
catch (Exception cause) {
for (StackTraceElement element : cause.getStackTrace()) {
        System.err.println(element.getFileName() + ":"
                        + element.getLineNumber()
                        + ">> "
                        + element.getMethodName() + "()");
    }
}
```


Das Definieren von Ausnahmeklassen

- Nutzen einer vorhandenen oder Definieren einer eigenen?
  - Braucht man eine eigene?
  - Würde man Nutzern damit helfen?
  - Tritt mehr als ein Ausnahmetyp auf?

- Beispiel: Eine Klasse für verkettete Listen (`LinkedList`) soll die folgenden Methoden haben:
  - **`objectAt(int n)`**: Objekt auf Position n; wirft eine Ausnahme, falls n<0 oder größer als die Anzahl
  - **`firstObject():`** Objekt auf der ersten Position; wirft eine Ausnahme, falls die Liste kein Objekt enthält
  - **`indexOf(Object o):`** Position von Objekt o; wirft eine Ausnahme, falls das Objekt nicht in der Liste ist.



Zusammenfassung

- Ausnahmen dienen zur Fehlerbehandlung
- Man verwendet die **`throw`**-Anweisung zusammen mit einerm Ausnahme-Objekt, um eine Ausnahme anzuzeigen
- Programme können geworfene Ausnahmen mit der Kombination von **`try-`**, **`catch-`** und **`finally-Blöcken`** behandeln:
  - **`try`** identifiziert einen Code-Block, in dem eine Ausnahme auftreten kann
  - **`catch`** identifiziert Code, der einen bestimmten Ausnahmetyp behandeln kann
  - **`finally`** identifiziert Code, der unabhängig vom Auftreten einer Ausnahme in jedem Fall ausgeführt wird.