ingenieur
wissenschaften
htw saar

Hochschule für
Technik und Wirtschaft
des Saarlandes
University of
Applied Sciences

Ausnahmebehandlung

Sprachelemente der Ausnahmebehandlung

```
try {
    Anweisungen und Ausdrücke, die Ausnahmen auslösen
   können  z. B.
           throw new Ausnahmetyp1();

}
catch (Ausnahmetyp1 a) {
    fängt Ausnahmen der Klasse Ausnahmetyp1 auf und
    kann sie behandeln
}
catch (Ausnahmetyp2 a) {
    fängt Ausnahmen der Klasse Ausnahmetyp2 auf und
    kann sie behandeln
}
finally {
    Abschlussaktionen nach dem Try-Block...
}
```

Mögliche Abläufe (1)

Fall 1: Im `try`-Block wird keine Ausnahme ausgelöst:

- Die `catch`-Routinen werden ignoriert.
- Eine eventuell vorhandene `finally`-Klausel wird ausgeführt.
- Das Programm wird hinter der `try`-Anweisung fortgesetzt.

Mögliche Abläufe (2)

Fall 2: Im `try`-Block wird eine Ausnahme ausgelöst:

- Eine Ausnahme wird entweder von der Virtual Machine ausgeworfen oder durch eine `throw`-Anweisung.
  - ⇨ Es wird dabei ein Objekt einer speziellen Ausnahmeklasse erzeugt.
- Wenn im `try`-Block eine Ausnahme ausgelöst wird, so wird an das Ende des `try`-Blockes verzweigt.
- Die dort aufgeführten `catch`-Routinen haben jeweils wie Methoden einen Parameter.
- Der Typ dieses Parameters wird mit dem Typ des ausgeworfenen Objektes verglichen.

Mögliche Abläufe (3)

Fall 2.1: Es gibt eine passende Ausnahmebehandlungsroutine nach dem `try`-Block

- Eine `catch`-Routine ist zur Ausnahme "passend", wenn ihr Parameter als Referenzvariable auf das Ausnahme-Objekt verweisen darf.
  - ⇨ Die erste passende `catch`-Routine wird ausgeführt.
- Wenn innerhalb der `catch`-Routine keine weitere Ausnahme ausgelöst wird, gilt die Ausnahme als behandelt.
- Eine eventuell vorhandene `finally`-Klausel wird ausgeführt.
- Es wird mit der Anweisung fortgesetzt, die auf die `try`-Anweisung folgt.
  - ⇨ Es erfolgt kein Rücksprung zur Programmstelle, wo die Ausnahme ausgelöst wurde.

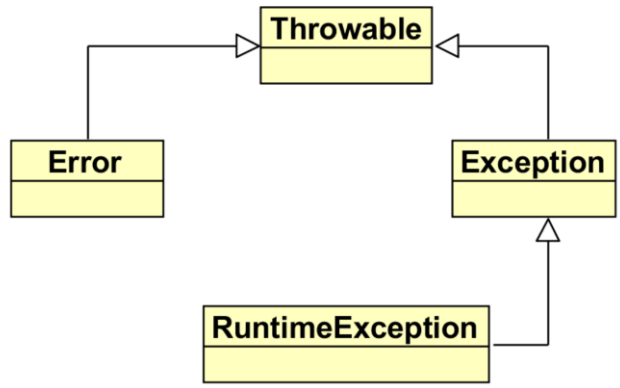
Mögliche Abläufe (4)

Fall 2.2: Es gibt keine passende Ausnahmebehandlungsroutine nach dem `try`-Block

- Eine eventuell vorhandene `finally`-Klausel wird ausgeführt.
- Die Kontrolle geht zum nächsten darüberliegenden `try`-Block über, der in der selben Methode aber auch in einer aufrufenden Methode sein kann.
- Dort wird nun dort nach einer passenden `catch`-Routine gesucht.
- Wird keine passende `catch`-Routine gefunden, so wird das Programm abgebrochen.

Die Java-Ausnahmeklassen

- Alle Ausnahmen in Java sind Unterklassen der Klasse `java.lang.Throwable`.
- `Throwable` ist eine allgemeine Ausnahmeklasse, die im wesentlichen eine Klartext-Fehlermeldung speichern und einen Auszug des Laufzeit-Stacks ausgeben kann.



Die Klasse java.lang.Error

- Die Klasse `java.lang.Error` ist Oberklasse aller schwerwiegenden Fehler.
- Diese werden hauptsächlich durch Probleme in der virtuellen Java-Maschine ausgelöst.
- Fehler der Klasse `Error` sollten in der Regel nicht abgefangen werden, sondern (durch die Standard-Fehlerbehandlung) nach einer entsprechenden Meldung zum Abbruch des Programms führen.
- Beispiele:
  `NoClassDefFoundError,`
  `OutOfMemoryError,`
  `StackOverflowError,`
  `VirtualMachineError, ...`
- aber leider auch: `AssertionError`

Die Klasse java.lang.Exception

- Alle Ausnahmen, die möglicherweise für die Anwendung selbst von Interesse sind, werden als Unterklasse von `java.lang.Exception` definiert.
- Beispiele:
  `ClassNotFoundException,`
  `CloneNotSupportedException,`
  `InterruptedException,`
  `IOException,`
  `SQLException, ...`

Die Klasse java.lang.RuntimeException

- Die Unterklassen der Klasse `java.lang.RuntimeException` sind spezielle Ausnahmen des Laufzeitsystems.
- Beispiele:
  `ArithmeticException,`
  `ClassCastException,`
  `IndexOutOfBoundsException,`
  `NegativeArraySizeException,`
  `NullPointerException,`
  `SecurityException, ...`

Methoden von `Throwable`

Methode	*Bedeutung*

`Throwable()`
`Throwable(String message)`
`Throwable(String message,`
`Throwable cause)`
`Throwable(Throwable cause)`
	Konstruktoren mit Meldung und/oder mit auslösender Ausnahme (`cause`)

`String getMessage()` `String getLocalizedMessage()`	Rückgabe der Meldung.

`void printStackTrace()`
`void printStackTrace`
`(PrintStream)`
`void printStackTrace`
`(PrintWriter)`
	Ausgabe des Ausführungsstacks.

`Throwable fillInStackTrace()`	Den aktuellen Ausführungsstack in das Ausnahmeobjekt eintragen (wird beim erneuten Auswerfen einer Ausnahme benötigt).

Methoden von `Throwable`

Methode	*Bedeutung*

`Throwable getCause()`
`Throwable initCause`
`(Throwable cause)`
	Grund für die Ausnahme erfragen bzw. neu setzen

`StackTraceElement getStackTrace()`
`void setStackTrace`
`(StackTraceElement[] stackTrace)`
	Direkter Zugriff auf den StackTrace

`String toString()`	Aufbereitung der Ausnahmeinformationen als String.

Beispiel: ExceptionTest1 (1)

```
/**
 * Einfache Tests mit Ausnahmen
 */
public class ExceptionTest1 {
    public static void main(String[] args) {
        try {
            throw new Exception("Dies ist eine Ausnahme");
        } catch(Exception e) {
            System.out.println("Ausnahme aufgefangen");
            System.out.println("e.getMessage(): "
                               + e.getMessage());
            System.out.println("e.toString(): "
                               + e.toString());
            System.out.println("e.printStackTrace():");
            e.printStackTrace(System.out);
        }
    }
}
```


Beispiel: ExceptionTest1 (2)

```
Ausnahme aufgefangen
e.getMessage(): Dies ist eine Ausnahme
e.toString(): java.lang.Exception: Dies ist eine Ausnahme
e.printStackTrace():
java.lang.Exception: Dies ist eine Ausnahme
       at ExceptionTest1.main(ExceptionTest1.java:11)
```

Beispiel: Ausnahme weiterleiten (1)

```
public class ExceptionTest2 {
    public void f() throws Exception {
        System.out.println("Beginn f()");
        if (true)  // Trick, damit der Compiler uebersetzt
            throw new Exception("f()-Ausnahme");
        System.out.println("Ende   f()");
    }

    public void g() throws Exception {
        System.out.println("Beginn g()");
        try {
            f();
        } catch(Exception e) {
            System.out.println("g(): e.printStackTrace()");
            e.printStackTrace(System.out);
            throw e; // Ausnahme weiterleiten
        }
        System.out.println("Ende   g()");
    }
```


Beispiel: Ausnahme weiterleiten (2)

```
 public static void main(String[] args) {
        System.out.println("Beginn main()");
        ExceptionTest2 et = new ExceptionTest2();
        try {
            et.g();
        } catch(Exception e) {
            System.out.println("main(): e.printStackTrace()");
            e.printStackTrace(System.out);
        }
        System.out.println("Ende   main()");
    }
}
```

Beispiel: Ausnahme weiterleiten (3)

```
Beginn main()
Beginn g()
Beginn f()
g(): e.printStackTrace()
java.lang.Exception: f()-Ausnahme
       at ExceptionTest2.f(ExceptionTest2.java:8)
       at ExceptionTest2.g(ExceptionTest2.java:15)
       at ExceptionTest2.main(ExceptionTest2.java:28)
main(): e.printStackTrace()
java.lang.Exception: f()-Ausnahme
       at ExceptionTest2.f(ExceptionTest2.java:8)
       at ExceptionTest2.g(ExceptionTest2.java:15)
       at ExceptionTest2.main(ExceptionTest2.java:28)
Ende   main()
```


Anwendung von finally (1)

```
public int f() {
    try {
        ...
        return 1;
    }
    finally {
        return 2;          // Rückgabe: 2
    }
}
```

Anwendung von finally (2)

```
while (Bedingung) {
    try { ...
        if (AbbruchBedingung)
           break;
    } finally {
        System.out.println ();
    }
// Vor Verlassen der Schleife wird noch
// ein Zeilenwechsel geschrieben
}
```


Geprüfte und ungeprüfte Ausnahmen

Ungeprüfte Ausnahmen:

- sind Ausnahmen, die von `Error` oder `RuntimeException` abgeleitet sind
- können jederzeit an jeder Programmstelle vorkommen
- müssen nicht explizit deklariert werden

Geprüfte Ausnahmen:

- sind alle sonstigen von `Exception` abgeleiteten Ausnahmen
- müssen in einer Methode entweder explizit behandelt werden oder im Kopf deklariert werden
- dazu wird die Methodendeklaration ergänzt um die Throws-Klausel:
  `throws` *`Ausnahmetyp1`* `[, Ausnahmetyp2, …]`
  `{ … }`

Eigener Ausnahmetyp

```
/**
 * Nicht-oeffentliche Klasse MyException, ist nur
 * innerhalb des Paketes bekannt und zugreifbar
 */
class MyException extends Exception {
    public MyException()         { super(); }
    public MyException(String s) { super(s); }
}
```


Deklaration von Ausnahmen (1)

```
public class ExceptionTest6 {

    /** Gepruefte Eingabe: Bediener darf nur Zahl zwischen
     *  0 und 100 eingeben. Die ausgeloeste Ausnahme muss
     *  deklariert werden
     */
 public int gepruefteEingabe() throws MyException {
        System.out.print("Zahl >= 0 und <=100 eingeben:");
        int zahl = input.nextInt();
        if (!(zahl >= 0 && zahl <= 100))
            throw new MyException(
                        "Falsche Zahl eingegeben: " + zahl);
        return zahl;
    }
```

Deklaration von Ausnahmen (2)

```
    public void teste()  {
        int zahl = -1;
        while (zahl != 0) {
            try {
                zahl = gepruefteEingabe();
                System.out.println(zahl + " eingegeben\n");
            }
            catch (MyException e) {
                System.out.println(e + "; zahl = "
                                        + zahl +"\n");
            }
        }
    }
```


Deklaration von Ausnahmen (3)

```
    public static void main(String args[]) {
        ExceptionTest6 et = new ExceptionTest6();
        try {
            et.teste();
        }
        catch (Throwable e) {
            System.out.println("main: " + e);
        }
    }
}
```

Deklaration von Ausnahmen (3)

```
/** Beispielhafte Ausgabe:
Zahl >= 0 und <=100 eingeben: 12
12 eingegeben

Zahl >= 0 und <=100 eingeben: -4
exceptiontest.MyException: Falsche Zahl
eingegeben: -4; zahl = 12

Zahl >= 0 und <=100 eingeben: 0
0 eingegeben
 */
```


Reihenfolge der Ausnahmebehandlung

```
try {
    ...
} catch (MyException e) {
    ...
} catch (NumberFormatException e) {
    ...
} catch (RuntimeException e) {
    ...
} catch (Exception e) {
    ...
} catch (Throwable e) {
    ...
```

Deklaration von Ausnahmen und Vererbung

- Beim Überschreiben von Methoden gibt es wichtige Regeln:
  - ⇨ Überschriebene Methoden in einer Unterklasse dürfen nicht mehr Ausnahmen auslösen, als schon beim throws-Teil der Oberklasse aufgeführt sind.
  - ⇨ Eine Methode der Unterklasse kann nur dieselben Ausnahmen wie die Oberklasse auslösen, oder
  - ⇨ Ausnahmen spezialisieren, oder
  - ⇨ Ausnahmen weglassen.

Deklaration von Ausnahmen und Vererbung

```
interface Info {
    public void info() throws Exception;
}

class ObenException extends Exception {}

class MitteException extends ObenException {}

class Oben implements Info {
    // Die Ausnahme muss eine Unterklasse von Exception
    // sein
    public void info() throws ObenException {
        System.out.println("Oben.info()");
        throw new ObenException();
    }
}
```

Deklaration von Ausnahmen und Vererbung

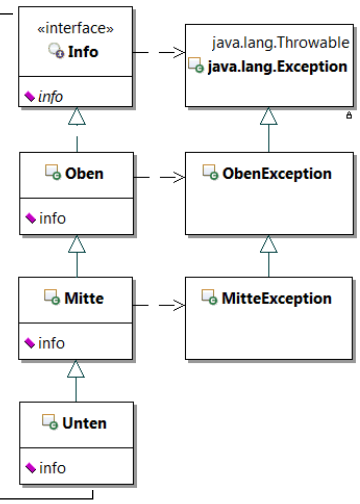
```
class Mitte extends Oben {
    // Die Ausnahme muss eine Unterklasse von
    // ObenException sein
    @Override
    public void info() throws MitteException {
        System.out.println("Mitte.info()");
        throw new MitteException();
    }
}

class Unten extends Mitte {
    public void info()  { // Keine Ausnahme mehr
        System.out.println("Unten.info()");
    }
}
```


Deklaration von Ausnahmen und Vererbung

```
public class VererbungTest {
    public static void main(String[] args)
   {
        Oben[] tab = { new Oben(),
                       new Mitte(),
                       new Unten()};
        for (Oben o : tab) {
            try {
                o.info();
            } catch (ObenException e) {
                System.out.println(e);
            }
        }
    }
}
```

Ausgabe:
```
Oben.info()
ObenException
Mitte.info()
MitteException
Unten.info()
```



try mit Ressourcen

- Mit der Garbage Collection (GC) ist es möglich, nicht mehr referenzierte Objekte freizugeben, aber was ist mit den folgenden Ressourcen?
  - ⇨ Dateisystem-Ressourcen von Dateien
  - ⇨ Netzwerkressourcen wie Socket-Verbindungen
  - ⇨ Datenbankverbindungen
  - ⇨ nativ gebundene Ressourcen vom Grafiksubsystem
  - ⇨ Synchronisationsobjekte (bei Thread-Programmierung)
- Die bisherige Lösung sah vor, dass so etwas in der finally-Klausel gelöst wird.
- Um das Schließen von Ressourcen zu vereinfachen, wurde in Java 7 eine besondere Form der try-Anweisung eingeführt, die **try mit Ressourcen** genannt wird.

try mit Ressourcen

Syntax:

```
try (Ressource1 [; Ressource2 …])  {
    Block
} [catch-Klauseln] [finally-Klauseln]
```

- Die hinter try definierte(n) Ressource(n) müssen vom Typ `java.lang.AutoCloseable` sein.
- Am Ende des Blocks wird dann immer automatisch die close()-Methode aufgerufen
- Es sind auch mehrere Ressourcendefinitionen möglich.

java.lang.AutoCloseable

```
package java.lang;
public interface AutoCloseable
{
  void close() throws Exception;
}
```

- Dieses Interface wird z. B. von den folgenden Klassen implementiert:

Typ	Signatur
java.io.Scanner	close() // ohne Ausnahme
java.io.FileInputStream	close() throws IOException
java.io.BufferedReader	close() throws IOException
java.sql.Connection	close() throws SQLException

Beispiel ohne catch-Klausel

```
// Mit try-with-resources
static String readFirstLine() {
    try (Scanner sc = new Scanner(System.in)) {
        return sc.nextLine();
    } // automatischer Aufruf von close
}
```

```
// ohne try-with-resources
static String readFirstLineWithFinallyBlock() {
    Scanner sc = new Scanner(System.in);
    try {
        return sc.nextLine();
    } finally {
        if (sc != null)
            sc.close();
    }
```

Beispiel mit catch-Klausel

```
// Mit try-with-resources
static String readFirstLine() {

    try (BufferedReader bin = new BufferedReader(
                              new InputStreamReader(
                                  System.in))) {
        return bin.readLine();
    } catch (IOException e) {
        System.out.println(e);
    } // automatischer Aufruf von close

    return null;
}
```