Grundlagen der Programmierung

Dr. Christian Herzog
Technische Universität München

Wintersemester 2007/2008

Kapitel 9: Ausnahmen

Überblick

- Ausnahme: ein Ereignis, das den normalen Programmfluss ändert.
- Ausnahmen in Java
- Programmierung von Ausnahmen
- Behandlung von Fehlern mit Ausnahmen

Ziel der Vorlesung

- Sie verstehen das Konzept der Ausnahme und Ausnahmebehandlung.
- Sie sind in der Lage das Java **`try-catch-finally`**-Konstrukt zu benutzen.
- Sie können Ausnahmen als Unterklassen von Java-Ausnahmen implementieren.
- Sie verstehen, wie man Ausnahmebehandlungen implementiert.

Fehlerbehandlung in Software-Systemen

- Eines der größten Problembereiche beim Entwurf von Software-Systemen ist der Umgang mit möglichen Fehlern. Beim Entwurf von Software müssen Sie sich immer die folgenden 2 Fragen stellen:
  - Was kann falsch laufen?
  - Was mache ich, wenn etwas falsch läuft?
- Falls z.B. der Aufruf von Methoden an Bedingungen geknüpft ist (wie: *„Element muss in der Menge enthalten sein.“*):
  - Was ist, wenn die Bedingungen falsch formuliert worden sind? Oder wenn gar keine gestellt wurden?
    - Dies kann bereits zur Entwicklungszeit oder erst zur Laufzeit erkannt werden
  - Was ist, wenn die Bedingungen nicht eingehalten werden?
    - Alternative: Abbrechen oder Weiterlaufen

Beispiel eines Fehlers

- Wenn die Länge der Reihung 0 ist, gibt es einen Fehler "Division-durch-0".

Schlechter Entwurf:
`mittelwert()` ist nicht vor **`arr.length==0`** geschützt.

```
public static int mittelwert(int arr[]) {
    int avg = 0;
    for (int k = 0; k < arr.length; k++)
        avg += arr[k];
    return avg / arr.length;
} // mittelwert()
```


Was machen wir mit Fehlern?

- Wir bauen die Fehlerbehandlung in das Programm ein
- 95% aller existierenden Systeme arbeiten so :-(

```
public static int mittelwert( int arr[] ) {
    int avg = 0;
    if (arr.length == 0) {
        System.err.println("FEHLER: Division durch 0");
        System.exit(99);
    }
    for (int k = 0; k < arr.length; k++)
        avg += arr[k];
  return avg / arr.length;
} // mittelwert()
```

Bricht den Programmablauf mit dem Fehlerschlüssel **`99`** ab.

- Alternative Fehlerbehandlung: Ausnahmen

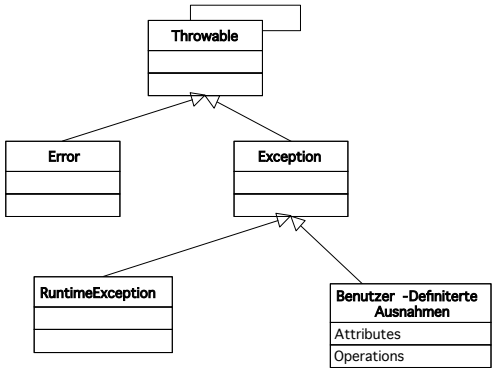
Ausnahmen

- **Definition Ausnahme:** Ein Ereignis, welches die normale Ausführungsreihenfolge in einem System unterbricht, da ein Fehler aufgetreten ist.
  - Eine Ausnahme ist eine Klasse, d.h. sie hat Methoden und Attribute, die die Unterscheidung verschiedener Fehlerursachen erlauben. Wenn diese Unterscheidung wichtig ist, sagen wir statt Ausnahme auch **Ausnahmetyp**.
- **Definition Ausnahmeobjekt:** Die Instanz einer Ausnahme. Wird oft ebenfalls als Ausnahme bezeichnet (Die Unterscheidung zwischen Ausnahme als Typ und Ausnahme als Objekt ist oft aus dem Kontext klar, in dem sie benutzt wird).
- **Definition Ausnahmebehandlung:** Nachdem eine Methode ein Ausnahmeobjekt erzeugt hat, geht der Programmfluss nicht normal weiter, sondern wird bei der Ausnahmebehandlung fortgesetzt. Das Ausnahmeobjekt enthält die Einzelheiten der Fehlerursache.

Taxonomie von Ausnahmen

- Ein wesentlicher Teil des Entwurfs, besonders des detaillierten Entwurfs ist die Modellierung von Ausnahmen.
- Ziel der Modellierung ist die Erstellung einer Taxonomie (Hierarchie), die es ermöglicht, Ausnahmen aus der Anwendungsdomäne und der Lösungsdomäne zu behandeln.
  - Ausnahmen sind gut für die Modellierung von potentiellen Problemen, z.B. bei der Eingabe von Werten in interaktiven Systemen.
- Die Programmiersprache Java stellt eine (erweiterbare) Taxonomie bereit.
  - Mithilfe von Spezialisierung können wir die modellierten Ausnahmen aus der Anwendungsdomäne und Lösungsdomäne als Unterklassen von Java's Ausnahmen-Hierarchie definieren.

Java's Ausnahmenhierarchie



- Jede Ausnahme ist Unterklasse von **`Throwable`**.
- **`Error`** ist eine Ausnahme, für die keine Ausnahmebehandlung möglich ist (z.B. Hardware-Fehler). Sie führt immer zum Programmabbruch.
- **`Exception`** ist eine Ausnahme, für die eine Ausnahmebehandlung möglich ist.
- Für jede Ausnahme vom Typ **`Exception`** muss der Benutzer angeben, ob und, wenn ja, wie sie zu behandeln ist, außer für Ausnahmen vom Typ **`RuntimeException`**.
- Wird eine Ausnahme nicht behandelt, so führt sie zum Programmabbruch durch das Java-System.

Die Klasse `Exception`

```
public class Exception extends Throwable {

// Attribut:
   private String message = …;    // beschreibt die Ausnahme;
                                  // Voreinstellung abhaengig vom
                                  // Ausnahmetyp
// Konstruktoren:
   public Exception() {...}       // Keine detaillierte Beschreibung

   public Exception(String s) {…} // Mit detaillierterer Beschreibung:
                                  // s ueberschreibt message

// ausgewaehlte Methoden (geerbt von Throwable):
 …
   public String getMessage() {…} // liefert Beschreibung
 …
   public void printStackTrace() {…}
      // druckt die Ausnahme und die aktuelle Methoden-Aufrufstruktur
      // auf die Standard-Fehlerausgabe System.err
 …
}
```


Beispiel für das Auftreten einer Ausnahme

```
class Example {
  public static int mittelwert (int[] arr) {
    int avg = 0;
    for (int k = 0; k < arr.length; k++)
        avg += arr[k];
    return avg / arr.length;
  } // mittelwert()

  public static void main (String[] args) {
    int[] arr = new int[0];
    System.out.println(mittelwert(arr));
  }
}
```

- Der Aufruf **`java Example`** führt zu folgender Ausgabe:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Example.mittelwert(Example.java:6)
        at Example.main(Example.java:11)
```

- Das Java-System hat die Ausnahme durch Aufruf der Methode **`printStackTrace()`** und Programmabbruch behandelt.

Spezialiserung bei Ausnahmen

- Ausnahmen sind Klassen, wir können also Unterklassen definieren.
- **Beispiel**: Beim Prüfen von Eingaben wollen wir oft sicherstellen, dass der eingegebene Wert kleiner als eine obere Grenze ist.
  - Ein spezieller Konstruktor sorgt dafür, dass die detaillierte Fehlerbeschreibung auch die aktuelle Obergrenze enthält.

```
//  Die Ausnahme IntOutOfRange wird ausgeloest, wenn eine
//  ganze Zahl einen bestimmten Wert Bound überschreitet.
public class IntOutOfRange extends Exception {
// Die beiden Standard-Konstruktoren für Ausnahmen:
    public IntOutOfRange () {
        super();
    }
    public IntOutOfRange (String s) {
        super(s);
    }
// Ein zusätzlicher speziellerer Konstruktor:
    public IntOutOfRange (int Bound) {
        this("Der Eingabewert ist größer als " + Bound);
    }
}
```

Mit **`super()`** wird jeweils der Konstruktor der Superklasse, hier also von **`Exception`**, aufgerufen.

Beispiel: Ein neuer Ausnahmetyp `MultipleElementListException`

- In der Methode **`insertElement()`** der Klasse **`OrderedList`** kann der Fehler auftreten, dass das einzufügende Element bereits enthalten ist.
- Die bisherige Fehlerbehandlung soll durch Auslösen einer Ausnahme ersetzt werden.
- Dafür definieren wir die Klasse **`MultipleElementListException`**:

```
public class MultipleElementListException extends ListException {
  public MultipleElementListException() {
    super();
  }
  public MultipleElementListException(String s) {
    super(s);
  }
  public MultipleElementListException(Comparable c) {
    this("multiple Element: " + c);
  }
}
```

`ListException` sei Oberklasse für alle Ausnahmen, die bei Listen auftreten können.

Das Auslösen („Werfen“) einer Ausnahme

- In der Methode **`insertElement()`** soll nun statt einer Fehlerausgabe am Bildschirm die Ausnahme **`MultipleElementListException`** ausgelöst werden.
- Ausnahmen werden mittels **`throw`** ausgelöst („geworfen“).
- Das (eventuelle) Auslösen einer Ausnahme muss mittels **`throws`** im Methodenkopf deklariert werden.

```
// Allgemeine Form:
public void m ( ... ) throws Exception1, … , ExceptionN {
      … … …
   // eventuelles Auslösen von Exception1:
   if ( … ) … throw new Exception1(…); …
      … … …
   // eventuelles Auslösen von ExceptionN:
   if ( … ) … throw new ExceptionN(…); …
      … … …
}
```

Deklaration der Ausnahmen

new instantiiert eine neue Ausnahme und `throw` löst sie aus.

- **Ausnahmeregelung**: Ausnahmen, die Unterklasse von **`Runtime-Exception`** sind, müssen **nicht** mittels **`throws`** deklariert werden.
  - Grund: Sie können praktisch überall ausgelöst werden.

Beispiel: Auslösen einer Ausnahme in `insertElement()`

```
public static OrderedList insertElement(Comparable c, OrderedList l)
                        throws MultipleElementListException {
  // Falls die Liste leer ist oder nur größere Elemente enthaelt:
  if (l == null || l.item.comparesTo(c) == 1) {
    return new OrderedList(c, l);
  }

  // Falls das erste Listenelement c enthaelt:
  if (l.item.comparesTo(c) == 0) {
     throw new MultipleElementListException(c);
  }

  // Ansonsten arbeite rekursiv mit der Nachfolger-Liste:
  l.next = insertElement(c, l.next);
  return l;
}
```

Deklaration der Ausnahme

new instantiiert die neue Ausnahme und `throw` löst sie aus.

- Bei Ausführung von **`throw`** wird die Methode **`insertElement()`** sofort beendet und die Ausnahme an die aufrufende Methode weitergereicht.
- Dort wird sie entweder **behandelt** („gefangen“) oder ebenfalls wieder an den Aufrufer weitergereicht.

Beispiel: Nichtbehandeln (Weiterreichen) einer Ausnahme

```
class ExceptionTest1 {
// Auf der Kommandozeile übergebene ganze Zahlen werden in eine
// sortierte Liste eingefügt und dann ausgedruckt:
    public static void main (String[] args)
                          throws MultipleElementListException {
    OrderedList list = null;
    for (int i=0; i<args.length; i++)
      list = OrderedList.insertElement(
                new MyInteger(Integer.parseInt(args[i])), list);
    System.out.println(list);
  }
}
```

- Die Methode **`insertElement()`** hat die Ausnahme **`MultipleElementList-Exception`** deklariert, weil sie eventuell ausgelöst wird.
- Deshalb löst der Aufruf von **`insertElement()`** eventuell diese Ausnahme aus.
- Da diese Ausnahme in **`main()`** nicht behandelt wird, wird sie an den Aufrufer (hier das Java-System) weitergereicht.
- Deshalb muss auch **`main()`** die Ausnahme **deklarieren**.
- **Regel**: **nicht behandelte Ausnahmen müssen deklariert werden** (außer **`RuntimeException`**).

Behandlung benutzerdefinierter Ausnahmen durch das Java-System

```
class ExceptionTest1 {
// Auf der Kommandozeile übergebene ganze Zahlen werden in eine
// sortierte Liste eingefügt und dann ausgedruckt:
    public static void main (String[] args)
                              throws MultipleElementListExceptiion {
    OrderedList list = null;
    for (int i=0; i<args.length; i++)
      list = OrderedList.insertElement(
                new MyInteger(Integer.parseInt(args[i])), list);
    System.out.println(list);
  }
}
```

- Benutzerdefinierte Ausnahmen werden vom Java-System wie jede andere Ausnahme behandelt (Aufruf von **`printStackTrace()`** und Abbruch).
- Beispiel: Das Kommando **`java ExceptionTest1 1 2 3 2`** führt zur Ausgabe:

```
Exception in thread "main" MultipleElementListException: multiple Element: 2
        at OrderedList.insertElement(OrderedList.java:53)
        at OrderedList.insertElement(OrderedList.java:57)
        at ExceptionTest1.main(ExceptionTest1.java:8)
```


Behandlung von Ausnahmen: `try-catch-finally`

- Ein Versuchs-Block (**`try{…}`**) enthält Anweisungen, in denen Ausnahmen ***eventuell ausgelöst*** werden können. Mit einem Versuchs-Block signalisieren wir unsere Bereitschaft, diese Ausnahmen zu behandeln.
- Die Ausnahmebehandlung ***fängt*** eine geworfene Ausnahme und besteht aus einem oder mehreren Fang-Blöcken (**`catch(…) {…}`**).
  - Jeder Fang-Block ist für eine Ausnahme zuständig. Der speziellste Ausnahmetyp sollte im ersten Fang-Block sein
  - Die Definition von Fang-Blöcken ähnelt der Definition von Methoden, und enthält einen formalen Parameter vom Typ (oft e genannt). Der Typ des Parameters ist der Ausnahmentyp.
- Der (optionale) Final-Block (**`finally{…}`**) macht die ***Aufräumarbeiten***.
  - Falls das Programm nicht abbricht, wird dieser Block ausgeführt, egal, ob eine Ausnahme erzeugt worden ist, oder nicht.

Behandlung von Ausnahmen

```
try {
   …
      // In diesem Versuchs-Block werden eventuell Ausnahmen
      // ausgelöst
   …
}
catch (Exception1 e) {
  System.err.println("Fehler: " + e);
   …  // Ausnahmebehandlung für Ausnahmen vom Typ Exception1
}
catch (Exception2 e) {
  System.err.println("Fehler: " + e);
   …  // Ausnahmebehandlung für Ausnahmen vom Typ Exception2
}
 …    // Ausnahmebehandlungen für weitere Ausnahmetypen

finally {
   …  // Aufräumarbeiten und andere Anweisungen
}
```


Beispiel: Behandeln einer Ausnahme - Version 1

```
class ExceptionTest2 {
// Auf der Kommandozeile übergebene ganze Zahlen werden in eine
// sortierte Liste eingefügt und dann ausgedruckt:
  public static void main (String[] args) {
    OrderedList list = null;
    try {
      for (int i=0; i<args.length; i++)
        list = OrderedList.insertElement(
                  new MyInteger(Integer.parseInt(args[i])), list);
    }
    catch (MultipleElementListException e) {
      System.err.println("Fehler: " + e);
    }
    finally {
      System.out.println(list);
    }
  }
}
```

Ausnahme wird in diesem Fall behandelt und muss nicht deklariert werden!

- Sobald eine Ausnahme ausgelöst wird, wird der **`try`**-Block verlassen und zuerst der **`catch`**-Block, danach der **`finally`**-Block ausgeführt.
- Dann ist das Ende der Methode erreicht. (Der **`try`**-Block wurde endgültig verlassen)
- **`java ExceptionTest2 1 2 3 4 3 5 5 6 7`** führt zur Ausgabe:

```
Fehler: MultipleElementListException: multiple Element: 3
1,2,3,4
```

Beispiel: Behandeln einer Ausnahme - Version 2

- In diesem Beispiel wird der **`try`**-Block nur um den Rumpf der Schleife gelegt.
  - **`finally`** wird nicht benötigt.

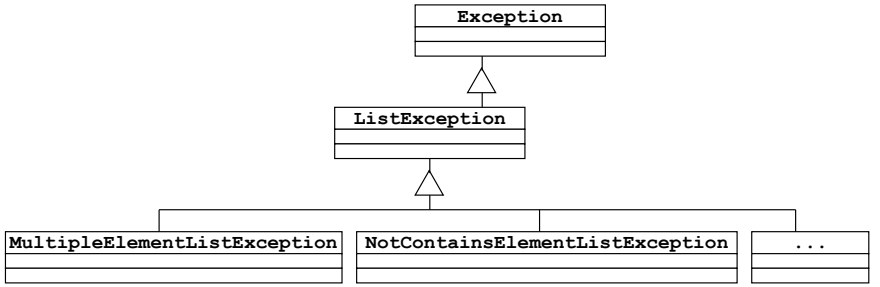
```
class ExceptionTest3 {
// Auf der Kommandozeile übergebene ganze Zahlen werden in eine
// sortierte Liste eingefügt und dann ausgedruckt:
  public static void main (String[] args) {
    OrderedList list = null;
    for (int i=0; i<args.length; i++)
      try {
        list = OrderedList.insertElement(
                  new MyInteger(Integer.parseInt(args[i])), list);
      }
      catch (MultipleElementListException e) {
        System.err.println("Fehler: " + e);
      }
    System.out.println(list);
  }
}
```

- **`java ExceptionTest3 1 2 3 4 3 5 5 6 7`** führt diesmal zur Ausgabe:

```
Fehler: MultipleElementListException: multiple Element: 3
Fehler: MultipleElementListException: multiple Element: 5
1,2,3,4,5,6,7
```


Taxonomie benutzerdefinierter Ausnahmen

- Die Definition benutzerdefinierter Ausnahmetypen sollte sorgfältig erfolgen.
  - Z.B. sollten Ausnahmetypen für alle Fehlerfälle eingeführt werden, die unterschiedlich behandelt werden werden sollen:



- Z.B. sollten auch Oberklassen zu ähnlichen Ausnahmetypen eingeführt werden.
- Dies erlaubt eine abgestufte Behandlung der Ausnahmen.

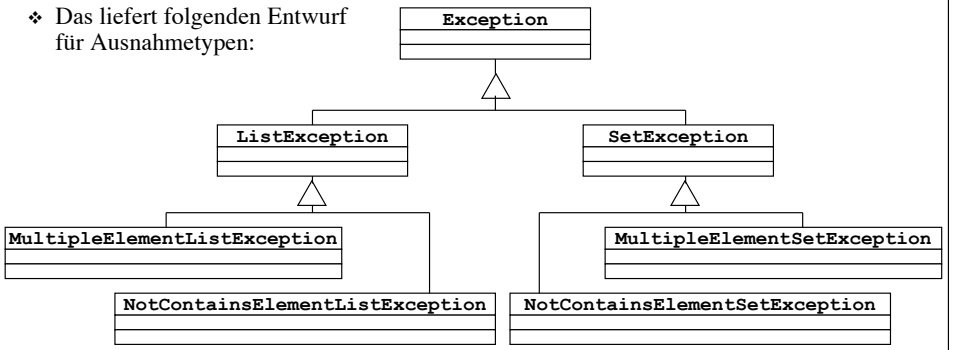
Abgestufte Behandlung von Ausnahmen

- Ausnahmehierarchien erlauben abgestufte Behandlung vom speziellsten zum allgemeinsten Fehlerfall.
- Der erste „passende“ Fang-Block wird ausgewählt :

```
try {
      // In diesem Versuchs-Block werden eventuell Ausnahmen
      // ausgelöst
}
// Spezielle Behandlung von NotContainsElementListException:
catch (NotContainsElementListException e) {
  System.err.println(”schwerer Fehler: " + e);
  System.exit(98);
}
// Ausnahmebehandlung für alle restliche Ausnahmetypen bei
Listen:
catch (ListException e) {
  System.err.println("Fehler bei Listen: " + e);
  // Hier nur Meldung, kein Programmabbruch
}
// Ausnahmebehandlungen für alle weiteren Ausnahmetypen
catch (Exception e) {
  System.err.println(”unerwarteter Fehler: " + e);
  System.exit(99);
}
```

Vervollständigung unseres Beispiels zur Mengendarstellung

- Wir unterscheiden folgende zwei Fehlerfäller für Listen:
  - In eine Liste soll ein Element eingefügt werden, das bereits vorhanden ist.
  - In einer Liste soll ein Element gelöscht werden, das gar nicht vorhanden ist.
- Und analog für Mengen:
  - In eine Menge soll ein Element eingefügt werden, das bereits vorhanden ist.
  - In einer Menge soll ein Element gelöscht werden, das gar nicht vorhanden ist.
- Das liefert folgenden Entwurf für Ausnahmetypen:



Beispiel: Die Methode `insert()` der Klasse `OrderedListSet`

- **`insert()`** muss die Ausnahme **`MultipleElementListException`** der Methode **`insertElement()`** behandeln ...
- ... und soll im Fehlerfall die Ausnahme **`MultipleElementSet-Exception`** auslösen:

```
public void insert(Object o) throws MultipleElementSetException {

 // stützt sich auf entsprechende Methode von OrderedList:

    try {
          list = OrderedList.insertElement((Comparable) o, list);
       }

    catch (MultipleElementListException e) {
          throw new MultipleElementSetException(e.getMessage());
    }
}
```


Ausnahmen und abstrakte Methoden

- Die von einer Methode ausgelösten Ausnahmen (**`throws`**-Konstrukt) gehören zur Funktionalität der Methode.
- Wenn eine Methode eine abstrakte Methode implementieren soll, muss die Funktionalität identisch sein.
- Bereits die abstrakte Methode (und damit alle anderen alternativen Implementierungen) muss also das identische **`throws`**-Konstrukt enthalten.
- Am Beispiel der abstrakten Klasse **`Set`**:

```
abstract class Set {
   …
    // Einfuegen eines Elementes:
    public abstract void insert(Object o)
                          throws MultipleElementSetException;

    // Entfernen eines Elementes:
    public abstract void delete(Object o)
                          throws NotContainsElementSetException;
   …
}
```


Noch ein Beispiel zum Behandeln von Ausnahmen:

```
class ExceptionTest4 {
// Auf der Kommandozeile übergebene ganze Zahlen werden in eine sortierte
// Menge eingefügt und dann ausgedruckt.
// Falsche Zahldarstellungen werden dabei ignoriert
  public static void main (String[] args) {
      OrderedListSet set = new OrderedListSet ();
      for (int i=0; i<args.length; i++)
         try {
            set.insert(new MyInteger(Integer.parseInt(args[i])));
         }
         catch (NumberFormatException e) {
           System.out.println("Achtung: " + args[i] + " ist keine Zahl.");
         }
         catch (MultipleElementSetException e) {
           System.out.println("Achtung: " + args[i] + " war schon da.");
         }
      System.out.println(set);
   }
}
```

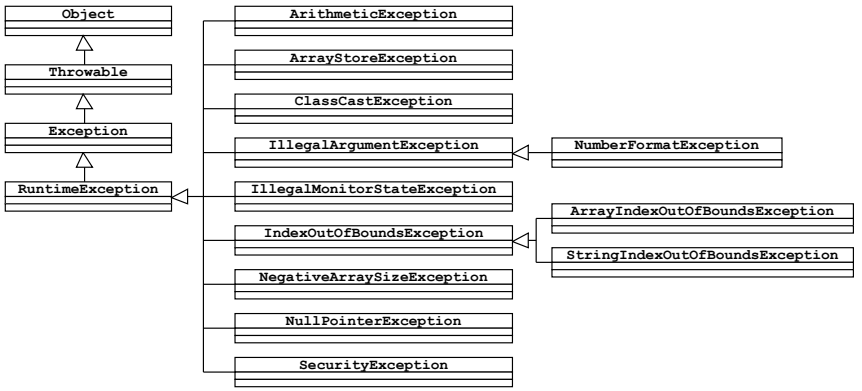
Kann Runtime-Exception **`NumberFormatException`** auslösen.

- **`java ExceptionTest4 5 3 2 xxx 3 6 1 4`** führt zur Ausgabe:

```
Achtung: xxx ist keine Zahl.
Achtung: 3 war schon da.
{1,2,3,4,5,6}
```


Ungeprüfte Ausnahmen

- Ungeprüfte Ausnahmen heißen so, weil vom Java Compiler nicht überprüft wird, ob sie deklariert sind.
  - Die Klasse `RuntimeException` und alle ihre Unterklassen sind ungeprüfte Ausnahmen.
  - Klassenhierarchie für `RuntimeException`



Ungeprüfte Ausnahmen

ArithmeticException	Division durch Null oder ein anderes arithmetisches Problem
ClassCastException	Ungültige Typ-Konvertierung eines Objektes in eine Klasse, von der es keine Instanz ist.
IllegalArgumentException	Methodenaufruf mit falschen Argumenten
NumberFormatException	Illegales Zahlenformat (z.B. beim Methodenaufruf)
IndexOutOfBoundsException	Ein Reihungs-oder Zeichenkettenindex ist ausserhalb der Grenzen
ArrayIndexOutOfBoundsException	Ein Index in einer Reihung ist kleiner als 0 oder ≥ als die Länge der Reihung
StringIndexOutOfBoundsException	Ein Zeichenkettenindex ist ausserhalb der Grenzen
NullPointerException	Referenz auf nicht instanziiertes Objekt

Heuristiken für Entwurf von Ausnahmen

- **Defensiver Entwurf.** Versuchen Sie, potentielle Probleme zu sehen, die die normalen Folge von Ereignissen verändern. Ein guter Ausgangspunkt sind Anwendungsfälle. Spezielle Anwendungsfälle sind gute Kandidaten für Ausnahmen. Auch die inkorrekte Eingabe von Werten bei interaktiven Systemen lässt sich oft gut als Ausnahme modellieren.
- **Lokale Behandlung.** Versuchen Sie Ausnahmen lokal zu behandeln.
- **Ausnahmen in Klassenbibliotheken.** Schreiben Sie **`catch`**-Blöcke für alle Ausnahmen, die Routinen aus Klassenbibliotheken auslösen könnten, sonst überlassen Sie Ihr Schicksal den oft unverständlichen Beschreibungen des Java-Systems.
- **Ausnahmen beschreiben Ausnahme-Situationen.** Benutzen Sie Ausnahmen nicht für die Behandlung von normalen Bedingungen.

Zusammenfassung

- Ausnahmen sind auch Klassen
- Behandlung von Ausnahmen: das **`try-throw-catch-finally`** Gerüst
- Geprüfte und ungeprüfte Ausnahmen:
  - Alle geprüften Ausnahmen müssen deklariert oder behandelt sein.
- Benutzerdefinierte Ausnahmen sind Spezialisierungen der Klassen **`Exception`**
- Ausnahmen sind ein gutes Werkzeug, um die Einhaltung von Vorbedingungen in Methoden zu überprüfen.

Hinweis

- Auf der Homepage der Vorlesung steht unter
  - **`Ausnahmen.tar.gz`** bzw.
  - **`Ausnahmen.zip`**

  die Hierarchie von Mengendarstellungen als generische Klassen aus Kapitel 8, ergänzt um Fehlerbehandlung durch Ausnahmen, zur Verfügung.