7.

Java: Weitere Sprachelemente

Interfaces, Exceptions, ArrayList, Generics

Buch Mark Weiss „Data Structures & Problem Solving Using Java“ siehe Seiten 134-136 (Interfaces); 83-87, 173-174 (Exceptions); 40-44 (ArrayList); 150 ff (Generics)

Lernziele Kapitel 7 Weitere Java-Sprachelemente

- Den Unterschied und Mehrwert von Interfaces relativ zu abstrakten Klassen erkennen und Interfaces beim Softwareentwurf adäquat einsetzen können
- Eigene Exceptions definieren können; Abfangen versus Weiterleiten von Ausnahmen in seinen Konsequenzen verstehen sowie mit *try*, *catch* und *throw* souverän umgehen können
- Den Unterschied dynamischer ArrayLists und statischer Arrays in seinen Konsequenzen bei der Anwendung verstanden haben
- Parametrisierte Typen („generics") hinsichtlich Typsicherheit verstanden haben und anwenden können

Thema / Inhalt

Interfaces und **Exceptions** sind bei der Erstellung grösserer Systeme durch mehrere Entwickler ein wichtiges Strukturierungsmittel, sie begünstigen einen änderungsfreundlichen und fehlerarmen Systementwurf. Typkonformität, die durch den Compiler überprüfbar ist, schliesst von vornherein viele Fehler aus, die sonst erst zur Laufzeit auftreten würden. Bei dynamischen Datentypen (wie z.B. **ArrayList**) dienen die „**generics**" diesem Zweck.

Deutsche Bezeichnung

1. Interfaces („Schnittstellen“)

Dienen primär dem Zweck der Spezifikation

- Interface = abstrakte Klasse, wo alle Methoden abstrakt sind
  - Methoden können aber evtl. eine Default-Implementierung haben

```
interface Menge
{ int cardinal();
  void insert (Object x);
  void remove (Object x);
}
```

Schlüsselwort interface statt class

Alle Methoden sind automatisch public abstract

- Ein Interface (d.h. alle seine Methoden) wird dann von anderen Klassen implementiert:

```
class S implements Menge
{ public int cardinal();
  { ...
    while ... i++ ...;
    return i;
  }
  ...
}
```

implements hat die gleiche Rolle wie extends bei „echten“ Klassen

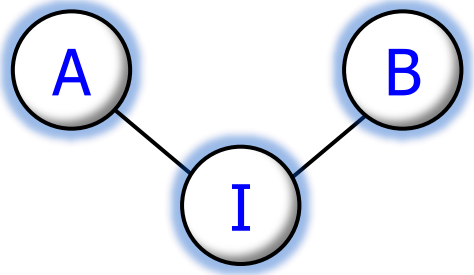
Es kann auch eine entsprechende Hierarchie entstehen

  - Default-Methoden des Interface brauchen nicht implementiert zu werden (können aber durch eine eigene Implementierung überschrieben werden)

Interface-Hierarchie, Mehrfacherweiterungen

- Interfaces dürfen mehrere andere erweitern, z.B.:

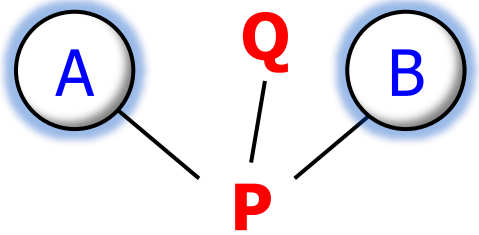
```
interface A {...}
interface B {...}
interface I extends A, B {...
   int m(); ...
}
```



  - I umfasst alle (abstrakten) Methoden von A und B (und zusätzlich m)

- Eine Klasse dagegen kann nur eine einzige Klasse erweitern (aber gleichzeitig mehrere Interfaces implementieren):

```
class P extends Q
        implements A,B,...
```



  - Die Einschränkung minimiert das Problem, dass evtl. gleichbenannte Attribute aus verschiedenen Oberklassen zu Konflikten führen

Interfaces versus abstrakte Klassen

- Interfaces und abstrakte Klassen sind syntaktisch ähnlich
  - Dienen aber unterschiedlichen Zwecken

```
interface Menge {
  int cardinal();
  void insert (Object x);
  void remove (Object x);
}
```

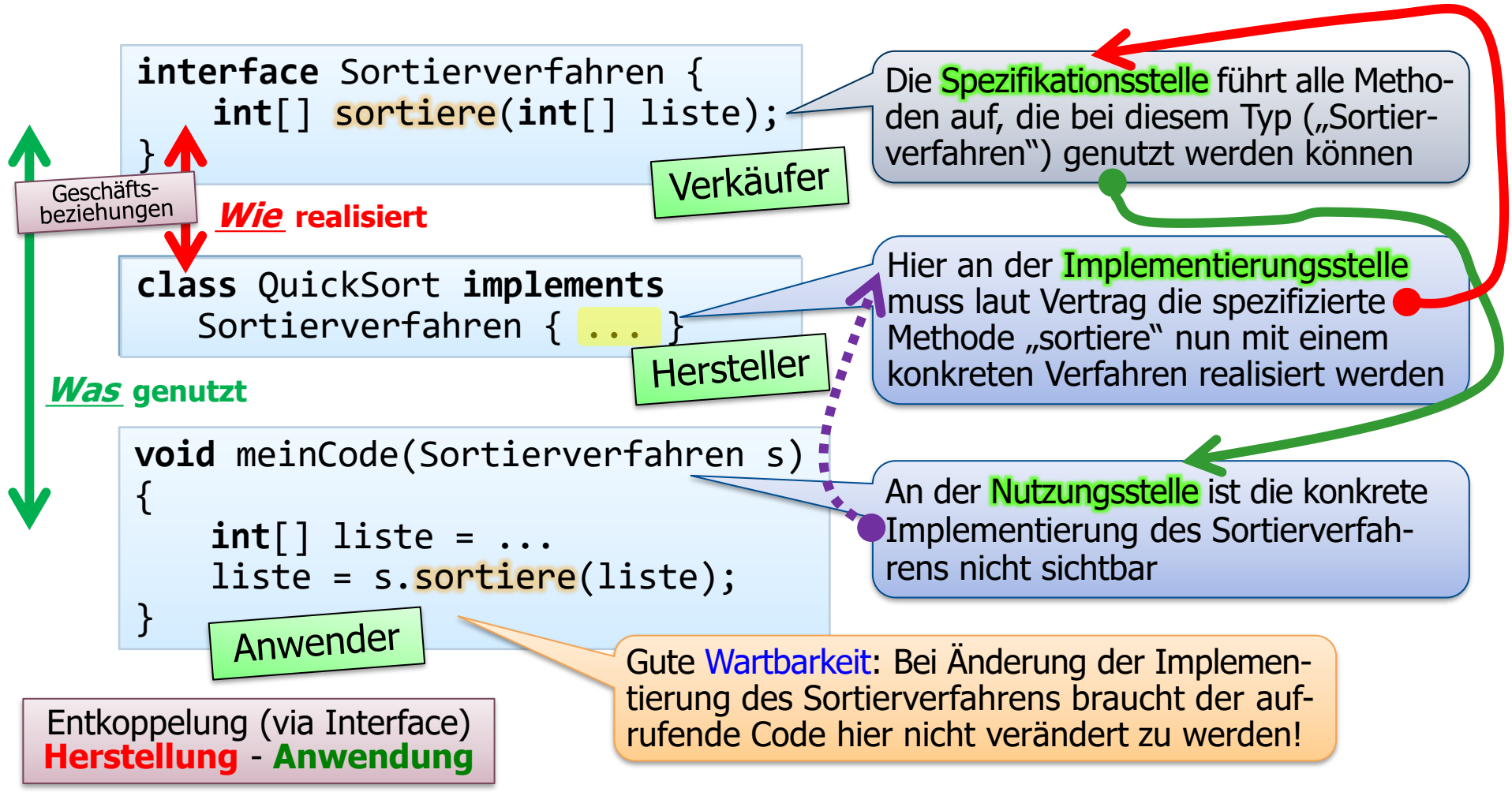
```
abstract class Menge {
  abstract int cardinal();
  abstract void insert (Object x);
  abstract void remove (Object x);
}
```

- Wann verwendet man was?
  - **Abstrakte Klassen**: Teil einer Vererbungshierarchie, bei der gemeinsame Konzepte für mehrere abgeleitete Klassen an einer einzigen Stelle zusammengefasst werden
  - **Interfaces**: Spezifikation von Eigenschaften (in Methodenform), die dann eine sie implementierende Klasse umsetzen muss

    Ausserdem unterstützen Interfaces die „Mehrfachvererbung“ (eine Klasse hingegen kann zwar mehrere Interfaces implementieren, aber nur von einer Klasse erben); auf das Konzept der Mehrfachvererbung gehen wir in dieser Vorlesung aber nicht ein

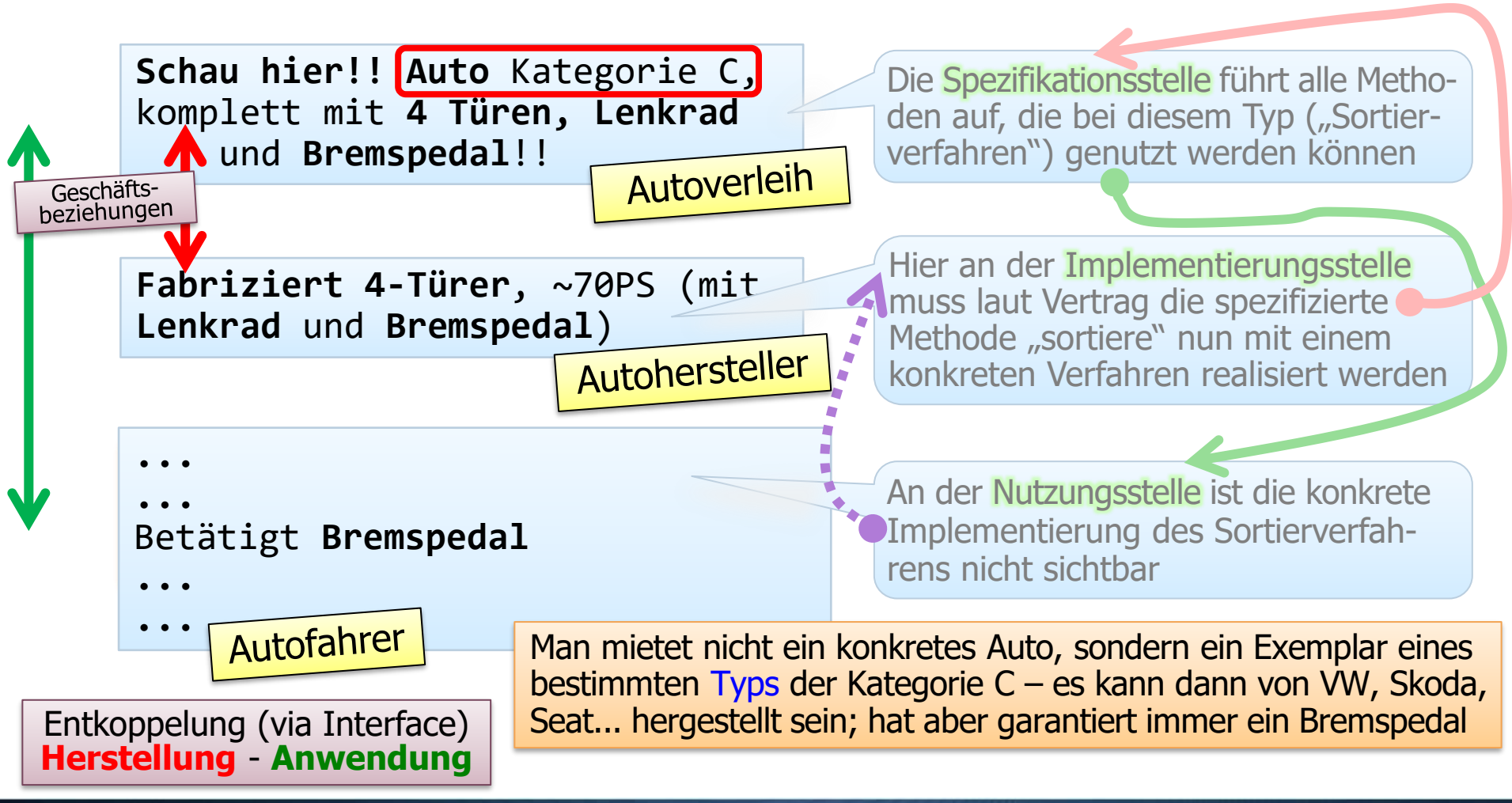
Einsatz von Interfaces beim Softwareentwurf

- Ein Interface spezifiziert die Funktionalität einer Softwarekomponente in Form eines von der Implementierung zu erfüllenden Vertrages



Einsatz von Interfaces – ein Gleichnis

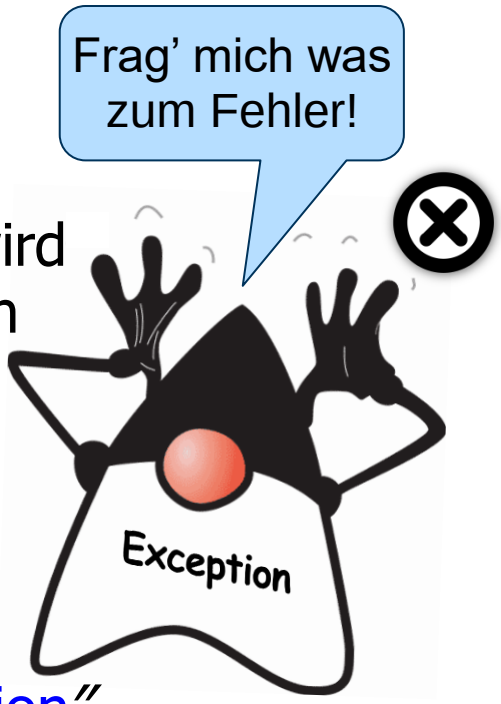
- Ein Interface spezifiziert die Funktionalität ~~einer Softwarekomponente~~ in Form eines ~~von der Implementierung~~ zu erfüllenden Vertrages



2. Exceptions

Zu Exceptions siehe Buch von M. A. Weiss, Kap. 2.5 und
https://docs.oracle.com/javase/tutorial/essential/exceptions/advantages.html

- Eine „Ausnahme“ (exception) ist ein Ereignis, das während normaler Programmausführung auftritt und die Ausführung unterbricht
- Oft ist die Ursache ein Fehlerbedingung – dann wird ein exception object erzeugt; es liefert Information
  - zum Fehlertyp,
  - Fehlerort,
  - Programmzustand zum Zeitpunkt des Auftretens
- “*Creating an exception object and handing it to the runtime system is called* throwing an exception”



Auf Deutsch: *Auslösen* einer Ausnahme bzw. eines Ausnahmeereignisses (nicht: „werfen“)

To throw: ..., eine unangenehme Überraschung bescheren, ausflippen, einen Rappel bekommen, hinklotzen, Zustände kriegen, den grossen Macker markieren, eine kalte Dusche verpassen, in die Parade fahren,...

Exceptions in Java

Z.B. StackOverflowError

- Ausnahmen als Fehlerereignisse
  - Werden oft vom System ausgelöst
  - Können aber auch explizit im Programm ausgelöst werden („**throw**“)
  - Sollten abgefangen und behandelt werden („**catch**“)
  - Fehlertypen sind in einer Klassenhierarchie angeordnet (z.B. Exception ⊃ RuntimeException ⊃ ArithmeticException)
- Strukturierung durch „**try**“ und „**catch**“
  - Fehlerbehandlung muss auf diese Weise nicht mit dem „normalen“ Programmcode verwoben werden

```
try {
   // Hier stehen Anweisungen, bei denen
   // Fehlerbedingungen eintreten können
   … … …
} catch (Fehlertyp_1) {
   // "Behandlung" dieses Fehlertyps;
} catch (Fehlertyp_2) {
   // "Behandlung" dieses Fehlertyps;
}
```

„To catch“ kann vieles bedeuten („I have to catch my train“), gemeint ist hier „abfangen“ (im Sinne von „eine Nachricht / einen Schlag abfangen“), nicht „fangen“ im Sinne von „Schmetterlinge fangen“!

Beispiel: Abfangen einer Exception

Von früher bekannte *altägyptische Multiplikation*; induziert evtl. „StackOverflowError“

```
static int f(int a, int b) {
  if (b == 1) return a;
  try
  {
    if (b%2 == 0) return f(2*a, b/2);
       else return a + f(2*a, b/2);
  }
  catch (StackOverflowError e)
  {
    //...
    System.out.println("Mist! " + a + " " + b);
    System.exit(1);
  }
  //...
}
```

Hier erwarten wir keine Fehler, daher ausserhalb des try-Blocks

e könnte man benutzen, um mehr über den Fehler zu erfahren (wann, wo,...?)

Notausstieg aus dem Programm
(könnten wir stattdessen etwas sinnvolleres tun?)

Fehlerarten

- Typ. Situationen, in denen Ausnahmen auftreten können:
  - Ein- / Ausgabe (z.B. *IOException*)
  - Netz (z.B. *MalformedURLException*)
  - Erzeugen von Objekten mit „new“
  - Typkonvertierung (z.B. *NumberFormatException*)

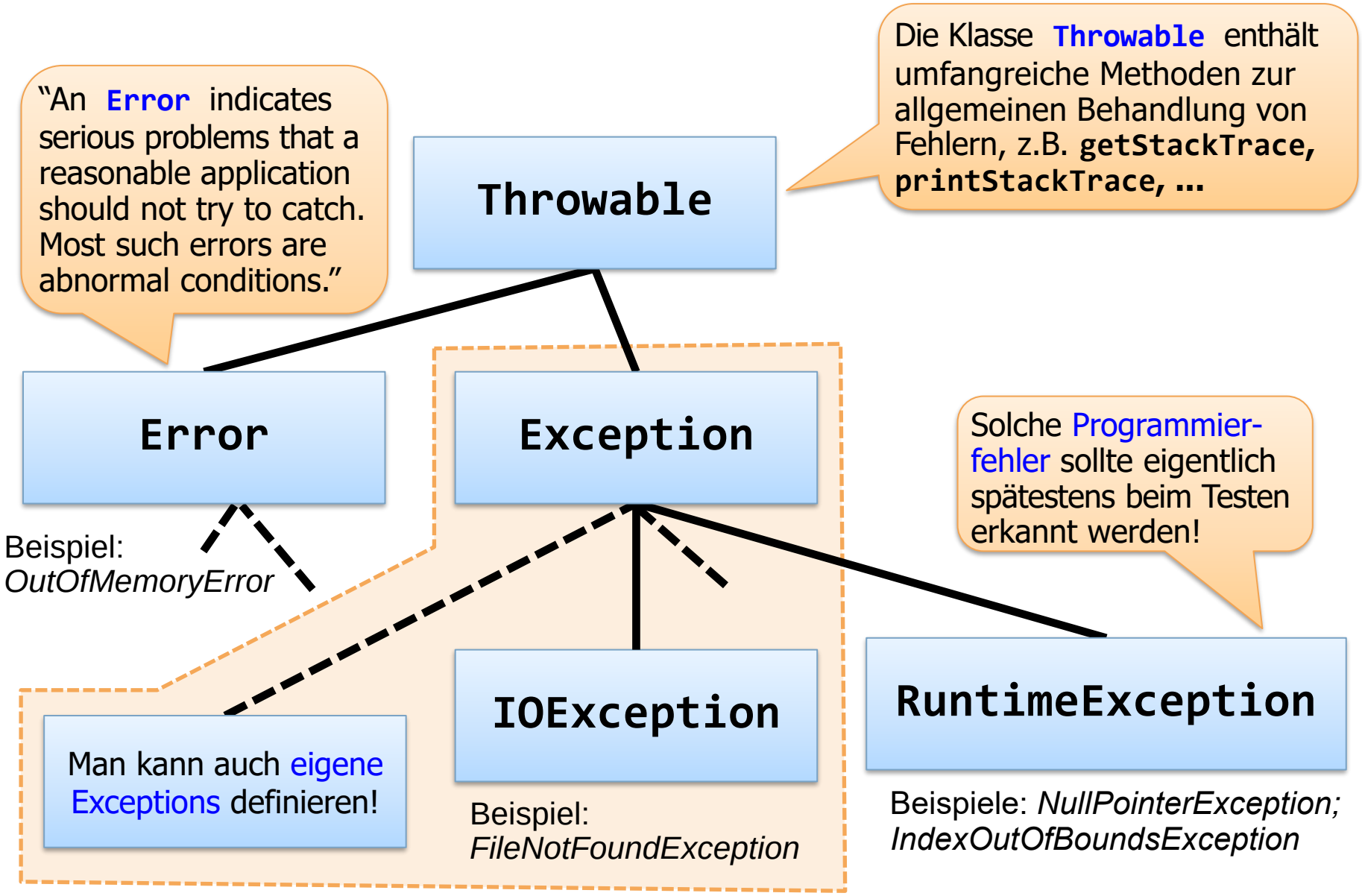
Ob solch ein Fehler auftritt, das hängt vom Kontext ab, den man meistens nicht selbst kontrolliert

- „Run time exceptions“ bilden eine relevante Teilmenge, z.B.:
  - Zugriffsversuch über Null-Referenz
  - Versuchte Division durch 0
  - Array-Indexfehler

→ Solche „Programmierfehler“ (die sich aber erst zur Laufzeit zeigen) können, aber müssen nicht abgefangen werden

```
try {
  value = value / x;
}
catch (ArithmeticException e){
  System.out.println
    ("Division durch 0?");
}
```

Ausnahmen: Hierarchie

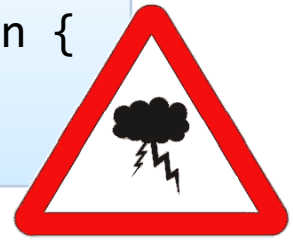


Ausnahmen abfangen / weiterleiten

- Alle Ausnahmen (ausser run time exceptions) sollten von einer Methode selbst abgefangen oder explizit weitergeleitet werden
  - Entweder durch try / catch in der Methode selbst
  - Oder durch Angabe (mittels „throws"), dass die Methode diese Ausnahme evtl. auslöst (und damit dem „Aufrufer" weiterreicht), z.B.:

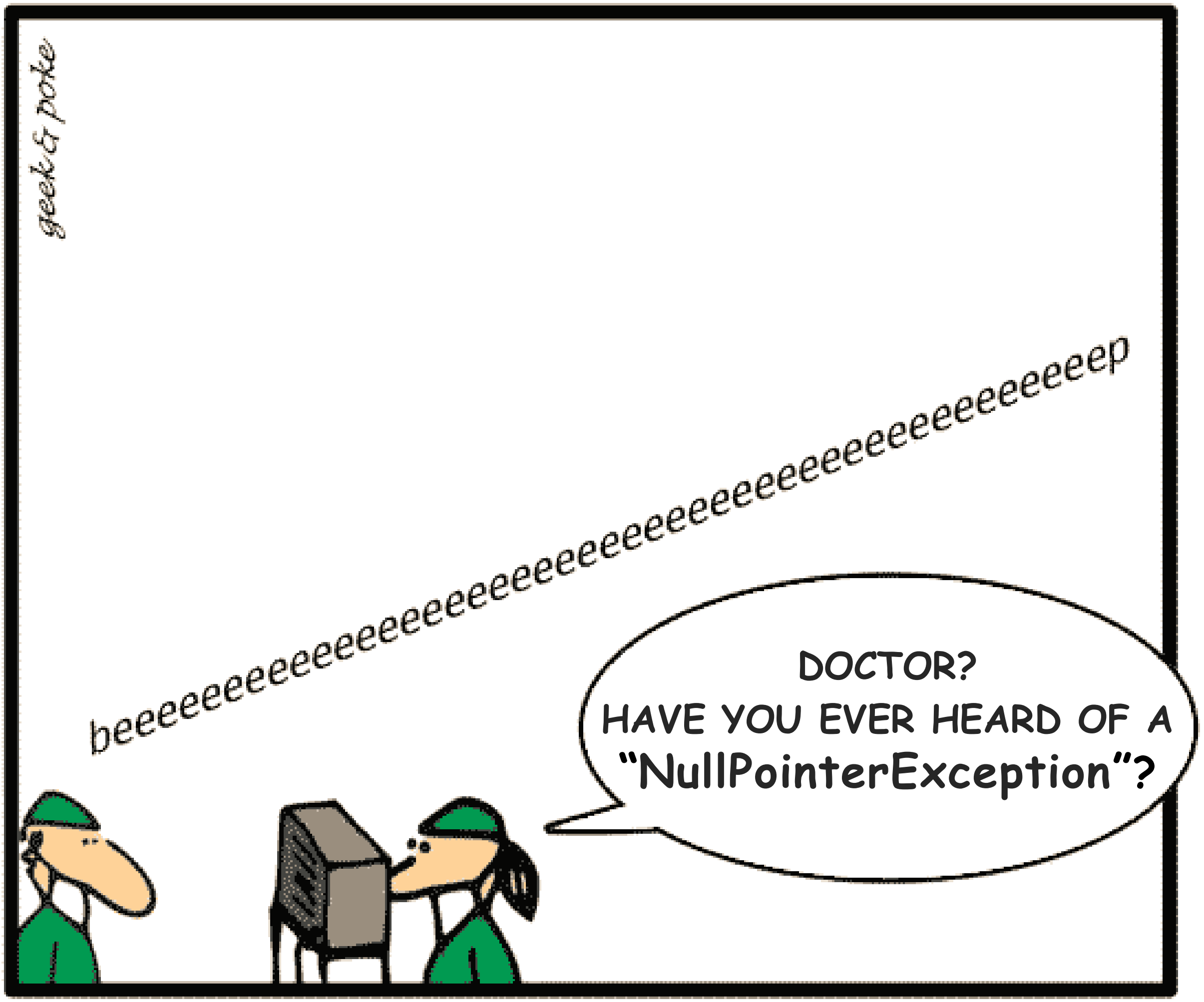
```
import java.io.*;
public eine_methode (...) throws java.io.IOException {
   ... read ...
}
```

Der Aufrufer muss dann mit dieser Ausnahme „rechnen"



- Eventuell auftretende Ausnahmen gehören so (wie die Parametertypen) zur Signatur einer Methode und sind damit explizit gemacht

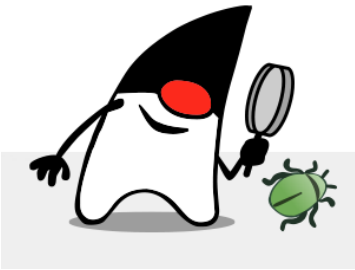
Aufrufer muss mit der Ausnahme rechnen...



http://geek-and-poke.com

RECENTLY IN THE OPERATING ROOM

Welche Ausnahmen gibt es in Java?



- Es gibt viele! (→ Dokumentation von Klassenbibliotheken)
  - Viele Entwicklungsumgebungen geben in Form von Tipps an, welche Exceptions im jeweiligen Kontext relevant sind
- Einige häufige Ausnahmen:
  - NullPointerException
  - ArrayOutOfBoundsException
  - IndexOutOfBoundsException
  - NegativeArraySizeException
  - ArithmeticException *(occurs for example, when you divide a number by zero)*
  - NumberFormatException *(occurs, when you try to convert a string to a numeric value and the String is not formatted correctly)*
  - IllegalArgumentException
  - CharConversionException
  - StringIndexOutOfBounds
  - IOException
  - FileNotFoundException
  - ClassCastException *(occurs, when you try to assign a reference variable of a class to an incompatible reference variable of another class)*

Im Jahr 2009 entschuldigt sich Tony Hoare für die Erfindung des Nullpointers:

I call it my billion-dollar mistake. It was the invention of the null reference in 1965. At that time, I was designing the first comprehensive type system for references in an object oriented language. My goal was to ensure that all use of references should be absolutely safe, with checking performed automatically by the compiler. But I couldn't resist the temptation to put in a null reference, simply because it was so easy to implement. This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years. My friend Edsger Dijkstra had a very principled objection to null references, because of possible misinterpretation of the model of reality: For example, a bachelor is often encoded by setting the wife field to null. This could give the impression that all bachelors are polyandrously married to the same wife. Bertrand Meyer merkt dazu an: *The null reference appears as necessary to the type systems of usable programming languages as zero — another troublemaker, the tormentor of division — to the number system of mathematics. What threatens to make programs crash is the risk of null dereferencing.*

Ausnahmen: E/A-Beispiel

Eingabe / Ausgabe

```
import java.io.*;
public class EA_Beispiel {
  // print "Hello World" to a file specified by the 1st input parameter
  public static void main(String args[]) {
    FileOutputStream out = null;

    // try opening the file; if we can't,
    // display error and quit
    try {
      out = new FileOutputStream(args[0]);
    }
    catch (Throwable e) {
      System.out.println("Error in opening file");
      System.exit(1);
    }

    PrintStream ps = new PrintStream(out);

    try {
      ps.println("Hello World");
      out.close();
    }
    catch (IOException e) {
      System.out.println("I/O Error");
      System.exit(1);
    }
  }
}
```

Da wir Fehler selbst abfangen, können wir auf „throws…“ verzichten

Diese Fehlerklasse liegt ganz oben in der Hierarchie und fängt damit alles ab

Z.B. Zugriffsrechte „falsch“

Evtl. Fehler hierbei würden nicht abgefangen

Über diese Variable e kann man mehr über den Fehler erfahren

Notausstieg aus dem Programm

Definieren und Auslösen eigener Ausnahmen

- Ausnahmen manifestieren sich als Objekte!

- Eigenen Ausnahmetyp ableiten von java.lang.Exception
- Kann mit „throw“ ausgelöst werden
- „super“: Aufruf des Konstruktors der Basisklasse (im Konstruktor der abgeleiteten Klasse)

```
class IllegalesDatum extends Exception {
  IllegalesDatum(int Tag, int Monat, int Jahr) {
    super("Fehlerhaftes Datum ist: "
          Tag + "." + Monat + "." + Jahr);
  }
}
```

Der Konstruktor von „Exception“ erwartet einen String, der als Fehlermeldung (mit dem Stack-Trace) ausgegeben wird

```
class Datum {
  ...
  void setzen(int T, int M, int J)
                          throws IllegalesDatum {
    Tag = T; Monat = M; Jahr = J;
    if (Tag > 31) throw new
       IllegalesDatum (Tag, Monat, Jahr);
  }
}
class Beispiel {
  ...
  d.setzen(35,05,1997); // Das ist Zeile 39 in
  ...                   // der Datei "Datum.java"
}
```

An der Nutzungsstelle ändert sich syntaktisch nichts

```
Fehlerhaftes Datum
ist: 35.5.97 at Datum.
setzen(Datum.java:27)
at Beispiel.main
(Datum.java:39)
```

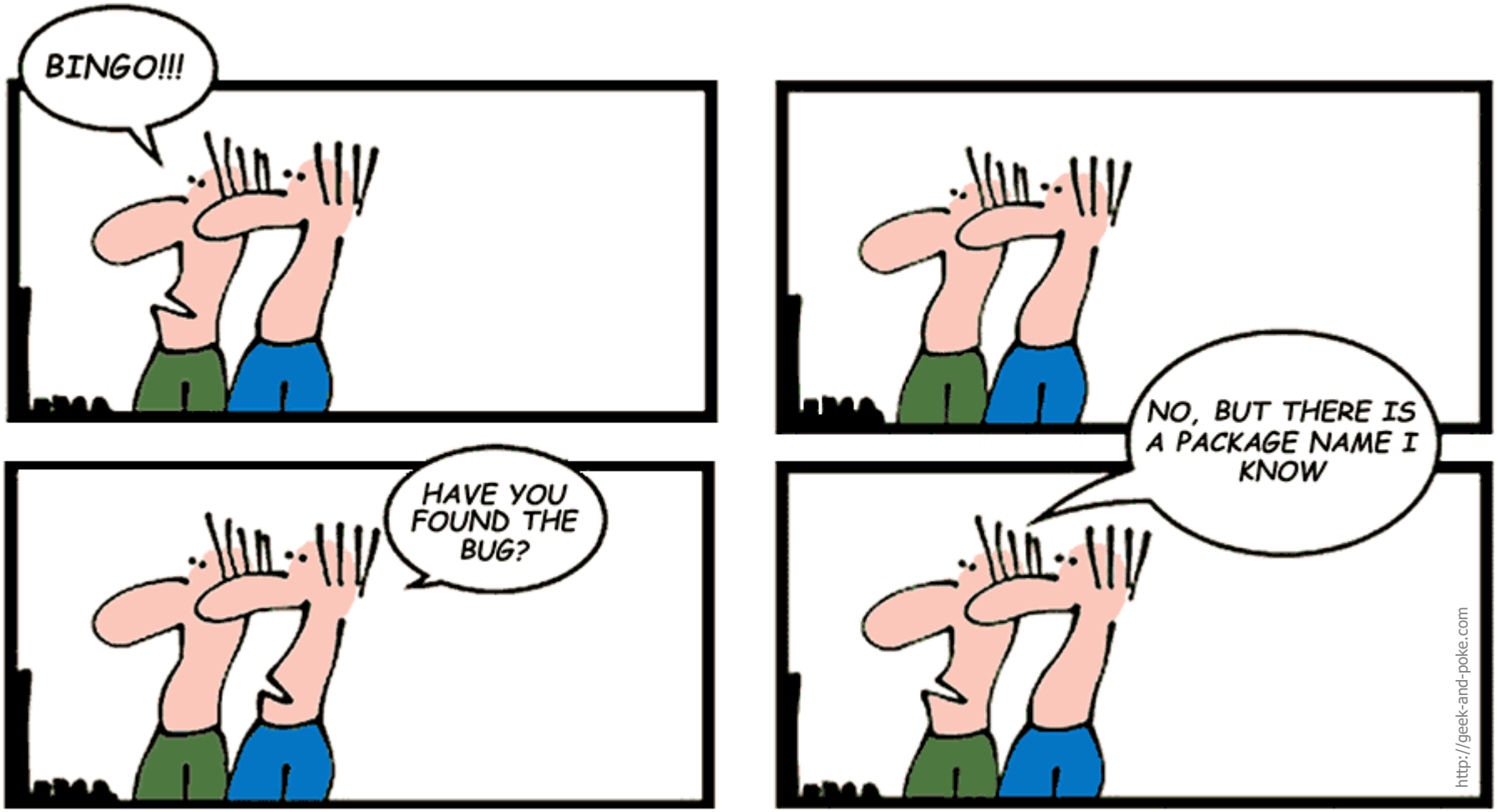
So sieht es dann manchmal in der Praxis aus…

java.lang.IllegalArgumentException: This type of node cannot have more than one incoming connection!

```
at org.jbpm.workflow.core.node.ActionNode.validateAddIncomingConnection(ActionNode.java:50)
at org.jbpm.workflow.core.impl.NodeImpl.addIncomingConnection(NodeImpl.java:100)
at org.jbpm.workflow.core.impl.ConnectionImpl.connect(ConnectionImpl.java:76)
at org.jbpm.workflow.core.impl.ConnectionImpl.<init>(ConnectionImpl.java:71)
at org.jbpm.bpmn2.xml.ProcessHandler.linkConnections(ProcessHandler.java:293)
at org.jbpm.bpmn2.xml.ProcessHandler.end(ProcessHandler.java:145)
at org.drools.xml.ExtensibleXmlParser.endElement(ExtensibleXmlParser.java:422)
at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.endElement(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.xs.XMLSchemaValidator.endElement(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanEndElement(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl$FragmentContentDriver.next(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.XMLDocumentScannerImpl.next(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.XMLNSDocumentScannerImpl.next(Unknown Source)
at com.sun.org.apache.xerces.internal.impl.XMLDocumentFragmentScannerImpl.scanDocument(Unknown Source)
at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.parsers.XML11Configuration.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.parsers.XMLParser.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.parsers.AbstractSAXParser.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl$JAXPSAXParser.parse(Unknown Source)
at com.sun.org.apache.xerces.internal.jaxp.SAXParserImpl.parse(Unknown Source)
at org.drools.xml.ExtensibleXmlParser.read(ExtensibleXmlParser.java:301)
at org.drools.xml.ExtensibleXmlParser.read(ExtensibleXmlParser.java:180)
at org.jbpm.compiler.xml.XmlProcessReader.read(XmlProcessReader.java:46)
at org.jbpm.compiler.ProcessBuilderImpl.addProcessFromXml(ProcessBuilderImpl.java:262)
at org.drools.compiler.PackageBuilder.addProcessFromXml(PackageBuilder.java:673)
at org.drools.compiler.PackageBuilder.addKnowledgeResource(PackageBuilder.java:709)
at org.drools.builder.impl.KnowledgeBuilderImpl.add(KnowledgeBuilderImpl.java:51)
at org.drools.builder.impl.KnowledgeBuilderImpl.add(KnowledgeBuilderImpl.java:40)
at com.sample.ProcessMain.readKnowledgeBase(ProcessMain.java:31)
at com.sample.ProcessMain.main(ProcessMain.java:20)
```

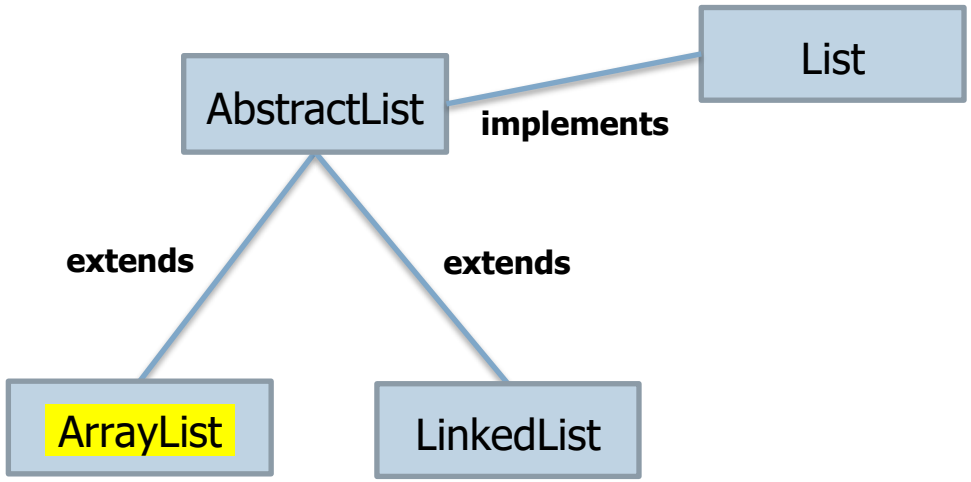
Ein „stack trace", um einen Fehler bis zu seiner eigentlichen (?) Ursache zurückverfolgen zu können

THANK GOD WE HAVE STACKTRACES!



3. ArrayList: Arrays dynamischer Länge

- Dynamisch wachsende Arrays (ähnlich zu std::vector bei C++)
- Einordnung im System der Java-Standardklassen:



Das Interface „List" spezifiziert Methoden für den Zugriff auf Listen (Einfügen und Entfernen von Elementen) sowie Methoden für komplexere Operationen (z.B. Suche in einer Liste)

- Die Klasse ArrayList stellt Möglichkeiten zur Verfügung, um in effizienter Weise direkt auf einzelne Elemente zuzugreifen (wahlfreier Zugriff, random access), analog zum indexbezogenen Zugriff bei normalen Arrays

Array und ArrayList im Vergleich

Array (statisch)

Initialisierung

```
// Zahl der Elemente ist fix
int[] x = new int[10];
```

Einfache Operationen: Hinzufügen, Zugriff, Zuweisung und Entfernen von Elementen

```
x[0] = 7; x[1] = 9; x[2] = 3;
int t = x[1];
x[1] = 8;
t = x[1];
// Entfernen von Elementen
// ist nicht möglich
```

Abfragen der Grösse

```
int laenge = x.length; // 10
```

ArrayList (dynamisch)

```
// ArrayList ist zunächst leer
ArrayList x = new ArrayList();
```

Wir geben keinen konkreten Typ (z.B. `int`) an!

`add`: Am Ende hinzufügen

```
x.add(7); x.add(9); x.add(3);
int t = (Integer)x.get(1); // 9
x.set(1, 8);
t = (Integer)x.get(1); // 8
x.remove(1);
```

`get`/`set`: Index angeben

Elemente rechts rücken eine Position nach links

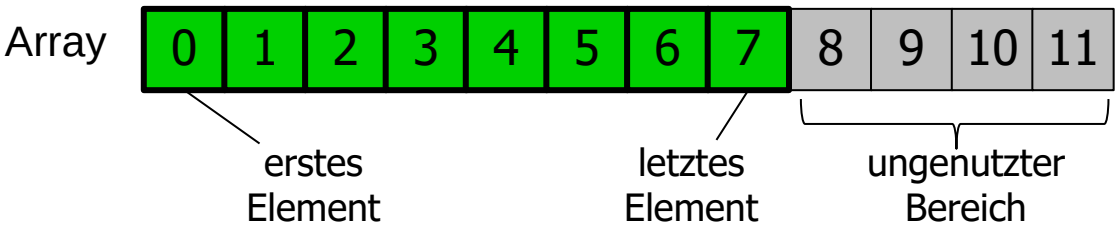
Typumwandlung nötig, da `get` eine Instanz vom Typ `Object` zurückgibt

```
int laenge = x.size(); // 2
```

java.util.ArrayList (Java) vs. std::vector (C++)

- Nur geringfügige Unterschiede in der Nutzung
  - C++: `int zahl = liste[index];`
    (Überladung des Operators [])
  - Java: `int zahl = liste.get(index);`

- Realisierung: Allokation eines Speicherbereichs, der grösser sein kann als die Menge der aktuell gespeicherten Elemente. Ist die maximale Länge des Bereichs erreicht, werden die vorhandenen Elemente automatisch in einen um einen Faktor x verlängerten Array-Bereich kopiert.
  - Java: x = 1.5 (Verlängerung um 50%)
  - C++: x = 2 (Verdopplung)



Vorsicht: Die in der Java Class Library vorhandene Klasse *java.util.Vector* gilt als veraltet!

4. Generics (Parametrisierte bzw. „generische" Typen)

- `ArrayList` kann Objekte beliebigen Typs speichern:

```
ArrayList liste = new ArrayList();
liste.add(7);
liste.add("Hallo");
int i = (Integer)liste.get(0);
String s = (String)liste.get(1);
```

Flexibel, aber fehleranfällig: Der Compiler kennt den Inhalt der Liste i.Allg. nicht und verlässt sich auf die (nötigen) Typumwandlungen. Dies kann bei falschen Annahmen zu Laufzeitfehlern führen, wie z.B. bei:
`int i = (Integer)liste.get(1);`

- → Typsicherheit durch Nutzung von Generics
  - Datentypfehler werden schon zur Übersetzungszeit erkannt:

```
ArrayList<Integer> liste =
    new ArrayList<Integer>();
liste.add(7);
liste.add("Hallo");
int i = liste.get(0);
String s = liste.get(1);
```

Fehlermeldung durch Compiler

Durch den Typparameter `<Integer>` weiss der Compiler, dass die Liste nur Elemente vom Typ „`Integer`" enthält. Ausserdem ist eine explizite Typumwandlung von „`Object`" in „`Integer`" nicht mehr nötig

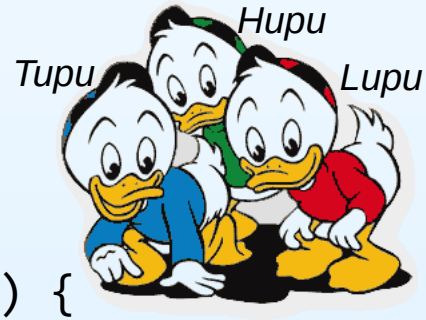
Beispiel einer ArrayList mit Generics

```
// Erstelle Liste und füge zwei Elemente hinzu
ArrayList<String> liste = new ArrayList<String>();
liste.add("Tupu");
liste.add("Hupu");

// Ausgabe aller Elemente
int groesse = liste.size();
for (int i = 0; i != groesse; i++) {
    System.out.println(i + ". Element: " + liste.get(i));
}

// Untersuche, ob "Lupu" ein Element der Liste ist
if (liste.contains("Lupu"))
    System.out.println("Lupu ist Teil der Liste");
else
    System.out.println("Lupu ist nicht Teil der Liste“);

// Lösche alle Elemente der Liste
liste.clear();
```



`contains` ist eine nützliche Methode von **`ArrayList`**, die über die Funktionalität normaler Arrays hinausgeht

Übersicht aller Methoden: Vgl. dazu die Java-API-Dokumentation:
https://docs.oracle.com/en/java/javase/ → API Documentation → java.base → java.util

Resümee des Kapitels

- Interfaces
  - Spezifikation entkoppelt Nutzung von Implementierung
- Parametrisierte Typen („generics“)
  - Flexible Typsicherheit
- ArrayList
  - Arrays dynamischer Länge
- Exceptions
  - Definieren eigener Ausnahmen (*extends Exception; throws; throw*)
  - Strukturieren mittels *try* und *catch*