Wiederholung/Erweiterung elementarer Java-Konzepte
(Skript: AP1, AP2 Kapitel 2, Anfang Kapitel 3)

- (Konventionen)
- (JavaDoc)
- (Paketkonzept)
- Ausnahmebehandlung
- Typparameter

Der Stoff des Kapitels wird eventuell in anderer Reihenfolge eingeschoben.

Teilweise wird er nicht behandelt (s. 1. Semester).

Motivation

- Teamarbeit erfordert **Stilregeln** und **Dokumentation**!

- Große Systeme erfordern ein Ordnungskonzept: **Pakete**

  (1. Semester)

- **Fehlermeldungen** müssen intern weitergereicht werden können ohne die Lesbarkeit zu beeinträchtigen.

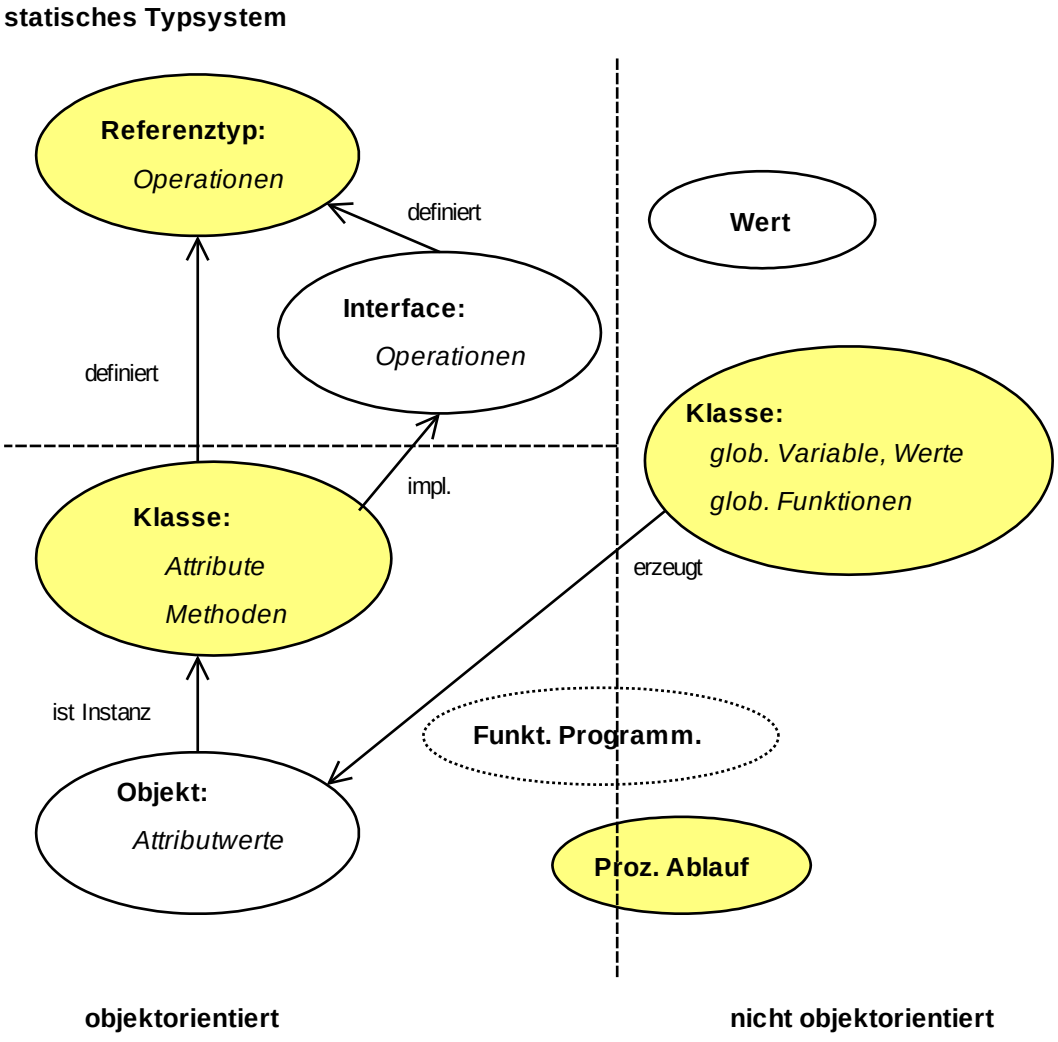
  (n**euer Stoff**)

- **Typangaben** sollen möglichst präzise sein. Sie sollen Fehlersicherheit, Lesbarkeit und Werkzeuge unterstützen.

  (n**euer Stoff**)

Aber das Ganze soll die Sprache nicht komplizierter machen :-)

Fokus: Layout, Dateien, Typen (aber nicht Objektorientierung)



Prozedurale Programmierung in Java

```
import static java.lang.Math.log;

public final class Mathematik {  // keine Vererbung
    private Mathematik() {}      // keine Objekterzeugung

                                 // globale Konstante
    public static final double LG2 = log(2)/log(10);

                                 // globale Funktion
    public static long faculty(int n) {
        return (n == 0) ? 1 : n * faculty(n - 1);
    }
}
```

… das ist nichts anderes als die prozedurale Programmierung in C.

Warum existiert der bedingte Ausdruck: *`Bedingung ? Ausdruck1 : Ausdruck2`* ?

Ist **`if`**`(! istSchwarz)` besser als **`if`**`(schwarz == `**`false`**`)` ?

Doch noch 1 mal: *im 2. Teil (Algorithmen) wird prozedurale Programmierung vorausgesetzt.*

Konventionen für Klassen (zum Nachlesen)

Schreibweise von Namen: Klasse:: groß (Substantiv), Variable: klein (Attribut), Methode: klein (Verb), Paket: klein, Konstante: durchgängig groß

- In der Regel sind ***alle Variablen*** `private`!
- Ein Objekt wird **möglichst vollständig über den *Konstruktor initialisiert*!**
- Wenn möglich, bevorzugt man ***unveränderliche Objekte*.**
- Es ist sinnvoll, ***unveränderliche Attribute*** als `final` zu deklarieren.
- Methoden und Konstruktoren, die nicht für allgemeine Verwendung gedacht sind, werden `private` deklariert.
- Klassen, Methoden und Konstruktoren, die nur innerhalb eines Pakets verwendet werden, deklariert man mit *Paketsichtbarkeit* (s.u.)
- Damit ein Objekt richtig ausgegeben wird, sollte man die Methode `toString` **überschreiben.**
- Bei Bedarf wird `equals` **und** `hashCode` **überschrieben** (kommt später).
- Fehler werden konsequent über ***Ausnahmebehandlung*** (s.u.) gemeldet.
- Klassen von denen man nicht ableiten will/soll, werden `final` deklariert.

Beispielklasse (zum Nachlesen)

```
package mymath;

public final class Bruch {                  // keine Vererbung
   private final long nenner;               // keine Veränderung
   private final long zaehler;

   public Bruch(long zaehler, long nenner) {
      if (nenner == 0) throw new ArithmeticException();
      this.zaehler = gekuerzter zaehler;
      this.nenner = gekuerzter nenner;
   }

   public Bruch addiere(Bruch summand) {
      return new Bruch( ... );
   }

   public String toString() { // wird z.T. automatisch gerufen
      return zaehler + "/" + String.valueOf(nenner);
   }

}
```

JavaDoc (zum Nachlesen)

Idee: Elemente der Schnittstelle müssen dokumentiert werden.
Die Dokumentation sollte als gut lesbarer Hypertext zur Verfügung stehen, gleichzeitig aber im Programmcode stehen und mit diesem gepflegt werden.

Lösung: besondere Kommentare /** … */ und besondere Tags (`@param`).

Der erste Satz eines Kommentars stellt eine Kurzbeschreibung dar.

Klassenkommentare beschreiben die Aufgabe der Klasse.

Methodenkommentare beschreiben die Aufgabe der Methode, die Bedeutung der Parameter und des Rückgabewertes sowie die relevanten Ausnahmen.

Alle öffentlichen Klassenelemente sollten kommentiert sein.

Nicht kommentieren: Selbstverständlichkeiten, die schon am Methodenkopf erkennbar sind.

JavaDoc, wichtige Tags (zum Nachlesen)

`@param` ***`Name`*** *`Erklärung`*	Der Parameter *Name* wird erläutert
`@return` `Erklärung`	Der Rückgabewert wird erläutert
`@throws` ***`Ausnahmeklasse`*** *`Erkl.`*	Die Ausnahme wird erläutert
`@see` ***`Klasse#Methode`***	Querverweis

Kommentare können Html-Tags enthalten:

`<code> .. </code>`	für Programmelemente (oder `<tt> .. </tt>`)
`<pre> .. </pre>`	für Programmbeispiele
`<p>`	für Absatz

Java Packages werden durch die Datei *package.html* im Paket-Verzeichnis dokumentiert. Der Inhalt ist in **`<body> .. </body>`** eingeschlossen.

Hinweis: In Eclipse muss bei erster Verwendung javadoc eingerichtet werden.

JavaDoc Beispiel(zum Nachlesen)

```
/**
 * Die Klasse Bruch stellt unveränderliche Brüche zur
 * Verfügung.
 */
public final class Bruch {

   /**
    * Erzeugt einen neuen Bruch. Der Nenner darf nicht 0
    * sein. Zähler und Nenner müssen nicht gekürzt sein.
    *
    * @param zaehler Zähler des Bruchs.
    * @param nenner Nenner des Bruchs
    * @throws ArithmeticException wenn der Nenner 0 ist
    */
   public Bruch(long zaehler, long nenner) { .. }

   /**
    * Gibt die Summe des Empfängerobjekts und des
    * Summanden zurück.
    * @param summand der Summand
    * @return Summe der beiden Brüche
    */
   public Bruch addiere(Bruch summand) {
```

Pakete gliedern Softwarebibliotheken (zum Nachlesen, Wiederholung)

Eine Klasse beschreibt gleichartige Objekte.
Zu einem Anwendungsbereich gehören viele Klassen.
Evtl. sind Klassen als Bibliothek gebündelt.
=> Klassen werden zu Paketen zusammengefasst:

Grundlagen:	`java.lang`
Ein-/Ausgabe:	`java.io`
Hilfsmittel:	`java.util`
GUI:	`java.awt`
	`javax.swing`
3rd party:	`com.sun.beans.decoder`

Namensregel: *umgekehrte URL . lokaler Paketname*

Was bedeuten Pakete für die Programmierung in Java, wie definiert und wie nutzt man Sie?

Definiton von Paketzugehörigkeit (zum Nachlesen)

Syntax:

`package` ***Namensbestandteil*** { `.` ***Namensbestandteil*** } `;`

Die Paketanweisung ist die 1. Anweisung einer Übersetzungseinheit und bezieht sich auf die gesamte Datei.

Alle Klassen dieser Datei gehören zu dem angegebenen Paket.

Der Paketname besteht aus mehreren durch Punkt getrennten Namensbestandteilen.

Die Teile sollen den logischen Zusammenhang der Pakete ausdrücken.

Für Java regelt der vollständige Paketname die Sichtbarkeit von Klassen.
(Es gibt keine Subpakete)

Fehlt die package-Deklaration, so gehören die Klassen der Übersetzungseinheit zu dem anonymen Defaultpaket.

Bei Android müssen Klassen in einem weltweit einmaligen Paket stehen.

Beispiel (Wiederholung)

Datei: `meinPaket/Klasse1.java`:

```
package meinPaket;

public class Klasse1 {
    private String text;                  // nur in der Klasse
    /* default */
        void paketUtility() {}            // nur im Paket
    protected void hilfsfunktion() {} // Paket und Vererbung
    public void oeffentlich() {}          // überall
}
```

Datei: `meinPaket/Klasse2.java`:

```
package meinPaket;

class Klasse2 {                           // nur für meinPaket
    void irgendwas(Klasse1 partner) {
        partner.paketUtility();           // erlaubt!
    }
}
```

Beispiel (Wiederholung)

```
// import meinPaket.Klasse1;
// import meinPaket.*;

class PaketTest {                    // im anonymen-Paket ist public nie nötig
    public static void main(String[] argv) { // beim main aber immer!
        meinPaket.Klasse1 k1;
        meinPaket.Klasse2 k2;  // FEHLER
        k1 = new meinPaket.Klasse1();
        k1.paketUtility();     // FEHLER
        k1.oeffentlich();
    }
    ...
}
```

Der volle Name eine Klasse besteht aus *Paketname.Klassenname*. Dadurch werden Namenskollisionen zwischen verschiedenen Paketen vermieden.

Die Import-Anweisung importiert nicht eine Datei, sondern nur einen Paketnamen!

Außerhalb eines Paketes ist nur **`public`** und **`protected`** sichtbar.

Die Importanweisung (zum Nachlesen)

```
import Paketname.Klassenname;
import Paketname.*;

import static Paketname.Klassenname.Elementname;
import static Paketname.Klassenname.*;
```

Die Importanweisung führt die Möglichkeit der Abkürzung ein.

Import ermöglicht, innerhalb der aktuellen Datei den Paketnamen wegzulassen. Die zweite Form (*) gilt für alle Klassen des Pakets.

Was Import nicht tut (die Antworten aus Klausuren):
es inkludiert nichts, bindet nichts ein, macht nichts verfügbar, lädt nichts in den Speicher, bindet keine Methoden ein, stellt nichts bereit...

Import ermöglicht nur das Abkürzen des Klassennamens!
Static import kürzt nur den Zugriff auf statische Klassenelemente ab.

Aufbau einer Java-Datei (Wiederholung)

```
package Paketname;                    // Paketzugehörigkeit
import Paketname.Klassenname;         // import einer Klasse
import Paketname.*;                   // import aller Klassen
import static Paket.Klasse.Element;   // import eines stat. Elements
import static Paket.Klasse.*;         // import aller stat. Elemente
```

Eine oder mehrere der folgenden Klassen oder Schnittstellendefinitionen:

```
(public)? enum Name { . . . }

(public)? (final)? class Name { … }

(public)? abstract class Name { … }

(public)? interface Name { … }
```

Maximal eine Klasse/Interface darf mit **`public`** gekennzeichnet sein.
Die Datei muss den Namen dieser öffentlichen Klasse/Interface tragen.

Suche nach Classfiles

Für die Übersetzung einer Java-Datei oder für die Ausführung müssen Classfiles lokalisiert werden. Die folgenden Regeln beziehen sich demnach auf javac und java. Sie gelten nicht für Eclipse! Jedes System kann seine eigenen Regeln definieren!!

Details müssen Sie auch selbst in den Manuals nachlesen!

1. Systembibliotheken (Bootstrap/Extension Classes) werden nach internen Regeln gesucht. Die wichtigste Datei ist `rt.jar`. Sie enthält die Java-API

2. Ausgangspunkt für die weitere Suche ist die Liste der Verzeichnisse (und jar-Archive) in der Umgebungsvariablen CLASSPATH. Wenn diese Variable nicht existiert, wird nur im aktuellen Verzeichnis gesucht.

3. Der Classfile `Klasse.class` für eine Klasse im anonymen Paket liegt unmittelbar in einem der `CLASSPATH`-Verzeichnisse.

4. Der Classfile der Klasse `ebene1.ebene2.Klasse` findet sich in einem (in der `CLASSPATH`-Liste vorhandenen) Unterverzeichnis `ebene1` in dem ein Unterverzeichnis `ebene2` existiert, das `Klasse.class` enthält. Der vollständige relative Pfad innerhalb eines Der CLASSPATH-Verzeichnisse lautet: `ebene1/ebene2/Klasse.class`

5. Wenn der Compiler die Quelle weiterer Dateien benötigt, orientiert er sich am CLASSPATH, bzw. am SOURCEPATH.

Pfade und Eclipse

Eclipse verwendet nicht die Systemvariablen. Statt dessen werden Pfade für jedes Projekt gesondert eingestellt.

Die Einstellungen findet man unter *Project Properties → Build Path*.
Dort gibt es Tabs:

Source : wo befinden sich die Quellen?
Libraries : welche Bibliotheken verwende ich?

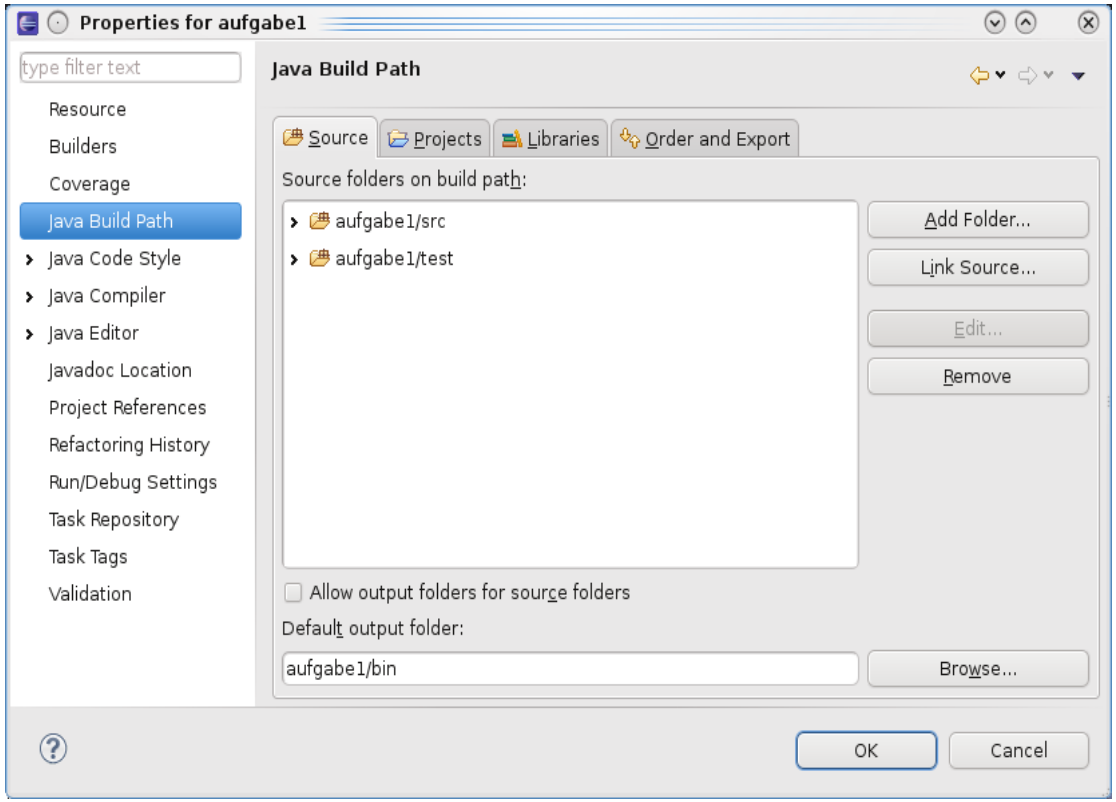
Sonderfall: Wenn ich Eclipse-Projekte exportiere, muss ich mich wieder um die Pfade kümmern.

Hinweis: Jar-Archive erlauben es, Anwendungen zu erzeugen, die man auch per Mausklick starten kann. Alles Nötige findet sich in der Java-Manuals!

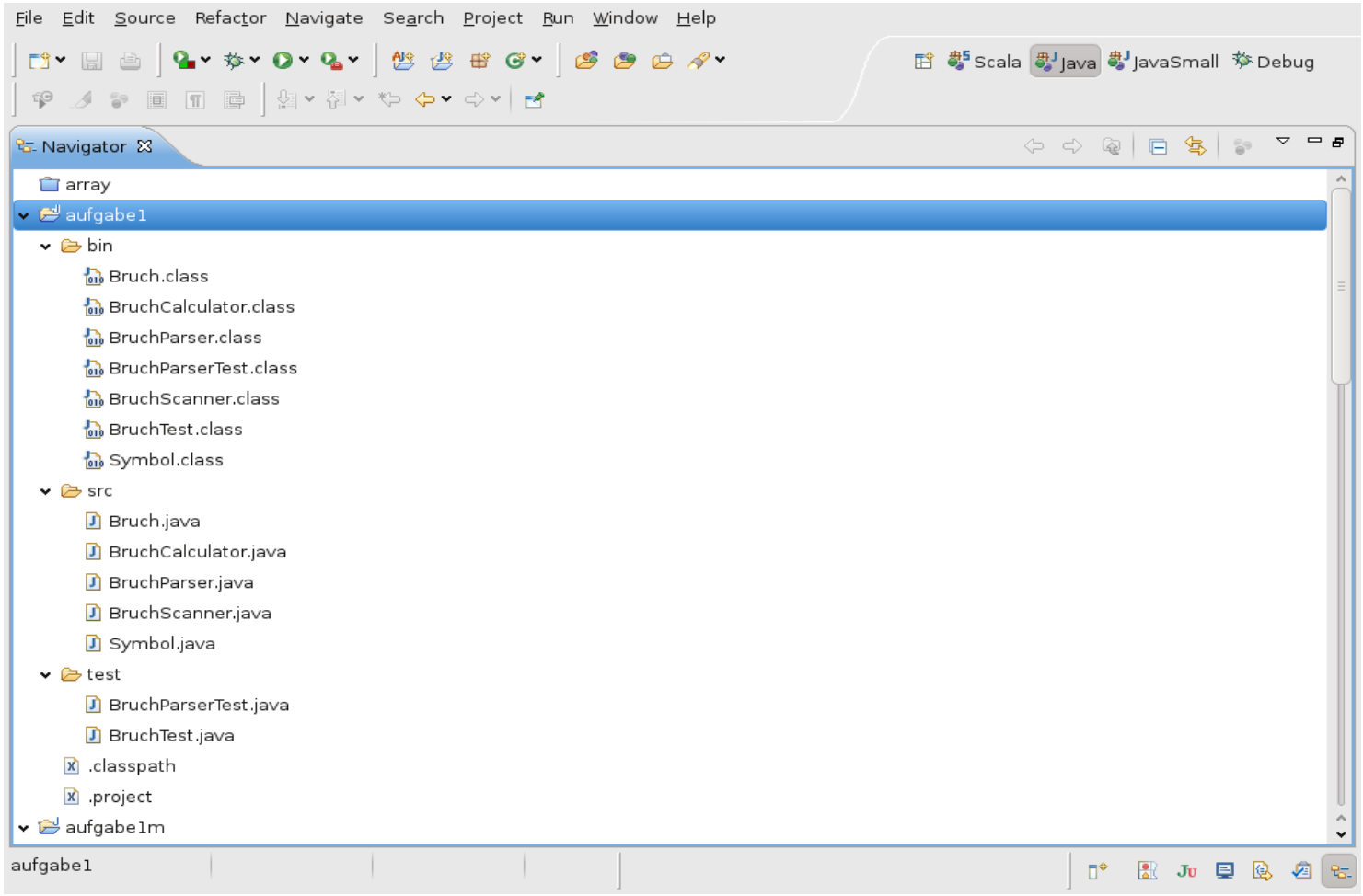
Pfade und Eclipse

Das Projekt „aufgabe1 ist bei mir so eingestell:

- Quelltexte unter `aufgabe1/src`
- Testquellen unter `aufgabe1/test`
- Classfiles unter `aufgabe1/bin`



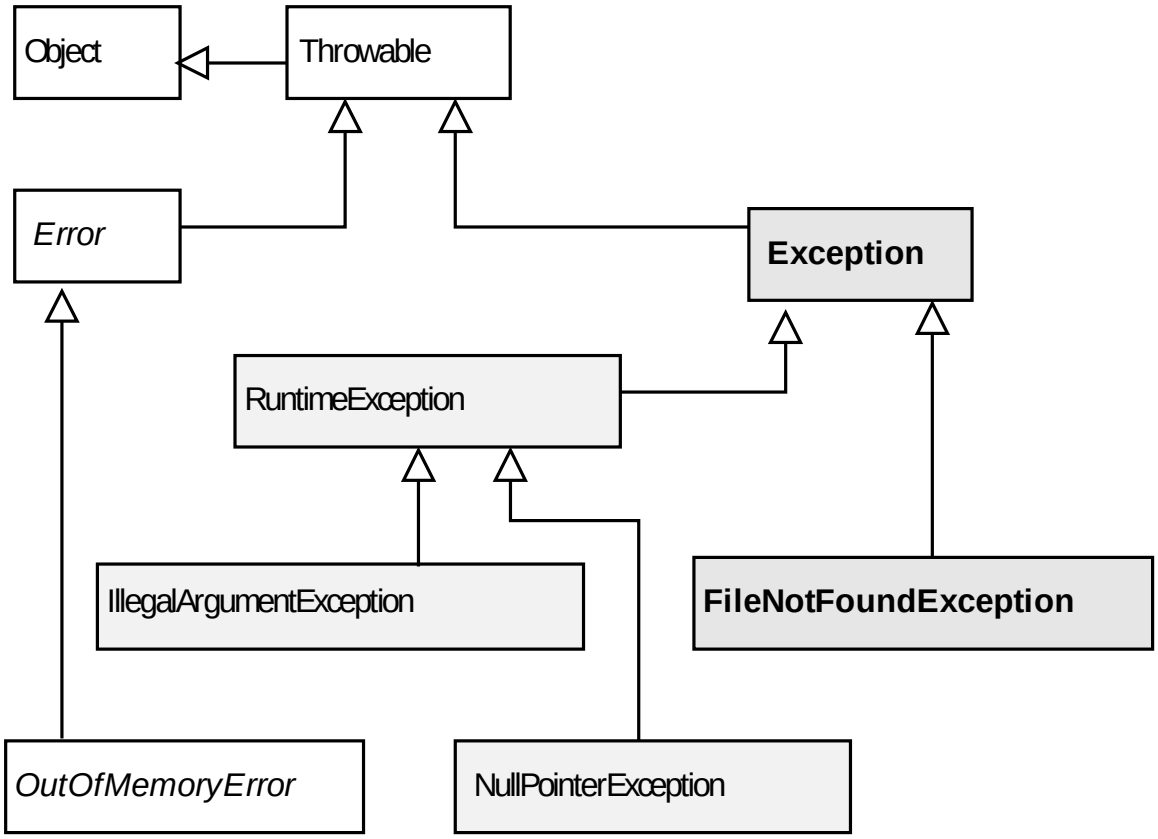
Darstellung des Projekts im Eclipse Navigator



Fehlerbehandlung in Java

- Syntaktische Fehler und die meisten Typfehler werden durch den **Compiler** gefunden.
- Fehler, die vom Programmablauf abhängen, werden erst zu **Laufzeit** entdeckt.
- Java als sichere Sprache findet grundsätzlich alle Fehler, die gegen die Sprachdefinition verstoßen.
- **Laufzeitfehler werden durch Ausnahmeobjekte (exception) gemeldet.**

Hierarchie der Exceptions (Überblick)



Error Runtime Exception Checked Exception

Erläuterung des Diagramms

`Throwable`: Superklasse von allem und gesamte Funktionalität

`Error`: fatale Fehler (z.B. Hardware), mit dem man meist nichts anfangen kann

`Exception`: meist unerwartetes Ereigniss

`RuntimeException`: unerwartetes Ereignis oder auch bloßer Programmierfehler

alle Exceptions außer `RuntimeException` sind **checked Exceptions**

checked Exceptions stehen für Ereignisse, mit denen der Programmierer rechnen muss.

„checked“ = geprüft bedeutet: der Compiler prüft, dass nur solche Ausnahmen von einer Klasse geworfen werden, die im Methodenkopf deklariert sind (throws-Klausel).

Motivation für Ausnahmen

Der Sinn einer Ausnahme besteht darin, dass Laufzeitfehler so gemeldet werden, dass ihre Ursache zur Laufzeit erkannt oder vom Programmierer behoben werden kann (Meldung an den Verursacher).

Dazu wird ein Fehlerobjekt erzeugt, dass die Art der Ausnahme (durch den Namen der Klasse), eine weitere Beschreibung (durch einen Meldungstext) und die Information über den Ort des Auftretens im Programmablauf (*stack trace*) enthält.

Im Defaultfall wird diese Information ausgegeben und das Programm wird beendet.

Es ist extrem dumm, wenn man Exceptions auffängt, nur um die Fehlermeldung zu unterdrücken!

Fehler, die die virtuelle Maschine schon erkennt, müssen nicht noch mal vom Programmierer geprüft (und damit verhindert werden).

Automatische Mechanismen sind meist besser als selbst programmierte!

Es ist nicht der Sinn eines Programms, Fehlermeldungen zu unterdrücken, sondern Fehler (wenn sie nicht vermieden werden können) sinnvoll zu melden!

Werfen einer Ausnahme

Wenn die passende Ausnahme nicht schon automatisch geworfen wird, benötigt man zur Fehlermeldung die Throw-Anweisung.

Beispiele:

```
throw new IllegalArgumentException();

if (param < 0)
   throw new IllegalArgumentException("Parameter " + param +
      " ist nicht erlaubt ");
```

Wirkung: der normale Programmfluss wird unterbrochen, jede Methode wird beendet und es wird solange zurückgesprungen, bis der Fehler aufgefangen wird.

Geprüfte Ausnahmen (checked exceptions), die eine Methode werfen kann, müssen im Methodenkopf mittels `throws` … angegeben werden. Eine Methode, die eine Schnittstellenoperation implementiert oder eine andere Methode überschreibt, darf nicht mehr geprüfte Ausnahmen werfen, als in der Schnittstelle/Oberklasse angegeben.

Ausnahmen die man kennen sollte (checked exceptions immer) sollten durch den Javadoc-Tag `@throws` *`Exception`* beschrieben werden!

Geprüfte und ungeprüfte Ausnahme

```
/**
 * Oeffnet eine Textdatei zum Lesen.
 * @param name Dateiname
 * @throws FileNotFoundException wenn die Datei nicht gefunden wird
 */
public oeffneDatei(String name) throws FileNotFoundException {
    return new Scanner(new FileReader(name));
}

/**
 * Liest eine ganze Zahl.
 * @param scanner Objekt zum Lesen
 * @throws InputMismatchException wenn das naechste Token keine Zahl ist
 * @throws NoSuchElementException wenn das Dateiende erreicht ist
 * @throws IllegalStateException wenn der Scanner geschlossen ist
 */
public int readNumber(Scanner scanner) {
    return scanner.nextInt();
}
```

Hinweise

Es ist dient der Lesbarkeit, wenn man die Fehlerbehandlung modularisiert (Sie können dazu auch eine Hilfsklasse schreiben. Durch Fehlermethoden werden Programme lesbarer!

Allgemeine Bedingungsprüfung mit IllegalArgumentException:

```
public static void require(boolean condition) {
    if (! condition) throw new IllegalArgumentException();
}
```

Die Anwendung sieht dann ganz einfach aus:

```
require(betrag > 0);
```

Während dies eine ganz allgemeine Funktion ist, kann man aber auch in einzelnen Klassen spezifische Bedingungen formulieren.

Die Assert.Anweisung stellt einen Sonderfall dar. Sie wird nur ausgeführt, wenn man dies durch eine Compilerangabe ausdrücklich will. Sie eignet sich damit auch für aufwändige Prüfungen, die nur im bei Fehlerverdacht durchgeführt werden sollen.

```
assert istSortiert(arrray), "das Array muss sortiert sein!";
```

Schema einer eigenen Ausnahmeklasse:

Wenn es keine passende Ausnahmeklasse in der API gibt, definiert man eigene Ausnahmeklassen. In aller Regel werden nur die Konstruktoren definiert. Im Prinzip kann die Ausnahmeklasse aber auch eigene Attribute und Operationen enthalten. Zwingend ist nur die Ableitung von einer anderen Ausnahmeklasse!

```
public class MyException extends RuntimeException {

    public MyException() {
        super();
    }

    public MyException(String message) {
        super(message);
    }
}
```

Auffangen einer Ausnahme und Finally

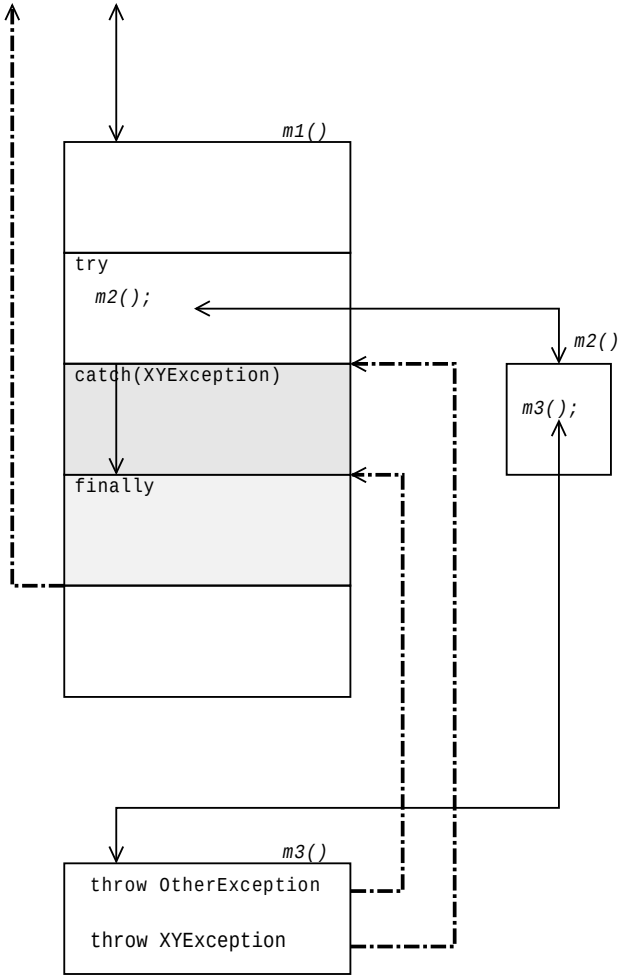
Wenn man einen aufgetretenen Fehler **sinnvoll** behandeln kann, wird man die Ausnahme *an geeigneter Stelle auffangen*.

Beispiel:

```
try {
    Anweisungen
}
catch (IllegalArgumentException e) {
    Fehlerbehandlung
}
catch (Exception alleAnderen) {
    Fehlerbehandlung
}
finally {
    in jedem Fall auszuführen
}
```

- Tritt eine Ausnahme im Try-Block auf, wird der erste passende Catch-Block ausgeführt.
- Der optionale Finally-Block wird in jedem Fall (auch ohne Fehler ausgeführt). Anschließend wird evtl. die Fehlerbehandlung fortgesetzt.
- Oberklassen passen auf Unterklassen-Ausnahmen.
- Neben dem Try-Block muss mindestens ein Catch- oder ein Finally-Block auftreten.

Kontrollfluss der Ausnahmebehandlung



Hinweise:

Ausnahmebehandlung ist kein „Programmabsturz“ sondern ein kontrollierter Umgang mit Ausnahmesituationen und Fehlern.

(Man *kann* Fehler nicht vermeiden – man muss mit Fehlern rechnen!)

Die Throws-Deklaration hat *keinerlei* Einfluss auf die Ausnahmebehandlung

durchgezogene Linie = normaler Ablauf

gestrichelt = Ablauf bei Ausnahme

Typparameter (Motivation)

`java.util.ArrayList` ist eine komfortable Alternative zu Java-Arrays.

Eine Array-Deklaration legt den Elementtyp eines Arrays fest:

```
String[] stringArray;
```

Wie ist das bei ArrayList?

```
ArrayList<String> stringArrayList;
```

In den spitzen Klammern steht der Typparameter der Klasse `ArrayList`.
Genauso geht das auch bei der Objekterzeugung:

```
new String[20];
new ArrayList<String>();
```

Ziel: exakte Typinformation für den Compiler und damit Fehlerprüfung!

```
stringArrayList.add("abc");            // ok
stringArrayList.add(new Bruch(1,2));   // Fehler
String s = stringArrayList.get(0);     // ok
```

Typparameter (schon mal vorweg: Interfaces)

```
class Bruch implements Comparable<Bruch>
```

heißt: Die Klasse Bruch hat die folgende Methode:

```
int compareTo(Bruch andererBruch) { ... }
```

Dies ist festgelegt durch ein Interface (wird demnächst besprochen):

```
public interface Comparable<T> {
    public int compareTo(T other);
}
```

Die Typparameter <**Bruch**> im Klassenkopf sagt, dass in unserem Fall **T** durch **Bruch** zu ersetzen ist.

Anm.: Dadurch, dass der Klassenkopf feststellt, dass Brüche `Comparable` (*vergleichbar*) sind, lassen sich auch für Brüche Bibliotheksfunktionen, wie `Arrays.sort`, nutzen.

Typparameter (Klassendefinition)

```
class Stack<T> {                    // mehr Stacks brauchen wir nicht!!
    private Object[] a;
    int top = 0;

    public Stack() {
        a = new Object[20];
    }
    public void push(T x) { a[top++] = x; }

    @SuppressWarnings("unchecked")
    public T pop() { return (T) a[--top]; }

    public boolean isEmpty() { return top  == 0; }
}
```

Anmerkungen:

- Typparameter sind nur für den Compiler da, nicht zur Laufzeit ! (***type erasure***)
- Als Typparameter sind nur Referenztypen erlaubt!
- Man kann keine Arrays mit dem Typ eines Parameters erzeugen: ~~`new T[n]`~~
- Ein Cast `(T)` ist manchmal nötig, aber bewirkt nichts (Warnung).

Typparameter (kleine Anwendung)

```
public static void main(...) {
    Stack<String> stringStack = new Stack<String>();

    stringStack.push("hello");
    stringStack.push("world");

    StringBuilder sb = new StringBuilder();
    while (! stringStack.isEmpty())
      sb.append(stringStack.pop());
      if (! stringStack.isEmpty()) sb.append(", ");
    }
    System.out.println(sb.toString()); // toString() nicht notwendig
    // Ausgabe: "world, hello"
}
```

Keine Wertdatentypen als Typparameter – aber Autoboxing!

```
public static void main(...) {
    Stack<Character> charStack = new Stack<Character>();
    // nicht Stack<char> !!!

    charStack.push(Character.valueOf('x'));
    // Autoboxing (d.h. automatisches Einpacken):
    charStack.push('x');

    char a = charStack.pop().charValue();
    // Autoboxing (d.h. automatisches Auspacken):
    char b = charStack.pop();

    ...
}
```

Autoboxing: automatisches Ein- und Auspacken von Wertdaten in die zugehörigen Verpackungsobjekte.

Wertdatentypen sind nicht objektorientiert !!

Nachmals Autoboxing

```
public static void main(...) {
    Object[] verschiedenes = new Object[10];

    verschiedenes[0] = "Hallo";
    verschiedenes[1] = Double.valueOf(1.5);   // bitte kein new!
    verschiedenes[2] = Boolean.valueOf(true); // bitte kein new!
    String s = (String) verschiedenes[0];
    double x = ((Double) verschiedenes[1]).doubleValue();

    // oder einfacher geschrieben:

    verschiedenes[3] = Math.PI;
    double pi = (Double) verschiedenes[3];


    ...
}
```

Autoboxing ist ein Workaround um die Tatsache, dass elementare Typen nicht objektorientiert sind.

Die Casts teilen dem Compiler mit, welchen Typ wir erwarten – mehr nicht!!

Nebenwirkungen ...

Das war nicht alles, leider ist das Java-Typsystem nicht einfach.

Ich werde aber versuchen, die Probleme zu umgehen.

Man kann sich zur Not immer mit Casts und mit @SuppressWarnings helfen.

Es gibt auch Literatur! Z.B.:

Gilad Bracha, *Generics in the Java Programming Language*

Ein Trost:

Java 10 wird vielleicht Java „richtig“ objektorientiert machen und das Typkonzept bereinigen.

Hilfe gibt es überall ... man muss sie nur nutzen :-)

