

Le Tour de



# DART

## Web-Technologien

Client- und Serverseitige Sprachen (Dart Teil I)

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

1



**Prof. Dr. rer. nat.  
Nane Kratzke**

*Praktische Informatik und  
betriebliche Informationssysteme*

- **Raum: 17-0.10**
- **Tel.: 0451 300 5549**
- **Email: [nane.kratzke@fh-luebeck.de](mailto:nane.kratzke@fh-luebeck.de)**

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

2

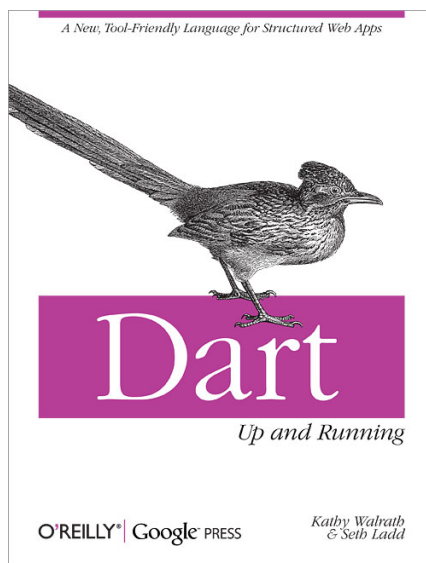
## Dart in a Nutshell ...



- Eine optional typisierte,
- Multiprogrammierparadigmen (imperativ, non dogmatic objektorientiert, funktional) unterstützende,
- Language VM (Virtual Machine) basierte Programmiersprache (die eine Transcompilation nach Javascript ermöglicht)
- für Scalable Web App Engineering,
- die sowohl In-Browser als auch On-Server (in Dart VM) ausführbar ist.

<https://www.dartlang.org/>

## Zum Nachlesen ...



**Chapter 1:**  
Quick Start

**Chapter 2:**  
A Tour of the Dart Language

**Chapter 3:**  
A Tour of the Dart Libraries

**Chapter 4:**  
Tools

**Chapter 5:**  
Walkthrough: Dart Chat

## Le Grande Boucle



University of Applied Sciences



### Tour de Dart

- Optional Typed
- Multiparadigm
- Byte Code (JavaScript Cross Compiled)
- In Browser and On Server

### Tour de Dart Libraries

- Library System
- Asynchronous Programming
- Math and Random
- Browser-Based Apps
- I/O
- Decoding and Encoding

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

5

## Tour de Dart



University of Applied Sciences



Kern-  
Konzepte

Variablen

Built-In  
Types

Kontrollfluss

Funktionen  
(Methoden)  
und Typedefs

Operatoren

Klassen  
(OO)

Generics

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

6

## Tour de Dart

|                                    |               |
|------------------------------------|---------------|
| Kern-Konzepte                      | Variablen     |
| Built-In Types                     | Kontrollfluss |
| Funktionen (Methoden) und Typedefs | Operatoren    |
| Klassen (OO)                       | Generics      |



## Erstens: Dart ist nicht dogmatisch!

Jedes Dart Programm beginnt (wie Java) in einer `main` Methode.

Dart kennt top level Funktionen (z.B. die `main`). Klassen sind nicht erforderlich.

```
var greet = "Hello World";  
main() {  
  print(greet);  
}
```

Dart ist optional typisiert (dies gilt für Methoden, Funktionen wie für Datenfelder und Variable).

Sie können, müssen aber keine Datentypen bei Deklarationen angeben.

```
main() {  
  print("Hello World");  
}
```

Dart kennt top level Variablen.  
Dart ist dynamisch typisiert.

```
String greet = "Hello World";  
void main() {  
  print(greet);  
}
```



## Runtime Modes

Jedes Dart Programm kann nach JavaScript kompiliert werden und läuft damit in jedem JavaScript fähigem Browser!

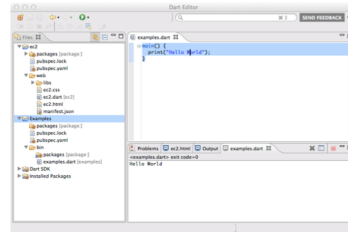
Dart Programme können auch in Dart Virtual Machine ausgeführt werden (ähnlich der Java VM, jedoch ist die Dart VM nicht ByteCode sondern Language-basiert). Damit können Dart Programme auf allen Systemen laufen, für die eine Dart VM existiert (z.B. Desktop, Server, Smartphone etc).

Dart wurde entworfen, um zunehmend komplexere Web Apps entwickeln zu können, aber gleichzeitig soll es einfach zu erlernen sein. Dart macht daher viele Anleihen bei bekannten Sprachen, wie bspw. Java.

Dart wird mit einer Entwicklungsumgebung Dart Editor bereitgestellt.

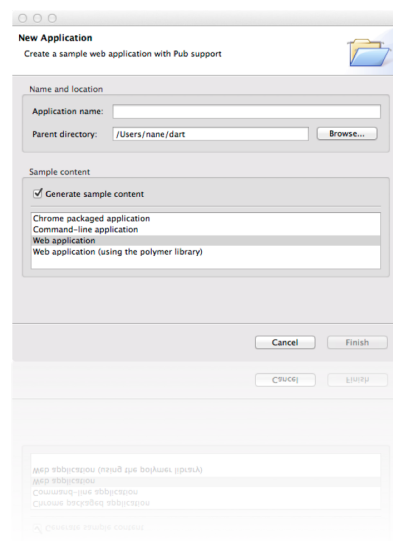
Dart kennt zwei Runtime Modes:

- Im **Production Mode** werden statische Typisierungsfehler ignoriert. Gleichzeitig ist die Ausführungsgeschwindigkeit optimiert (für die Auslieferung).
- Im **Checked Mode** werden statische Typisierungsfehler geprüft und ggf. Exceptions geworfen (für Test- und Entwicklung) sowie Assertions geprüft.



## Install Dart is easy

- Download and Unzip from:
  - <http://www.dartlang.org/downloads.html>
- Start the DartEditor Executable File
- Create an App
  - File -> New Application
  - Command Line Application (for non browser Apps)
  - Web Application (for browser context)



## Hello Darties (I)

```
<!DOCTYPE html>

<html>
  <head>[...]</head>
  <body>
    <h1>Hello ...</h1>
    <p>Static "Hello world" from HTML!</p>

    <div id="hello"></div>

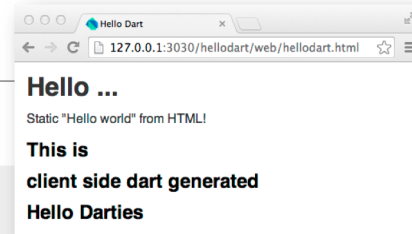
    <script type="application/dart" src="hellodart.dart"></script>
    <script src="packages/browser/dart.js"></script>
  </body>
</html>
```

## Hello Darties (II)

```
import 'dart:html';

main() {
  querySelector("#hello").innerHTML = generateHelloWorld();
}

String generateHelloWorld() {
  final greetings = [
    "This is",
    "client side dart generated",
    "Hello Darties"
  ];
  return greetings.map((g) => "<h2>$g</h2>").join("\n");
}
```



## Tour de Dart



## Variable, Immutables und Konstanten

In Dart sind alle Variablen Referenzen auf Objekte. D.h. der Defaultwert für jede (Referenz)variable ist null.

Das gilt auch für „primitive Datentypen“.

```
String n;  
int a;  
bool b;  
assert (n == null);  
assert (a == null);  
assert (b == null);
```

```
const MAENNLICH = "Max Mustermann";  
const WEIBLICH = "Maren Musterfrau";
```

```
final n = "Tessa Loniki";  
assert (n == "Tessa Loniki");
```

```
n = MAENNLICH;  
// Error: Erneute Wertzuweisung
```

Dart kennt Konstante (const), d.h. Werte, die bereits zur Compilerzeit feststehen.

Dart kennt Immutables (final), d.h. Werte, die einmal festgelegt, im Nachhinein nicht mehr geändert werden können.

## Alles natürlich optional typisiert

In Dart können Variablen auch immer ohne Typ definiert werden (Schlüsselwort `var`). Der Defaultwert bleibt logischerweise `null`.

```
var n = 42;  
var a = "Hello World";  
var b = true;  
assert (n == 42);  
assert (a == "Hello World");  
assert (b == true);
```

```
const String MAENNLICH = "Max Mustermann";  
const String WEIBLICH = "Maren Musterfrau";  
  
final String n = "Tessa Loniki";
```

```
var n;  
var a;  
var b;  
assert (n == null);  
assert (a == null);  
assert (b == null);
```

Initialisierung von Variablen ist aber ggf. verständlicher (insbesondere bei untypisiert definierten Variablen).

Konstante (`const`) und Immutables (`final`) können natürlich auch typisiert definiert werden.

## Wann nutze ich Immutables? **final** first

Wie sie noch aus Programmieren II (Multithread Programmierung) wissen, sind die Kriterien für Thread Safeness die

- Unveränderlichkeit bzw.
- Korrekte Synchronisierung
- oder korrekte Kapselung

Grundsätzlich ist es also immer sinnvoll Immutables zu nutzen, um Software auf gute Parallelisierbarkeit und Thread Safeness auszuliefern.

Wenn Sie Variablen oder Datenfelder deklarieren, nutzen sie doch immer erst einmal `final` (definieren sie also erst einmal Immutables). Erst wenn sie keine pragmatische Lösung mit `finals` finden, können sie immer noch `var` nutzen.

Je mehr funktionale Programmierkonzepte sie nutzen, desto weniger `var` benötigen sie, desto weniger zustandsbehaftet programmieren sie, desto weniger schwer zu findende Fehler programmieren sie!

## final first Geht das denn?

```
final table = [
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9, 10, 11, 12],
  [13, 14, 15, 16]
];

print(toHtmlTable(table));
```

Gegeben ist links stehende Datenstruktur. Diese soll mit einer Funktion `toHtmlTable()` in unten stehende HTML Zeichenkette überführt werden.

Aus Programmieren I wissen Sie, dass Sie solche Probleme mit zwei ineinander geschachtelten `for` Schleifen gut realisieren können.

```
<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>9</td><td>10</td><td>11</td><td>12</td></tr>
<tr><td>13</td><td>14</td><td>15</td><td>16</td></tr>
</table>
```

## final first Geht das denn?

```
final table = [
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9, 10, 11, 12],
  [13, 14, 15, 16]
];

print(toHtmlTable(table));
```

```
String toHtmlTable(List<List> rows) {
  var ret = "";
  for (List row in rows) {
    ret += "<tr>";
    for (var col in row) {
      ret += "<td>$col</td>";
    }
    ret += "</tr>\n";
  }
  return "<table>\n$ret</table>";
}
```

Ihre Lösung sieht vermutlich in etwa wie folgt aus.

**Problem:** Die Variable `ret` ist nicht Immutable. Damit ist der Code schlecht parallisierbar, da von einem zentralen und sich ändernden Zustand `ret` abhängig.

```
<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>9</td><td>10</td><td>11</td><td>12</td></tr>
<tr><td>13</td><td>14</td><td>15</td><td>16</td></tr>
</table>
```

## final first Geht das denn?

```
final table = [
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9, 10, 11, 12],
  [13, 14, 15, 16]
];

print(toHtmlTable(table));
```

```
String toHtmlTable(List<List> rows) {
  var ret = "";
  rows.forEach((row) {
    ret += "<tr>";
    row.forEach((col) {
      ret += "<td>$col</td>";
    });
    ret += "</tr>\n";
  });
  return "<table>\n$ret</table>";
}
```

Dart bietet mit der `forEach` Methode eine Möglichkeit an, Listen per Callback zu durchlaufen.

**Problem:** Eine Variable `ret` bleibt weiter erforderlich. Letztlich ist `forEach` nur eine elegantere `for` Schleife.

```
<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>9</td><td>10</td><td>11</td><td>12</td></tr>
<tr><td>13</td><td>14</td><td>15</td><td>16</td></tr>
</table>
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

19

## final first Geht das denn?

```
final table = [
  [1, 2, 3, 4],
  [5, 6, 7, 8],
  [9, 10, 11, 12],
  [13, 14, 15, 16]
];

print(toHtmlTable(table));
```

```
String toHtmlTable(List<List> rows) {
  final table = rows.map((row) {
    final zeile = row.map((col) {
      return "<td>$col</td>";
    }).join();
    return "<tr>$zeile</tr>\n";
  }).join();
  return "<table>\n$table</table>";
}
```

Dart bietet mit der `map` Methode aber auch eine Möglichkeit an, Listen rein funktional (d.h. zustandslos) zu verarbeiten.

**Vorteil:** Variable sind nicht mehr erforderlich. Der Code wäre damit gut parallelisierbar.

```
<table>
<tr><td>1</td><td>2</td><td>3</td><td>4</td></tr>
<tr><td>5</td><td>6</td><td>7</td><td>8</td></tr>
<tr><td>9</td><td>10</td><td>11</td><td>12</td></tr>
<tr><td>13</td><td>14</td><td>15</td><td>16</td></tr>
</table>
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

20

## final first Geht das denn?

### Offenbar schon ...

Mehr zu Funktionaler Programmierung mit Dart kommt noch.

```
String toHtmlTable(List<List> rows) {  
  final table = rows.map((row) {  
    final zeile = row.map((col) {  
      return "<td>$col</td>";  
    }).join();  
    return "<tr>$zeile</tr>\n";  
  }).join();  
  return "<table>\n$table</table>";  
}
```

## final first Geht das denn?

Wenn sie also in Zukunft eine Variable (Zustand) mittels `var` definieren wollen, sollte in ihrem Kopf folgender Prozess ablaufen.

```
var zustand = 3 + x + 7 + y + 8 + z;
```

```
var zustand = 3 + x + 7 + y + 8 + z;
```

```
final ausdrück = 3 + x + 7 + y + 8 + z;
```

Meistens definiert man nämlich keinen Zustand, sondern will nur einen komplizierten Ausdruck abkürzen, um ihn an anderer Stelle einzusetzen.

Wieder was für Parallelisierbarkeit getan. Funktionale Programme kennen keinen sich verändernden Zustand (nur Immutables) und sind daher sehr gut parallelisierbar.

## The Downfall of Imperative Programming

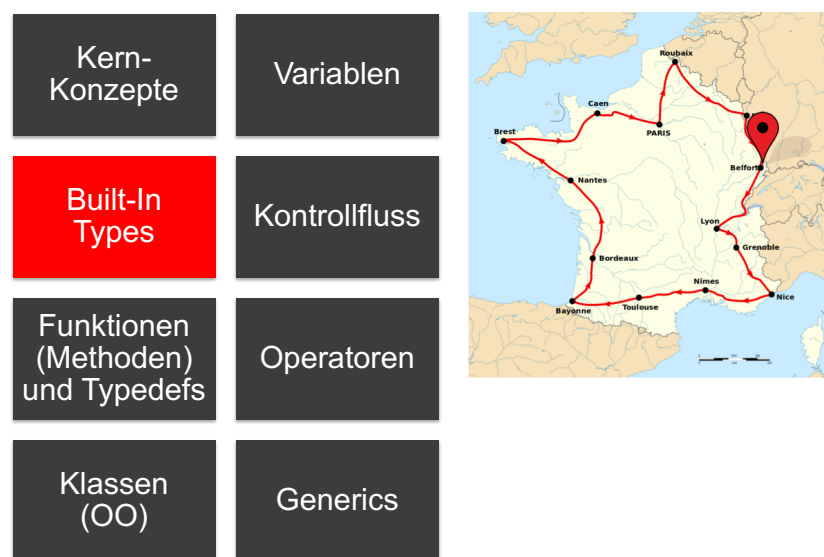
*„If programmers were electricians, parallel  
programmers would be bomb disposal experts.  
Both cut wires [...]“*

Bartosz Milewski, „The Downfall of Imperative Programming“

Quelle (letzter Zugriff am 22.12.2013):

<https://www.fpcomplete.com/business/blog/the-downfall-of-imperative-programming/>

## Tour de Dart





## Darts built-in Types int, double, num

Numbers

Strings

Booleans

Lists/Sets

Maps

```
// Ganzzahlige Werte koennen so definiert werden
// Werte koennen beliebig groß werden
var i1 = 42;
int i2 = 42;
int earthPopulation = 8000000000;
assert (i1 == 42);
assert (i2 == 42);

// Fließkommawerte koennen so definiert werden
// und haben doppelte Genauigkeit
var d1 = 42.0;
double d2 = 42.7;
assert (d1 == 42.0);
assert (d2 == 42.7);
```

num bezeichnet numerische Wertetypen, d.h. double und int.

## Darts built-in Types int, double, num

Numbers

Strings

Booleans

Lists/Sets

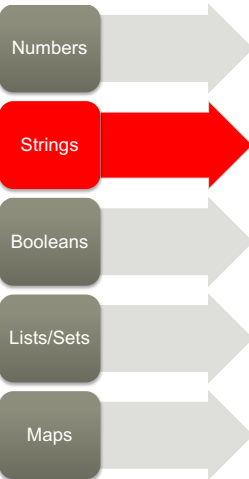
Maps

```
var i1 = 42;
int i2 = 42;
var d1 = 42.0;
double d2 = 42.7;

// int und double sind in Dart Objekte
// d.h. es gibt hilfreiche Methoden/getter
assert (i1.isEven);
assert (!i2.isOdd);
assert (d1.floor() == 42);
assert (d2.ceil() == 43);
```

## Darts built-in Types

### Strings



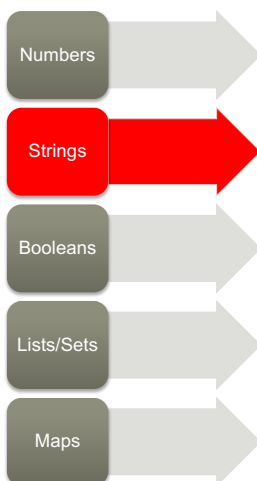
```
// In Dart koennen sie Strings single oder
// double quoted definieren.
var singlequote = 'Hello my name is rabbit';
String doublequote = "This is rabbit's home.";

// Dart kennt die ueblichen Escapesequenzen \
var escaped = 'This is rabbit\'s home';
assert (escaped == "This is rabbit's home");

// Dart kennt raw Strings (ohne Escape oder
// Interpolation)
var raw = r"This is rabbit\n home";
```

## Darts built-in Types

### Strings



```
// Zeichenkettenkonkatenation funktioniert
// (wie in Java) mit dem + Operator
var concat = "Hello" + " " + "World";
assert (concat == "Hello World");

// Oder auch so (Zeichenketten hintereinander)
var anotherConcat = "Hello" " " "World";
assert (anotherConcat == "Hello World");

// Das funktioniert natuerl. in der double, single
// oder raw Variante (und Kombinationen)
var crazyConcat = "Hello" ' ' r"World";
assert (crazyConcat == "Hello World");
```

## Darts built-in Types Strings

Numbers

Strings

Booleans

Lists/Sets

Maps

```
// Dart kennt String Interpolation
// String Interpolation wird durch ein $
// aehnlich wie in PHP gekennzeichnet
var name = "Max";
assert ("Hello $name" == "Hello Max");

// Aber Dart kann mehr als PHP
// Man kann auch Ausdruecke auswerten
assert ("Hello ${name.toUpperCase()}" == "Hello MAX");

// D.h. wir koennen auch rechnen
assert ("1 + 1 = ${1 + 1}" == "1 + 1 = 2");

// oder Konstante auswerten (und das kann PHP nicht)
const pi = 3.14;
assert ("PI = $pi" == "PI = 3.14");
```

## Darts built-in Types bool

Numbers

Strings

Booleans

Lists/Sets

Maps

```
// Wahrheitswerte werden in Dart mittels dem Datentyp
// bool ausgedrückt. Ein bool kann zwei Werte annehmen:
// true und false.
var unkorrekt = false;
bool korrekt = true;

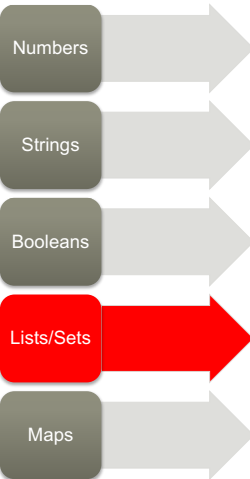
// Wenn Dart einen Wahrheitswert erwartet, dann wird nur
// true
// zu wahr ausgewertet. Alle anderen von true verschiedenen
// Werte, werden zu falsch ausgewertet (anders als in C,
// PHP,
// JavaScript, ... !!!).
var person = 'Bob';
if (person) {
  print('You have a name!');
  // Würde in JavaScript ausgegeben werden (denn != 0)!
  // Aber in Dart: person != true => false
  // Im Production Mode würde nichts ausgegeben werden
  // Im Checked Mode würde eine Exception ausgelöst werden
  // (String != bool)
}
```

## Darts built-in Types (Lists)

[https://api.dartlang.org/docs/channels/stable/latest/dart\\_collection/Li](https://api.dartlang.org/docs/channels/stable/latest/dart_collection/Li)  
[stBase.html](#)



University of Applied Sciences



```
// Lists sind mit Index 0 startende Sequenzen von Werten.  
// list.length - 1 bezeichnet den Index des letzten Elements  
// einer Liste. Darts Listennotation entspricht  
(untypisiert)  
// bspw. der von JavaScript.  
var list = [1,2,3];  
assert(list.length == 3);  
assert(list[1] == 2);  
  
// Listen können natürlich auch typisiert angelegt werden.  
// Hierzu bedient man sich der Generic Notation, vglb.  
// der Java Notation.  
List<String> typedList = <String>["1", "2", "3"];  
assert(typedList.length == 3);  
assert(typedList[0] == "1");  
assert(typedList[0] != 1);  
  
// Listen können auch typisiert angelegt, aber untypisiert  
// referenziert werden.  
var anotherList = <int>[1, 2, 3];
```

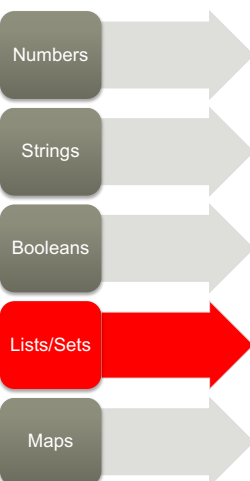
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

31

## Darts built-in Types Lists (forEach, map)



University of Applied Sciences



```
// Listen haben mehrere praktische Methoden zum Durchlaufen  
var teas = ["green", "black", "roisboos", "earl grey"];  
  
// Wir können auch Listen mittels einer forEach Methode  
// durchlaufen, ähnlich wie mit einer foreach Schleife.  
teas.forEach((tea) => print("I drink tea $tea"));  
  
// Dies ist gleichbedeutend mit folgender Kontrollstruktur  
for (final tea in teas) {  
  print("I drink tea $tea");  
}  
  
// Mittels map kann eine Funktion  
// auf alle Elemente einer Liste angewendet werden.  
// forEach/map? If in doubt use map  
final loudTeas = teas.map((tea) => tea.toUpperCase());  
print(loudTeas);  
// Erzeugt auf Konsole: (GREEN, BLACK, ROISBOOS, EARL GREY)
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

32

## Darts built-in Types

### Lists (any, every)



Numbers

Strings

Booleans

Lists/Sets

Maps

```
// Mit den Methoden any und every stehen Ihnen ferner
// die aus der Prädikatenlogik bekannten Quantoren
// "es existiert (mindestens ein)" (any)
// und "für alle" (every) zur Verfügung.

// Sehr praktisch um Vor- und Nachbedingungen für Methoden
// auszudrücken.

// Beispiel
// Bedingungsdefinition für koffeinfreie Teesorten t
// Rotbusch- und Kamillentees sind koffeinfrei (teeinfrei)
caffeineFree(t) => ["rooibos", "chamomile"].contains(t);

var someTeas = ["green", "black", "rooibos", "earl grey"];
assert(someTeas.any(caffeineFree));
// es existieren koffeinfreie Tees
assert(!someTeas.every(caffeineFree));
// aber nicht alle Tees sind koffeinfrei
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

33

## Darts built-in Types

### Lists (where, contains, isEmpty)



Numbers

Strings

Booleans

Lists/Sets

Maps

```
// Mittels der Methode where können wir aus Listen
// Elemente herausfiltern. Mittels contains prüfen
// ob ein Element Bestandteil einer Liste ist.
// Hier alle koffeinhaltigen Teesorten

caffeineFree(t) => ["rooibos", "chamomile"].contains(t);
var someTeas = ["green", "black", "rooibos", "earl grey"];

print(someTeas.where((t) => !caffeineFree(t)));
// Ergibt auf der Konsole: (green, black, earl grey)

// Und ganz banale Dinge wie prüfen ob eine Liste leer ist
// können wir natürlich auch (isEmpty).
// isEmpty ist dabei als getter implementiert
// (mehr dazu folgt noch)

if (someTeas.where(caffeineFree).isEmpty) {
  print("Verzeihung, keine Koffeinfreien Heißegetränke.");
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

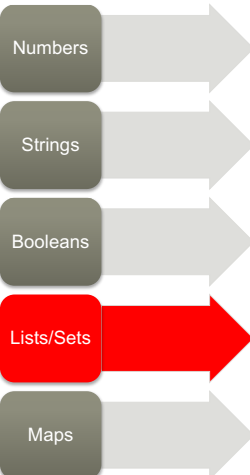
34

## Darts built-in Types (Sets)

[https://api.dartlang.org/docs/channels/stable/latest/dart\\_core/Set.html](https://api.dartlang.org/docs/channels/stable/latest/dart_core/Set.html)



University of Applied Sciences



```
// Eine Menge ist eine nicht indizierte Liste von Elementen  
// ohne Doppelungen.  
// Auf Mengenelemente kann daher nicht per Index  
// zugegriffen werden. Mengen haben (leider) auch keine  
// Sprachlitterale, sondern müssen  
// mittels Konstruktoren angelegt werden.
```

```
var ingredients = new Set.from(  
  ["gold", "titanium", "xenon"]  
); // untypisiert  
  
Set<String> nobleGases = new Set<String>.from(  
  ["xenon", "argon"]  
); // typisiert  
assert(ingredients.length == 3);  
assert(nobleGases.length == 2);
```

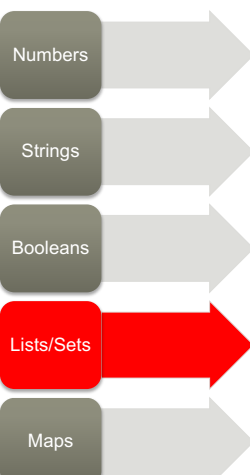
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

35

## Darts built-in Types Sets (intersect, union)



University of Applied Sciences



```
// Das hinzufügen von doppelten Elementen zu einer Menge hat  
// keine Auswirkungen.  
nobleGases.add("argon");  
assert(nobleGases.length == 2);  
assert(nobleGases.contains("xenon"));  
assert(nobleGases.contains("argon"));
```

```
// Dafür können wir jetzt die Mengenoperationen  
// intersection (Schnittmenge),  
// union (Vereinigung)  
// und containsAll (Teilmenge von) nutzen.  
var intersect = nobleGases.intersection(ingredients);  
assert(intersect.length == 1);  
assert(intersect.contains("xenon"));
```

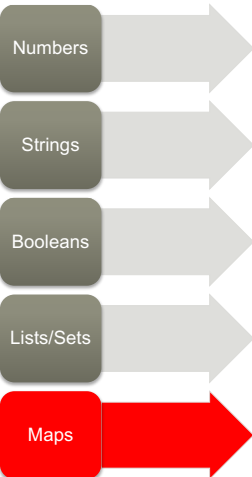
```
var allElements = nobleGases.union(ingredients);  
assert(allElements.length == 4);  
assert(allElements.containsAll(nobleGases));  
assert(allElements.containsAll(ingredients));
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

36

## Darts built-in Types (Maps)

[https://api.dartlang.org/docs/channels/stable/latest/dart\\_core/Map.html](https://api.dartlang.org/docs/channels/stable/latest/dart_core/Map.html)



```
// Maps sind eine ungeordnete Liste von Key Value Paaren.  
// Maps verknüpfen einen Schlüssel mit einem Wert.  
// Maps können mit folgenden Literalen { : } angelegt  
werden.  
  
var hawaiianBeachMap = {  
  "oahu" : ["waiki", "kailua", "waimanalo"],  
  "big island" : ["wailea bay", "pololu beach"],  
  "kauai" : ["hanalei", "poipu"]  
}; // untypisiert  
  
Map<String, List<String>> typedBeachMap = {  
  "oahu" : ["waiki", "kailua", "waimanalo"],  
  "big island" : ["wailea bay", "pololu beach"],  
  "kauai" : ["hanalei", "poipu"]  
}; // typisiert  
  
// Maps können auch über einen Konstruktor angelegt werden  
var searchTerms = new Map();  
// gleich: var searchTerms = {};  
var typedTerms = new Map<String, List<String>>();  
// gleich: Map<String, List<String>> typedTerms = {};
```

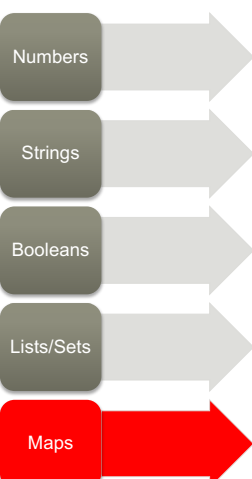
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

37

## Darts built-in Types Maps (keys, [])



University of Applied Sciences



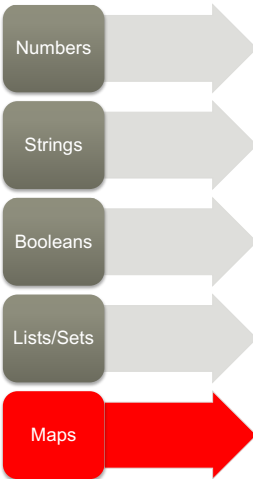
```
var hawaiianBeaches = {  
  "oahu" : ["waiki", "kailua", "waimanalo"],  
  "big island" : ["wailea bay", "pololu beach"],  
  "kauai" : ["hanalei", "poipu"]  
};  
  
// Mittels eckiger Klammern kann man über Schlüssel auf  
// Werte  
// einer Map zugreifen.  
assert (hawaiianBeaches["florida"] == null);  
assert (hawaiianBeaches["kauai"].contains("hanalei"));  
  
// Mittels des Properties keys (Iterable) kann man alle  
// Schlüssel einer Map ermitteln  
var keys = hawaiianBeaches.keys;  
assert (keys.length == 3);  
assert (keys.contains("oahu"));  
assert (!keys.contains("florida"));
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

38

## Darts built-in Types

### Maps (containsKeys, forEach)



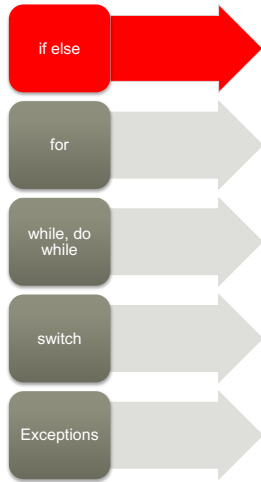
```
var hawaiianBeaches = {  
  "oahu" : ["waiki", "kailua", "waimanalo"],  
  "big island" : ["wailea bay", "pololu beach"],  
  "kauai" : ["hanalei", "poipu"]  
};  
  
// Mittels containsKeys kann geprüft werden, ob ein Key  
// vorhanden ist.  
assert (hawaiianBeaches.containsKey("oahu"));  
assert (!hawaiianBeaches.containsKey("florida"));  
  
// Alle key value Paare einer Map lassen sich wie folgt  
// durchlaufen.  
for (String i in hawaiianBeaches.keys) {  
  print ("Visiting $i to swim at ${hawaiianBeaches[i]}");  
}  
  
// Was gleichbedeutend mit dieser forEach Methode ist  
hawaiianBeaches.forEach((i, beaches) {  
  print ("Visiting $i to swim at $beaches");  
});
```

## Tour de Dart





## Kontrollfluss



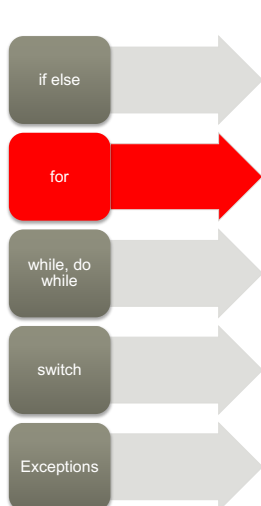
```
// Bedingte Anweisungen

bool isRaining = false;
bool isSnowing = true;

if (isRaining) {
  you.bringRainCoat();
} else if (isSnowing) {
  you.wearJacket();
} else {
  car.putTopDown();
}

// Wichtig!
// Dart behandelt alle Wert != true als false!
```

## Kontrollfluss for, for in, forEach



```
// Dart kennt natürlich die klassische Zählschleife.
var candidates = ["Bob", "Eliza", "Mike", "Tara"];
for (int i = 0; i < candidates.length; i++) {
  print ("Interviewing ${candidates[i]}");
}

// Collections lassen sich aber auch mit for in
// durchlaufen
for (String candidate in candidates) {
  print ("Interviewing $candidate");
}

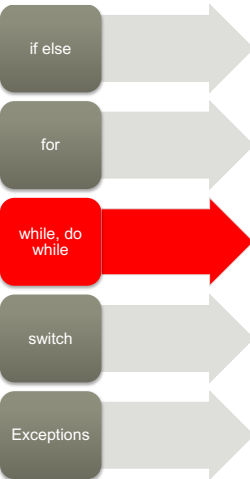
// Oder aber mit einem Closure in einer forEach
// Methode
candidates.forEach((candidate) {
  print ("Interviewing $candidate");
});
```

## Kontrollfluss

### while, do while



University of Applied Sciences



```
// Kaum erstaunlich. Dart kennt abweisende while
var lst = ["A", "B", "C"];
while (lst.isNotEmpty) {
  print (lst.removeLast());
}

// und nicht abweisende do while Schleifen.
var furtherList = [];
var eingabe = "";
do {
  eingabe = stdin.readLineSync();
  if (eingabe != "STOPP") furtherList.add(eingabe);
} while (eingabe != "STOPP");
print ("You entered the following list:"
      $furtherList);
// Wichtig! In Dart sind alle Werte != true, false!
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

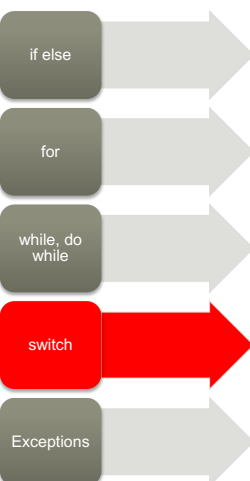
43

## Kontrollfluss

### switch



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

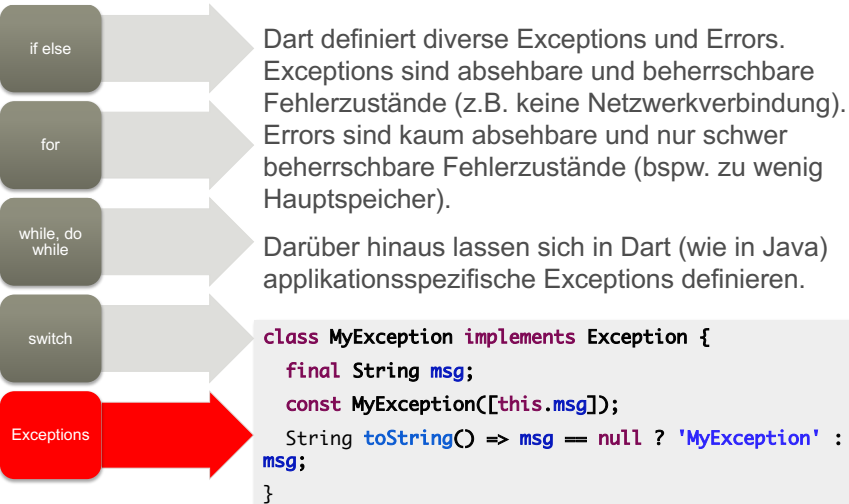


```
// Auch Dart hat diese eigenwillige switch
// Anweisung (Mehrfachverzweigung).
// Do not forget the break!
var command = "OPEN";
switch (command) {
  case "OPEN":
    print ("Opening file");
    break;
  case "CLOSE":
    print ("Closing file");
    break;
  case "COPY":
    print ("Copying file");
    break;
  default:
    print ("Not touching the file");
}
```

44

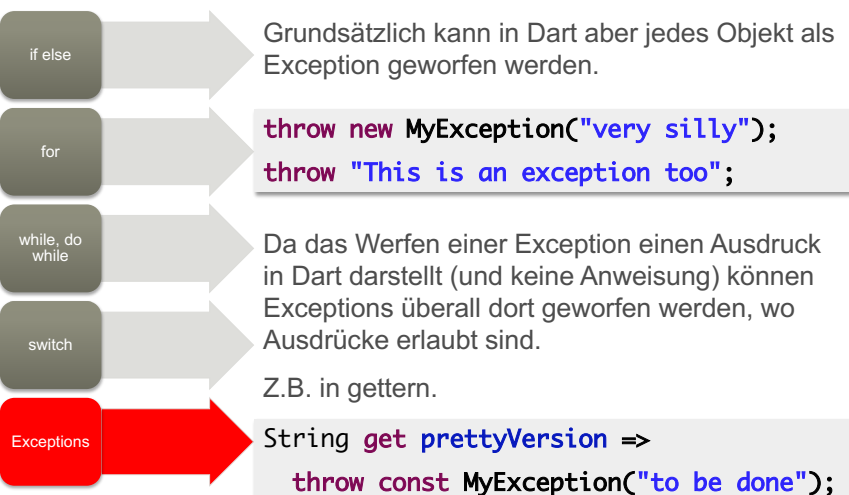
## Kontrollfluss

### Exceptions definieren (implements Exception)



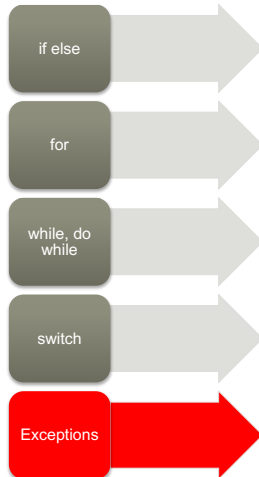
## Kontrollfluss

### Exceptions werfen (throw)



## Kontrollfluss

### Exceptions behandeln (try, catch, on, finally)



Das behandeln einer Exception stoppt die Propagierung entlang des Call Stacks.

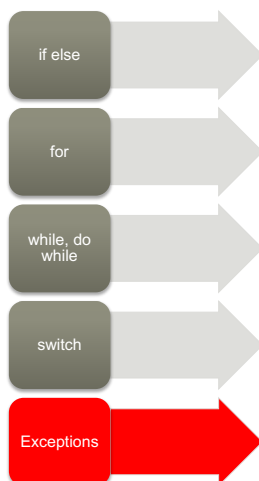
Wie in Java können Code Bereiche in Dart dazu unter Exception Kontrolle innerhalb einer try Blocks laufen.

Können in einem try Block mehr Exception Typen auftreten, lassen sich auch mehrere catch Klauseln hierzu definieren.

Ist man nur an dem Typ der Exception interessiert nutzt man on, ist auch das Exception Objekt für die Behandlung von Interesse nutzt man (ergänzend) catch.

## Kontrollfluss

### Exceptions behandeln (try, catch, on, finally)



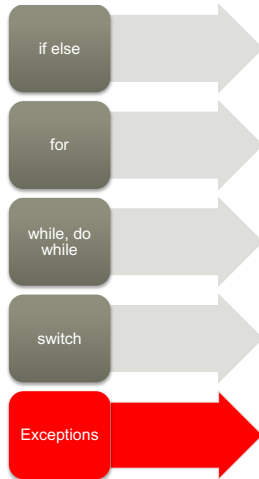
```
try {  
  breedMoreLlamas();  
} on OutOfLlamasException {  
  // Exception mit spezifischem Typ  
  // Exception Objekt unnötig  
  buyMoreLlamas();  
} on Exception catch(e) {  
  // Alles andere, Hauptsache Exception  
  // Jedoch sind wir am Exception Objekt  
  // interessiert  
  print('Unknown exception: $e');  
} catch(e) {  
  // Kein Exceptiontyp angegeben, wird alles fangen  
  print('Something really unknown: $e');  
}
```

## Kontrollfluss

Exceptions behandeln (**try**, **catch**, **on**, **finally**)



University of Applied Sciences



Code im **finally** Block stellt sicher, dass Anweisungen sowohl im Exceptionfall als auch im fehlerfreien Fall ausgeführt werden.

```
try {  
    breedMoreLlamas();  
} on OutOfLlamasException {  
    buyMoreLlamas();  
} catch(e) {  
    print('Something really unknown: $e');  
} finally {  
    cleanLlamaStalls();  
    // Immer aufräumen (finally wird immer ausgeführt)  
    // Auch wenn eine Exception geworfen wurde.  
}
```

**Hinweis:** Wenn keine catch Klausel auf eine Exception greift (bzw. definiert wurde), wird diese nach Abarbeitung des **finally** Blocks entlang des Call Stacks propagiert.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

49

## Tour de Dart



University of Applied Sciences



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

50

## Funktionen Funktionsdefinitionen



```
// Funktionen werden im wesentlichen definiert wie in anderen Sprachen auch.
// Per Konvention werden Funktionen üblicherweise typisiert definiert.
String helloXYt(String vn, String nn) {
    return "Hello $vn $nn";
}

// Funktionen können aber natürlich auch
// untypisiert definiert werden. Dies reduziert
// jedoch häufig die Verständlichkeit.
helloXY(vn, nn) {
    return "Hello $vn $nn";
}

// Enthält eine Funktion nur einen Ausdruck, so kann die Funktion short-hand
// ausgedrückt werden. Üblicherweise wird diese Variante untypisiert eingesetzt.
helloXYsh(vn, nn) => "Hello $vn $nn";

// Man kann short-hand functions aber natürlich auch typisiert definieren.
String helloXYsht(String vn, String nn) => "Hello $vn $nn";
```

```
void main() {
    print(helloXYt("Max", "Mustermann"));
    print(helloXY("Max", "Mustermann"));
    print(helloXYsh("Max", "Mustermann"));
    print(helloXYsht("Max", "Mustermann"));
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

51

## Funktionen Optional named/positional parameters



```
// Parameter können optional sein, ggf. default Werte (:) haben und benannt ({} )
// werden.
helloXYop({vn : "Max", nn : "Mustermann"}) => "Hello $vn $nn";

// Optionale Parameter können aber auch anhand ihrer Position ([]) definiert werden und
// ebenfalls ggf. default Werte haben (=).
helloXYopp(String vn, [String nn, String greet = "Hello"]) {
    // Um zu prüfen, ob ein Parameter gesetzt wurde kann dieser gegen null geprüft
    // werden.
    return "$greet $vn ${nn != null ? nn : ""}";
}

// Optionale Parameter (ob benannt oder nicht) folgen immer den required (normalen)
void main() {
    assert(helloXYop() == "Hello Max Mustermann");
    assert(helloXYop(nn: "Musterfrau") == "Hello Max Musterfrau");
    assert(helloXYopp("Maren") == "Hello Maren ");
    assert(helloXYopp("Maren", "Musterfrau") == "Hello Maren Musterfrau");
    assert(helloXYopp("Maren", "Musterfrau", "Hi") == "Hi Maren Musterfrau");
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

52

## Funktionen

### Funktionen sind first-class Objekte (und ggf. anonym)

```
void printElement(elem) {  
    print(elem);  
}  
  
// Funktionen können als Parameter anderen Funktionen übergeben werden.  
final list = [1, 2, 3];  
list.forEach(printElement); // Gibt 1 2 3 aus  
  
// Funktionen können auch an Variable/Immutableables zugewiesen werden.  
final method = printElement;  
  
// Funktionen können anonym definiert werden (short hand oder klassisch).  
final loudify = (String s) => "!!! ${s.toUpperCase()} !!!";  
assert (loudify("hello world") == "!!! HELLO WORLD !!!");  
  
final calmify = (String s) {  
    return s.toLowerCase();  
};  
assert(calmify("HELLO WORLD") == "hello world");
```

## Funktionen

### Funktionen als closures

```
// Funktionen können sich Variablen/Immutableables des umgebenden Scopes merken.  
// (und diese damit in ihre Implementierung einschließen, daher closure).  
// Folgendes Beispiel definiert eine Methode makeAdder(), die einen Parameter n  
// definiert und auf Basis dessen eine neue anonyme Funktion erzeugt.  
// Die erzeugte anonyme Funktion schließt diesen Parameter in ihre Implementierung  
// ein  
// und "erinnert" sich an n wenn sie aufgerufen wird.  
  
// Versuchen sie das nicht mit Java (7) :-)  
  
function makeAdder(num n) { // erzeugt eine anonyme Funktion (Inkrement um n)  
    return (num i) => n + i;  
}  
  
final add2 = makeAdder(2);  
final add4 = makeAdder(4);  
assert (add2(3) == 5);  
assert (add4(5) == 9);
```

## Funktionen

Damit steht ihnen die Tür zur funktionalen Programmierung weit offen!

```
String toHtmlTable(List<List> rows) {  
  final table = rows.map((row) {  
    final zeile = row.map((col) {  
      return "<td>$col</td>";  
    }).join();  
    return "<tr>$zeile</tr>\n";  
  }).join();  
  return "<table>\n$table</table>";  
}
```

Das geht natürlich noch „dartiger“ 😊

```
String toHtmlTable(rows) {  
  final td = (elem) => "<td>$elem</td>";  
  final tr = (row) => "<tr>" + row.map(td).join() + "</tr>\n";  
  return "<table>\n" + rows.map(tr).join() + "</table>";  
};
```

## Funktionen

### Typedefs (I)

```
// Da in Dart Funktionen ganz normale Parameter sein können, müssen ihre Signaturen  
// als Parametertypen angegeben werden können.  
// Exemplarisch definieren wir erläuternd eine reduce Funktion.  
  
// Untypisiert - wir können nicht ablesen was Rückgabe, vs oder folder sein sollen.  
reduce(vs, folder) {  
  return vs.reduce(folder);  
}  
  
// Typisiert - Rückgabe und Parameter sind ablesbar. Funktionstyp ist "ungewöhnlich".  
num reduceTyped(List<num> vs, num folder(num a, num b)) {  
  return vs.reduce(folder);  
}  
  
// Typisiert - Rückgabe und Parameter sind gut erkennbar  
// Reducer ist als typedef ausgeworfen. Dies macht es übersichtlicher.  
typedef num Reducer(num a, num b);  
num reduceTypedefed(List<num> vs, Reducer folder) {  
  return vs.reduce(folder);  
}
```



## Funktionen

### Typedefs (II)



```
final oneToFive = [1, 2, 3, 4, 5];

// Wir definieren uns nun eine Additionsfunktion (typisiert und untypisiert)
final addTyped = (num a, num b) => a + b;
final addUntyped = (a, b) => a + b;

// Und eine Maximumsfunktion (typisiert)
final maxTyped = (num a, num b) => a > b ? a : b;

// Wenden wir diese "folder" Funktion nun auf eine Liste von Werten an, erhalten wir
das
// "reduzierte" Ergebnis. Der Typcheck erfolgt zur Compilezeit, nicht zur Laufzeit.
assert(reduce(oneToFive, addUntyped) == 15);
assert(reduceTyped(oneToFive, addTyped) == 15);
assert(reduceTyped(oneToFive, addUntyped) == 15);

// Dank funktionaler Definition können wir jetzt auch einfach das Maximum
// einer Liste von Werten bestimmen.
assert(reduce(oneToFive, maxTyped) == 5);
assert(reduceTyped(oneToFive, maxTyped) == 5);
assert(reduceTyped(oneToFive, maxUntyped) == 5);
```

Praktische Informatik und betriebliche Informationssysteme

57

## Funktionen

### Typedefs (III)



University of Applied Sciences

```
// Wenden wir die untypisierten Varianten an, können wir sogar
// mehr machen als nur addieren und Maximum bestimmen.
final strings = ["Finally", " ", "I", " ", "got", " ", "it", "."];
assert(reduce(strings, addUntyped) == "Finally I got it.");

// Typisierung erhöht also die Lesbarkeit und Typsicherheit
// (d.h. die Chance Fehler zur Compilezeit zu finden) aber reduziert die
// Wiederverwendbarkeit. Untypisierter Code ist wiederverwendbarer.
// Fehler finden sich aber häufig erst zur Laufzeit.
// Um Code typsicher und wiederverwendbar zu machen, dazu mehr
// wenn wir zu Generics kommen.
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

58

## Miniübung:



[https://api.dartlang.org/docs/channels/stable/latest/dart\\_core/List.html](https://api.dartlang.org/docs/channels/stable/latest/dart_core/List.html)

Beide Methoden reduzieren eine Liste von Werten mittels einer Funktion `combine` auf einen einzelnen Wert.

Bei `reduce` muss dabei der Datentyp erhalten bleiben (bspw. Liste von Zahlen zu einer Zahl verdichten).

Bei `fold` kann der Datentyp der Listenelemente sich vom Ergebnistyp unterscheiden (bspw. Aus einer Liste von Zeichenketten, die Länge der längsten Zeichenkette (`num`) bestimmen).

**abstract E reduce(E combine(E value, E element))**

inherited from `Iterable`

Reduces a collection to a single value by iteratively combining elements of the collection using the provided function.

Example of calculating the sum of an iterable:

```
iterable.reduce((value, element) => value + element);
```

**abstract dynamic fold(initialValue, combine(previousValue, E element))**

inherited from `Iterable`

Reduces a collection to a single value by iteratively combining each element of the collection with an existing value using the provided function.

Use `initialValue` as the initial value, and the function `combine` to create a new value from the previous one and an element.

Example of calculating the sum of an iterable:

```
iterable.fold(0, (prev, element) => prev + element);
```

## Miniübung:



Gegeben sei eine Liste von Strings, z.B.

```
final strings = ["Finally", " ", "I", " ", "got", " ", "it", "."];
```

Gegeben sei ferner folgende Funktion:

```
num count(List<String> s, num f(String, num)) {  
  return s.fold(0, f);  
}
```

Geben sie bitte eine Funktion `counter` an, so dass der folgende Aufruf

```
assert(count(strings, counter) == 17);
```

die Gesamtlänge aller Zeichenketten bestimmt.

## Miniübung:



Gegeben sei eine Liste von Strings, z.B.

```
final strings = ["Finally", " ", "I", " ", "got", " ", "it", "."];
```

Gegeben sei ferner folgende Funktion:

```
foo(s, f) => s.fold(0, f)
```

Geben sie bitte eine Funktion max an, so dass der folgende Aufruf

```
assert(foo(strings, max) == 7);
```

die Länge der längsten Zeichenkette in strings bestimmt.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

61

## Miniübung:



Gegeben sei eine Liste von Strings, z.B.

```
final strings = ["Finally", " ", "I", " ", "got", " ", "it", "."];
```

Gegeben sei ferner folgende Funktion:

```
magic(ls, red) => ls.reduce(red);
```

Bestimmen sie bitte unter Nutzung von magic die längste Zeichenkette in einer Liste von Zeichenketten.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

62

## Tour de Dart



## Operatoren

Dart bietet die üblichen **arithmetischen**, **relationalen**, **zuweisenden** sowie **logischen** Operatoren an. Ebenso wie Operatoren zum **Type Testing** sowie **bitweise** Operatoren.

In Dart lassen sich einige der Operatoren **überladen**, das heißt mit einer anderen Bedeutung versehen.

Dart orientiert sich dabei an der gängigen **Operatorbindung** (siehe rechts, Operatorbindung von oben nach unten abnehmend).

| Description                                | Operator                                                           |
|--------------------------------------------|--------------------------------------------------------------------|
| unary postfix and argument definition test | <code>expr++ expr-- () [] .</code>                                 |
| unary prefix                               | <code>~expr !expr ~expr ++expr --expr</code>                       |
| multiplicative                             | <code>* / % ~/</code>                                              |
| additive                                   | <code>+ -</code>                                                   |
| shift                                      | <code>&lt;&lt; &gt;&gt;</code>                                     |
| relational and type test                   | <code>&gt;= &gt; &lt;= &lt; as is !</code>                         |
| equality                                   | <code>== !=</code>                                                 |
| bitwise AND                                | <code>&amp;</code>                                                 |
| bitwise XOR                                | <code>^</code>                                                     |
| bitwise OR                                 | <code> </code>                                                     |
| logical AND                                | <code>&amp;&amp;</code>                                            |
| logical OR                                 | <code>  </code>                                                    |
| conditional                                | <code>expr1?expr2:expr3</code>                                     |
| cascade                                    | <code>..</code>                                                    |
| assignment                                 | <code>= *= /= ~/= %= += -= &lt;&lt;= &gt;&gt;= &amp;= ^=  =</code> |

Quelle: Walrath and Ladd, Dart Up and Running, O'Reilly, 2012

## Operatoren

### Arithmetische Operatoren

#### Arithmetische Operatoren

**Dart Besonderheit:** Es gibt zwei Divisionen.

- `/` (Ergebnis ggf. Fließkomma)
- `~/` (Ergebnis ganzzahlig)

| Operator           | Meaning                                                                  |
|--------------------|--------------------------------------------------------------------------|
| <code>+</code>     | Add                                                                      |
| <code>-</code>     | Subtract                                                                 |
| <code>-expr</code> | Unary minus, also known as negation (reverse the sign of the expression) |
| <code>*</code>     | Multiply                                                                 |
| <code>/</code>     | Divide                                                                   |
| <code>~/</code>    | Divide, returning an integer result                                      |
| <code>%</code>     | Get the remainder of an integer division (modulo)                        |

Quelle: Walrath and Ladd, Dart Up and Running, O'Reilly, 2012

#### Inkrement Operatoren

Keine Dart Besonderheiten. Jedoch die üblichen Überraschungen mit Pre- und Post-Inkrementen und -Dekrementen.

| Operator           | Meaning                                          |
|--------------------|--------------------------------------------------|
| <code>++var</code> | $var = var + 1$ (expression value is $var + 1$ ) |
| <code>var++</code> | $var = var + 1$ (expression value is $var$ )     |
| <code>--var</code> | $var = var - 1$ (expression value is $var - 1$ ) |
| <code>var--</code> | $var = var - 1$ (expression value is $var$ )     |

## Operatoren

### Relationale Operatoren

#### Relationale Operatoren

Dart Besonderheit. Der `==` Operator (Gleichheitsoperator) funktioniert etwas anders als in anderen Sprachen.

`x == y`

1. Wenn `x` oder `y` `null` sind, wird nur `true` zurückgegeben, wenn `x` und `y` gleich `null` sind, ansonsten `false`.
2. Der `==` Operator ist in Dart eigentlich eine Methode. `x == y` ist nur eine Dart Notation für `x.equals(y)` (die Java-Entsprechung wäre `x.equals(y)`).

| Operator           | Meaning                     |
|--------------------|-----------------------------|
| <code>==</code>    | Equal; see discussion below |
| <code>!=</code>    | Not equal                   |
| <code>&gt;</code>  | Greater than                |
| <code>&lt;</code>  | Less than                   |
| <code>&gt;=</code> | Greater than or equal to    |
| <code>&lt;=</code> | Less than or equal to       |

Quelle: Walrath and Ladd, Dart Up and Running, O'Reilly, 2012

Mehr dazu, wenn es um Operatorüberladung geht.

## Operatoren Zuweisungsoperatoren

### Zuweisungsoperatoren

Dart kennt den Zuweisungsoperator `=` sowie alle Kombinationen mit den arithmetischen und bitweisen Operatoren.

Wie in vielen anderen Sprachen auch, ist eine Zuweisung in Dart auch immer ein Ausdruck.

D.h. wir können bspw. schreiben:

```
num a = 10;  
assert("${a} ~/= 3}" + "=$a" == "3=3");
```

Dies ermöglicht uns auch Zuweisungen dieser Art zu machen.

```
num a, b, c, d, e;  
a = b = c = d = e = 10;
```

|                 |                  |                        |                     |
|-----------------|------------------|------------------------|---------------------|
| <code>=</code>  | <code>*=</code>  | <code>%=</code>        | <code>&amp;=</code> |
| <code>+=</code> | <code>/=</code>  | <code>&lt;&lt;=</code> | <code>^=</code>     |
| <code>-=</code> | <code>~/=</code> | <code>&gt;&gt;=</code> | <code> =</code>     |

Quelle: Walrath and Ladd, Dart Up and Running, O'Reilly, 2012

Und natürlich gilt die übliche Regel für kombinierte Operatoren für alle oben stehenden Operatoren:

```
num x = 5, y = 5;  
assert ((x = x + 3) == (y += 3));
```

## Operatoren Logische Operatoren

### Logische Operatoren

Dart kennt die üblichen logischen Operatoren NICHT, UND und ODER, die auf die übliche Art eingesetzt werden.

Logisches ODER und logisches UND sind in der gebräuchlichen short-circuit Variante implementiert.

| Operator                | Meaning                                                                  |
|-------------------------|--------------------------------------------------------------------------|
| <code>!expr</code>      | inverts the following expression (changes false to true, and vice versa) |
| <code>  </code>         | logical OR                                                               |
| <code>&amp;&amp;</code> | logical AND                                                              |

Quelle: Walrath and Ladd, Dart Up and Running, O'Reilly, 2012

```
bool sunny = true;  
bool rainy = true;  
  
if (sunny || rainy) {  
  print("Ok, there is weather outside.");  
}  
  
if (sunny && rainy) {  
  print("Rainbows likely.");  
}  
  
if (!sunny && !rainy) {  
  print("Then it might be cloudy or it is night.");  
}
```

## Operatoren Bitweise Operatoren

### Bitweise Operatoren

Just to be complete.

Dart kennt die gebräuchlichen bitweisen Operatoren.

Haben Sie vermutlich in der C-Programmierung und in der Lehrveranstaltung Betriebssysteme (Low Level Programmierung) kennengelernt.

In dieser Lehrveranstaltung (bzw. der High-Level Programmierung) werden diese eigentlich so gut wie gar nicht benötigt.

| Operator | Meaning                                               |
|----------|-------------------------------------------------------|
| &        | AND                                                   |
|          | OR                                                    |
| ^        | XOR                                                   |
| ~expr    | Unary bitwise complement (0s become 1s; 1s become 0s) |
| <<       | Shift left                                            |
| >>       | Shift right                                           |

Quelle: Walrath and Ladd, Dart Up and Running, O'Reilly, 2012

## Operatoren Type Test Operatoren

### Type Test Operatoren

Dart kennt die folgenden drei Operatoren zum Testen und Casten von Typen zur Laufzeit.

Sie werden wie unten gezeigt eingesetzt.

**Dart Besonderheit:** Der TypeCast Operator ist etwas verbosier als in anderen Sprachen.

| Operator | Meaning                                    |
|----------|--------------------------------------------|
| as       | Typecast                                   |
| is       | True if the object has the specified type  |
| is!      | False if the object has the specified type |

Quelle: Walrath and Ladd, Dart Up and Running, O'Reilly, 2012

```
var a = "10";  
if (a is String) { // Runtime Type Check  
  print ("a hat die Länge ${a.length}");  
}  
  
// Mit einem Type Cast lässt sich dies kürzer ausdrücken  
print ("a hat die Länge ${(a as String).length}"); // Type Cast
```

## Operatoren Weitere Operatoren

### Weitere Operatoren

Just to be complete.

Dart kennt wie viele anderen Sprachen auch Operatoren die sie wie selbstverständlich nutzen. Bspw. Die Funktionsapplikation (), der Zugriff über einen Index auf ein Listenelement [], natürlich die bedingte Auswertung ?:, der Zugriff auf Datenfelder oder Methoden eines Objekts . (Member Access).

### Dart Besonderheit:

Dart kennt einen sogenannten Cascade Operator .. der es ermöglicht, den Aufruf mehrerer Methoden hintereinander auf einem Objekt etwas effizienter zu notieren. Mehr dazu im Abschnitt Objektorientierung.

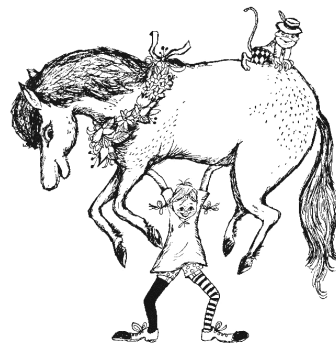
| Operator                                   | Name                 | Meaning                                                                                                                        |
|--------------------------------------------|----------------------|--------------------------------------------------------------------------------------------------------------------------------|
| ()                                         | Function application | Represents a function call                                                                                                     |
| []                                         | List access          | Refers to the value at the specified index in the list                                                                         |
| <i>expr1</i> ? <i>expr2</i> : <i>expr3</i> | Conditional          | If <i>expr1</i> is true, executes <i>expr2</i> ; otherwise, executes <i>expr3</i>                                              |
| .                                          | Member access        | Refers to a property of an expression; example: <i>foo</i> . <i>bar</i> selects property <i>bar</i> from expression <i>foo</i> |
| ..                                         | Cascade              | Allows you to perform multiple operations on the members of a single object; described in                                      |

Quelle: Walrath and Ladd, Dart Up and Running, O'Reilly, 2012

## Operatoren überladen

In Dart kann man den folgenden Operatoren für Klassen eine neue Bedeutung geben.

```
< + | []  
> / ^ []=  
<= ~/ & ~  
>= * << ==  
- % >>
```



Getreu dem Motto: „Ich mach die Welt,  
wie sie mir gefällt.“



## Operatoren

### Beispiel für 2D Vektoren

Die + und – Operatoren auf Vektoren sind nicht definiert. Wir können sie aber schnell nachrüsten.

```
class Vector {  
    final int x;  
    final int y;  
    const Vector(this.x, this.y);  
  
    Vector operator +(Vector v) => new Vector(x + v.x, y + v.y);  
    Vector operator -(Vector v) => new Vector(x - v.x, y - v.y);  
    String toString() => "($x, $y)";  
}  
  
main() {  
    final v = new Vector(2,3);  
    final w = new Vector(2,2);  
    assert(v.x == 2 && v.y == 3);  
    assert((v+w).x == 4 && (v+w).y == 5); // v+w == (4,5)  
    assert((v-w).x == 0 && (v-w).y == 1); // v-w == (0,1)  
    assert("$v" == "(2, 3)");           // toString  
}
```

## Miniübung:



In Python gehen „krude“ Dinge wie `"Hallo" * 3 == "HalloHalloHallo"`.

In Dart geht nicht einmal `"Hello" + 4 == "Hello4"` (das kann sogar Java, nun gut Java kennt dafür auch keine Ausdrucksinterpolation in Zeichenketten).

Wir wollen das aber auch können und müssen daher beide Operationen auf `BetterStrings` definieren.

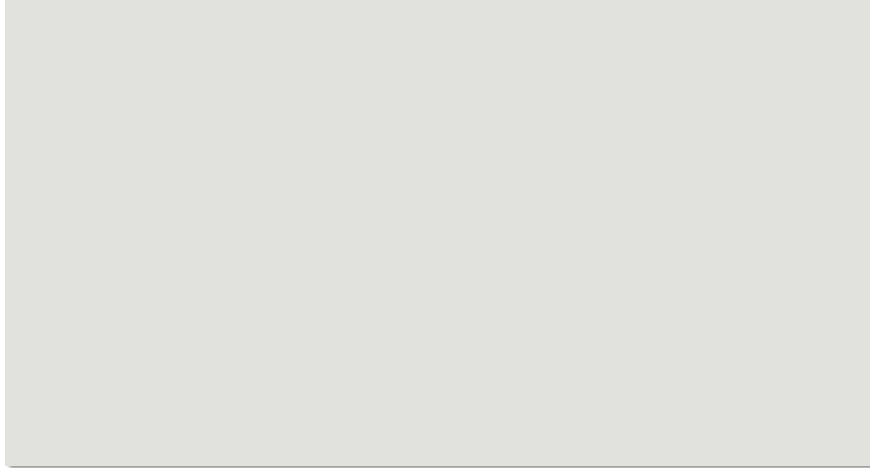
Definieren sie also bitte eine Klasse `BetterString` die den **Multiplikationsoperator**, den **Additionsoperator** und den **Gleichheitsoperator** definiert. Diese Operatoren sollen die folgenden asserts ermöglichen.

```
main() {  
    final s = new BetterString("Hello");  
    final v = new Vector(3, 6);           // definiert wie auf Folie vorher  
    assert (s + 4 == new BetterString("Hello4"));  
    assert (s + " World" == new BetterString("Hello World"));  
    assert (s + " Vektor " + v == new BetterString("Hello Vektor (3, 6)"));  
    assert (s * 4 == new BetterString("HelloHelloHelloHello"));  
}
```

## Miniübung:



Das ginge bspw. so:



## Tour de Dart

|                                    |               |
|------------------------------------|---------------|
| Kern-Konzepte                      | Variablen     |
| Built-In Types                     | Kontrollfluss |
| Funktionen (Methoden) und Typedefs | Operatoren    |
| Klassen (OO)                       | Generics      |



## Klassen



University of Applied Sciences

Dart ist eine klassenbasierte objektorientierte Sprache die single inheritance unterstützt. Folglich kann man Klassen mit Datenfelder und Methoden definieren und aus diesen Klassen Objekte ableiten. Und zwar ziemlich genauso wie in jeder anderen klassenbasierten OO-Sprache auch (natürlich typisiert oder untypisiert).

```
class PersonT {  
    String vorname;    // Datenfeld (typisiert)  
    String nachname;   // Datenfeld (typisiert)  
  
    // Konstruktor (typisiert).  
    PersonT(String this.vorname, String this.nachname);  
  
    // Methode um Hallo zu sagen (typisiert).  
    String sayHello() => "Hello $vorname $nachname";  
}  
  
void main() {  
    var p = new PersonT("Max", "Mustermann");  
    PersonT q = new PersonT("Maren", "Musterfrau");  
  
    assert (p.sayHello() == "Hello Max Mustermann");  
    assert (q.sayHello() == "Hello Maren Musterfrau");  
}
```

```
class PersonUT {  
    var vorname;    // Datenfeld (untypisiert)  
    var nachname;   // Datenfeld (untypisiert)  
  
    // Konstruktor (untypisiert).  
    PersonUT(this.vorname, this.nachname);  
  
    // Methode um Hallo zu sagen (untypisiert).  
    sayHello() => "Hello $vorname $nachname";  
}  
  
void main() {  
    var r = new PersonUT("Max", "Mustermann");  
    PersonUT s = new PersonUT("Maren", "Musterfrau");  
  
    assert (r.sayHello() == "Hello Max Mustermann");  
    assert (s.sayHello() == "Hello Maren  
Musterfrau");  
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

77

## Konstruktoren



University of Applied Sciences

Konstruktoren heißen wie ihre Klassen und dienen dazu den Zustand eines Objekts zu initialisieren. Jede Klasse hat einen parameterlosen Default Konstruktor und wie in Java erben Unterklassen keine Konstruktoren von ihrer Vaterklasse.

Dart Besonderheit: Konstruktoren können benannt werden. Und es gibt Syntactic Sugar um langweilige `this.x = x;` Anweisungen in Konstruktoren zu vermeiden.

```
class Person {  
    String vorname; String nachname;  
  
    // Konstruktor wie aus Java  
    Person(String vn, String nn) { this.vorname = vn; this.nachname = nn; }  
  
    // Kann vereinfacht werden zu (Syntactic Sugar):  
    Person(this.vorname, this.nachname);  
  
    // Benannter Konstruktor  
    Person.fromMap(Map data) { this.vorname = data['vn']; this.nachname = data['nn']; }  
  
    // Benannter Konstruktor mit Initializer List (Syntactic Sugar)  
    Person.fromMap(Map data) : vorname = data['vn'], nachname = data['nn'] {  
        // Hier kann dann was komplexeres realisiert werden.  
    }  
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

78

## Konstruktoren in Klassenhierarchien

### super, this



University of Applied Sciences

Um den Konstruktor einer Superklasse (`super`) oder als redirecting Konstruktor (`this`) aufrufen zu können, muss man in Dart eine recht eigenwillige Syntax bemühen (dieser erfolgt nämlich in der Initializer List).

Das Aufrufen von Methoden der Vaterklasse aus einer Unterklasse erfolgt hingegen wie man es erwarten kann.

```
class Person {  
  String vorname;  
  String nachname;  
  
  // Konstruktor  
  Person(this.vorname, this.nachname);  
  
  // Methode  
  String toString() => "$vorname $nachname";  
}
```

```
class Student extends Person {  
  int matrNr;  
  
  // Ein Superkonstruktoraufwurf erfolgt in der Initializer List  
  Student(String vn, String nn, int mn) : super(vn, nn), matrNr = mn;  
  
  // Auch redirecting des Konstruktors erfolgt in der Initializer List  
  Student.fromMap(Map data) : this(data['vn'], data['nn'], data['mn']);  
  
  // Aufruf der überschriebenen Methode erfolgt wie in Java auch  
  String toString() => "${super.toString()} ($matrNr)";  
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

79

## Factory Konstruktoren

Dart unterstützt über das Schlüsselwort `factory` Konstruktoren die als Factories eingesetzt werden. Factories sind „Konstruktoren“ die nicht immer ein neues Objekt erzeugen, sondern ggf. nur die Referenz auf ein bereits existentes Objekt.

```
Person p1 = new Person.fromNames("Max", "Mustermann");  
assert(Person.created.length == 1);  
Person p2 = new Person.fromNames("Maren", "Musterfrau");  
assert(Person.created.length == 2);  
Person p3 = new Person.fromNames("Max", "Mustermann");  
assert(Person.created.length == 2);  
assert(identical(p1, p3));
```

```
class Person {  
  String vorname;  
  String nachname;  
  
  // Personenverwaltung  
  static Map<String, Person> created = {};  
  
  // Personen  
  Person(this.vorname, this.nachname);  
  
  // Personenfactory  
  factory Person.fromNames(String vn, String nn) {  
    var p = created["$vn $nn"];  
    if (p != null) {  
      return p;  
    } else {  
      p = new Person(vn, nn);  
      Person.created["$vn $nn"] = p;  
      return p;  
    }  
  }  
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

80

## Getter und Setter



Auch in Dart definieren Methoden das Verhalten von Objekten. Typische Instanzmethoden haben wir schon kennen gelernt (`sayHello()`). Es gelten hier die üblichen Regeln der Objektorientierung.

In Dart lassen sich aber **getter** und **setter** Methoden nachträglich definieren, ohne bestehenden Code anpassen zu müssen.

Dieser ungewöhnliche Weg führt dazu, dass man die Programmierung mit gewöhnlichen Instanzvariablen beginnen kann, und nachträglich „schützende“ getter und setter nachtragen kann.

Nach außen ergibt sich hierdurch kein Unterschied.

```
class Rectangle {  
  num left;  
  num top;  
  num width;  
  num height;  
  
  Rectangle(this.left, this.top, this.width, this.height);  
  
  // Mit right und bottom definieren wir nach außen  
  // zwei „Datenfelder“ die im inneren Zustand gar nicht  
  // existieren  
  num get right => left + width;  
  set right(num value) => left = value - width;  
  num get bottom => top + height;  
  set bottom(num value) => top = value - height;  
}  
  
void main() {  
  var rect = new Rectangle(3, 4, 20, 15);  
  assert(rect.left == 3);  
  rect.right = 12;  
  assert(rect.left == -8);  
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

81

## Sichtbarkeiten

Dart kennt kein klassisches `private`, `protected`, `public`



University of Applied Sciences

Viele klassenbasierte OO-Sprachen bieten die drei Zugriffsmodifizierer `private`, `protected` und `public`. Diese Modifizierer sind entweder klassenbasiert (bspw. Java) oder objektbasiert (bspw. Ruby) definiert.

Dart geht hier einen anderen Weg. Sichtbarkeiten und (die damit zusammenhängenden Zugriffsmöglichkeiten) werden in Dart auf der Ebene von Libraries (vgl. `package` in Java) definiert.

Per Konvention sind in **Dart Bezeichner** (für Datenfelder, Methoden aber auch Top Level Variablen, Immutables und Funktionen) die mit einem **Underscore** `_` beginnen **Library private** (d.h. nur innerhalb der Library zu sehen in der der Bezeichner definiert wurde, in Java entspricht das in etwa der `package` Sichtbarkeit).

Das ist sicher pragmatisch, reduziert aber die Schutzmöglichkeiten innerhalb von Libraries (Klassenprivat lässt sich bspw. nicht definieren, auch `protected` entlang von Vererbungslinien lässt sich nicht mehr wie gewohnt einsetzen).

Die Gefahr unnötig hoher Daten/Zustandsabhängigkeiten steigt dadurch vermutlich.

```
class Rectangle {  
  num _left; // Nun library private  
  num _top; // Nun library private  
  num _width; // Nun library private  
  num _height; // Nun library private  
  
  Rectangle(  
    this._left, this._top, this._width, this._height  
  );  
  
  // Die getter und setter bleiben public  
  num get right => _left + _width;  
  set right(num value) => _left = value - _width;  
  num get bottom => _top + _height;  
  set bottom(num value) => _top = value - _height;  
}  
  
void main() {  
  var rect = new Rectangle(3, 4, 20, 15);  
  assert(rect.left == 3);  
  rect.right = 12;  
  assert(rect.left == -8);  
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

82

## Abstrakte Klassen

Besonderheit in Dart ist, dass abstrakte Methoden einfach nur keinen Rumpf haben müssen.

```
abstract class Figur {
  num _x, _y, _a, _b;

  Figur(this._x, this._y, this._a, this._b);
  num get flaeche; // Abstrakter getter
  String toString(); // Abstrakte Methode
}

class Rechteck extends Figur {
  Rechteck(x, y, a, b) : super(x, y, a, b);
  // Konkrete Implementierungen
  num get flaeche => (_a * _b).abs();
  String toString() => "Rechteck ($_x, $_y) Fläche $flaeche";
}

class Dreieck extends Figur {
  Dreieck(num x, num y, num a, num b) : super(x, y, a, b);
  // Konkrete Implementierungen
  num get flaeche => (_a * _b).abs();
  String toString() => "Dreieck ($_x, $_y) Fläche $flaeche";
}
```

```
// Es lässt sich eine Liste über die
// abstrakte Klasse anlegen und alles von Figur abgeleitete
// darin ablegen (und wie eine Figur verarbeiten)
List<Figur> figuren = [
  new Rechteck(3, 4, 2, 1),
  new Rechteck(2, 1, 3, 4),
  new Dreieck(1, 2, -2, 1),
  new Dreieck(2, 1, 2, -2)
];

// Und egal ob Rechteck oder Dreieck einheitlich mittels
// toString ausgeben
figuren.forEach((Figur f) => print("$f"));

// Und wir können auch die auf Figur bereits definierten
// getter aufrufen
final f = figuren.fold(0, (num v, Figur f) => v + f.flaeche);
print("Die Gesamtfläche beträgt $f");
```

In klassenbasierten OO-Sprachen dienen abstrakte Klassen dazu, Objektverhalten nur vorbereiten zu müssen und Verhaltensimplementierungen ererbenden Klassen aufzuerlegen.

Wie in Java dient in Dart hierzu das Schlüsselwort `abstract`. Eine Klasse die mindestens eine abstrakte Methode deklariert muss als `abstract` gekennzeichnet werden.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

83

## Implizite Schnittstellen (I)



```
class Person {
  String _vorname; // Im Interface aber library private
  String _nachname; // Im Interface aber library private
  Person(this._vorname, this._nachname); // Nicht im Interface
  String toString() => "$_vorname $_nachname"; // Im Interface
}

class Student extends Person { // Klassische OO-Erweiterung
  num _mnr;
  Student(vn, nn, mnr) : super(vn, nn), _mnr = mnr;
  String toString() => "StudentIn ${super.toString()} ($_mnr)";
}

// Mitarbeiter implementiert Personeninterface
class Mitarbeiter implements Person {
  String _vorname, _nachname;
  String _job;
  Mitarbeiter(this._vorname, this._nachname, this._job);
  String toString() => "MitarbeiterIn $_vorname $_nachname ($_job)";
}

// Ein WorkingStudent ist ein Student und ein Mitarbeiter
class WorkingStudent extends Student implements Mitarbeiter {
  String _job;
  WorkingStudent(vn, nn, m, j) : super(vn, nn, m), _job = j;
  String toString() => "$_vorname $_nachname [Mnr. $_mnr] (Stud. + $_job)";
}
```

In Dart definiert jede Klasse automatisch eine implizite Schnittstelle (Datenfelder, getter, setter und Methoden, keine Konstruktoren).

Dadurch kann man entscheiden ob, man eine Klasse per `extends` erweitern möchte oder deren implizite Schnittstelle mittels `implements` implementieren möchte.

Mittels Schnittstellen kann man das Verhalten mehrerer Klassen auf eine übertragen. Da bei Schnittstellen alle Methoden in der implementierenden Klasse neu implementiert werden müssen, kann es nicht zu Kollisionen kommen (wie bei Mehrfachvererbungen). Man hat aber eine vergleichbare Typflexibilität wie bei Mehrfachvererbung (allerdings verstößt man häufiger gegen DRY).

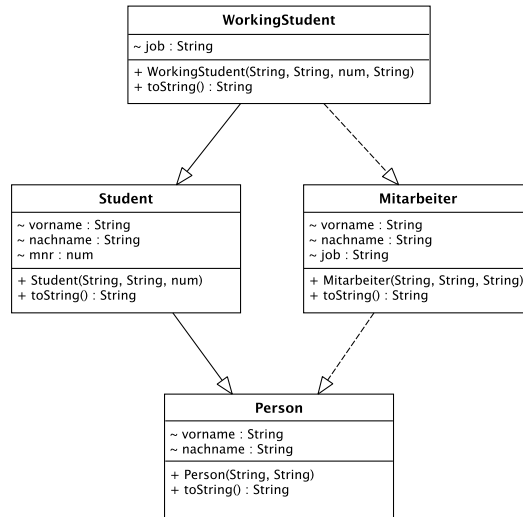
Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

84

## Implizite Schnittstellen (II)



University of Applied Sciences



In UML sähe das gezeigte Beispiel etwa wie folgt aus (Achtung dies ist kein korrektes UML!).

Man kann also nur einen Extends Strang haben (deswegen ist Dart eine single inheritance OO-Sprache) jedoch weitere Implements Stränge in Klassenhierarchien.

Durch implizite Schnittstellen ist man aber recht nah an der Mehrfachvererbung.

Formal handelt es sich aber um keine Mehrfachvererbung, wie folgendes (korrekte) UML Diagramm zeigt und man vermeidet dadurch auch alle die Probleme der Mehrfachvererbung.

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

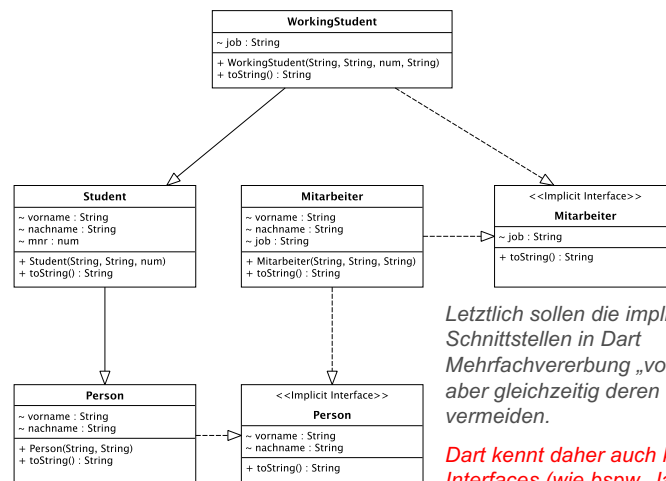
85

## Implizite Schnittstellen (III)



University of Applied Sciences

Da UML keine Implements Beziehung zwischen Klassen kennt, wird das „vollständige UML“ Diagramm etwas komplizierter, da wir nun die impliziten Schnittstellen ausweisen müssen.



Letztlich sollen die impliziten Schnittstellen in Dart Mehrfachvererbung „vortäuschen“, aber gleichzeitig deren Probleme vermeiden.

*Dart kennt daher auch keine expliziten Interfaces (wie bspw. Java).*

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

86

## Cascade Operator (Method Chaining a la Java)

```
class ChainShape {  
  num _x, _y, _w, _h;  
  ChainShape(this._x, this._y, this._w, this._h);  
  
  ChainShape move(num dx, num dy) { this._x += dx; this._y += dy; return this; }  
  ChainShape flipHorizontal() { this._w *= -1; return this; }  
  ChainShape flipVertical() { this._h *= -1; return this; }  
  ChainShape showMe() { print("$this"); return this; }  
  String toString() => "Shape at ($_x, $_y) has width=$_w and height=$_h";  
}
```

```
main() {  
  var t = new ChainShape(2, 2, 5, 3);  
  t.showMe()  
  .move(3, 2)  
  .showMe()  
  .flipHorizontal()  
  .showMe()  
  .flipVertical()  
  .showMe();  
}
```

### Konsoleausgabe:

```
Shape at (2, 2) has width=5 and height=3  
Shape at (5, 4) has width=5 and height=3  
Shape at (5, 4) has width=-5 and height=3  
Shape at (5, 4) has width=-5 and height=-3
```

Um Methoden auf Objekten verkettet auszuführen, lässt man Methoden häufig eine Referenz auf das eigene Objekt zurückgeben. Für die Logik der Routine ist das nicht erforderlich. Es würde eigentlich eine void Rückgabe reichen.

## Cascade Operator (Method Chaining a la Dart, Syntactic Sugar)

```
class Shape {  
  num _x, _y, _w, _h;  
  Shape(this._x, this._y, this._w, this._h);  
  
  void move(num dx, num dy) { this._x += dx; this._y += dy; }  
  void flipHorizontal() { this._w *= -1; }  
  void flipVertical() { this._h *= -1; }  
  void showMe() => print("$this");  
  String toString() => "Shape at ($_x, $_y) has width=$_w and height=$_h";  
}
```

```
main() {  
  var s = new Shape(2, 2, 5, 3);  
  s..showMe()  
  ..move(3, 2)  
  ..showMe()  
  ..flipHorizontal()  
  ..showMe()  
  ..flipVertical()  
  ..showMe();  
}
```

### Konsoleausgabe:

```
Shape at (2, 2) has width=5 and height=3  
Shape at (5, 4) has width=5 and height=3  
Shape at (5, 4) has width=-5 and height=3  
Shape at (5, 4) has width=-5 and height=-3
```

Dart kennt hierfür etwas Syntactic Sugar, d.h. einen eigenen Operator .. (**Cascade Operator**).

Dies befreit die Methoden von der Rückgabe der Selbstreferenz (this). Dies macht OO-Code übersichtlicher.



## Klassenvariablen und -methoden



University of Applied Sciences

Wie bspw. in Java nutzt auch Dart das Schlüsselwort **static** um **Klassenvariablen** und **Klassenmethoden** zu kennzeichnen. Klassenvariablen sind nützlich für klassenweit genutzte Status oder Konstanten.

Statische Methoden arbeiten nicht auf einer Instanz der Klasse und haben demzufolge keinen Zugriff auf die **this** Referenz.

```
class Color {  
  // Statische Klassenkonstanten (immutables)  
  static final Color RED = new Color('rot');  
  static final Color GREEN = new Color('grün');  
  static final Color BLUE = new Color('blau');  
  // Eine Instanzvariable (immutable)  
  final String name;  
  // Eine veränderliche Klassenvariable (mutable)  
  static num colors = 0;  
  // Konstruktor zum Erstellen einer Farbe  
  Color(this.name) { Color.colors += 1; }  
}  
  
main() {  
  assert(Color.RED.name == 'rot');  
  assert(Color.GREEN.name == 'grün');  
  assert(Color.BLUE.name == 'blau');  
  assert(Color.colors == 3);  
}
```

```
import 'dart:math';  
  
class Point {  
  num x, y;  
  Point(this.x, this.y);  
  // Statische Klassenmethode  
  static num distanceBetween(Point a, Point b) {  
    final dx = a.x - b.x;  
    final dy = a.y - b.y;  
    return sqrt(dx * dx + dy * dy);  
  }  
}  
  
main() {  
  final a = new Point(2, 2);  
  final b = new Point(4, 4);  
  assert(Point.distanceBetween(a, b) == sqrt(8));  
}
```

Dart kennt auch top level Funktionen. Ggf. ist die Definition einer top level Funktion in einer Library sinnvoller als eine statische Methode in einer Klasse!

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

89

## Tour de Dart



University of Applied Sciences



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

90

## Generics

Aus Sprachen wie Java kennt man das Konzept sogenannter Generics. In Klassen können Typplatzhalter vorgesehen werden, die erst zur Instantiierung einer Klasse durch konkrete Typen ersetzt werden.

Generics erlauben es somit in statisch typisierten Programmiersprachen mit Typplatzhaltern arbeiten zu können, dennoch Typisierungsfehler zur Compilezeit aufdecken zu können. Dynamisch typisierte Programmiersprachen sehen Generics selten vor, da Typkonflikte dort eh erst zur Laufzeit auftreten.

Nun ist Dart optional typisiert. Konsequenterweise sollte es dann beides können.

Wir wollen dies an einem kleinen Beispiel verdeutlichen. Artikel sollen an Kundenadressen versendet werden.

```
class Article {  
  String _name;  
  num _id;  
  Article(this._name, this._id);  
  String toString() => "Article: $_name ($_id)";  
}
```

```
class Address {  
  String _name, _street, _postalCode, _town;  
  int _nr;  
  
  Address(  
    this._name, this._street, this._nr,  
    this._postalCode, this._town  
  );  
  
  String toString() =>  
    "$_name, "  
    "$_street $_nr, "  
    "$_postalCode $_town";  
}
```

## Generics (II)

Wir können uns nun eine Klasse Orders definieren, in der wir pflegen, welche Artikel an welche Kundenadressen zu senden sind.

Das funktioniert wunderbar für unsere definierten Datentypen Article und Address.

```
class Orders {  
  
  var _orders = {};  
  
  Address addOrder(Article item, Address shipTo) =>  
    _orders[item] = shipTo;  
  
  String toString() {  
    return _orders.keys.map((k) {  
      return "$k shipped to ${_orders[k]}";  
    }).join("\n");  
  }  
}
```

```
final a1 = new Address("Max Muster", "Musterweg", 1, "27356", "Luebeck");  
final a2 = new Address("Maren Muster", "Mustergasse", 3, "12345", "Exempel");  
final a3 = new Address("Tessa Loniki", "Urlaub", 7, "27356", "Mallorca");  
  
final i1 = new Article("Fernseher", 123456);  
final i2 = new Article("Drucker", 654321);  
final i3 = new Article("Rechner", 123321);  
  
final Orders orders = new Orders();  
orders.addOrder(i1, a3);  
orders.addOrder(i2, a2);  
orders.addOrder(i3, a1);  
print("$orders");
```

## Generics (II)

Nun sollen aber nicht Artikel sondern auch bspw. Abonnements an Kundenadressen versendet werden.

Wir können jetzt einfach die Typisierung von `addOrder` löschen, dann können wir damit beliebige Datentypen verarbeiten (auch Abonnements) jedoch können wir auch Adressen an Artikel versenden (eigentlich ein Typfehler).

```
final a1 = new Address("Max Muster", "Musterweg", 1, "27356", "Luebeck");
final a2 = new Address("Maren Muster", "Mustergasse", 3, "12345", "Exempel");
final a3 = new Address("Tessa Loniki", "Urlaub", 7, "27356", "Mallorca");

final i1 = new Article("Fernseher", 123456);
final i2 = new Article("Drucker", 654321);
final i3 = new Article("Rechner", 123321);

final UTOorders = new UTOorders();
utorders.addOrder(i1, a3);
utorders.addOrder(i2, a2);
utorders.addOrder(a2, i1); // Hier überstrapazieren wir jetzt die dynamische Typisierung
print("$utorders");
```

```
class UTOorders {
  var _orders = {};

  addOrder(item, shipTo) => _orders[item] = shipTo;

  String toString() {
    return _orders.keys.map((k) {
      return "$k shipped to ${_orders[k]}";
    }).join("\n");
  }
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

93

## Generics (III)

Wenn wir unsere Orderlisten typerein haben wollen, aber dennoch nicht für jede neue Produktkategorie der Marketingabteilung eine neue Orderliste implementieren wollen, können wir generische Typen nutzen.

Wir können jetzt einfach die Typisierung von `addOrder` generisch halten, und auch bspw. auf Abo's instantiieren).

```
Address a1 = new Address("Max Muster", "Musterweg", 1, "27356", "Luebeck");
Address a2 = new Address("Maren Muster", "Mustergasse", 3, "12345", "Exempel");
Address a3 = new Address("Tessa Loniki", "Urlaub", 7, "27356", "Mallorca");

Article i1 = new Article("Fernseher", 123456);
Article i2 = new Article("Drucker", 654321);
Article i3 = new Article("Rechner", 123321);

TOrders<Article, Address> torders = new TOrders<Article, Address>();
torders.addOrder(i1, a3);
torders.addOrder(i2, a2);
torders.addOrder(a2, i1); // Typverletzung wird durch den Compiler erkannt

TOrders<Abo, Address> aboorders = new TOrders<Abo, Address>(); // Orderlisten für bel. Typen und Ziele
```

```
class TOrders<I, A> {
  Map<I, A> _orders = <I, A>{};

  A addOrder(I item, A to) => _orders[item] = to;

  String toString() {
    return _orders.keys.map((k) {
      return "$k shipped to ${_orders[k]}";
    }).join("\n");
  }
}
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

94

## Generics (List und Maps)



University of Applied Sciences

Die Dart Collections `List` und `Map` sind solche auf generischen Typen definierte Klassen. Da sie spezielle Literale haben, um angelegt zu werden, ist die Angabe der Typisierung etwas ungewöhnlich. Gewöhnungsbedürftig ist ferner, dass man die optionale Typisierung der Referenz mit der optionalen Typisierung der Datenstruktur mischen kann.

```
var list1 = ["A", 1, 3.7]; // untypisierte Liste, nicht typrein befüllt
var list2 = ["A", "B", "C"]; // immer noch nicht typisiert, nur zufaellig typrein.
var list3 = <String>["A", "B", "C"]; // typisierte Liste aber Referenz untypisiert
List<String> list4 = <String>["A", "B", "C"]; // typisierte Liste und Referenz typisiert
```

```
// untypisiertes Mapping, nicht typrein befüllt
var map1 = { 1 : "A", 2 : "B", 3 : "C", 4 : "D" };

// untypisiertes Mapping, nur zufaellig typrein
var map2 = { 1 : "A", 2 : "B", 3 : "C", 4 : "D" };

// typisiertes Mapping, aber Referenz untypisiert
var map3 = <int, String>{ 1 : "A", 2 : "B", 3 : "C", 4 : "D" };

// typisiertes Mapping, Referenz typisiert
Map<int, String> map4 = <int, String>{ 1 : "A", 2 : "B", 3 : "C", 4 : "D" };
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

95

## Generics (und ihre inneren Typen)



University of Applied Sciences

In Dart lässt sich auch der innere Typ einer Collection zur Laufzeit prüfen. In Java kann man beispielsweise nur prüfen, ob eine Referenz eine Liste ist, aber nicht ob eine Referenz eine Liste über Strings ist. Dies hat in Java damit zu tun, dass Java Generics durch Type Erasure realisiert (vgl. Programmieren II und damit den inneren Typ „wegwirft“).

In Dart bleibt der innere Typ bspw. für Type Checks erhalten. Sie können also so etwas hier machen (dafür kennt Dart dann auch keine Upper und Lower Bounds für generische Typen):

```
var list = <String>["A", "B", "C"];
var map = <int, String>{ 1 : "A", 2 : "B", 3 : "C", 4 : "D" };

assert(list is List<String>);
assert(map is Map<int, String>);

// In Java ginge nur (kein lauffähiger Java Code, nur zur Erläuterung):
// assert(list is List);
// assert(map is Map);
```

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

96

## Le Grande Boucle



University of Applied Sciences



### Tour de Dart

- Optional Typed
- Multiparadigm
- Byte Code (JavaScript Cross Compiled)
- In Browser and On Server



### Tour de Dart Libraries

- Library System
- Asynchronous Programming
- Math and Random
- Browser-Based Apps
- I/O
- Decoding and Encoding



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

97

## Tour de Dart Libraries



University of Applied Sciences



Es geht in Teil II weiter mit:

**dart:library**  
Das Library  
System von Dart

**dart:async**  
Asynchronous  
Programming

**dart:io**  
Streams, Files,  
Directories

**dart:html**  
Manipulating the  
DOM

**dart:servers**  
HTTP Clients and  
Servers

**dart:convert**  
Decoding and  
Encoding JSON  
and more

Und ja: Es ist noch weit bis Paris ☺

Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

98

## Zusammenfassung



FACH  
HOCHSCHULE  
LÜBECK  
University of Applied Sciences

- **Dart**
  - macht viele Anleihen bei Java
  - ist aber nicht dogmatisch auf OO oder statische Typisierung ausgerichtet
  - beinhaltet funktionale Sprachbestandteile
  - kann nach JavaScript kompiliert werden
  - ist On Server und On Client ausführbar
- **Sprachbesonderheiten**
  - Optionale Typisierung
  - Checked und Production Mode
  - Operator Overloading
  - Short hand functions, anonymous functions, closures
  - setter und getter sowie factory Schlüsselworte für OO
  - Implizite Schnittstellen von Klassen (quasi multiple inheritance)
  - Typ-Literale für generische Listen/Maps



Prof. Dr. rer. nat. Nane Kratzke  
Praktische Informatik und betriebliche Informationssysteme

99