Le Tour de

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

DART

Web-Technologien

Client- und Serverseitige Sprachen (Dart Teil I)

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
1

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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**

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
2

Dart in a Nutshell ...

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

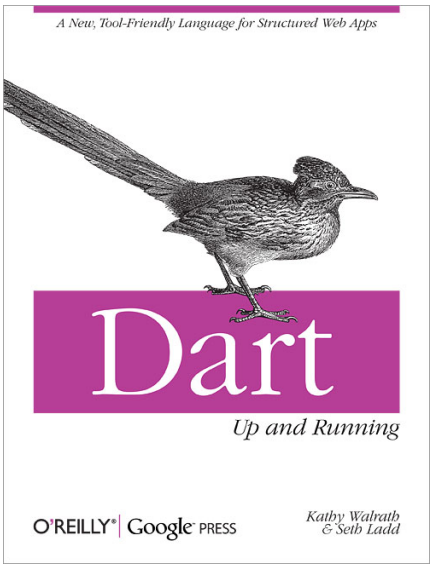
Dart

- 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 ...

FACH HOCHSCHULE LÜBECK
University of Applied Sciences



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

FACH HOCHSCHULE LÜBECK
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

FACH HOCHSCHULE LÜBECK
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

FACH HOCHSCHULE LÜBECK
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
7

Erstens: Dart ist nicht dogmatisch!

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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

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

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

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

Dart kennt top level Variablen.

Dart ist dynamisch typisiert.

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

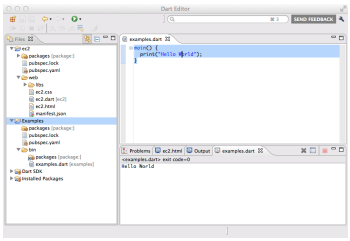
Sie können, müssen aber keine Datentypen bei Deklarationen angeben.

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

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
8

Runtime Modes

FACH HOCHSCHULE LÜBECK
University of Applied Sciences



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 sonder 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 mit einer Entwicklungsumgebung Dart Editor bereitgestellt.

Dart kennt zwei Runtime Modes:

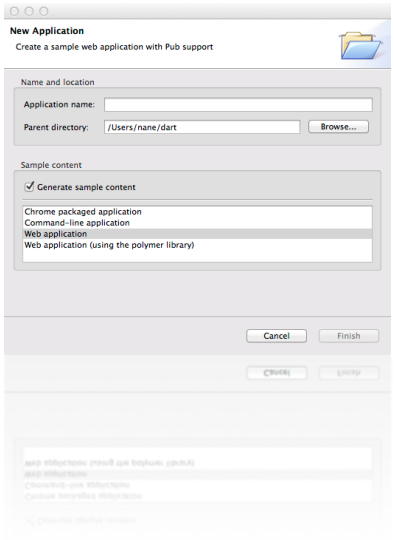
- Im **Production Mode** werden statische Typisierungsfehler ignoriert. Gleichzeitig ist die Ausführungsgeschwindigkeit optimiert (für die Auslieferung).
- Im **Checked Mode** werden statische Typisierungsfehler geprüft und ggf. Exceptions geworfen (für Test- und Entwicklung) sowie Assertions geprüft.

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
9

Install Dart is easy

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- 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)



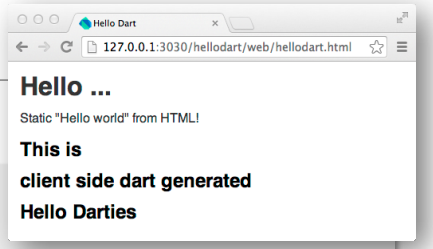
Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
10

Hello Darties (I)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

```
<!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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
13

Variable, Immutables und Konstanten

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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 Compilierzeit feststehen.

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

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
14

Alles natürlich optional typisiert

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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);
```

```
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).

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

final String n = "Tessa Loniki";
```

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

Wann nutze ich Immutables? `final` first

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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 auszulegen.

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 `final`s 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?

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

```
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>
```


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>
```

`final` first
Geht das denn?

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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?

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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 ausdruck = 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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

„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/

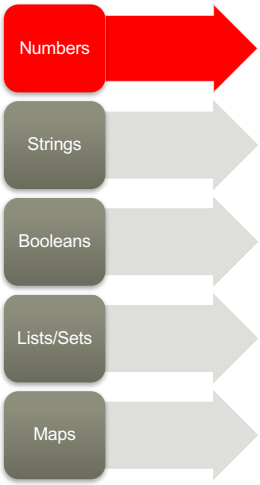
Tour de Dart

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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

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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

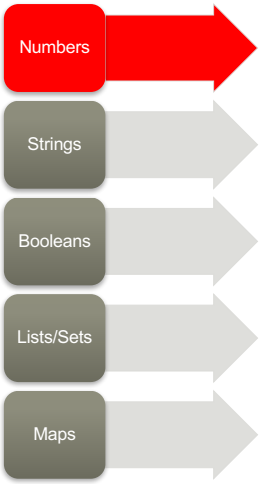



```dart
// Ganzzahlige Werte koennen so definiert werden
// Werte koennen beliebig groß werden
var i1 = 42;
int i2 = 42;
int earthPopulation = 8000000000;
assert (i1 == 42);
assert (i2 == 42);

// 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`

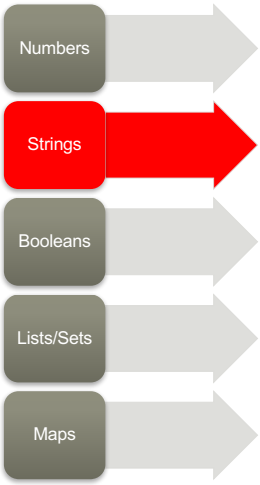
FACH HOCHSCHULE LÜBECK
University of Applied Sciences




```dart
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

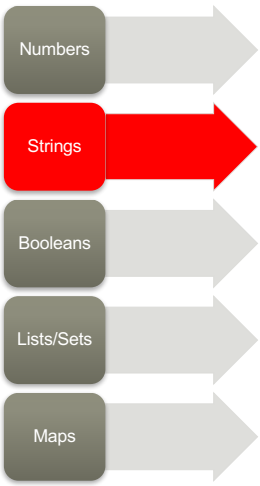
FACH HOCHSCHULE LÜBECK
University of Applied Sciences




```dart
// 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

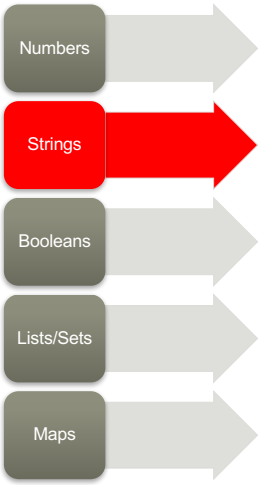
FACH HOCHSCHULE LÜBECK
University of Applied Sciences




```dart
// 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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

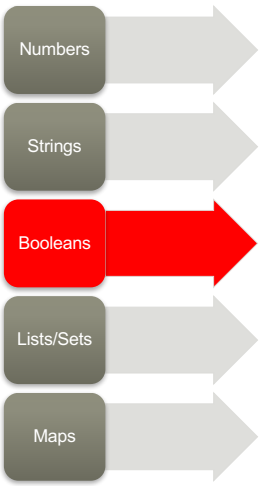



```
// Dart kennt String Interpolation
// String Interpolation wird durch ein $
// aehnlich wie in PHP gekennzeichnet
var name = "Max";
assert ("Hello $name" == "Hello Max");

// 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");
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
29

Darts built-in Types
bool

FACH HOCHSCHULE LÜBECK
University of Applied Sciences



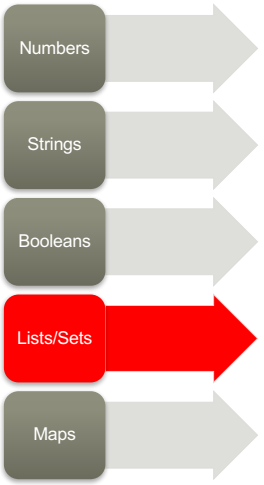

```
// Wahrheitswerte werden in Dart mittels dem Datentyp
// bool ausgedrückt. Ein bool kann zwei Werte annehmen:
// true und false.
var unkorrekt = false;
bool korrekt = true;

// 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)
}
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
30

Darts built-in Types (Lists)

https://api.dartlang.org/docs/channels/stable/latest/dart_collection/ListBase.html

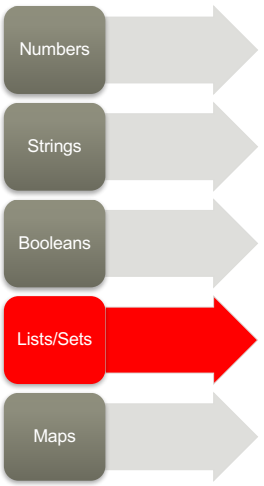
FACH HOCHSCHULE LÜBECK
University of Applied Sciences




```dart
// 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];
```


Darts built-in Types
Lists (forEach, map)

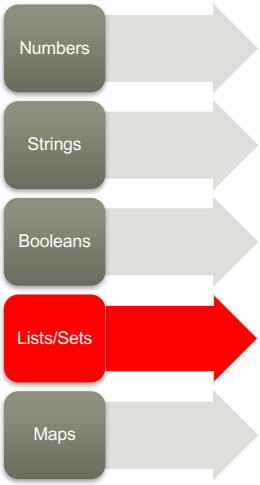
FACH HOCHSCHULE LÜBECK
University of Applied Sciences




```dart
// 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)
```

Darts built-in Types
Lists (any, every)

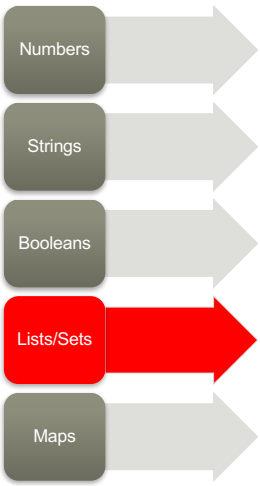
FACH HOCHSCHULE LÜBECK
University of Applied Sciences




```
// 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) => ["rooisbos", "chamomile"].contains(t);

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


Darts built-in Types
Lists (where, contains, isEmpty)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

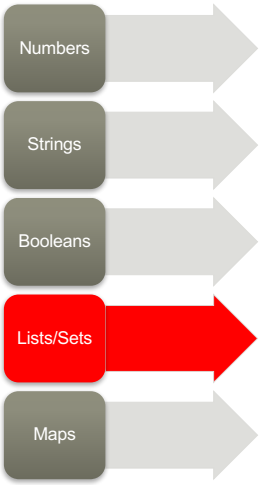



```
// Mittels der Methode where können wir aus Listen
// Elemente herausfiltern. Mittels contains prüfen
// ob ein Element Bestandteil einer Liste ist.
// Hier alle koffeinhaltigen Teesorten

caffeineFree(t) => ["rooisbos", "chamomile"].contains(t);
var someTeas = ["green", "black", "rooisbos", "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 Heißgetränke.");
}
```

Darts built-in Types (Sets)

https://api.dartlang.org/docs/channels/stable/latest/dart_core/Set.html

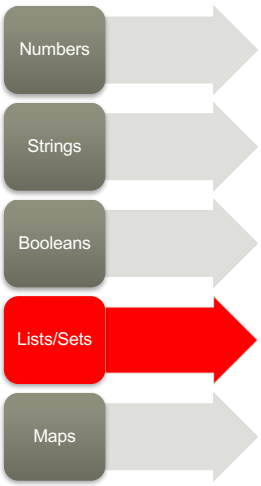
FACH HOCHSCHULE LÜBECK
University of Applied Sciences




```dart
// Eine Menge ist eine nicht indizierte Liste von Elementen
// ohne Dopplungen.
// Auf Mengenelemente kann daher nicht per Index
// zugeriffen werden. Mengen haben (leider) auch keine
// Sprachliterale, 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);
```


Darts built-in Types Sets (intersect, union)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

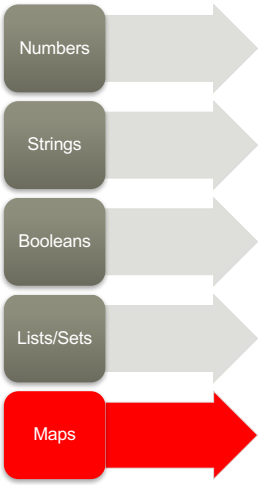



```dart
// Das hinzufügen von doppelten Elementen zu einer Menge hat
// keine Auswirkungen.
nobleGases.add("argon");
assert(nobleGases.length == 2);
assert(nobleGases.contains("xenon"));
assert(nobleGases.contains("argon"));

// 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));
```

Darts built-in Types (Maps)

https://api.dartlang.org/docs/channels/stable/latest/dart_core/Map.html

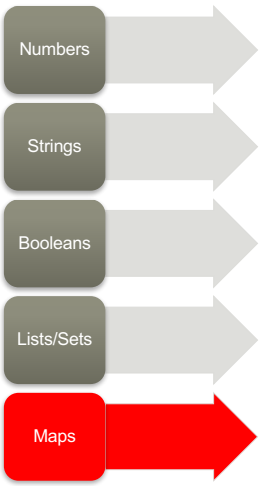
FACH HOCHSCHULE LÜBECK




```dart
// 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 = {};
```


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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences




```dart
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"));
```

Darts built-in Types
Maps (containsKeys, forEach)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- Numbers
- Strings
- Booleans
- Lists/Sets
- Maps

```
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");
});
```

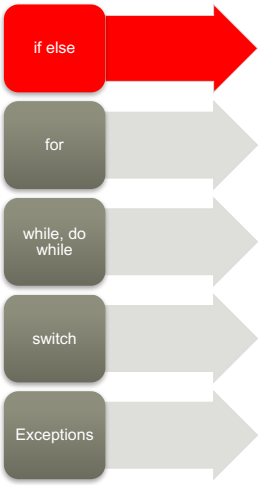

Tour de Dart

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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

Kontrollfluss

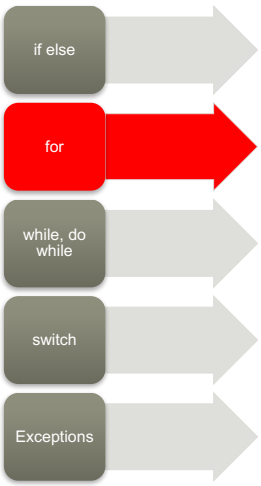
FACH HOCHSCHULE LÜBECK
University of Applied Sciences




```dart
// 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

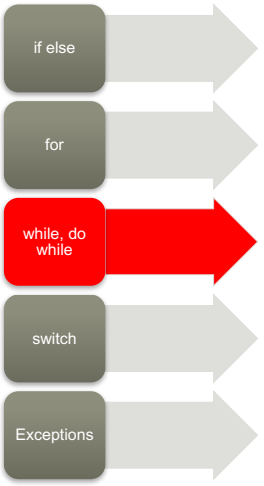
FACH HOCHSCHULE LÜBECK




```dart
// 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

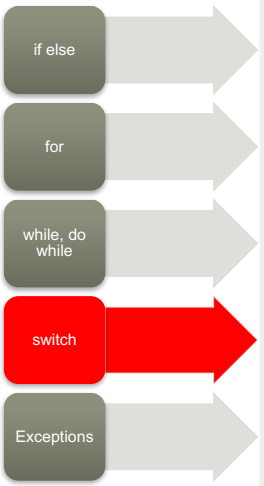
FACH HOCHSCHULE LÜBECK
University of Applied Sciences




```
// 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!
```


Kontrollfluss
switch

FACH HOCHSCHULE LÜBECK

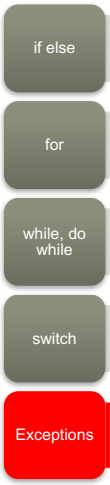



```
// Auch Dart hat diese eigenwillige switch
// Anweisung (Mehrfachverzweigung).
// Do not forget the break!
var command = "OPEN";
switch (command) {
  case "OPEN":
    print ("Opening file");
    break;
  case "CLOSE":
    print ("Closing file");
    break;
  case "COPY":
    print ("Copying file");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

Kontrollfluss

Exceptions definieren (implements Excpetion)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences



Dart definiert diverse Exceptions und Errors. Exceptions sind absehbare und beherrschbare Fehlerzustände (z.B. keine Netzwerkverbindung). Errors sind kaum absehbare und nur schwer beherrschbare Fehlerzustände (bspw. zu wenig Hauptspeicher).

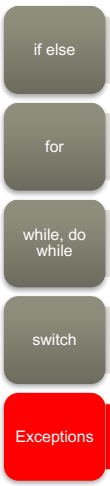
Darüber hinaus lassen sich in Dart (wie in Java) applikationsspezifische Exceptions definieren.

```
class MyException implements Exception {
  final String msg;
  const MyException([this.msg]);
  String toString() => msg == null ? 'MyException' :
msg;
}
```


Kontrollfluss

Exceptions werfen (throw)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences



Grundsätzlich kann in Dart aber jedes Objekt als Exception geworfen werden.

```
throw new MyException("very silly");
throw "This is an exception too";
```

Da das Werfen einer Exception einen Ausdruck in Dart darstellt (und keine Anweisung) können Exceptions überall dort geworfen werden, wo Ausdrücke erlaubt sind.

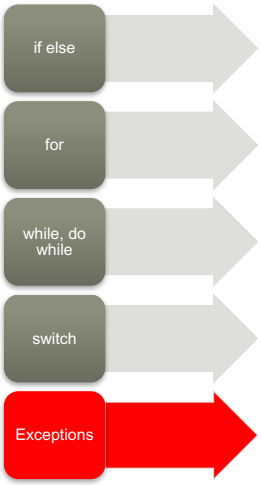
Z.B. in gettern.

```
String get prettyVersion =>
  throw const MyException("to be done");
```

Kontrollfluss

Exceptions behandeln (try, catch, on, finally)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences



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.

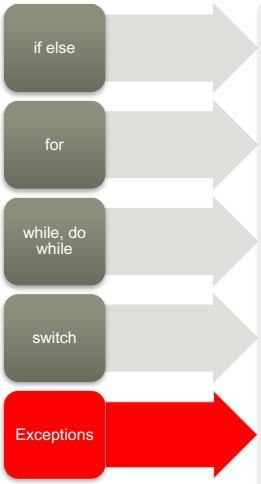
Können in einem `try` Block mehre Exception Typen auftreten, lassen sich auch mehrere catch Klauseln hierzu definieren.

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)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences




```dart
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)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

- if else
- for
- while, do while
- switch
- Exceptions

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 Excpetion 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.

Tour de Dart

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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

Funktionen
Funktionsdefinitionen

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

```
// 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"));
}
```


Funktionen
Optional named/positional parameters

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

```
// 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");
}
```

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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

```
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/Immutables 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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

```
// Funktionen können sich Variablen/Immutables 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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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)

FACH HOCHSCHULE LÜBECK

```
// 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)

FACH HOCHSCHULE LÜBECK

```dart
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(reduceTypedefed(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(reduceTypedefed(oneToFive, maxTyped) == 5);
```


Funktionen
Typedefs (III)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

```dart
// Wenden wir die untypsierten 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.
```

Miniübung:



FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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:



FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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:



FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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.

Miniübung:



FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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.

Tour de Dart

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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

Operatoren

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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

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

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

Operatoren
Arithmetische Operatoren

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Arithmetische Operatoren

Dart Besonderheit: Es gibt zwei Divisionen.

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

Operator	Meaning
+	Add
–	Subtract
-*expr*	Unary minus, also known as negation (reverse the sign of the expression)
*	Multiply
/	Divide
~/	Divide, returning an integer result
%	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
++*var*	*var* = *var* + 1 (expression value is *var* + 1)
var++	*var* = *var* + 1 (expression value is *var*)
--*var*	*var* = *var* – 1 (expression value is *var* – 1)
var--	*var* = *var* – 1 (expression value is *var*)

Operatoren
Relationale Operatoren

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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.==(y)` (die Java-Entsprechung wäre `x.equals(y)`).

Mehr dazu, wenn es um Operatorüberladung geht.

Operator	Meaning
==	Equal; see discussion below
!=	Not equal
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

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

Operatoren
Zuweisungsoperatoren

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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;
```

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

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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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
!expr	inverts the following expression (changes false to true, and vice versa)
||	logical OR
&&	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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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 verboser 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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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 Besondertheit:

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
`expr1 ? expr2 : expr3`	Conditional	If `expr1` is true, executes `expr2`; otherwise, executes `expr3`
.	Member access	Refers to a property of an expression; example: `foo.bar` selects property `bar` from expression `foo`
..	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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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:



FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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:



FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Das ginge bspw. so:

Tour de Dart

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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

Klassen

FACH HOCHSCHULE LÜBECK
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).

```dart
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");
}
```

```dart
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

FACH HOCHSCHULE LÜBECK
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.

```dart
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.
  }
}
```

```dart
Person m1 = new Person("Max", "Mustermann");
Person m2 = new Person.fromMap({ 'vn': "Max", 'nn' : "Mustermann"});
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
78

Konstruktoren in Klassenhierarchien
super, this

FACH HOCHSCHULE LÜBECK
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 Student extends Person {
  int matrNr;

  // Ein Superkonstruktoraufruf 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)";
}
```

```
class Person {
  String vorname;
  String nachname;

  // Konstruktor
  Person(this.vorname, this.nachname);

  // Methode
  String toString() => "$vorname $nachname";
}
```


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.

```
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;
    }
  }
}
```

```
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));
```

Getter und Setter

FACH HOCHSCHULE LÜBECK

Auch in Dart definieren Methoden das Verhalten von Objekten. Typische Intanzmethoden 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);
}
```


Sichtbarkeiten

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

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

Viele klassenbasierte OO-Sprachen bieten die drei Zugriffsmodifier `private`, `protected` und `public`. Diese Modifier 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 (vglb. `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);
}
```

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 laesst sich eine List ueber 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 koennen 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 erbenden 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.

Implizite Schnittstellen (I)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

```
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)";
}
```

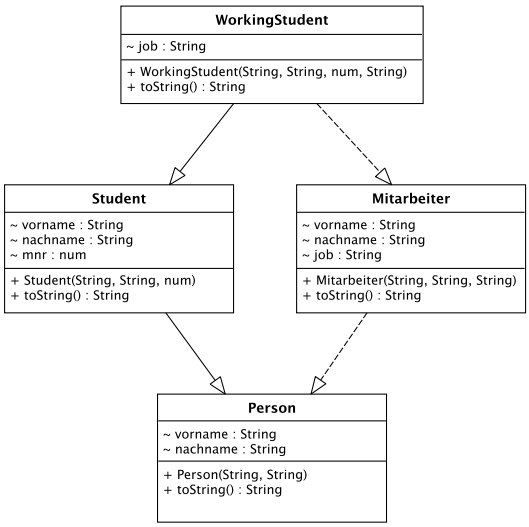
In Dart definiert jede Klasse automatisch eine implizite Schnittstelle (Datenfelder, getter, setter und Methoden, keine Konstruktoren).

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).

Implizite Schnittstellen (II)

FACH HOCHSCHULE LÜBECK
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.

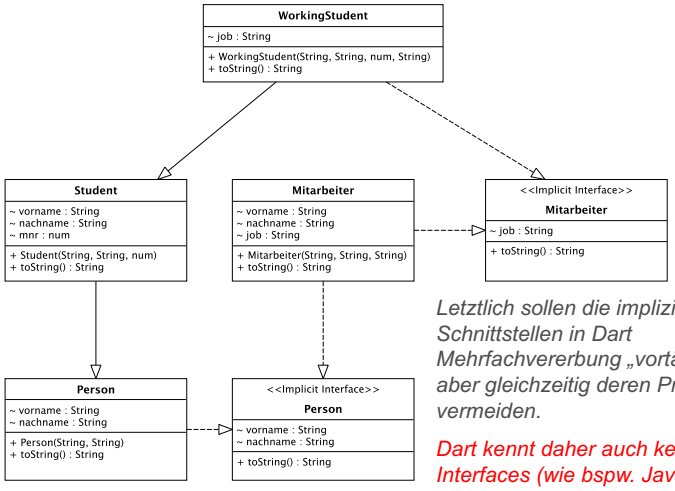
Durch implizite Schnittstellen ist man aber recht nah an der Mehrfachvererbung.

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

Implizite Schnittstellen (III)

FACH HOCHSCHULE LÜBECK
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).

Cascade Operator
(Method Chaining a la Java)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

```
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();
}
```

Konsolenausgabe:

```
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)

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

```
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();
}
```

Konsolenausgabe:

```
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

FACH HOCHSCHULE LÜBECK
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.

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!

```dart
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);
}
```

```dart
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));
}
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
89

Tour de Dart

FACH HOCHSCHULE LÜBECK
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
90

Generics

FACH HOCHSCHULE LÜBECK
University of Applied Sciences

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 Adress {
  String _name, _street, _postalCode, _town;
  int _nr;

  Adress(
    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 `Adress`.

```
class Orders {

  var _orders = {};

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

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

```
final a1 = new Adress("Max Muster", "Musterweg", 1, "27356", "Luebeck");
final a2 = new Adress("Maren Muster", "Mustergasse", 3, "12345", "Exempel");
final a3 = new Adress("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).

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

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

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

```
final a1 = new Adress("Max Muster", "Musterweg", 1, "27356", "Luebeck");
final a2 = new Adress("Maren Muster", "Mustergasse", 3, "12345", "Exempel");
final a3 = new Adress("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 UTOrders utorders = new UTOrders();
utorders.addOrder(i1, a3);
utorders.addOrder(i2, a2);
utorders.addOrder(a2, i1); // Hier überstrapazieren wir jetzt die dynamische Typisierung
print("$utorders");
```

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
93

Generics (III)

Wenn wir unsere Orderlisten typrein 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 Abos instantiieren).

```
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");
  }
}
```

```
Adress a1 = new Adress("Max Muster", "Musterweg", 1, "27356", "Luebeck");
Adress a2 = new Adress("Maren Muster", "Mustergasse", 3, "12345", "Exempel");
Adress a3 = new Adress("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, Adress> torders = new TOrders<Article, Adress>();
torders.addOrder(i1, a3);
torders.addOrder(i2, a2);
torders.addOrder(a2, i1); // Typverletzung wird durch den Compiler erkannt

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

Prof. Dr. rer. nat. Nane Kratzke
Praktische Informatik und betriebliche Informationssysteme
94

Generics (List und Maps)

FACH HOCHSCHULE LÜBECK
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 befuellt
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" };
```


Generics (und ihre inneren Typen)

FACH HOCHSCHULE LÜBECK
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);
```

Le Grande Boucle

FACH HOCHSCHULE LÜBECK
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

Tour de Dart Libraries

FACH HOCHSCHULE LÜBECK
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

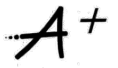
`dart:html`
Manipulating the DOM

`dart:servers`
HTTP Clients and Servers

`dart:convert`
Decoding and Encoding JSON and more

Und ja: Es ist noch weit bis Paris ☺

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