HOCHSCHULE FURTWANGEN UNIVERSITY | HFU

Kurzskript

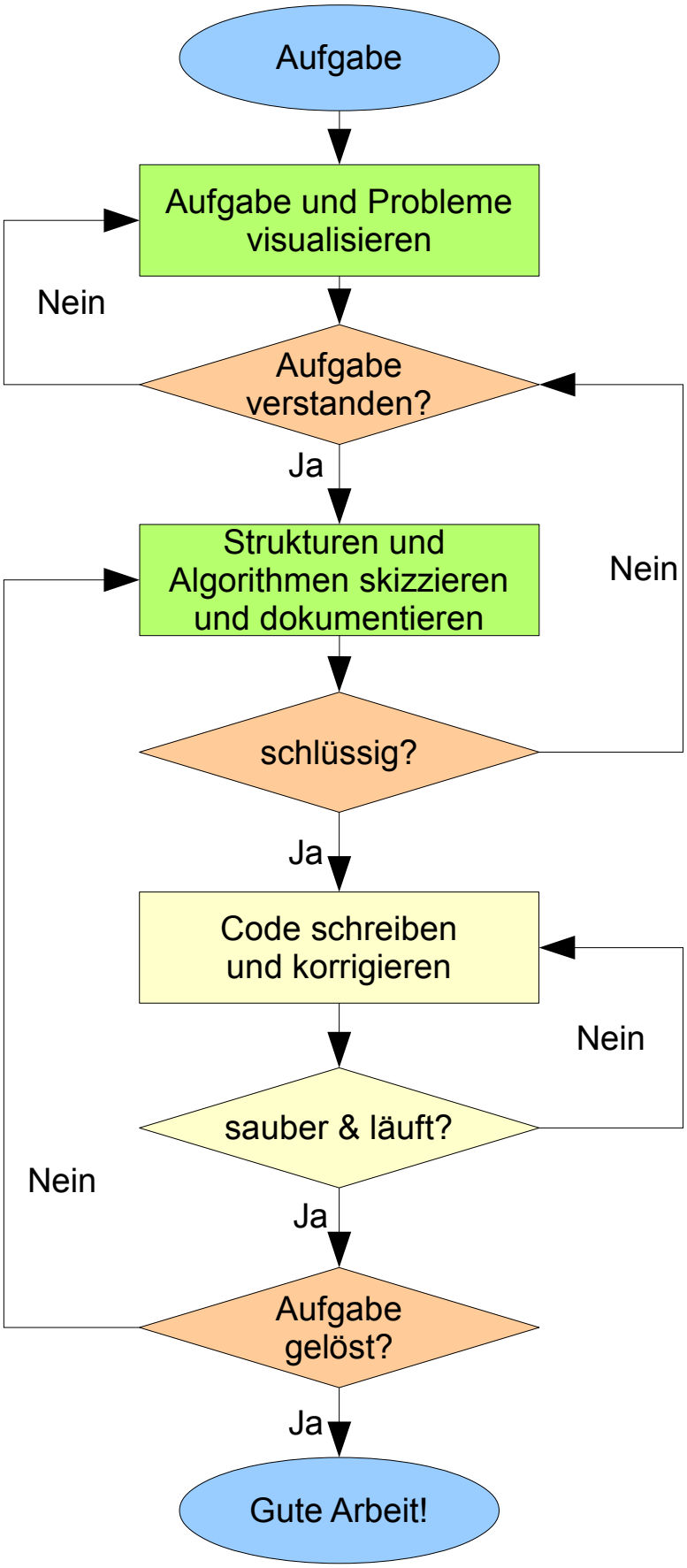
zur Veranstaltung „Programmieren“

Prof. Dipl.-Ing. Jirka R. Dell'Oro-Friedl

V2.2

Inhaltsverzeichnis

Arbeitsfluss

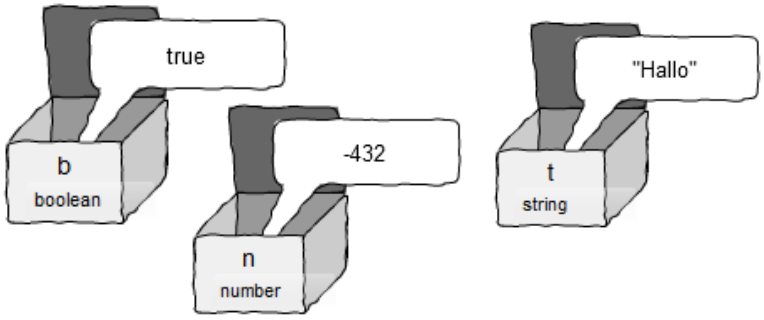


Stil

Programme können schnell sehr komplex werden. Daher ist es wichtig, sich an Stil-Regeln zu halten, um sie möglichst verständlich zu schreiben. In diesem Kurs gelten folgende Stil-Regeln:

1. Code sollte sich so gut wie möglich selbst erklären. Hierzu sind sprechende Variablen-, Funktions- und Klassennamen erforderlich. Kurze Namen sind nur in kleinen Gültigkeitsbereichen oder bei klarer Bedeutung (z.B. `y` für vertikale Position) erlaubt.
2. Variablen erhalten explizite Typ-Notationen und Anweisungen werden mit einem Semikolon beendet (auch wenn TypeScript diese automatisch einfügen bzw. inferieren kann)
3. Variablen- und Funktionsnamen beginnen mit Kleinbuchstaben und folgen der Kamelnotation, d.h. bei zusammengesetzten Namen beginnen die Wortteile im Inneren mit einem Großbuchstaben (z.B. `animalLion`). Funktionsnamen beschreiben dabei eine Aktivität (z.B. `calculateHorizontalPosition(...)`) oder Frage (z.B. `isHit()`)
4. Die Namen formaler Parameter in Funktionen beginnen mit einem Unterstrich.
5. Die Namen von Klassen, Interfaces und Modulen beginnen mit einem Großbuchstaben und folgen ebenfalls der Kamelnotation.
6. Die Namen von Enumerations und deren Elemente werden durchgehend mit Großbuchstaben geschrieben. Wortteile werden bei Bedarf mit Unterstrich getrennt.
7. Literale Zeichenketten werden in doppelte Anführungszeichen geschrieben.
8. Kommentare werden eingesetzt, um Programmteile abzugrenzen und die Verständlichkeit zu erhöhen. Programmteile, die von anderen Skripts genutzt werden sollen, werden im JSDoc-Format kommentiert (/** … */)

Variablen, Datentypen, Literale



Zuweisungsoperator =
`x = 10;`
Der Wert zur Rechten wird der Variablen zur Linken zugewiesen. Rechts können komplexe Ausdrücke (Terme) stehen.

Typ	Bedeutung	Wertebereich	Literalsyntax	Syntax Deklaration & Definition
boolean	Wahrheitswert	true, false	`true` `false`	`var b: boolean = (7 < 9);`
string	Zeichenkette	0 – .length( ) Zeichen	`"........."`	`var t: string = "Hallo";`
number	Zahl	-1.79e+308 bis 1.79e+308	`0.00234, 3, -653.13e-5`	`var n: number = -432;`
any	dynamisch	variabel		`var a: any = x;`
void	nichts			`function f(): void { }`

In anderen Programmiersprachen gibt es noch weitere Datentypen wie z.B.
`float, double` für Fließkommazahlen unterschiedlicher Präzision und Speicherbedarfe,
`byte, word, int, long` für Ganzzahlen mit 8, 16, 32 und 64 Bit, oder
`char` für einzelne Zeichen

Operatoren

Arithmetische und kombinierte Operatoren

	Name	Beispiel	Anmerkung
+	Addition	`x + 5;`	Liefert das Ergebnis der Addition des linken und rechten Wertes ohne Speicherung.
-	Subtraktion	`17 - x;`	Liefert das Ergebnis der Subtraktion des linken und rechten Wertes ohne Speicherung.
*	Multiplikation	`5 * x;`	Liefert das Ergebnis der Multiplikation des linken und rechten Wertes ohne Speicherung.
/	Division	`x / 2;`	Liefert das Ergebnis der Division des linken durch den rechten Wert ohne Speicherung.
%	Modulo	`x % 10;`	Liefert den Rest der ganzzahligen Division des linken durch den rechten Wert ohne Speicherung.
+=	Addend Additionszuweisung	`x += 19;`	Der Wert der Variablen wird um den rechtsstehenden Wert erhöht. Gleichbedeutend mit `X = X + 19;`
-=	Subtrahend Subtraktionszuweisung	`x -= 3;`	Der Wert der Variablen wird um den rechtsstehenden Wert vermindert. Gleichbedeutend mit `X = X - 3;`
*=	Faktor Multiplikationszuweisung	`x *= 100;`	Der Wert der Variablen wird um den rechtsstehenden Faktor erhöht. Gleichbedeutend mit `X = X *100;`
/=	Divisor Divisionszuweisung	`x /= 4;`	Der Wert der Variablen wird um den rechtsstehenden Divisor vermindert. Gleichbedeutend mit `X = X / 4;`
++	Inkrement	`x++;`	Der Wert der Variablen wird um 1 erhöht. Gleichbedeutend mit `X += 1;`
--	Dekrement	`x--;`	Der Wert der Variablen wird um 1 vermindert. Gleichbedeutend mit `X -= 1;`

Vergleichsoperatoren

==	Gleichheit	`x == "AB"`	Liefert den Wert `true`, wenn die Werte auf der linken und rechten Seite gleich sind. Vorsicht bei floats!!
!=	Ungleichheit	`x != "AB"`	Liefert den Wert `true`, wenn die Werte auf der linken und rechten Seite unterschiedlich sind.
>	Größer	`x > 2.32`	Liefert den Wert `true`, wenn der linke Wert größer als der rechte ist.
<	Kleiner	`x < 2.32`	Liefert den Wert `true`, wenn der linke Wert kleiner als der rechte ist.
>=	Größergleich	`x >= 2.32`	Liefert den Wert `true`, wenn der linke Wert größer als der rechte oder genau gleich ist.
<=	Kleinergleich	`x <= 2.32`	Liefert den Wert `true`, wenn der linke Wert kleiner als der rechte oder genau gleich ist.

Logische Operatoren

&&	Und	`x>2 && x<9`	Liefert den Wert `true`, wenn der linke und der rechte Ausdruck beide den Wert `true` haben. Hier, wenn x zwischen 2 und 9 liegt.
||	Oder	`x<2 || x>9`	Liefert `true`, wenn wenigstens einer der beiden Ausdrücke `true` ist. Hier, wenn x außerhalb des Bereichs 2 bis 9 liegt.
!	Nicht	`!(x > 10)`	Negiert den Ausdruck, liefert also `true`, wenn der folgende Ausdruck `false` ist. Hier, wenn x <= 10.

Daneben gibt es noch Bit-Operatoren zur gezielten Manipulation von Bitmustern. In JavaScript werden zudem die Operatoren === und !== verwendet, die auf Wert- und Typgleichheit prüfen.

Kontrollstrukturen

Block

```
{
    ...
}
```

Mehrere Anweisungen werden mit geschweiften Klammern zusammen gefasst.

Bedingung

```
if (Ausdruck) {
    ...
}
else {
    ...
}
```

Liefert der Ausdruck `true`, wird der obere Block ausgeführt, ansonsten der untere.

Mehrfachbedingung

```
switch (Ausdruck) {
    case Vergleichswert1:
        ...
        break;
    case Vergleichswert2:
    case Vergleichswert3:
        ...
        break;
    default:
        break;
}
```

Ist der Wert des Ausdrucks genau gleich einem der Vergleichswerte, werden die zugeordneten Anweisungen ausgeführt. Ansonsten jene unter `default`
Anm.: `break` beachten!

Konditionaloperator ? :

```
var sig:string = (x<0) ? "neg":"pos";
```

Ist `x<0` wahr, wird der Ausdruck links vom Doppelpunkt ausgewertet, ansonsten der rechte.

Blöcke können beliebig verschachtelt werden. In einer Schleife kann ein `switch` ausgeführt werden, in dessen Alternativen `if`-Blöcke stehen, darin Schleifen etc.

while-Schleife

```
while (Ausdruck) {
    ...
}
```

Der Block wird so lange ausgeführt, wie der Ausdruck `true` liefert. Findet keine Änderung statt, welche bewirkt, dass der Ausdruck `false` liefert, läuft die Schleife endlos.

do-while-Schleife

```
do  {
    ...
} while (Ausdruck)
```

Wie `while`-Schleife, der Block wird aber auf jeden Fall wenigstens einmal ausgeführt.

for-Schleife

```
for (Initialisierung; Ausdruck; Änderung) {
    ...
}
```

Der Block wird so lange ausgeführt, wie der Ausdruck `true` liefert. Gegenüber der `while`-Schleife sind die relevanten Steuermechanismen in einer Anweisung zusammen gefasst. Deren Positionen können aber auch leer gelassen werden.

for-in-Schleife

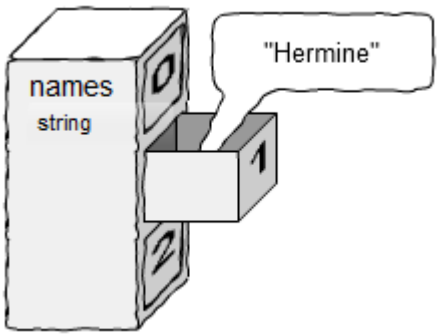
```
for (var k in arr) {
    ...
}
```

Die Schleife iteriert über alle Elemente von `arr`. `k` wird bei jedem Schritt der Index bzw. der Schlüssel zugewiesen.

Weitere Steuerbefehle für Schleifen

`break;`	`continue;`
Schleife beendet sofort	Startet sofort die nächste Iteration.

Arrays



Arrays erlauben es, mehrere Informationen zusammen unter einem Namen zu speichern. Die einzelnen Datenfelder werden dann über einen Index referenziert, der bei Null (!) beginnt.
Arrays sind komplexe Datenstrukturen, die über eigene Eigenschaften (`length`) und Methoden (wie `push, pop, reverse, sort`) verfügen. Diese sind über den Namen der Referenz und die Punkt-Syntax erreichbar.

Deklaration und Erzeugung mit [ ] und new

```
var names: string[] = new Array(3);
oder
var names: string[] = [];
oder
var names: string[] = new Array("Harry","Hermine","Ron");
oder
var names: string[] = ["Harry","Hermine","Ron"];
```

Der Datentyp `string[]` gibt an, dass die Variable names auf ein Array verweisen soll, welches Informationen vom Typ `string` halten soll. Allein die Deklaration erzeugt aber noch kein Array. `new Array(..)` bzw. `[...]` erzeugt schließlich das Array und reserviert den Speicherbereich für die Daten, den `names` nach der Zuweisung referenziert. Ist die Länge oder sind Elemente angegeben, wird das Array entsprechend dimensioniert und besetzt.

Zugriff und Manipulation

```
var values: number[] = new Array();
values[0] = 143.2; // 143.2 wird Element 0 zugewiesen
values[1] = values[0]; // kopiert 143.2 in Element 1
values.push(7.97); // fügt ein Element am Ende ein
console.log(values.length); // gibt 3 aus
console.log(values[3]); // gibt undefined aus, denn Element 3 gibt es noch nicht
```

Mit Hilfe der eckigen Klammern wird auf ein einzelnes Element des Array zugegriffen. Mit der Punkt-Syntax wird auf die Eigenschaft `length` und auf die Methoden (hier `push`) zugegriffen.

Arrays und Schleifen

```
for (var index:number = 0; index < values.length; index++)
    console.log(values[index]);
```

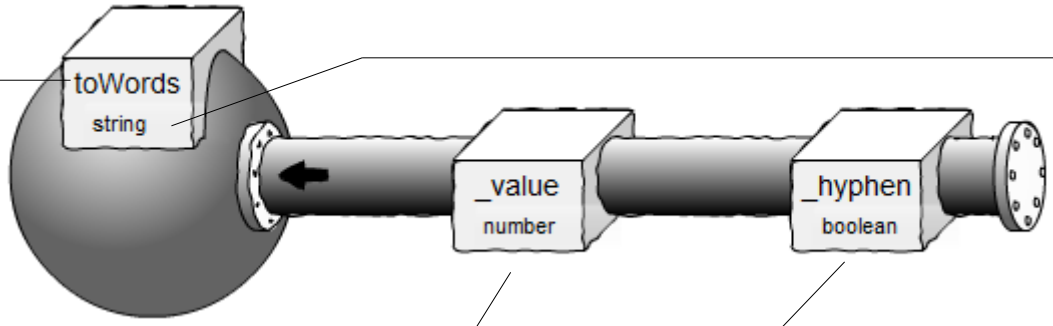
Ein Array kann ein große Menge von Daten halten, die Verarbeitung findet daher meistens in Schleifen statt. Im Beispiel werden alle Elemente des Array `values` ausgegeben.

Mehrdimensionale Arrays

```
var tictactoe: number[][] = [[0, 0, 0], [0, 0, 0], [0, 0, 0]];
tictactoe[2][0] = 1;
```

Elemente eines Array können wiederum auf Arrays verweisen. Im Beispiel hält das Array `tictactoe` Verweise auf drei Arrays mit je drei Elementen vom Typ `number`. Das entspricht einem Spielfeld von 3x3 Feldern. Das Feld in der 2. Zeile und der 0. Spalte (links unten) wird mit dem Wert 1 besetzt. Mit diesem Prinzip können leicht Felder von beliebiger Dimension erstellt und verwaltet werden.

Funktionen, Methoden



Name der Funktion

Liste formaler Parameter, durch Kommata getrennt. Für jeden Parameter Name und Datentyp

Datentyp des Rückgabewertes

```
function toWords(_value: number, _hyphen: boolean): string {
    var word: string[] =
    ["Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine"];
    var res: string = "";
    for (; !(_value < 1); _value /= 10) {
        _value = Math.floor(_value);
        var digit = _value % 10;
        res = (res.length > 0 ? (_hyphen ? "-" : " ") : "") + res;
        res = word[digit] + res;
    }
    return res;
}
```

Körper der Methode als Code-Block

Beendet die Methode und gibt Wert zurück

Aufruf:
```
var words: string = toWords(4362, true);
console.log(words);
```

Ausgabe:
```
Four-Three-Six-Two
```

Call by Value

```
function test(_val: number)
```

Ist ein Parameter von simplem Datentyp, wird beim Aufruf `test(a)` der Wert der Variablen `a` dort hineinkopiert. Änderungen des Wertes von `_val` innerhalb der Methode haben keine Auswirkung außerhalb.

Polymorphie, Signatur

In anderen Programmiersprachen kann es mehrere Funktionen/Methoden gleichen Namens (Polymorphie) geben, die sich bezüglich der Typen und/oder der Anzahl der Parameter (Signatur) unterscheiden.

Call by Reference

```
function test(_ref: number[])
```

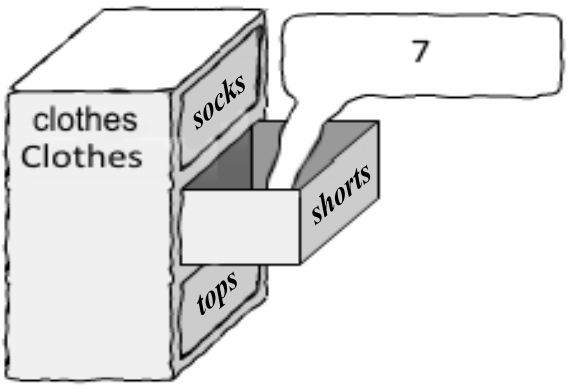
Ist ein Parameter von komplexem Datentyp, wie Arrays oder Klassen, wird beim Aufruf nur der Verweis auf die Daten im Speicher übergeben. Eine Änderung der referenzierten Daten innerhalb der Methode wirkt auch außerhalb. Im Beispiel bedeutet also eine Änderung in `_ref` die Änderung des übergebenen Arrays.

Hinweis: `_ref` ist nur eine Kopie des Verweises, genau genommen handelt es sich also auch um „Call by Value“. Ein echtes „Call by Reference“, wobei eine nach außen wirksame Änderung der Referenz selbst vorgenommen werden kann, ist in Programmiersprachen wie C++ möglich.

Assoziative Arrays und Interfaces

Bei assoziativen Arrays werden die einzelnen Elemente nicht über einen fortlaufenden Index, sondern über einen sogenannten Schlüssel referenziert. Meist ist der Schlüssel eine Zeichenkette, er kann aber auch von anderem Typ sein. Zur literalen Deklaration eines assoziativen Arrays wird folgende Syntax genutzt:

```
{ key : value, key : value, ... };
```



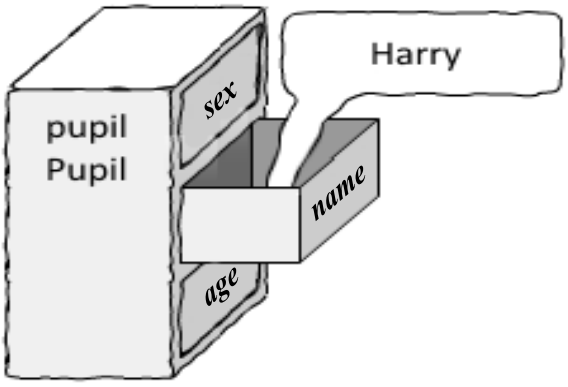
Homogenes assoziatives Array

```
interface Clothes {
    [key: string]: number;
}

var clothes: Clothes;
clothes = { "socks": 3, "shorts": 7 };
clothes["tops"] = 4;
console.log(clothes);
```

Das `interface` deklariert, dass assoziative Arrays vom Typ `Clothes` Werte vom Typ `number` über Schlüssel vom Typ `string` referenzieren (Name `key` ist irrelevant). Dabei kommen eckige Klammern zum Einsatz.

Interfaces dienen dazu, die Struktur von Daten festzulegen. So kann für assoziative Arrays angegeben werden, von welchem Typ die Schlüssel und die Daten sein sollen, und man erstellt ein homogenes Array. Oder man legt die Schlüsselliterale fest und weist deren zugeordneten Werten unterschiedliche Typen zu. So kann man Informationen unterschiedlicher Bedeutung und Datentypen in einer heterogenen Struktur speichern.



Heterogenes assoziatives Array

```
interface Pupil {
    sex: boolean;
    name: string;
    age: number;
}

var pupil: Pupil =
    { name: "Harry", age: 0, sex: true };
pupil.age = 13;
```

Das `interface` deklariert, dass assoziative Arrays vom Typ `Pupil` je drei Werte vom Typ `boolean`, `string` und `number` aufnehmen, welche über die Schlüssel `gender`, `name` und `age` referenziert werden. Dabei kann die Punkt-Syntax genutzt werden.

JavaScript-Objects

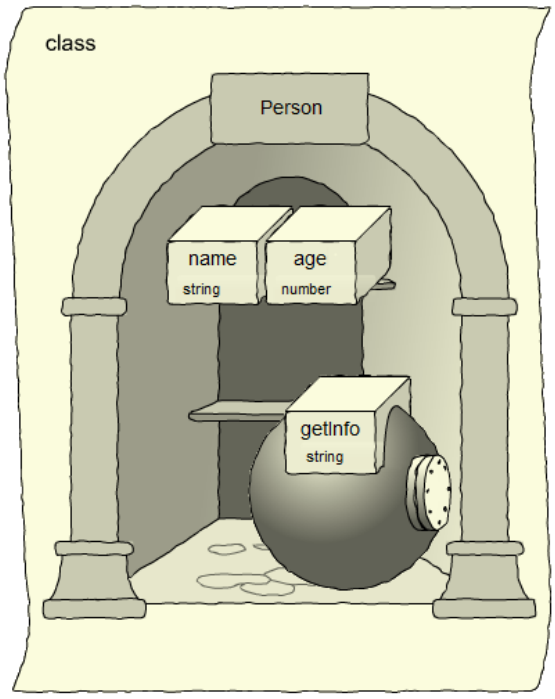
Die Versuchung kann groß sein, assoziative Arrays mit Hilfe des Datentyps `any` zu deklarieren. Damit ist die Wahl der Schlüssel- und Wertetypen völlig frei, so wie es bei JavaScript üblich ist. TypeScript kann dann aber nicht mehr bei der Fehlersuche helfen. Davon ist abzuraten!

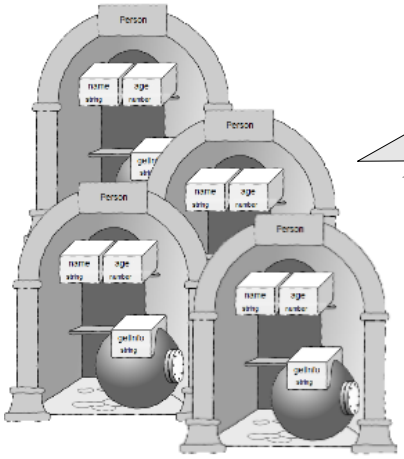
Klassen und Objekte

```
class Person {
    name:string;
    age:number;

    constructor(_name:string) {
        this.name = _name;
        this.age = 0;
    }

    getInfo():string {
        var res:string = "Name: " + this.name;
        res += "\nAge: " + this.age;
        return res;
    }
}
```





Instanzieren

Die Klasse enthält den „Bauplan“ für konkrete Objekte (Instanzen) und beschreibt deren Eigenschaften (Attribute, z.B. `name` und `age`) und Funktionalität (Methoden, z.B. `getInfo`). Es können beliebig viele Instanzen erzeugt werden. Jede erhält ihre individuellen Attribute.

Objekt-Methode

Funktion, die innerhalb der Klasse ohne das Schlüsselwort `function` definiert wird. Objektmethoden greifen über `this` direkt auf die Attribute und weitere Methoden des Objektes zu.

Punkt-Syntax

```
p.age = 24;
```

Mit Hilfe der Referenz und des Punktes kann auf die Attribute und Methoden eines Objektes zugegriffen werden.

this

```
this.age = 0;
```

Wird der Code eines Objektes abgearbeitet, verweist darin `this` auf das Objekt selbst.

Instanzieren

```
var p:Person = new Person("Eva");
```

`new` erzeugt eine Instanz (ein Objekt) der Klasse im Speicher. Der Variable des entsprechenden Typs wird die Referenz auf den Speicherort dieses Objektes zugewiesen.

Konstruktor

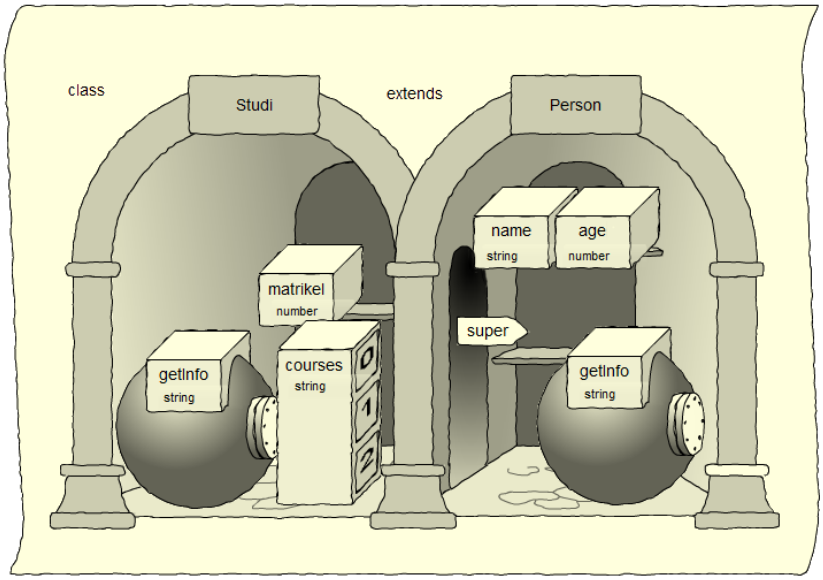
```
constructor(_name:string) {
    ...
}
```

Spezielle Methode innerhalb der Klasse mit Namen `constructor`. und ohne Typannotation Damit können z.B. sofort bei der Instanzierung Parameter (hier „Eva“) übergeben, Startzustände für Objektattribute (z.B. `age` = 0) hergestellt oder andere Objekte informiert werden etc.

Vererbung und Polymorphie

extends

Durch Erweiterung einer Klasse mit `extends` wird eine Subklasse erschaffen, die alle Attribute und Methoden „erbt“ und um zusätzliche ergänzt werden kann. Im Beispiel erweitert `Studi` die Klasse `Person` um eine Matrikelnummer und ein String-Array.




```
class Studi extends Person {
    matrikel: number;
    courses: string[] = new Array(3);

    constructor(_name: string, _mat: number) {
        super(_name);
        this.matrikel = _mat;
    }

    getInfo(): string {
        var res: string = super.getInfo();
        res += "\nMatrikel: " + this.matrikel;
        return res;
    }
}
```

Überschreiben

Attribute und Methoden der Superklasse können in der Subklasse neu definiert werden. `Studi` überschreibt im Beispiel `getInfo()`

super

`super` verweist auf das Basisobjekt. Damit können aus der Subklasse heraus gezielt Attribute oder Methoden der Superklasse referenziert werden. Im Beispiel ruft der Konstruktor der Subklasse explizit den Konstruktor der Superklasse auf und `getInfo()` nutzt zunächst die Methode `getInfo` der Person.

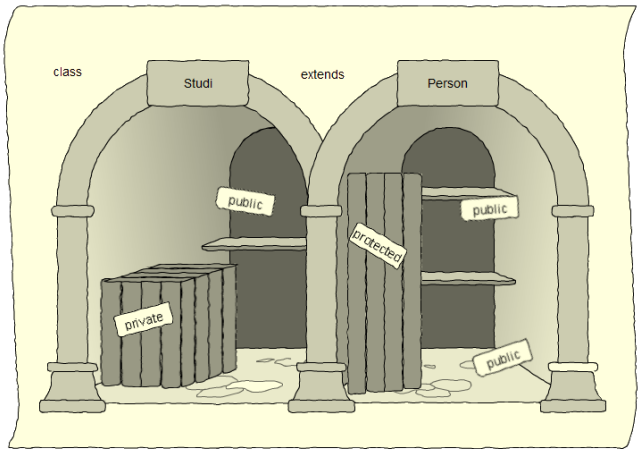
Polymorphie, Substitution

var eva:Person = new Studi("Eva", 123456);

Ein Objekt einer Subklasse kann als Objekt der Basisklasse referenziert werden. Vorteil: viele unterschiedliche Objekte können gleich behandelt werden, sofern Sie dieselbe Basisklasse besitzen.
Im Beispiel ist die Referenz vom allgemeinen Typ `Person`, das referenzierte Objekt aber vom speziellen Typ `Studi`. `eva.getInfo()` ruft die in der Studi-Klasse definierte Methode auf.
Merke: Studis sind auch Menschen...

Synonyme
Basisklasse, Superklasse, Vaterklasse, Elternklasse, Oberklasse
Subklasse, Sohnklasse, Abgeleitete Klasse, Kindklasse, Unterklasse
Methode, Elementfunktion, Objektfunktion, Memberfunktion
Attribut, Eigenschaft, Member-Variable Datenelement
Statische Methode, Klassenfunktion, Metafunktion
Statisches Attribut, Klassenattribut, Klassenvariable

Sichtbarkeit, Gültigkeit und Zugriffsmodifikation



In komplexeren Programmen sollte die Gefahr, während der Entwicklung Fehler zu implementieren, möglichst reduziert werden. Hierzu lässt sich die Sichtbarkeit von bzw. der Zugriff auf Attribute und Methoden gezielt einschränken, so dass Objekte nur im vorgesehenen Rahmen manipuliert werden können.

Gültigkeitsbereiche von Variablen

```
module ScopeTest {
    var scope:string = "module";

    class ScopeTest {
        scope: string = "class";
        global(): void {
            console.log(this.scope, scope);
        }
        local(): void {
            var scope: string = "method";
            console.log(this.scope, scope);
        }
    }

    var s: ScopeTest = new ScopeTest();
    s.local();
    s.global();
}
```

Module, Klassen und Funktionen separieren Namensräume. Somit können Variablennamen mehrfach verwendet werden, ohne dass es zwangsläufig Konflikte gibt. Im Beispiel wird auf allen Ebenen je eine Variable `scope` deklariert und definiert. Die Methoden liefern folgende Ausgaben:

`s.local()` → `"class" "method"`
`s.global()` → `"class" "module"`

Das Objektattribut ist also in den Methoden des Objektes sichtbar, ebenso die Modulvariable (siehe `s.global`). Variablen, die in Funktionen deklariert werden, sind nur innerhalb dieser gültig und können gleichnamige Variablen höherer Ebenen überdecken (siehe `s.local`)

let

Variablen, die mit `let` deklariert wurden, sind nur innerhalb des umschließenden Blocks gültig.

Modifikatoren

```
modifier x: number;
modifier calc(): boolean
```

Der Modifikator wird vor die Deklaration des Attributs bzw. Methode geschrieben.

public

Sichtbar im ganzen Programm (Standard)

protected

Sichtbar nur innerhalb des Objektes und dessen Erweiterungen (z.B. über `super` aber auch implizit)

private

Außerhalb des Objektes unsichtbar, auch für Erweiterungen.

static

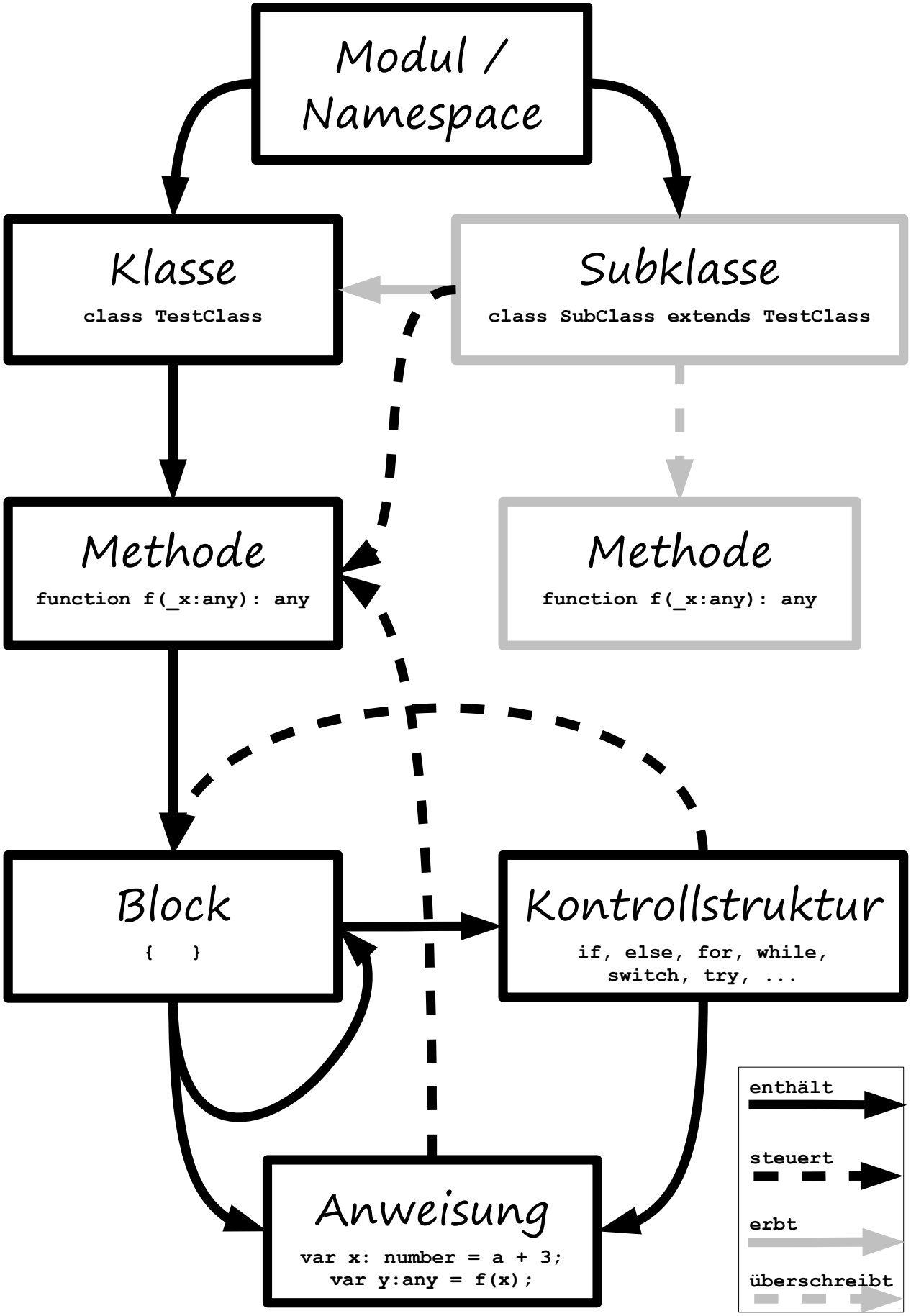
```
static private x: number;
static public calc(): boolean
```

Attribut bzw. Methode wird nicht mit einer Instanz verwendet, sondern direkt mit der Klasse. Daher nennt man diese dann auch Klassenmethode bzw. -attribut. Sie werden mit der Punkt-Syntax nicht nach dem Objekt-, sondern dem Klassennamen referenziert, z.B. `TestClass.calc();`

import / export

In Modulen organisierte Inhalte sind zunächst verborgen und werden mit `export` über Datei- und Modulgrenzen hinweg verfügbar gemacht. Mit `import` wird ein Modul in ein anderes eingebunden.

Überblick Programmhierarchie



Weiterführendes

Exception

```
function testException() {
    var a: HTMLElement;
    try {
        console.log("try");
        console.log(a.id);
    } catch (_e) {
        console.log("catch - " + _e);
        throw (new Error("thrown"));
    } finally {
        console.log("finally");
    }
}

try {
    testException();
}
catch (_e) {
    console.log("caller - " + _e);
}
```

In der Funktion `testException()` wird auf `a.id` zugegriffen, wobei `a` zwar deklariert, ein Objekt aber nicht instanziert wurde. Das Programm würde hier die Ausführung mit einer Fehlermeldung in der Konsole abbrechen.
Da der Zugriff aber in einem `try`-Block geschieht, wird dem `catch`-Block ein Fehlerobjekt übergeben, welches dort ausgewertet werden kann. Im Beispiel wird hier ein neues Fehlerobjekt erzeugt und an den aufrufenden Programmteil übergeben (`throw`), wodurch die Funktion beendet wird. Der `finally`-Block wird aber auf jeden Fall zuvor noch ausgeführt.
Das Beispiel erzeugt folgende Ausgabe:

```
"try"
"catch - TypeError: a is undefined"
"finally"
"caller - Error: thrown"
```

Anmerkung: Parameter der `catch`-Anweisung ohne Typannotation!.

Interface

```
interface GameItem {
    paint(): void;
    move(_x: number, _y: number): boolean;
}

class Ball implements GameItem {
    paint(): void {
        ...
    }
    move(_x: number, _y: number): boolean {
        ...
    }
}

var b: GameItem = new Ball();
```

Interfaces definieren auch Klassenschnittstellen. Klassen, welche Interfaces implementieren, müssen die angegebenen Methoden und Attribute enthalten. Damit ist schon beim Kompilieren sicher gestellt, dass alles erforderliche zumindest vorhanden ist. Variablen vom Typ des Interfaces können Objekte von Klassen referenzieren, die das Interface implementieren.
Es ist möglich, in einer Klasse mehrere Interfaces zu implementieren, sie werden mit Komma getrennt nach `implements` angegeben.

Generics

```
class GnArr<T> {
    arr:T[] = new Array(10);
}

var g:GnArr<string> = new GnArr<string>();
g.arr[0] = "123";
```

Die Klasse wird erst durch Angabe eines zu verwendenden Typs in spitzen Klammern konkretisiert. Im Beispiel erzeugt `GnArr`, ein Array des angegebenen Typs, hier `string`.
Somit können Klassen entworfen werden, die mit Datentypen arbeiten, welche noch nicht bei der Implementierung bekannt sind. `T` stellt innerhalb der Klasse den zu verwendenden Typ dar.

Type Assertion

```
var x: T = <T> y;
```

Einer bereits bestehenden Information wird explizit der Typ `T` zugewiesen, sofern kompatibel

Enumeration

```
enum TIER { HUND, KATZE, MAUS = 4 };
var cat: TIER = TIER.KATZE;
```

Es wird ein Datentyp definiert, der nur die angegebenen Werte aufnehmen kann. Intern sind dies Zahlen, ggf. explizit definiert.

Codebeispiel

```
module Vector {
   export class Vector2D {
      public x: number;
      public y: number;

      constructor(_x: number, _y: number) {
         this.x = _x;
         this.y = _y;
      }

      getDistanceTo(_v: Vector2D): number {
         var dx: number = this.x - _v.x;
         var dy: number = this.y - _v.y;
         return Math.sqrt(dx * dx + dy * dy);
      }
   }
}
```

```
module Herd {
   export class Dog extends Sheep {
      private target: Sheep;
      private speed: number;

      constructor(_x: number, _y: number) {
         super(_x, _y);
         this.speed = 0.2;
         this.selectTarget();
      }

      update(): void {
         if (this.huntSheep()) {
            this.target.runHome();
            this.selectTarget();
         }
      }

      render(): void {
         crc2.save();
         crc2.beginPath();
         crc2.strokeStyle = "#FF0000";
         crc2.lineWidth = 4;
         crc2.arc(this.pos.x, this.pos.y, 7, 0, 2 * Math.PI);
         crc2.stroke();
         crc2.restore();
      }

      private huntSheep(): boolean {
         this.pos.x += (this.target.pos.x - this.pos.x) * this.speed;
         this.pos.y += (this.target.pos.y - this.pos.y) * this.speed;
         return (this.pos.getDistanceTo(this.target.pos) < 2);
      }

      private selectTarget(): void {
         do {
            var i: number = Math.floor(  Math.random() *
                                         Sheep.herd.length);
            this.target = Sheep.herd[i];
         } while (this.target == this);
      }
   }
}
```

```
module Herd {
   import V2 = Vector.Vector2D;

   export class Sheep {
      public static herd: Sheep[];
      public pos: V2;

      constructor(_x: number, _y: number) {
         this.pos = new V2(_x, _y);
      }

      update(): void {
         this.pos.x += Math.random() * 6 - 3;
         this.pos.y += Math.random() * 6 - 3;
      }

      render(): void {
         crc2.fillStyle = "#ffffff";
         crc2.beginPath();
         crc2.arc(this.pos.x, this.pos.y, 5, 0, 2 * Math.PI);
         crc2.fill();
      }

      runHome(): void {
         this.pos.x = crc2.canvas.width / 2;
         this.pos.y = crc2.canvas.height / 2;
      }
   }
}
```

```
module Herd {
   import V2 = Vector.Vector2D;
   var size: V2 = new V2(400, 400);
   Setup.size(size.x, size.y);
   Setup.addEventListener(
            EVENTTYPE.MOUSEDOWN, addSheep);

   Sheep.herd = new Array(20);
   for (var i: number = 0; i < Sheep.herd.length; i++) {
      Sheep.herd[i] = new Sheep(size.x / 2, size.y / 2);
   }
   Sheep.herd.push(new Dog(Math.random() * size.x,
                           Math.random() * size.y));
   update();

   function update(): void {
      crc2.fillStyle = "#10a000";
      crc2.fillRect(0, 0, size.x, size.y);

      for (var i: number = 0; i < Sheep.herd.length; i++) {
         Sheep.herd[i].update();
         Sheep.herd[i].render();
      }
      Setup.setTimeout(update, 50);
   }

   function addSheep(_event: Event): void {
      Sheep.herd.unshift(new Sheep(Setup.pointerX,
                                   Setup.pointerY));
   }
}
```