adesso business. people. technology.

JavaScript

Konstantin Pentarakis, Leiter JavaScript-CC

- Leiter JavaScript-Abteilung
- Frontend-Architekt
- 15 Jahre JavaScript-/Web-Erfahrung

Projekte

vodafone
BADER MODE. WOHNEN. LEBEN.
ERGO Direkt
chrono24 The World's Watch Market
Santander
MEDIA - SATURN
Deka Investments
Bundeswehr
DAIMLER
pirobase imperia
REMONDIS IM AUFTRAG DER ZUKUNFT

Arbeitgeber

adesso
EXXETA CONSULTING AND TECHNOLOGIES
CGI
IBM

Introduction of participants

Hey, I am _____ !

My previous knowledge ...

My (technical) background ...

What do I expect about the workshop? What do I look for?

Organization

Did you install the tools?

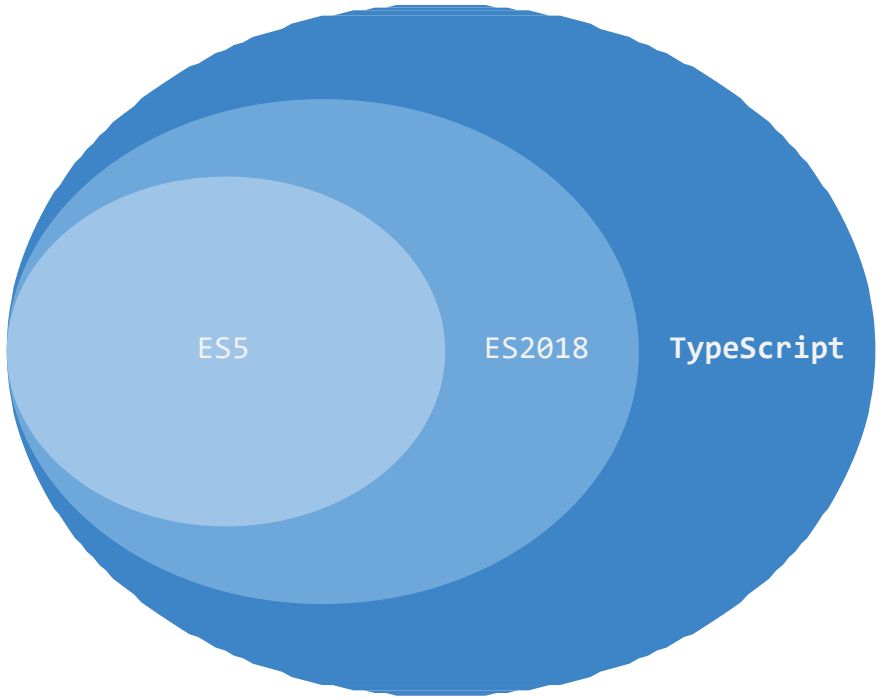
Can you access the internet?

Organization

Don’t hesitate to ask questions all the time!

TypeScript is a superset

- Superset of EcmaScript
- Compiles to JavaScript
- **Optional Types**

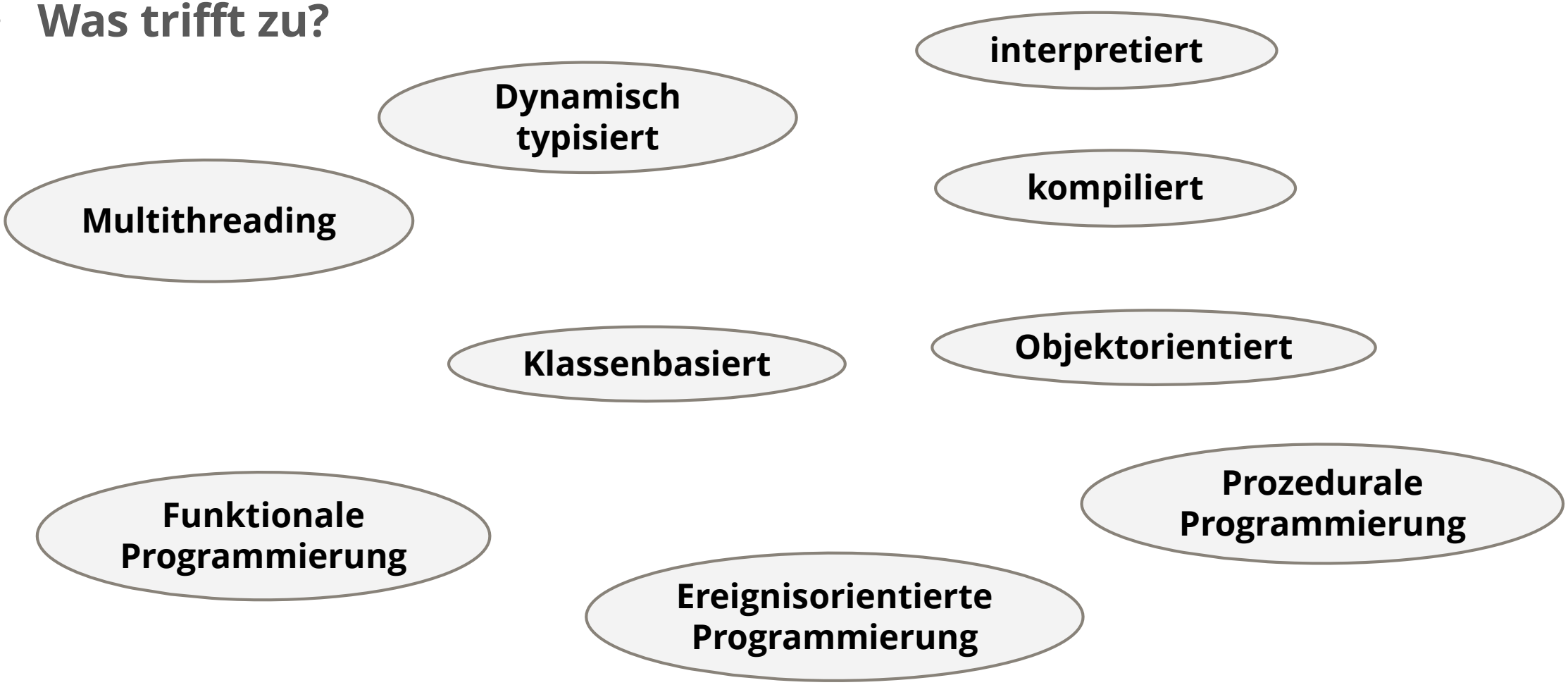


TypeScript

- **„Superset“ (Übermenge) von JavaScript**
  - JavaScript-Code ist gültiger TypeScript-Code
  - JS und TS können gemischt werden
- **Statische Typisierung**
- **Klassen, Vererbung, Interfaces, Generics, …**
- **ES 2015+ Sprach-Features (z.B. let, Fat Arrow)**
- **TS läuft nicht im Browser!**
- **Muss zu JS kompiliert werden**

JavaScript-Eigenschaften

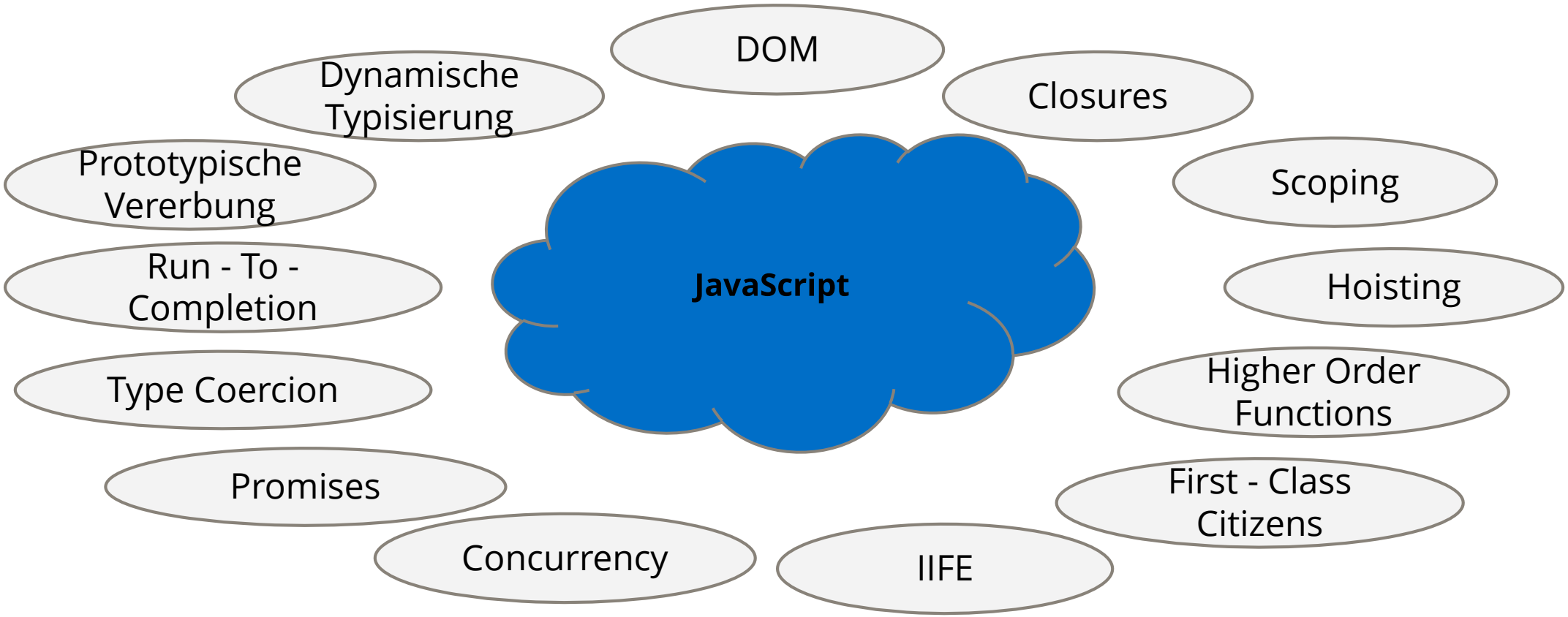
- **Was trifft zu?**



JavaScript-Eigenschaften

- **Dynamisch typisiert**
- **Objektorientiert** (aber klassenlos)
- **Prozedurale Programmierung**
- **Funktionale Programmierung**
- **Ereignisorientierte Programmierung**
- **Single-threaded**
- **Interpertiert**

Themenübersicht



Geschichte

Brendan Eich

Erfinder von JavaScript
Netscape
1995

“Make it look like Java.”
“Make it for beginners.”
“Make it control almost everything in Netscape.”

JavaScript

ist nicht

Java

ECMAScript

- **Offizieller Name er Sprache**
- **ECMA-262: Name des Standards**
- **Alle modernen Brower unterstützen ES5.1**
- **Aktueller Standard: Unübersichtlich! ☹**

  https://www.w3schools.com/jsref/

  https://caniuse.com/

Versionsgeschichte von ECMAScript (ECMA-262)

	1995	Erste Implementierung (LiveScript) – noch kein ECMA-Standard
1	1997	Erste ECMAScript-Version
2	1998	Änderungen zwecks Kompatibilität zu anderen Standards
3	1999	Erweiterung des Sprachumfangs
4	abgebrochen	Keine Einigung bzgl. Weiterentwicklung der Sprache
5	2009	Wichtige Verbesserungen wie z.B. Strict-Mode
5.1	2011	Heute „kleinster gemeinsamer Nenner“ aktueller Browser
2015	2015	Erweiterungen wie Klassen, Module, let, const, …
2016	2016	Weitere Erweiterungen wie z.B. Potenzfunktion
2017	2017	u.a. Asynchroner Code mit async/avait
2018	2018	u.a. rest/spread properties
Next	2019	

Einsatzgebiete von JavaScript

- **Dynamische Manipulation von Webseiten**
- **Validierung von Formulareingaben vor dem Senden**
- **Anzeige von Dialogfenstern**
- **Senden und Empfangen von Daten ohne Seitenwechsel**
- **Vorschlagen von Suchbegriffen während der Eingabe**
- **Werbebanner oder Laufschriften**
- **Lokale Datenspeicherung (Cookies, WebStorage ...)**
- **...**

Struktierung des Programmcodes

- **Kein public static void main!**
- **Ausführung von Funktionen**
- **Beliebig viele Funktionen in einer Datei möglich**
- **Event Handling**

Einbindung von JavaScript in Webseiten

JavaScript direkt im HTML:

```
<script>
     alert('Hallo Welt!');
</script>
```

JavaScript referenzieren:

```
<script src="hallo-welt.js"></script>
```

Syntax

JS-Sprachgrundlagen

- Statements werden mit **;** beendet
- Mehrere Statements in einer Zeile
- Blöcke werden mit **{}** umschlossen
- Einfache und doppelte Anführungszeichen erlaubt

```
if (a) {
      a = 'Hello ' + a;   alert(a);
}
```

Basistypen

- **Number** `3.14`
- **String** `'Hallo'`
- **Boolean** `true`

Primitiv

- **Function** `function (){}`
- **Object** `{}`
- **Array** `[]`

Komplex

Variables - Primitive types

Call by value

```
let a = 'Hello World';
let b = a; // Only value is copied
a = 4;

console.log(b);
// => 'Hello World'
```

Variables - Object types

Call by reference

```
let a = [1, 2, 3];
let b = a; // Copy the reference
a[0] = 99; // Modify the array using the reference

console.log(b);
// => [99, 2, 3]
```

Showcase

typeof

Operatoren

Operatoren (Auswahl)

=	a = 1	Zuweisung
+	1 + 2	Addition/Konkatination
+=	a += 1	Zuweisung und Addition/Konkatination
- * /	2 * 3	Subtraktion, Multiplikation, Division
++	a++	Inkrement
==	a == true	Vergleichsoperator
===	a === true	Strikter Vergleichsoperator
!=	a != true	Ungleich-Operator
!==	a !== true	Strickter Ungleich-Operator
&&	a && b	Logische Und-Verknüpfung
||	a || b	Logische Oder-Verknüpfung

Operatoren (Auswahl)

!	!a	Logischer Negations-Operator
>	a > 1	Größer als
>=	a >= 1	Größer-Gleich
<	a < 1	Kleiner als
<=	a <= 1	Kleiner-Gleich
cond ? val1 : val 2	a = (a < 1) ? 1: a	Konditional-Operator
typeof	typeof a	Typ-Bestimmung / gibt Typ als String aus
instanceof	a instanceof Array	Instanz-Bestimmung / true / false
delete	delete a.b	Lösch-Operator für Objekt-Properties

VARIABLEN

Variablen

Deklaration mit Schlüsselvor “var”, “let” oder “const” und Namen

```
var value;
var $value___123;
```

Beinahme beliebige Namen möglich. Ausnahmen:

=> Leerzeichen, 1. Zeichen numerisch, Bindestrich, JS-Schlüsselwörter

Variablen

Sogar UTF-8 Zeichen!

```
var π = Math.PI;

var ლ_ʊ益ʊ_ლ = 42;

var T̈̈̈ = 'Zalgo';
```

Variablen

Speichern das Ergebnis eines Ausdrucks

```
var helloString = 'Hello World';
var helloFunction = function(){};
var returnValue = getValueOfSomeFunction();

var x;
x === undefined; // true
```

Variablen ohne Zuweisung sind `undefined` !

Variablen

Können primitive Datentypen speichern

```
var a = 3.14;   // Deklaration und Initialisierung
var b = a;      // Kopiere wert in eine neue Variable
a = 4;          // Wert der ursprünglichen Variable ändern
alert(b);       // Zeigt 3.14 an; Kopie wurde nicht geändert
```

Variablen

Können auf Objekte zeigen (Objekt-Referenz)

```
var a = [1,2,3];  // Initialisierung mit Array-Referenz
var b = a;  // Kopiert Referenz (NICHT das Array!)
a[0] = 99; // Ändern des Array über die ursprüngliche Variable
alert(b);  // Anzeige des Arrays über die neue Variable
```

```
// => [99,2,3]
```

null
undefined
NaN

undefined / null / NaN

- **undefined**
  - Implizit für „keine Wert“ / „keine Zuweisung
- **null**
  - Explizit für „kein Wert“
- **NaN**
  - „Not a Number“

Showcase
null / undefined / NaN
+ delete

Bedingungen

Bedingungen

```
if (a > 10) {
        console.log('groß');
} else if (a > 5) {
        console.log('mittel');
} else {
        console.log('klein');
}
```

Bedingungen

```
switch (a) {
    case 'groß':
        console.log('viel');
        break;
    case 'klein':
        console.log('wenig');
        break;
    default:
        console.log('normal');
}
```

FUNKTIONEN

Deklaration und Ausführung

```
var welcome1 = function() {
      alert('Hello JavaScript');
};
welcome1();


function welcome2() {
       alert('Hello JavaScript')
}
welcome2();
```

Funktionen

“First-Class Citizens”

Kapselt/Definiert eine bestimme Logik

Funktionen sind in JS auch Objekte (!)

```
var go = function() { alert('Hello JavaScript') };
go();
go.foo = 'bar';
go.foo; // 'bar'
```

Funktionen

Mit und ohne Parameter

```
var go = function(a) {
      alert('Hello ' + a);
};
go('JavaScript');
go();
```

- go() kann auch ohne Parameter aufgerufen werden!
- Parameter können auch ignoriert werden

Funktionen

Können explizite Rückgabewerte haben

```
var go = function() {
  return 'JavaScript';
};
var a = go();
a; // 'JavaScript'
```

```
var go = function() {
  console.log('JavaScript');
};
var a = go();
a; // undefined
```

Wenn kein Rückgabewert angegeben wird, wird **undefined** zurück gegeben (Ausnahme: Constructor Functions!).

OBJECTS

Objects

Objekte erzeugen

```
var a = {}; // Empfohlen in JS
var b = new Object();
var c = Object.create(Object.prototype);

typeof a // 'object'
typeof b // 'object'
typeof c // 'object'
```

Objects

Enthalten Key-Value-Paare

```
// Gleichwertige 'Objekte
var a = {};
a.x = 5;
a.y = function(){ return 5; };

var b = {
      x: 5,
      y: function(){ return 5; }
};
```

Objects

Setzen von Properties:

```
var obj = {x:5};
console.log(obj.x); // 5
console.log(obj['x']); // 5
```

Zugriff auf Properties mit Leerzeichen:

```
var obj = {'JS rocks':5};
console.log(obj['JS rocks']); // 5

var name = 'JS rocks';
console.log(obj[name]); // 5
```

Objects

Löschen von Properties

```
var obj = {};
obj.a = 5;
console.log(obj.a); // 5

delete obj.a; // Property a löschen
console.log(obj.a); // undefined
```

Schlüsselwort `delete` löscht Properties

JSON

- **JavaScript Object Notation**

Im Programmcode:

```
{
  name: 'Einstein',
  vorname: 'Albert'
}
```

Datenaustausch:

```
{
  "name": "Einstein",
  "vorname": "Albert"
}
```

Objects vs. Maps

- **Objekte werden in JS auch als Maps genutzt**
- **Maps: Ab ES 2015**

ARRAYS

Arrays

Arrays sind Wertelisten

```
var myArray = ['a','b'];
// oder
var myArray = new Array('a','b');
```

Zugriff auf Array-Elemente

```
var myArray = ['a','b'];
console.log(myArray[0]); // a
```

Arrays sind **sortiert** – im Gegensatz zu Objects!

Arrays

Stack-Nutzung von Arrays [push() / pop()]

```
var myArray = ['a'];
myArray.push('b');
console.log(myArray.pop()); // b
console.log(myArray.pop()); // a
console.log(myArray.pop()); // undefined
```

Hinzufügen und Löschen von Elementen mit splice()

```
var arr = [1,2,3];
arr.splice(1,1,4,5,6)      splice(index, howMany, [element,..])
console.log(arr);
// [1,4,5,6,3]
```

Arrays

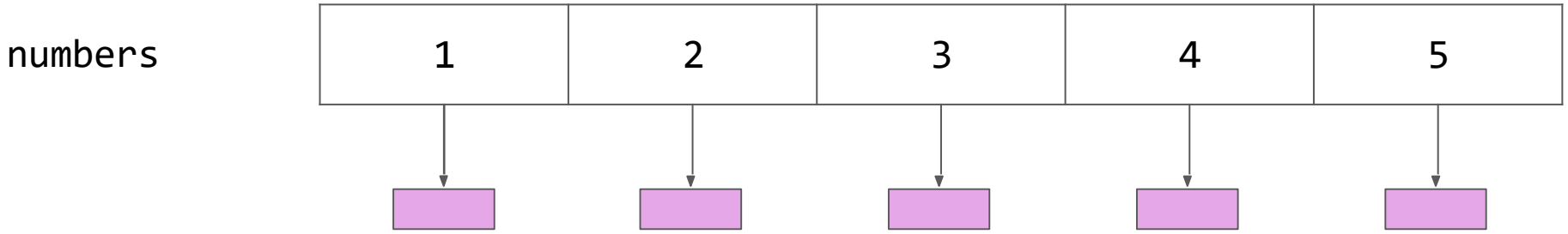
Iterieren mit For-Schleife

```
var arr = ['a','b'], length = arr.length;
for (var i = 0; i < length; i++) {
    console.log(arr[i]);
}
```

Arrays - Iterators

Array.forEach()

```
const myArray = [1,2,3,4,5];
myArray.forEach(elem=>console.log(elem));
```



`forEach()` is **slower than** using a `for` loop**!**

Weitere Schleifen

```
const arr = [3, 5, 7];

for (var i in arr) {
    console.log(i); // logs "0", "1", "2
}

for (const i of arr) {
    console.log(i); // logs "3", "5", "7"
}
```

Weitere Schleifen

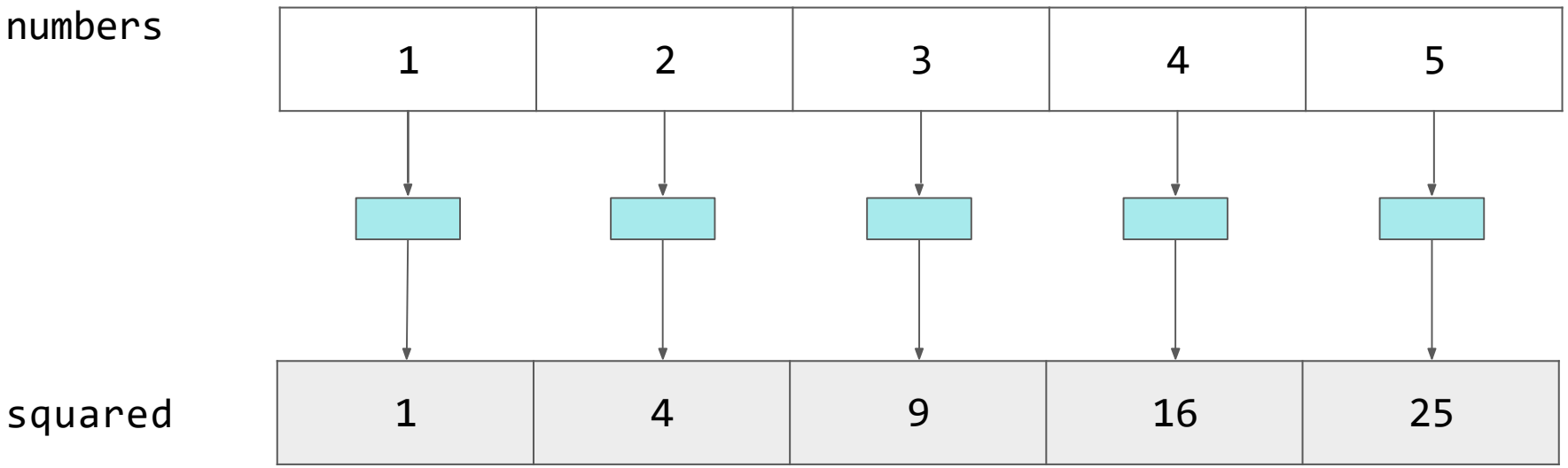
```
while (condition) {

}


do {

} while (condition);
```

Arrays - Transformations

Array.map()

```
const numbers = [1, 2, 3, 4, 5];
const squared = numbers.map(num => num * num);
//  squared is [1, 4, 9, 16, 25]
```

Transforming an array

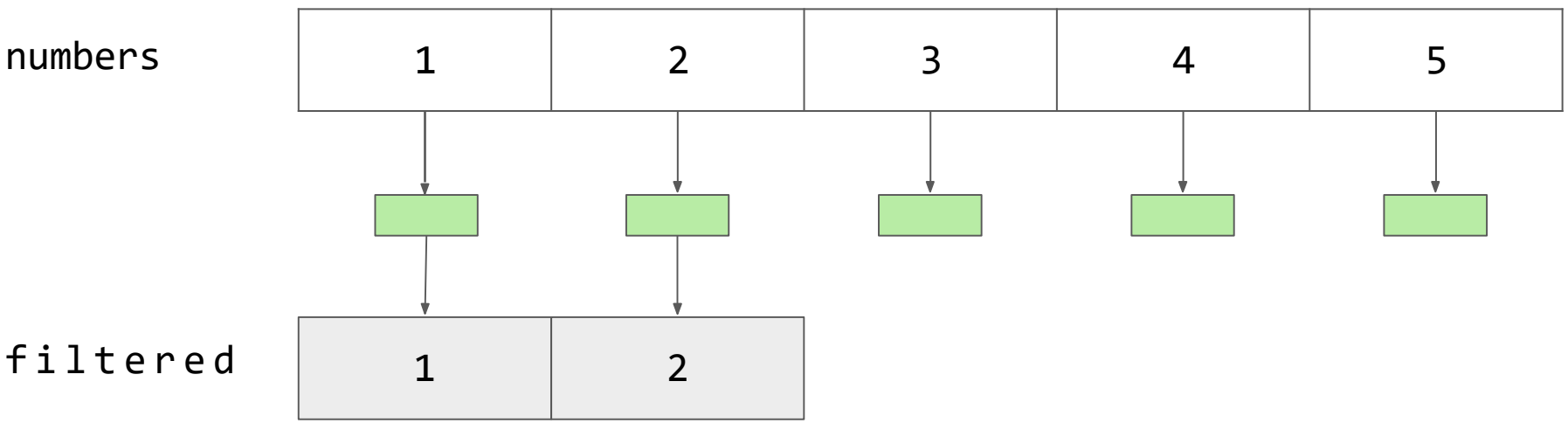


Arrays - Transformations

Array.filter()

```
const numbers = [1, 2, 3, 4, 5];
const filtered = numbers.filter(num => num < 3);
// filterd is [1, 2]
```

Filtering an array



Arrays

Array.some()

```
var arr = [1,2,3,4,5];
arr.some(function(num) {
  return num < 3;
}); // true
```

Überprüfen:
Mindestens ein Element, das …

Array.every()

```
var arr = [1,2,3,4,5];
arr.every(function(num) {
  return num < 10;
}); // true
```

Überprüfen:
Für alle Elemente gilt, dass …

Fallstricke

Object Types

- **Number, String, Boolean**
- **Primitive Typen und Boxed-Object-Typen**
- **Autoboxing / Unboxing ähnlich wie in Java**

```
var a = new Number(2);
var b = 5;
a + b // 7
```

```
var a = new String('Hello');
var b = 'World';
a + ' ' + b // 'Hello World'
```

```
var a = new Boolean(false);
var b = true;
a && b // true (!!!)
```

Vorsicht mit dem Boolean **Object Type!**

Number-Fallstricke

Zum Beispiel:

```
0.1 + 0.2 // 0.30000000000000004
```

Boolean-Fallstricke

Achtung bei Boolean-Objects!

```
var a = new Boolean(false);
if (a) {
      // Wird ausgefürt
}
var b = false;
if (b) {
      // wird NICHT ausgeführt!!!
}
```

Alle Objekt-Referenzen, die nicht `undefined` oder `null` sind, werden zu `true` evaluiert.

Konvetierung mit Boolean()

```
var a = new Boolean('Hello JavaScript');
// a ist ein Boolean Object (true).

var b = Boolean('Hello JavaScript'); // Boolean als function!
// b ist a ein Boolean Primitive (true).

a == b // true … implizite Type Coercion … später mehr!
a === b // false
```

Ausführungsmodell

Ausführungsmodell (Execution Model)

- **Single Threaded**
  - JS-Programme werden nur in einem Thread ausgeführt
  - Keine Möglichkeit, neue Threads zu erzeugen
  - (Ausnahme: Web Workers)

- **Run-To-Completion**
  - JS-Programme können nicht unterbrochen werden
  - Ausführung aller Anweisungen bis zum Schluss

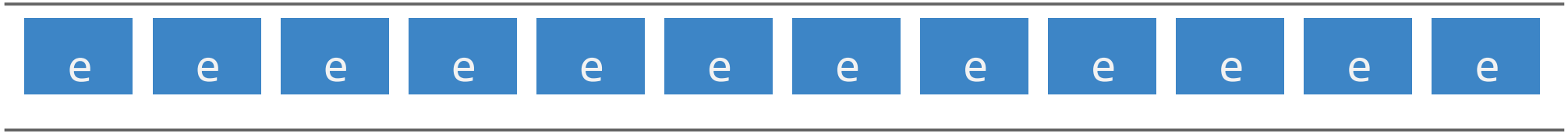
Event Loop

event based / event loop / event queue

Pseudo Code:

```
// event loop (1 thread)
while (true) {
        e = getEvent(); // Blockierender Aufruf
        invokeHandler(e); // Run-To-Completion
}
```

Execution Model




```
// event loop (1 thread)
while (true) {
        e = getEvent(); // Blocking
        invokeHandler(e); // Run-To-Completion
}
```

Long running tasks

after ~10seconds

Showcase

DOM

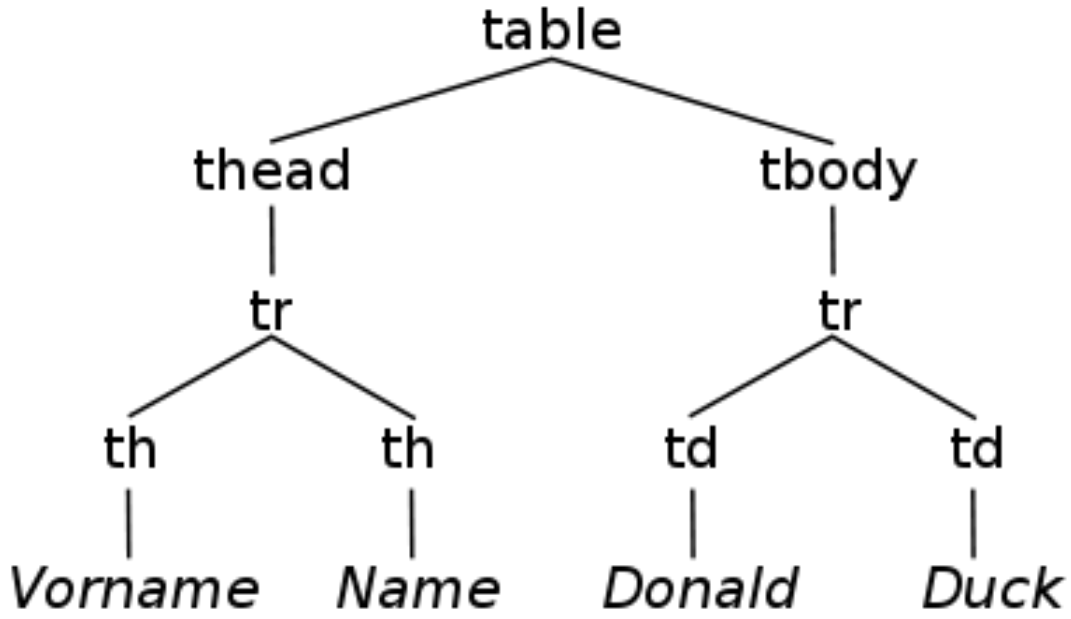
(Document Object Model)

DOM

- **Schnittstelle zwischen HTML und JavaScript**
- **HTML-Dokumente als Baum-Struktur**
- **Veränderung des HTML-Dokuments durch JavaScript**
  - Entfernen von Elementen
  - Hinzufügen von Elementen
  - Manipulation von Eigenschaften (HTML + CSS)

DOM

```
<table>
  <thead>
    <tr>
      <th>Vorname</th>
      <th>Name</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Donald</td>
      <td>Duck</td>
    </tr>
  </tbody>
</table>
```



DOM


> document.querySelectorAll('table')
▼[table]
▼0: table
accessKey: ""
align: ""
assignedSlot: null
▶attributes: NamedNodeMap {length: 0}
baseURI: "file:///C:/dev/pri/ista/html2/index.html"
bgColor: ""
border: ""
caption: null
cellPadding: ""
cellSpacing: ""
childElementCount: 2
▶childNodes: (5) [text, thead, text, tbody, text]
▶children: (2) [thead, tbody]
▶classList: [value: ""]
className: ""

Showcase

Event Binding

Event Binding

```
// Funktion „starteApp“ startet erst beim durch Event „DOMContentLoaded“
document.addEventListener('DOMContentLoaded', starteApp, false);
```

```
// Click-Handler zu Button hinzufügen (programmatisch)
let element = document.getElementById('myButton');
element.addEventListener('click', onClick);
```

```
// Click-Handler zu Button hinzufügen (HTML)
<input type="button" id="myButton" value="Klick!" onclick="onClick()">
```

TYP-UMWANDLUNG

(TYPE COERCION)

Quiz

http://pentarakis.de/javascript/#/

Typ-Umwandlung (Type Coercion)

Vergleich mit “==“ erzwingt Typ-Umwandlung

```
null == undefined;              // true
false == 0;                     // true
'' == 0;                        // true
'5' == 5;                       // true

null === undefined;     // false
false === 0;            // false
'' === 0;               // false
'5' === 5;              // false
```

Typ-Umwandlung (Type Coercion)

Überglick für ==

```
1 == 1      // returns true
"1" == 1    // returns true ("1" is converted to 1)
1 == true   // returns true
0 == false  // returns true
"" == 0     // returns true ("" is converted to 0)
"   " == 0  // returns true ("   " converted to 0)
0 == 1      // returns false
1 == false  // returns false
0 == true   // returns false
```

Typ-Umwandlung (Type Coercion)

Überblick für !=

```
0 != false // returns false
'' != 0    // returns false ('' is converted to 0)
'  ' != 0  // returns false ('  ' is converted to 0)
0 != 1     // returns true
1 != false // returns true
0 != true  // returns true
```

Typ-Umwandlung (Type Coercion)

Überblick für ===

```
1 === 1     // returns true
'1' === 1   // returns false ('1' is not converted)
1 === true  // returns false
0 === false // returns false
'' === 0    // returns false ('' is not converted)
'  ' === 0  // returns false ('  ' is not converted)
0 === 1     // returns false
1 === false // returns false
0 === true  // returns false
```

Typ-Umwandlung (Type Coercion)

Objekte Vergleich mit ==

```
var x, y;      // declare x and y
  x = {};      // create an object and assign it to x
  y = x;       // point y to x
  x == y;      // returns true (refers to same object in memory)
  x == {};     // returns false (not the same object)
```

Typ-Umwandlung (Type Coercion)

Objekt-Vergleich mit !=

```
var x, y;   // declare x and y
 x = {};    // create an object and assign it to x
 y = x;     // point y to x
 x != y;    // returns false (refers to same object)
 x != {};   // returns true (not the same object)
```

Typ-Umwandlung (Type Coercion)

Objekt-Vergleich mit ===

```
var x, y;           // declare x and y
    x = {};         // create an object and assign it to x
    y = x;          // point y to x
    x === y;        // returns true (refers to same object)
    x === {};       // returns false (not the same object)
    y === {};       // returns false (not the same object)
```

Vergleich von Objekten: Kein Unterschied zwischen:

== u. === !

Typ-Umwandlung (Type Coercion)

Primitive Box Types

```
new Number(3) == new Number(3) // false (not the same object)
new Number(3) === new Number(3) // false (not the same object)
new String('A') == new String('A') // false (not the same object)
new String('A') === new String('A') // false (not the same object)
new Boolean(true) == new Boolean(true) // false (not the same object)
new Boolean(true) === new Boolean(true) // false (not the same object)
```

Vergleich von Objekten: Kein Unterschied zwischen:
== u. === !

Truthy & Falsy

Falsy:

```
false
0                (Null)
''               (Leerstring)
'            '   (beliebig viele Leerzeichen)
null
undefined
NaN
```

Truthy: Alles, was nicht “falsy” ist!

Typ-Umwandlung (Type Coercion)

- **Vergleiche mit === sind performanter**
- **Vergleiche mit == solten vermieden werden**

Typ-Umwandlung (Type Coercion)

Umwandlung in if-Condition

```
if ('') // false
if ('5') // true
if (0) // false
if (-1) // true
if (null) // false
if (undefined) // false
```

SCOPING

Scopes

var ist function-scoped

```
var variable = 'parent scope';
function fn() {
      var variable = 'local scope';
      console.log(variable);
}
fn();
// => local scope

console.log(variable);
// => parent scope
```

var ist function-scoped

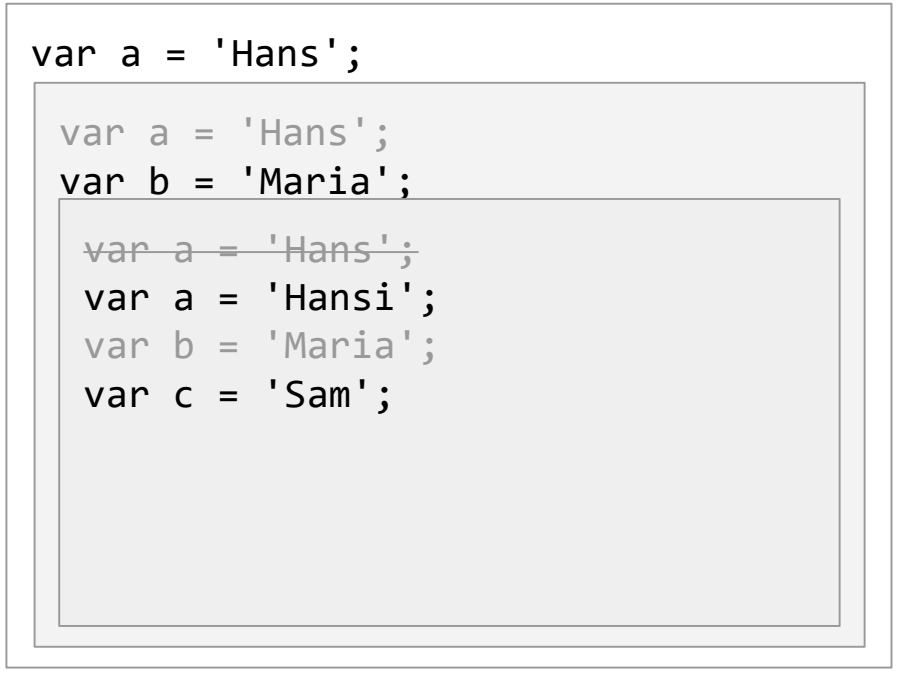
```
var something = 1;
if (true) {
        var something = 2;
        console.log('Inside: ' + something);
}
console.log('Outside: ' + something);

// => Inside: 2
// => Outside: 2
```

let ist block-scoped

```
let something = 1;
if (true) {
        let something = 2;
        console.log('Inside: ' + something);
}
console.log('Outside: ' + something);

// => Inside: 2
// => Outside: 1
```

Scoping

```
(function first() {
    let a = 'Hans';
    (function second() {
        let b = 'Maria';
        (function third(){
            let c = 'Sam';
            let a = 'Hansi';
            console.log(a,b,c);
        })();
    })();
})();
```



const

Reassigning throws an error

```
const dateOfBirth = new Date();

dateOfBirth = new Date(); // compile error!
```

const with objects

Only the reference is immutable

```
const myObject = {
  name: 'John',
  dateOfBirth: '1981-10-22'
};

// Object is mutable!
myObject.name = 'Andreas';

// but you cannot change the reference
myObject = {name: 'Peter'}; // this throws an error!
```

Immediately Invoked Function Expression (IIFE)

JS ist function-scoped.

Deshalb macht der Einsatz von IIFEs Sinn!

```
(function() {
      // Here is a new scope!
      // From outside nobody can access this scope!
})();
```

IIFE Pattern

Parameter-Übergabe

```
(function($) {
      // Here is a new scope!
      // From outside nobody can access this scope!
})(jQuery);
```

IIFE Pattern

Rückgabewert

```
var myService = (function() {
        // Here is a new scope!
        // From outside nobody can access this scope!

        var a = 42;
        var b = 'TEST';

        return {
                getA: function() { return a; },
                getB: function() { return b; }
        };
})();
```

HOISTING

Hoisting

Achtung! Die JS-Runtime “hebt” alle Variablen-Deklarationen an den Anfang einer Funktion.

```
var name = 'Emma';
function nameHer() {
    var name;
    console.log(name);
    name = 'Audrey';
}
```

Hoisting

Achtung! Die JS-Runtime “hebt” alle Variablen-Deklarationen an den Anfang einer Funktion.

```
var name = 'Emma';
function nameHer() {
    var name;
    console.log(name); // undefined
    name = 'Audrey';
}
```

Hoisting

Achtung! Die JS-Runtime “hebt” alle Variablen-Deklarationen an den Anfang einer Funktion.

```
var name = 'Emma';
function nameHer() {
    console.log(name);
    name = 'Audrey';
    var name;
}
```

Hoisting

Achtung! Die JS-Runtime “hebt” alle Variablen-Deklarationen an den Anfang einer Funktion.

```
var name = 'Emma';
function nameHer() {
    console.log(name); // undefined
    name = 'Audrey';
    var name;
}
```

Hoisting

Achtung! Die JS-Runtime “hebt” alle Variablen-Deklarationen an den Anfang einer Funktion.

```
var name = 'Emma';
function nameHer() {
    console.log(name);
    var name = 'Audrey';
}
```

Hoisting

Achtung! Die JS-Runtime “hebt” alle Variablen-Deklarationen an den Anfang einer Funktion.

```
var name = 'Emma';
function nameHer() {
    console.log(name); // undefined
    var name = 'Audrey';
}
```

Hoisting

Achtung! Die JS-Runtime “hebt” alle Variablen-Deklarationen an den Anfang einer Funktion.

```
console.log(hi());
// function declaration
function hi() {
    return 'Hi!';
}
```

Function hi() is executed.

```
console.log(hi());
// function expression
var hi = function () {
    return 'Hi!';
}
```

Function hi() is `undefined`.

HIGHER ORDER FUNCTIONS

Higher Order Functions

Funktionen die eine Funktion als Parameter entgegen nehmen

```
var doGetRequest = function(callback) {
      // do something...
      callback();
};

doGetRequest(function() {
      console.log('Ready!');
});
```

Higher Order Functions

Funktionen, die Funktionen zurück geben

```
var createAdder = function() {
       return function(a, b) {
              return a + b;
   };
};

var add = createAdder();
add(2, 3); // 5


createAdder()(2, 3)
```

CLOSURES

Closures

What happens with the variable after the function is terminated?

```
function getNumber() {
    let myNumber = 13;
    return myNumber;
}
getNumber();
```

Closures

The result is?

```
let createFunction = () => {
    let localVar = 123;

    return function(value) {
        return localVar + value;
    };
};

createFunction()(10); // ???
```

Closures

Funktionen, die Variablen “einschließen” (enclose)

```
let createFunction = () => {
    let localVar = 123;

    return function(value) {
        return localVar + value;
    };
};
createFunction()(10); // 133
```

- The inner function encloses *localVar* because it has read access to *localVar*.
- The **inner anonymous function** is a so-called **closure**.

Closures

Higher Order Functions und Closures für sehr elegante Konstrukte kombinieren:

```
var createLogger = function(loggerName) {
       return function(msg) {
              console.log('[' + loggerName + '] ' + msg);
   };
};

var info = createLogger('INFO');

info('User successfully logged in!');
```

Closures

Higher Order Functions und Closures für sehr elegante Konstrukte kombinieren:

```
var createLogger = function(loggerName) {
       return function(msg) {
              console.log('[' + loggerName + '] ' + msg);
   };
};

var info = createLogger('INFO');

info('User successfully logged in!');
// [INFO] User successfully logged in!
```

Closures

Achtung bei Closures und asynchronem Code in Schleifen!

```
for (var i = 0; i < 3; i++) {
    setTimeout(function() {
        console.log(i);
    }, 3000);
}
// log-output?
```

Closures

Achtung bei Closures und asynchronem Code in Schleifen!

```
for (var i = 0; i < 3; i++) {
    setTimeout(function() {
        console.log(i);
    }, 3000);
}
// => 3
// => 3
// => 3
```

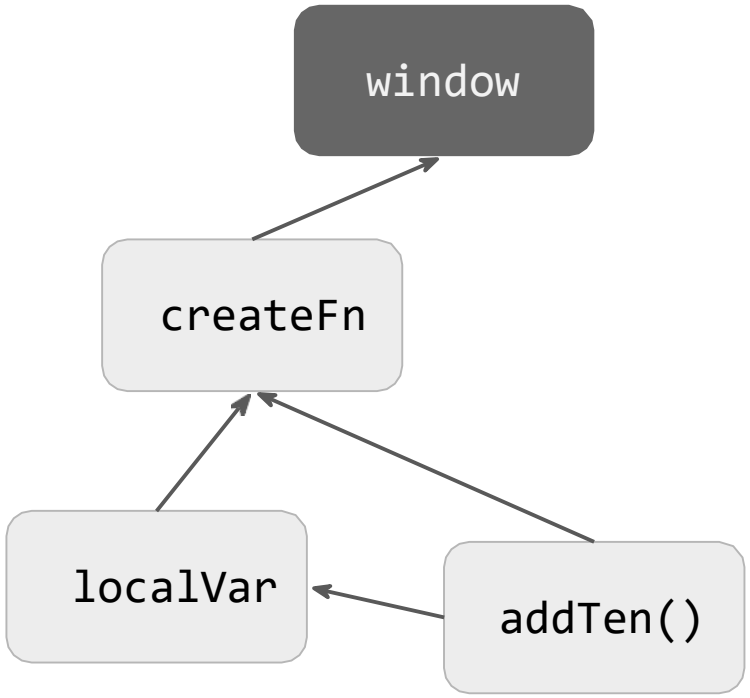
Closures

Achtung bei Closures und asynchronem Code in Schleifen!

```
for (var i = 0; i < 3; i++) {
    (function(i) {
        setTimeout(function() {
            console.log(i);
        }, 3000);
    })(i);
}
// => 0
// => 1
// => 2
```

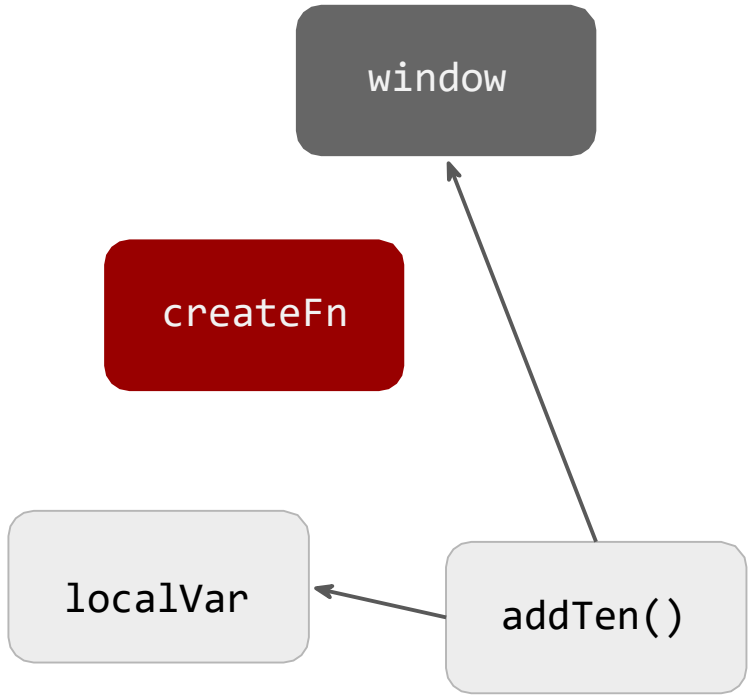
Garbage Collection

```
let createFunction = function() {
    let localVar  = 123;

    return  function() {
        return localVar + 10;
    };
};

let addTen = createFunction();
addTen(); // 133
```



Garbage Collection

- Mark & sweep
- Reference counting
- Elements without refs are garbage collected



AJAX
&
fetch

AJAX

- **Asynchronous JavaScript and XML**
- **Asynchrone Datenübertragung**
- **Kein Page Reload**
- **Weniger Overhead**
- **Kein „Flackern“**
- **XHR / XMLHttpRequest-Objekt**
- **Voraussetzung für Single-page Applications!**

AJAX

```
// Beispiel AJAX-Request
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function () {
   if (this.readyState == 4 && this.status == 200) {
      // Asynchroner Code / Callback
      document.getElementById("demo").innerHTML =
      this.responseText;
   }
};
xhttp.open("GET", "ajax_info.txt", true);
xhttp.send();
```

Einfacher mit fetch-API

```
fetch('https://server/api/resource/1')
  .then(response => {
    if (response.status === 200) {
      return response.json();
    } else {
      throw new Error('Something went wrong on api server!');
    }
  })
  .then(jsonData => {
    console.log(jsonData);
  }).catch(error => {
    console.error(error);
  });
```

Hands-On: fetch

https://www.anapioficeandfire.com/api/characters?pageSize=20

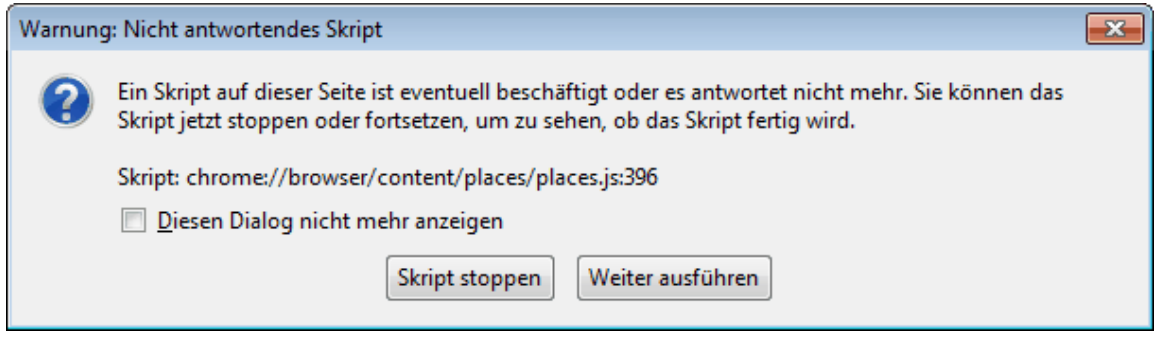
https://anapioficeandfire.com/Documentation

PROMISES & ASYNCHRONITÄT

```
((value) => {
  setTimeout(() => {
    console.log('d');
  }, 100);

  setTimeout(() => {
    console.log('u');
  }, 0);

  console.log(value);

})('e');
// => ?
```

Async / Non-Blocking Calls

Nochmal: JavaScript ist single-threaded



Don’t block the event loop!

Async / Non-Blocking Calls

Asynchroner Code ist allgegenwärtig!

```
var url = 'http://mydomain.com/api/user/42';

var user;

makeHttpRequest(url, function(response) {
      user = response.data.user;
}); // makeHttpRequest() is async / non-blocking!

console.log('User data', user); // user is undefined!
```

Async / Non-Blocking Calls

Asynchroner Code ist allgegenwärtig!

```
var url = 'http://mydomain.com/api/user/42';

var user;

makeHttpRequest(url, function(response) {
      user = response.data.user;
      console.log('User data', user); // user is defined!
}); // makeHttpRequest() is async / non-blocking!
```

Async / Non-Blocking Calls

Verschachtelung von Async-Calls

```
// ...
makeHttpRequest(url, function(response) {
 performOperation(response.data.user, function(result) {
       calc(result, function(calcResult) {
             doSomething(calcResult, function() {
                   ...
                   });
             });
       });
});
```

“Pyramid of Doom”

Verschachtelung von Async-Calls

```
// ...
makeHttpRequest(url, function(response) {
 performOperation(response.data.user, function(result) {
       calc(result, function(calcResult) {
             doSomething(calcResult, function() {
                   ...
                   });
             });
       });
});
```

Was ist mit Error-Handling?

Async / Non-Blocking Calls

Try-Catch funktioniert nicht!

```
function doSomethingAsync() {
    setTimeout(function() {
        console.log('doSomethingAsync() called!');
        throw new Error('Something went wrong!');
    }, 0);
}

try {
    doSomethingAsync();
} catch(e) {
    // This won't catch the error thrown in doSomethingAsync()!
}
```

Sinnvolle Fehlerbehandlung?

Async / Non-Blocking Calls

Wie vermeiden wir die “Pyramid of Doom”?
Was ist mit Error-Handling?

Promises!

Promises
Wieso?

Callback-Hölle und ***Pyramide of Doom*** vermeiden

Ausführung von oben nach unten

Error-Handling

Wiederherstellung im Fehlerfall

Promises

Offener Standard

Promises/A+

Implementierungen

ES2015, Q, avow, D.js, PinkySwear.js, rsvp.js, ...

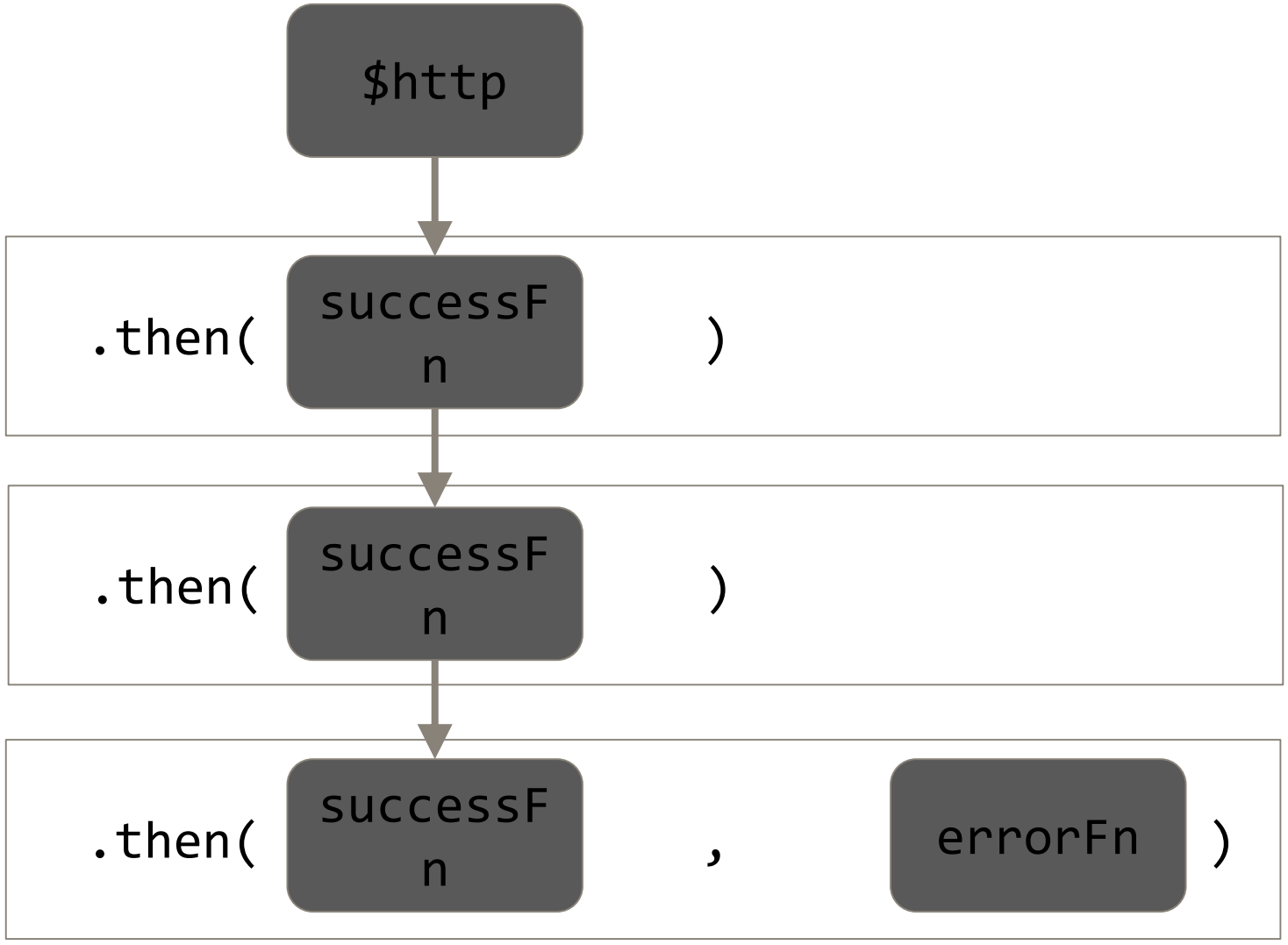
“A promise represents the eventual result of an asynchronous operation.”

Verschachtelte Async-Calls mit Promises

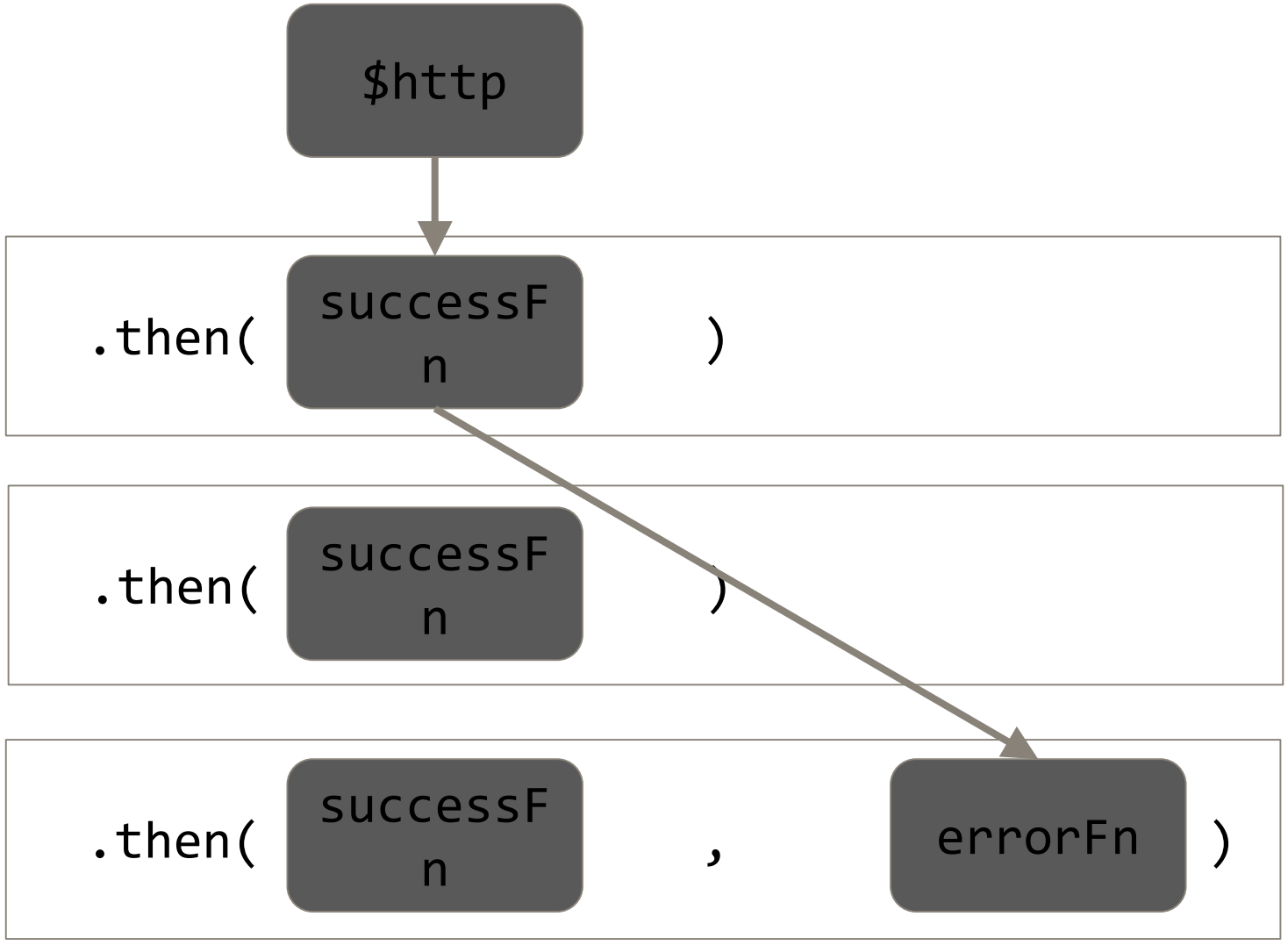
```
makeHttpRequest(url)
.then(function(response) {
      return response.data.user;
})
.then(performOperation)
.then(calc)
.then(doSomething, function(error) {
      console.log('An error occurred!', error);
});
```

Downstream Ergebnis- und Fehler-Weitergabe

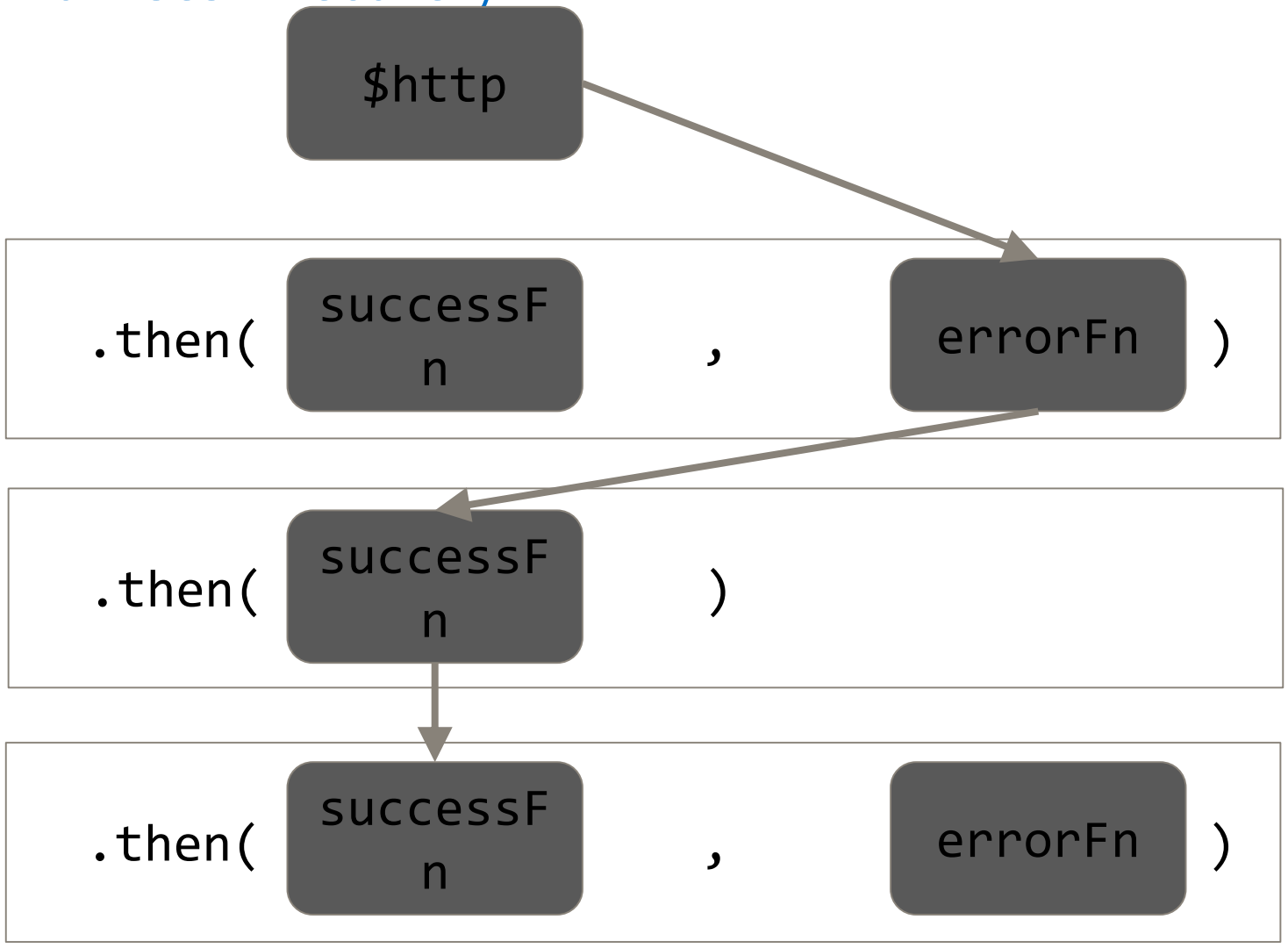
Promises - Success



Promises - Error



Promises - Recovery



Einfaches Beispiel: Asynchrone Operation

```
var isEvenAsync = function(input, callbackFn) {
	setTimeout(function() {
		var even = (input % 2) === 0;
		callbackFn(even);
	}, 0);
};

// invocation
isEvenAsync(13, function(result) {
	console.log('13 is even: ' + result);
});
```

Promises

Und jetzt mit Promises:

```
var isEvenAsync = function (input) {
  return new Promise(function (resolve) {
    setTimeout(function () {
      var even = (input % 2) === 0;
      resolve(even);
    }, 0);
  });
};

// invocation
isEvenAsync(13).then(function(result) {
  console.log('13 is even: ' + result);
});
```

PROTOTYPEN & OBJEKTORIENTIERUNG

Ein Objekt erbt alle Properties von seinem Prototype

```
var audi = {
       sayAudi: function() { console.log('I am an Audi!'); }
};
audi.sayAudi(); // 'I am an Audi!'

var audiR8 = Object.create(audi);
audiR8.sayAudiR8 = function() { console.log('I am an Audi R8!'); };
audiR8.sayAudi(); // 'I am an Audi!'
audiR8.sayAudiR8(); // 'I am an Audi R8!'
```

Prototypes

Wurde kein **Prototype** angegeben, wird `Object.prototype` verwendet

```
var audi = {
       [...]
};

var audiR8 = Object.create(audi);
[...]

Object.getPrototypeOf(audiR8) === audi // true
Object.getPrototypeOf(audi) === Object.prototype // true
```

Prototypes

Ein **Prototype** sollte nach der Objekt-Erzeugung nicht geändert werden!

> *Mutating the* `[[Prototype]]` *of an object, using either this method or the deprecated* `Object.prototype.__proto__`*, is strongly discouraged, because it is very slow and unavoidably slows down subsequent execution in modern JavaScript implementations.*

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/setPrototypeOf

Functions und Prototypes für Pseudo-Klassen

```
function Person(name) {
    this.name = name;
};

Person.prototype.sayName = function() {
    console.log('My name is ' + this.name);
};

var p = new Person('Max Mustermann');
Object.getPrototypeOf(p) === Person.prototype // true
```

Prototypes

Vererbung kann auch emuliert werden

```
// Person "class" is defined (see previous slide)
function Employee(name, salary) {
        Person.call(this, name); // call Person constructor with created object.
        this.salary = salary; // add property specific to Employee objects
}

// Employee extends Person
Employee.prototype = Object.create(Person.prototype);
Employee.prototype.constructor = Employee;

// add method specific to Employee objects
Employee.prototype.saySalary = function() {
        console.log('My salary is ' + this.salary + ' EUR / year.');
};

var frank = new Employee('Frank Sinatra', 68000);
frank.sayName(); // 'My name is Frank Sinatra'
frank.saySalary(); // 'My salary is 68000 EUR / year.'
```

Prototypes

Sehr umständlich, oder?

this

this in Functions

- **this ist abhängig von der Art des Aufrufs**
- **'use strict' verhält sich änders**

this – Arrow Functions

In arrow functions, `this` is set lexically, i.e. it's set to the value of the enclosing execution context's `this`.

```
const outerContext = this
const fatArrowFunction = () => this === outerContext

fatArrowFunction() // ==> true
```

this in Objekten

```
var obj = {
   answer: 42,
   fn: function() {
      return this.answer;
   }
};

console.log(obj.fn()); // => 42
```

this in Constructor-Funktionen

- In Construktur-Funktion ist **this** an das neue Objekt gebunden
- Constructor gibt standardmäßig **this** zurück

```
function MyConstructor() {
  this.a = 42
}
const myInstance = new MyConstructor()  // this is returned per default
console.log(myInstance.a) // ==> 42

function MyC2() {
     this.a = 30
     return {a: 20}
}
const i2 = new MyC2()
console.log(i2.a) // ==> 20
```

this ohne ‘use strict’

- **Window-Objekt!**

this mit ‘use strict’

- **undefined**

ES 2015+

Classes

- Code is more readable
- Syntactic sugar over prototype-based inheritance
- Not introducing a new class-based inheritance model

Classes

Class can have a constructor, attributes and methods.

```
class Person {
  bornOn;

  constructor(public name) {
    this.bornOn = new Date();
  }

  shout() {}
}
```

Classes - Instances

Create new instances with the `new` keyword.

```
class Person {…}

const john = new Person('John');

john.bornOn; // => a Date object

john.shout(); // => nothing but alerts
```

Classes - Inheritance

You can inherit from another class. Use super to call the constructor.

```
class Person {
  constructor(public name) {…}
}

class Employee extends Person {
  constructor(name, public salary) {
    super(name);
    //  …
  }
}
```

Functions - Rest/Spread parameter

An abitrary amout of parameters can be stored in an array.

```
function buildName(firstName, ...restOfNames) {

  let allNames = [firstName, ...restOfNames];
  // names = [firstName, restOfName[0], restOfName[1] …]

  return allNames.join(' ');
}
```

Template Strings

Strings - Template string

Variables in strings (multiline support!)

```
const name = 'Tom';

const string =
      `My name
      is ${name}!`;

// =>My name
      is Tom!
```

Destructuring

Destructuring - Objects

Get multiple local variables from an object with destructuring.

```
let circle = {radius: 10, x: 140, y: 70};

let {x, y}  = circle;
// let x  = circle.x;
// let y  = circle.y;

console.log(x, y)
//  => 140, 70
```

Destructuring - Arrays

Get multiple local variables from an object with destructuring.

```
let coords = [51, 6];

let [lat, lng] = coords;
// let lat = coords[0];
// let lng = coords[1];

console.log(lat, lng)
// => 51, 6
```

async / await

await

- **Stellt sicher, dass eine Funktion ein Promise zurück gibt**
- **Wrapped Nicht-Promise-Returns automatisch in einem Promise**

```
async function func() {
  return 1;
}

func().then(console.log); // 1
```

await

- Asynchroner Code kann linear geschrieben werden
- Nachfolgender Code wird erst ausgeführt, wenn Promise resolved wurde.
- Steht nur in async-Funktions zur Verfügung
- Kein echtes Warten!

```
async function fn() {
  let promise = new Promise((resolve, reject) => {
    setTimeout(() => resolve("done!"), 1000);
  });
  let result = await promise; // „wait“ for result

  alert(result); // "done!"
}
fn();
```

Vielen Dank!

adesso business. people. technology.

adesso AG
Stockholmer Allee 20
44269 Dortmund
T +49 231 7000-7000
F +49 231 7000-1000
www.adesso.de