Verteilte Systeme 1

Technologien des World Wide Web

christian.zirpins@hs-karlsruhe.de

JavaScript

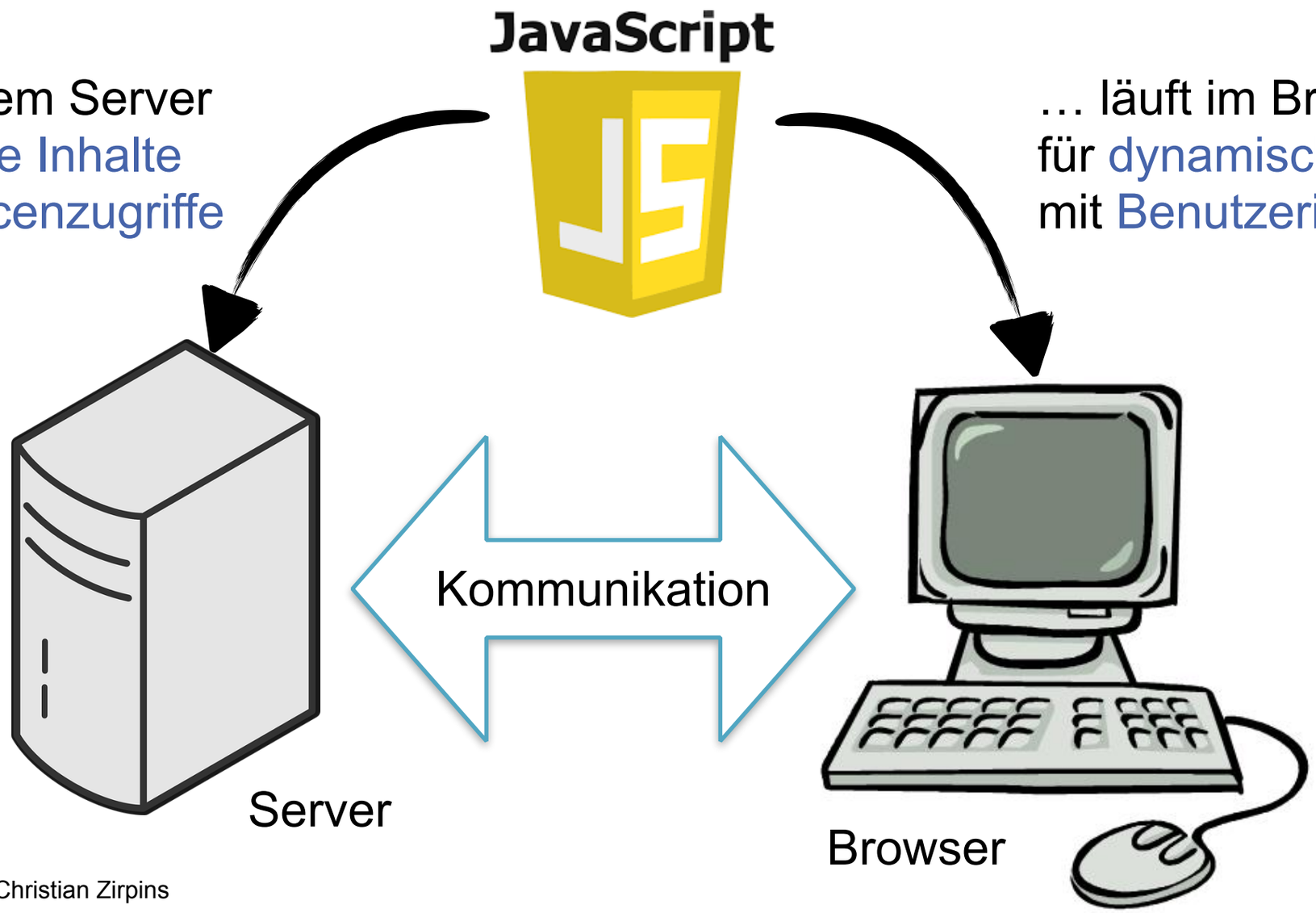
Hochschule Karlsruhe
Technik und Wirtschaft
UNIVERSITY OF APPLIED SCIENCES

Lernziele

Nach dieser Vorlesung können Sie…

- … **Scripting** in der Webentwicklung einordnen
- … Algorithmen mit **JavaScript Programmen** implementieren
- … **OO-Prinzipien** in JavaScript verwenden

… läuft auf dem Server
für individuelle Inhalte
und Ressourcenzugriffe



… läuft im Browser
für dynamische Webseiten
mit Benutzerinteraktion

Ein wenig Kontext

Geschichte

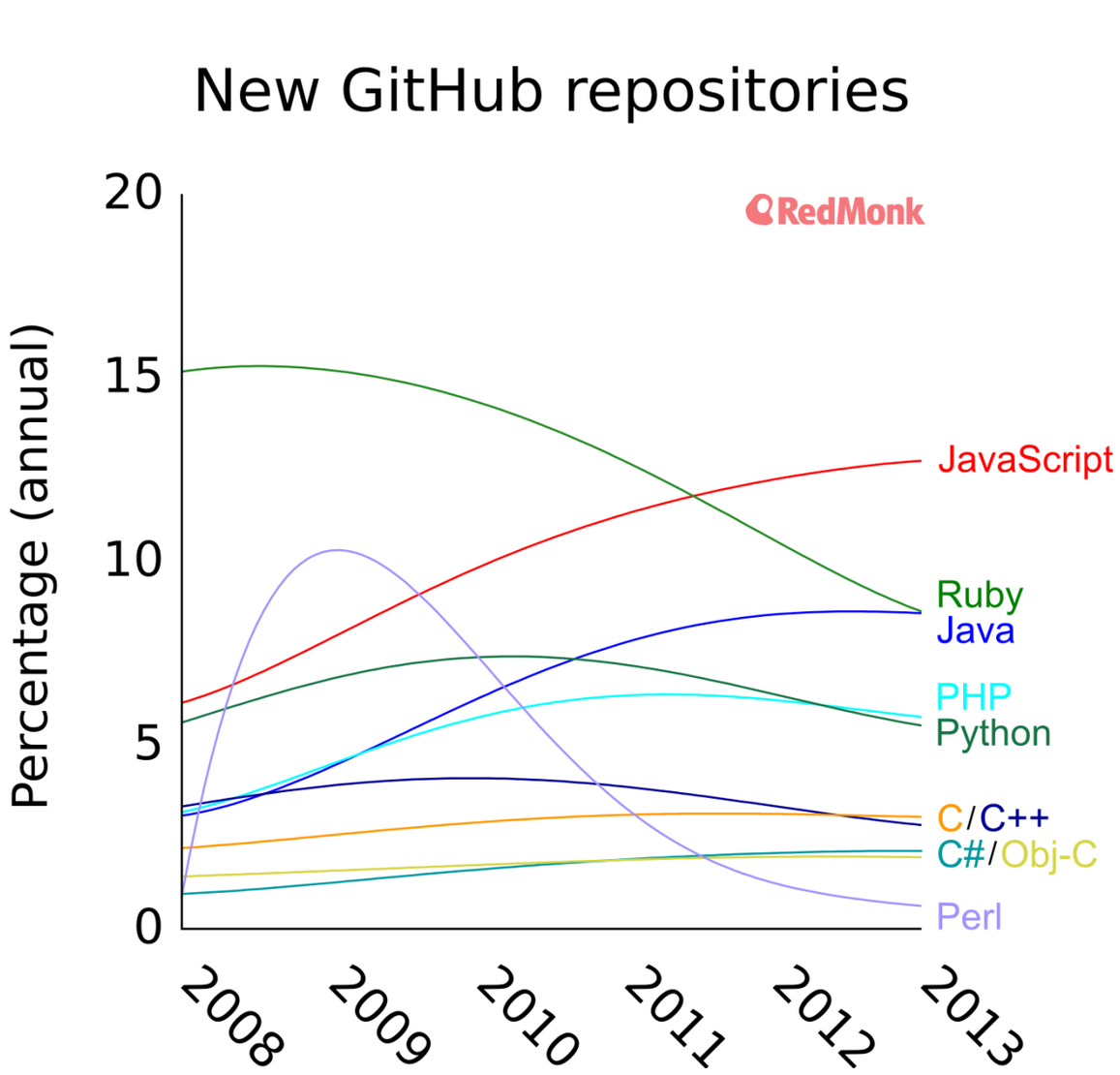
- **1995**: Brendan Eich entwickelt **LiveScript** für den Netscape Browser Client **(**Navigator 2.0)
- **1996**: Umbenennung in **JavaScript (**Version 1.1) für Netscape Navigator 3 und Internet Explorer 3.0.
- **1998**: JavaScript 1.3 wird im Standard **ECMA-262** (ECMAScript) festgehalten.
- …
- **2010**: JavaScript 1.8.5 wird im Standard **ECMAScript 5** Compliance festgehalten ➔ Basis dieser Einführung
- …

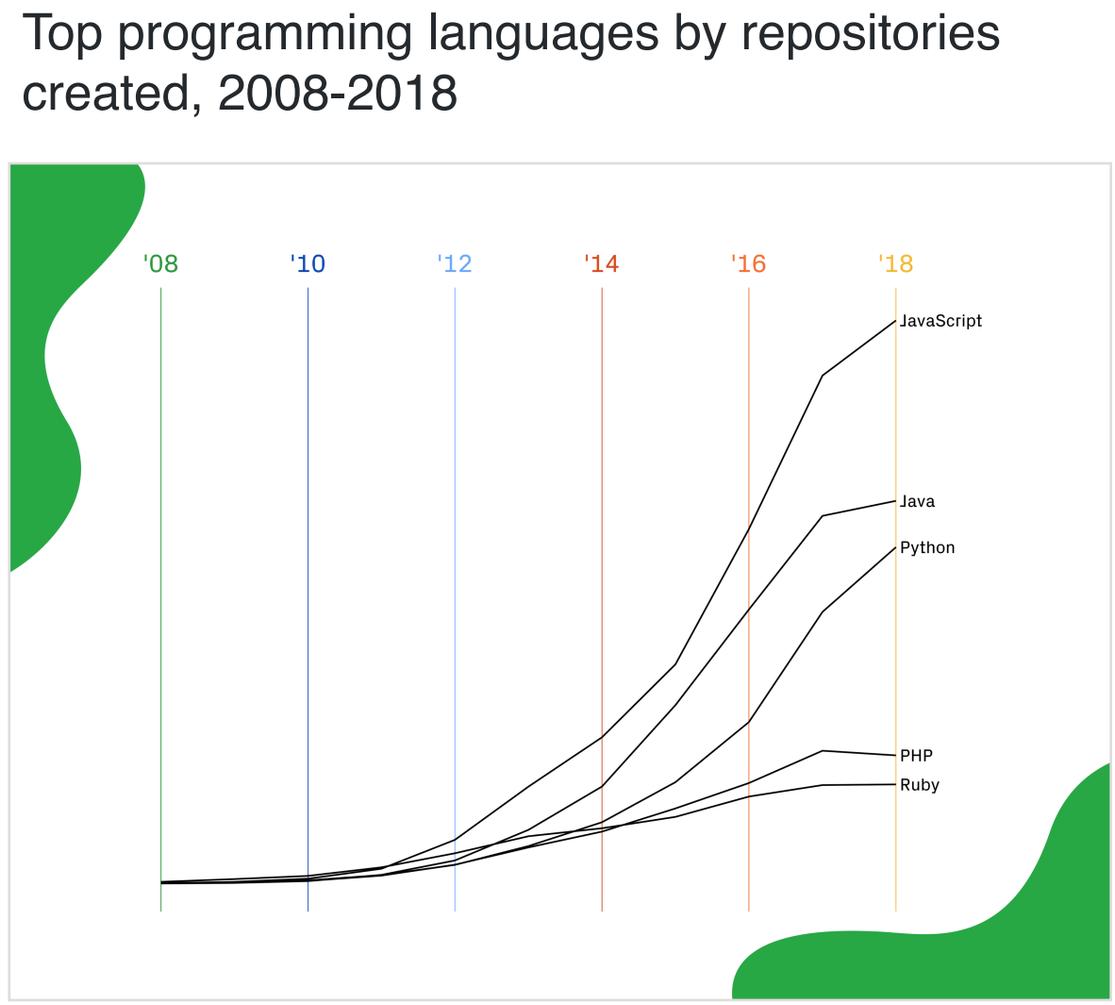
Die Syntax von Java ähnelt der von JavaScript, die Semantik ist teilweise aber deutlich anders.

Der Ruf von JavaScript

- Bis vor kurzem eher eine "Spielsprache"
- Heute: eine sehr wichtige Sprache moderner Webarchitekturen
- Tooling hat sich stark verbessert (Debugger, Test-Frameworks, etc.)
- JavaScript-Laufzeit-Engines sind effizient (V8)
- Hardware stark genug für JavaScript-basierte Spiele
- Auch für mobile Geräte (Cordova) und Desktop (Electron)

Was heißt "sehr wichtig"?





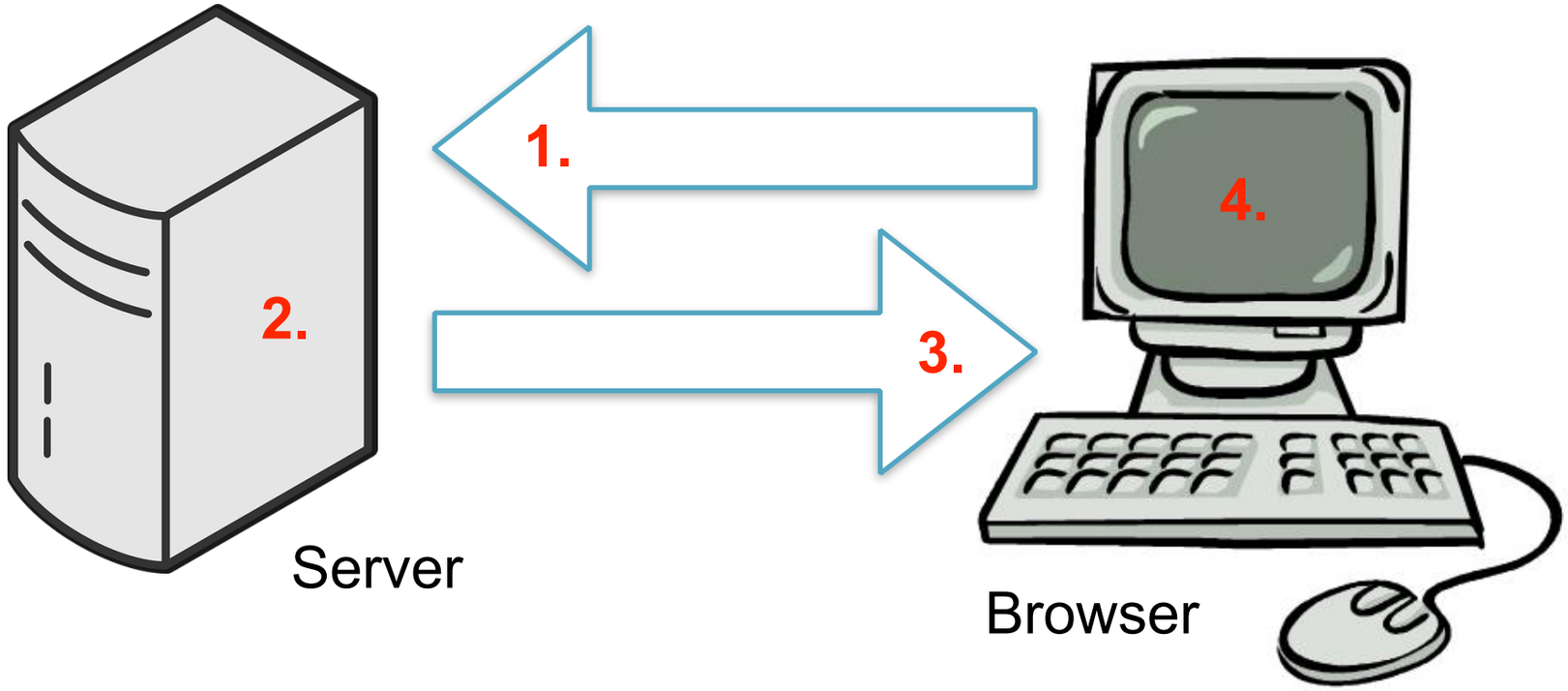
Siehe Diskussion bei **Redmonk**

Siehe **GitHub Blog**

Scripting Uberblick

Anfordern und Erzeugen einer Webseite

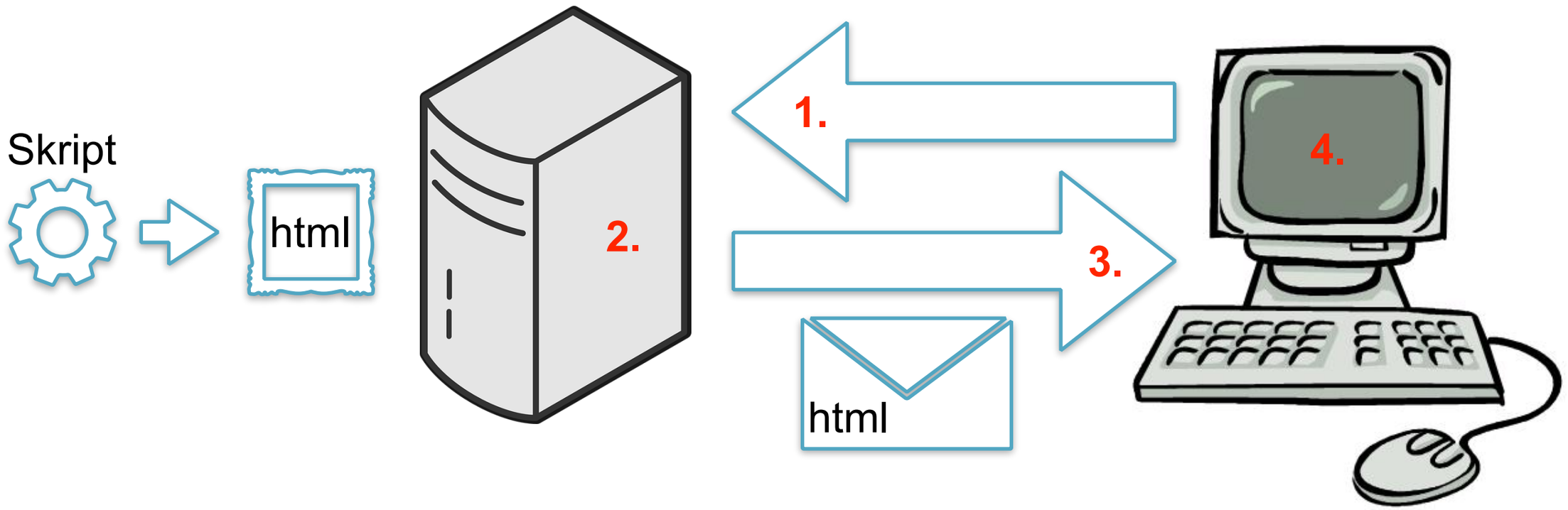
1. Browser sendet GET Request für Webseite an Webserver
2. Webserver führt **serverseitige Skripte** aus (PHP, node.js, etc.)
3. Web-Server sendet (generierte) HTML-/CSS-/Skript-Dateien an Browser
4. Browser führt **clientseitige Skripte** (JavaScript) aus und *rendert* Seite



JavaScript macht Web-Apps **interaktiv** und **reaktiv** auf Benutzeraktionen

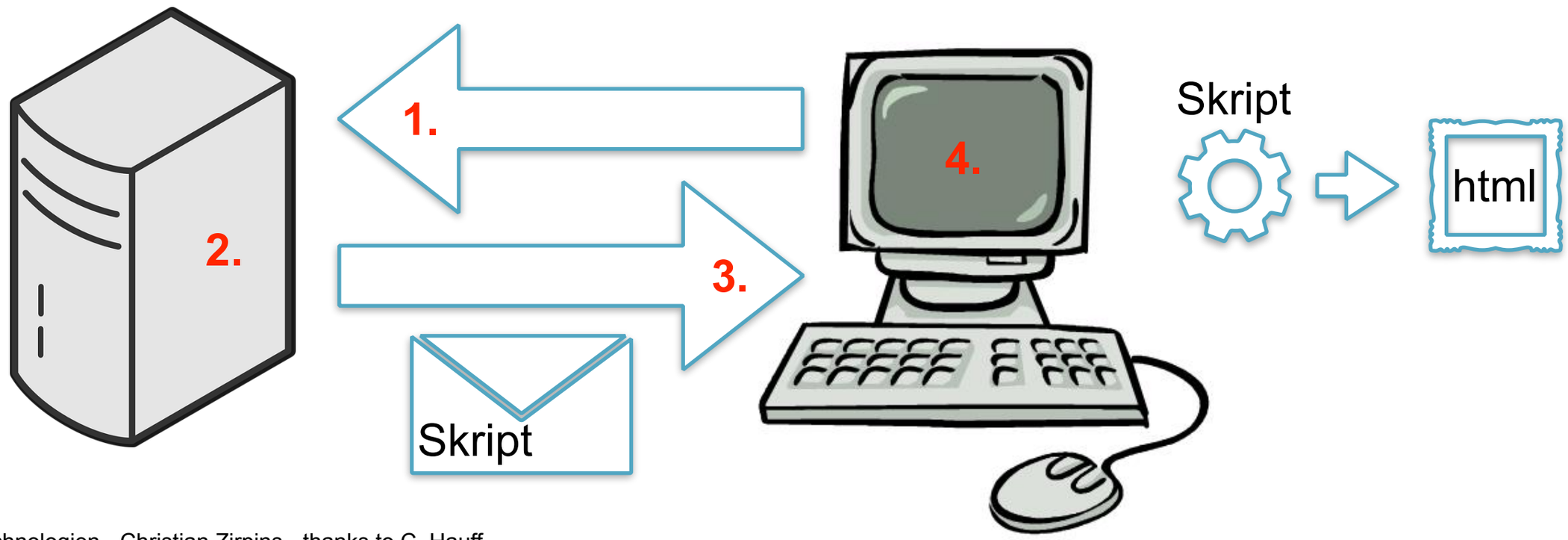
Serverseitiges Scripting

- Quellcode ist privat, nur das **Ergebnis** der Skriptausführung wird zurückgegeben (in HTML), nicht das Skript selbst
- Serverseitige Skripte können auf zusätzliche Ressourcen (einschließlich Datenbanken) zugreifen
- Serverseitige Skripte können nicht-standardmäßige Sprachfunktionen verwenden (Software des Servers ist bekannt)



Clientseitiges Scripting

- Quellcode ist für alle sichtbar ("View Page Source")
- Skriptausführung auf Browser reduziert Last des Webservers
- Alle nötigen Rohdaten (z.B. für Visualisierungen) müssen vom Client heruntergeladen und verarbeitet werden
- JavaScript ist ereignisgesteuert: Codebausteine werden oft als Reaktion auf Benutzeraktionen ausgeführt (Klick, "Hover", Verschieben usw.)



JavaScript in der Webentwicklung

- JavaScript ist die vorherrschende Sprache für Webbrowser. Deshalb erübrigt sich praktisch die Frage nach einer Technologie für den **clientseitigen Teil** einer Webanwendung.
- Für den **serverseitigen Teil** einer Webanwendung kann man jede beliebige Technologie wählen, die die grundlegenden Protokolle beherrscht (z.B. HTTP). JavaScript ist hier eine moderne Alternative.
- Meist ist die Entwicklung von Webanwendungen stark durch Auswahl und Einsatz von **Web Frameworks** geprägt.
  - Frameworks erleichtern die Entwicklung durch Vorgabe von Mustern, Programmiermodellen und Infrastruktur für generische Aufgaben.
  - Für JavaScript gibt es diverse solche Frameworks auf Clientseite (z.B. jQuery, Angular etc.) und Serverseite (z.B. Node.js und Express).

JavaScript-Grundlagen

LWAD Kapitel 4

- Wie man JavaScript in einer Webanwendung integrieren kann
- Wichtige JavaScript-integrierte Typen
- Wie benutzt man JavaScript-Kontrollstrukturen (`if`, `for`, `while`)
- Wie deklariert man Variablen und Funktionen
- Zweck von `console.log()`
- Arbeiten mit Arrays
- Wie man grundlegende jQuery-Funktionen verwendet

… wir wiederholen und vertiefen zunächst diese **Grundlagen** und bauen dann **objektorientierte Entwurfsmuster** darauf auf…

Typische Merkmale von Skriptsprachen

- Mit Interpreter ausgeführt, keine Compilierung erforderlich
- Implizit deklarierte Variablen, schwache Typisierung: Variablentypen werden zur Laufzeit erkannt ➔ Ducktyping
- Dynamische Programmänderung: Hinzufügen von Klassen, Methoden und Attributen zur Laufzeit
- Automatische Speicherverwaltung wie in Java und C#
- Oft weniger „geschwätzig“ ➔ höherer Abstraktionsgrad, domänenspezifische Sprachen (DSLs)

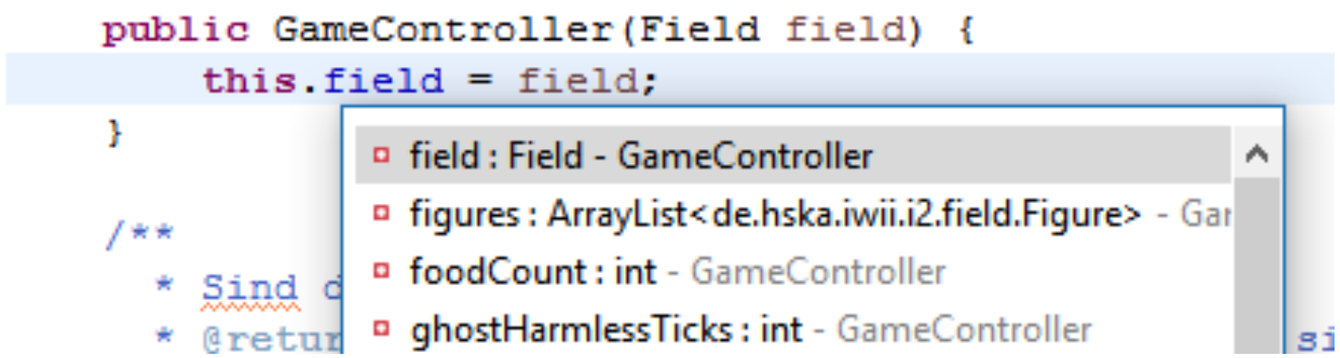
Skriptsprachen: Pro und Contra

Vorteile von Skript-Sprachen:

- schnelle Entwicklung (Rapid Prototyping)
- kein Compiler-Schritt erforderlich
- Codeaustausch zur Laufzeit
- Domänenspezifische Sprachen (DSLs)

Nachteile von Skript-Sprachen (z.T. schon behoben, bzw. Vorurteile)

- Typfehler erst zur Laufzeit sichtbar → viel mehr testen
- schlechte IDE-Unterstützung (Typen fehlen), Vorschläge wie hier bei Java sind nicht so genau möglich:
- Geringere Geschwindigkeit
- Weniger "Best-Practices"



JavaScript Grundlagen

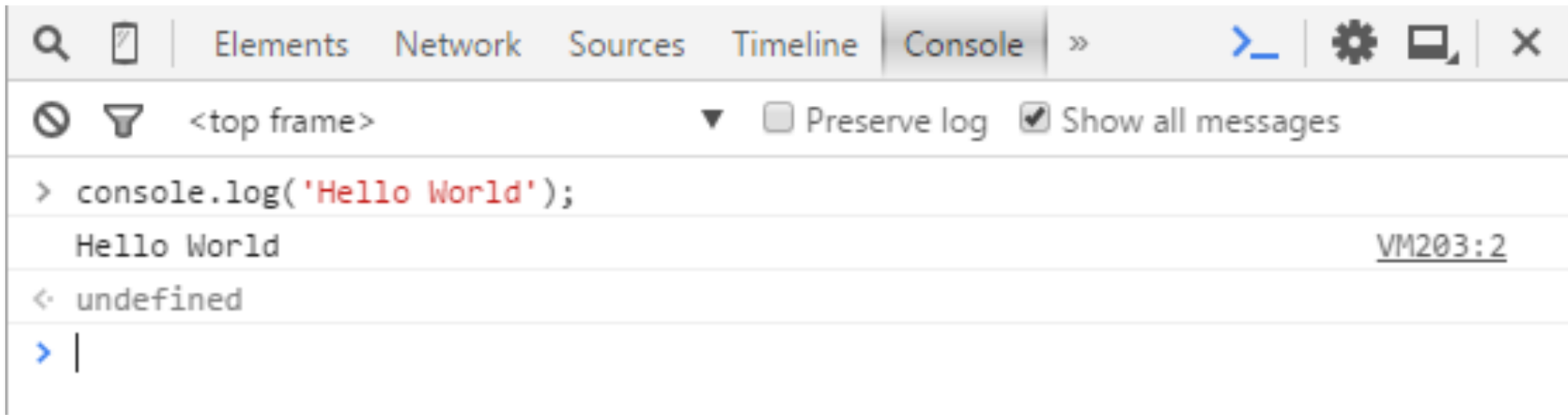
Das erste JavaScript Programm

JavaScript in der Browser Konsole

- Als Laufzeitumgebung für die folgenden kleinen Beispiele eignet sich die Konsole des Chrome- oder des Firefox-Browsers:
  - **Chrome**: Tools ➔ JavaScript-Konsole
  - **Firefox**: Entwickler-Werkzeuge ➔ JavaScript-Umgebung zur Eingabe und Entwickler ➔ Browser-Konsole zur Ausgabe
- Eingabe in die Konsole (wie `System.out.println` in Java):

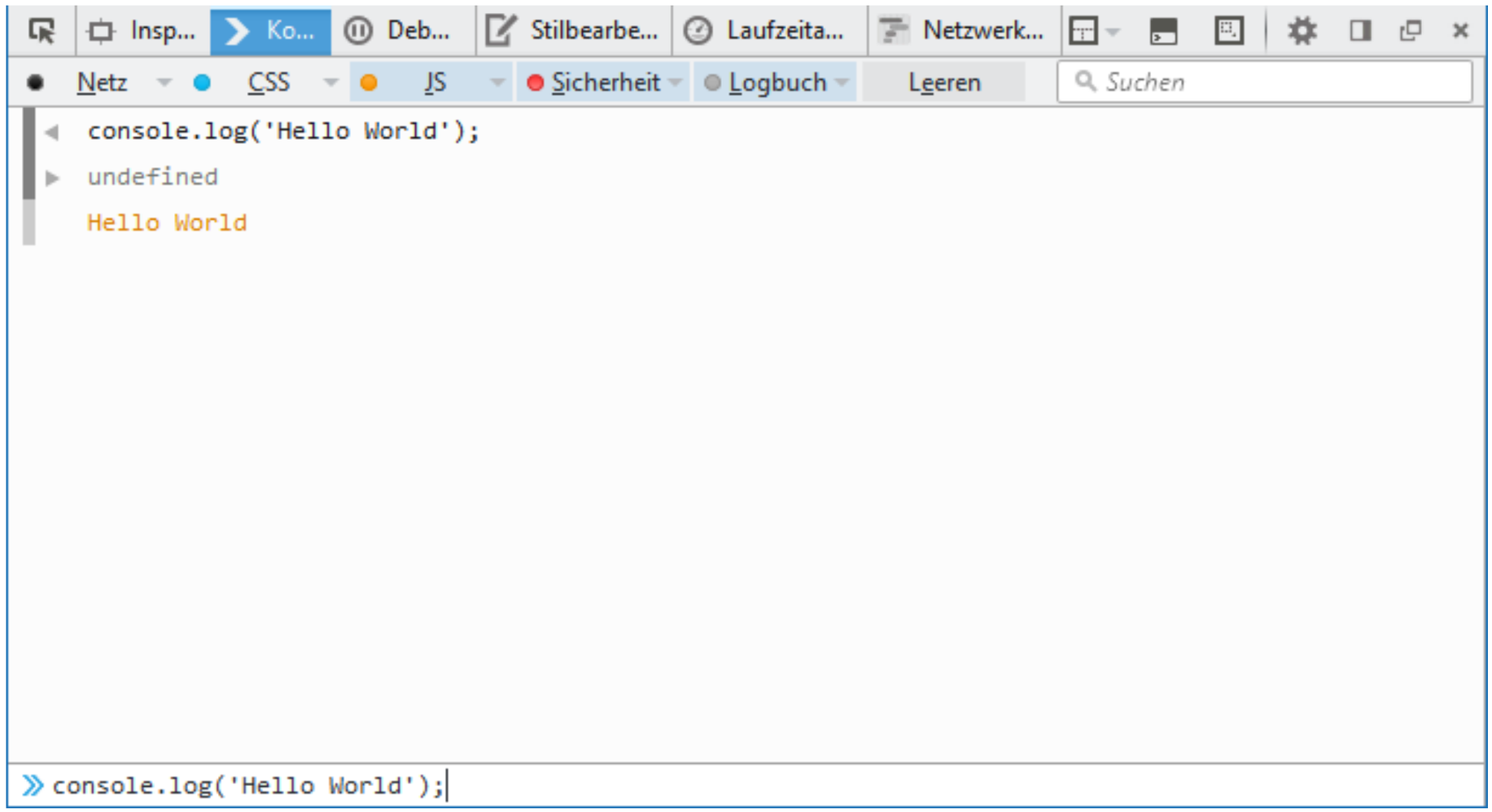
```
console.log('Hello World');
```

- Ein- und Ausgabe auf Chrome:



JavaScript in der Browser Konsole (Firefox)

- Ein- und Ausgabe auf Firefox:



- Es sind einfache und doppelte Anführungszeichen erlaubt.
- **console** ist ein im Browser vorhandenes JavaScript-Objekt.

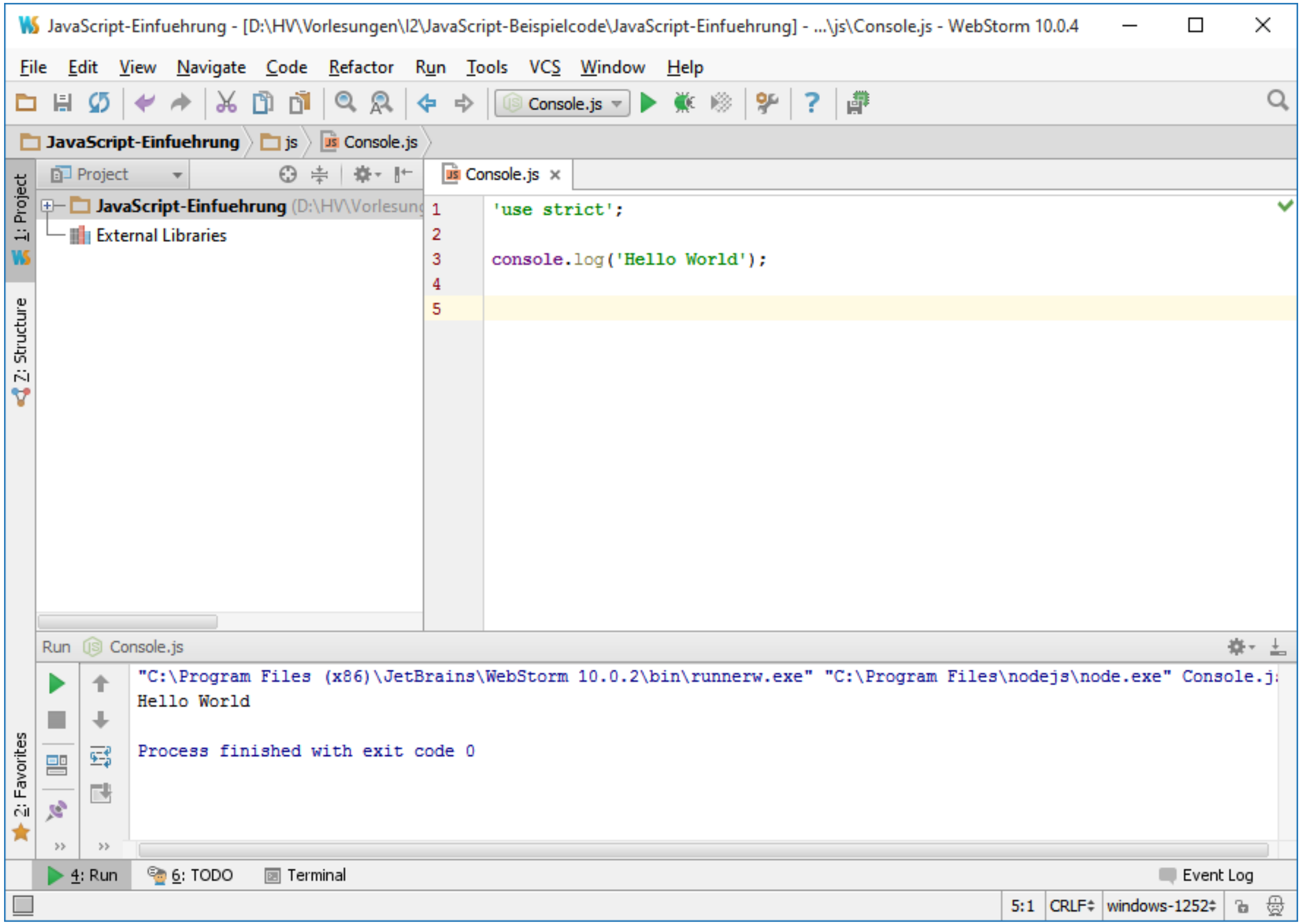
JavaScript in der IDE

- Ab jetzt: Beispiele in der **WebStorm-IDE**:
  - Node.js von http://nodejs.org/#download herunterladen und installieren
  - WebStorm von http://www.jetbrains.com/webstorm/download/ herunterladen (Lizenz für Studierende anfordern), installieren
  - WebStorm starten und in File ➔ Settings ➔ JavaScript ➔ Browser den Pfad auf **node.exe** eintragen (falls nicht automatisch gefunden, je nach Betriebssystem hat die Datei eine andere oder keine Endung).
- Neues, leeres Projekt anlegen
- Zum Projekt eine JavaScript-Datei `Console.js` hinzufügen und

```
console.log('Hello World');
```

eintragen.

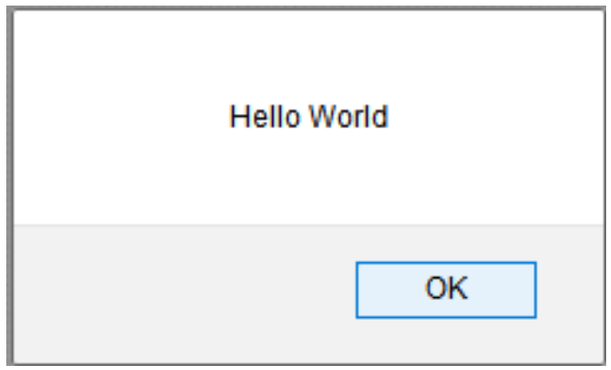
JavaScript in der IDE (WebStorm)



JavaScript in einer Webseite

- 1. Möglichkeit: Einbetten von JavaScript in eine HTML-Datei

```
<!DOCTYPE html>
<html>
<head lang="de">
  <meta charset="UTF-8">
  <title>Hello World</title>
  <script>
    alert('Hello World');
  </script>
</head>
<body>
<h1>Titelzeile</h1>
</body>
</html>
```



- `alert()` ist eine Browser-Funktion, die einen Dialog öffnet.
  - Der Dialog wird beim Laden der Seite und vor dem Anzeigen des Inhalts eingeblendet.

JavaScript in einer Webseite

- 2. Möglichkeit: Laden einer externen JavaScript-Datei

```
<!DOCTYPE html>
<html>
<head lang="de">
  <meta charset="UTF-8">
  <title>Hello World</title>
  <script type="text/javascript"
    src="./js/External.js">
  </script>
</head>
<body>
  <h1>Hello World</h1>
  <script>
    showMessage('Hello World')
  </script>
</body>
</html>
```

```
function showMessage(message) {
  alert(message);
}
```

Laden der Datei

Aufruf der Funktion

- Erstes `script`-Tag lädt die im `src`-Attribut gegebene JavaScript-Datei.
- Das zweite `script`-Tag enthält den Aufruf der geladenen Funktion. Der Aufruf erfolgt, wenn dieser Teil der HTML-Seite geladen ist.

JavaScript Grundlagen

Datentypen

JavaScript-integrierte Datentypen

- Es gibt in JavaScript nur wenige Datentypen:
  - `object`: Objekte, vergleichbar `Maps` in Java. Es können Methoden und Attribute hinzugefügt werden. Auch Arrays sind Objekte.
  - `string`: Zeichenketten, etwa wie `String`-Klasse in Java.
  - `number`: Zahlen
  - `boolean`: Wahrheitswerte
  - `null`, `undefined`, `symbol`: Spezialwerte
- Eine Variable wird ohne Typ angelegt: `var year = 2000;`
- Typ eines **Variablenwertes** zur Laufzeit auslesen mit `typeof`:

```
var year = 2000;
console.log(typeof year); // Ausgabe: number
year = 'HsKA';
console.log(typeof year); // Ausgabe: string
```

Datentypen: `object`

- Objekte sind eigentlich nur Maps:
  - Attribute bestehen aus Schlüssel (Attributnamen) und (Attribut-)Wert
  - Attributwerte können von beliebigem Typ sein ➔ auch Funktionen!
  - Attribute lassen sich dynamisch hinzufügen und mit `delete` entfernen.
  - Sie würden in Java einer `Map<String, Object>` entsprechen.
- Deklaration eines leeren Objektes als Literal:

```
var obj1 = {};
```

- `obj1` ist eine Variable mit einer Referenz auf das Objekt.
- Ein weiteres Objektliteral mit Attributen (auch „Members“ genannt):

```
var obj2 = {
  lastname: 'Zirpins',
  firstname: 'Christian'
};
```

Datentypen: `object`

- Zugriff auf die Attribute mit „.“ oder „[]“
  - Unter Angabe des Attributnamens wird dessen Wert zurückgegeben.

```
console.log(obj2.lastname);
console.log(obj2['lastname']);
```

Datentypen: `object`

- Attribute können jederzeit überschrieben werden:

```
obj2.lastname = 'Vogelsang';
```

- Es dürfen neue Attribute hinzugefügt werden:

```
obj2.age = 42;
obj2.role = function() { return 'Dozent'; };
```

- Da Funktionen Objekte sind, dürfen sie einem Attribut als Wert zugewiesen werden.
- Entfernen eines Attributs aus einem Objekt:

```
delete obj2.age;
console.log(obj2.age); // Ausgabe: undefined
```

- Typ eines Objekts:

```
console.log(typeof obj2); // Ausgabe: object
```

Datentypen: `object` (Array)

- Erzeugen eines Arrays:

```
var values = [1,2,3,4];
```

- Typ eines Arrays:

```
console.log(typeof values); // Ausgabe: object
```

- Zugriff auf Array-Elemente:

```
console.log(values[0]); // Ausgabe: 1
console.log(values); // Ausgabe: [1,2,3,4]
```

- Ändern der Array-Elemente, es müssen nicht alle Daten im Array denselben Typ haben:

```
values[ 2 ] = 'Name';
console.log(values); // Ausgabe: [1,2,'Name',4]
```

Datentypen: `object` (Array)

- Arrays sind im Gegensatz zu Java in ihrer Größe veränderlich.
- Anhängen neuer Werte an das Ende des Arrays mit **push**:

```
values.push(42, 66);
console.log(values); //Ausgabe [1,2,'Name',4,42,66]
```

Datentypen: `object` (Array)

- Entfernen von Werten aus dem Array mit `splice`:

```
values.splice(1, 2); // an Pos. 1 werden 2 Elemente entfernt
console.log(values); // Ausgabe [ 1, 4, 42, 66 ]
```

- Mit `splice` lassen sich auch Werte einfügen (siehe Literatur).
- Länge eines Arrays:

```
var values = [1, 2, 3, 4];
console.log(values.length); // Ausgabe: 4
```

- Die Länge kann vergrößert oder verkleinert werden:

```
values.length = 6;
console.log(values); // Ausgabe: [ 1, 2, 3, 4, ,  ]
console.log(values[ 5 ]); // Ausgabe: undefined
values.length = 3;
console.log(values); // Ausgabe: [ 1, 2, 3 ]
```

Datentypen: `object` (Array)

- Es darf auch über die Array-Grenzen hinaus geschrieben werden. Dabei wird das Array automatisch vergrößert:

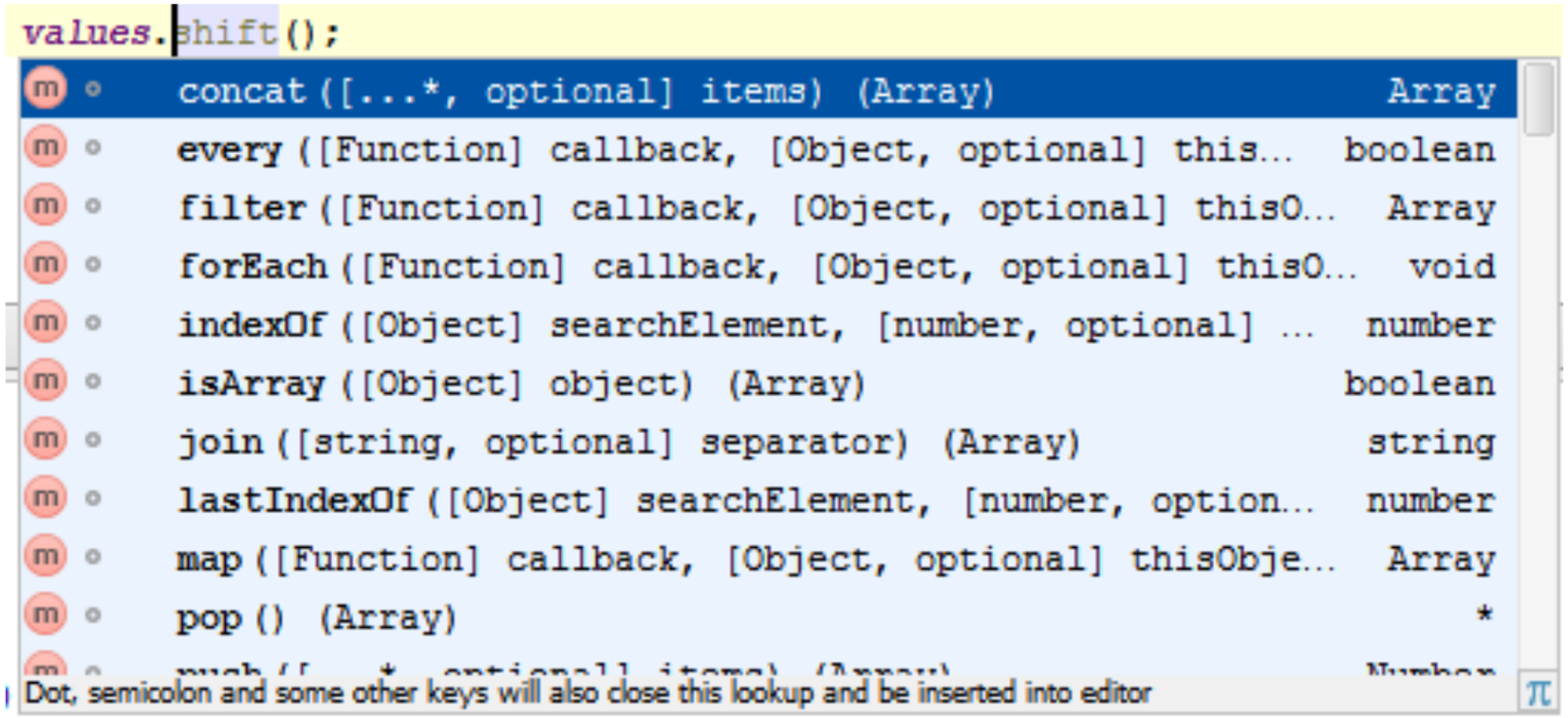
```
values = [1, 2, 3, 4];
values[ 8 ] = 8;
console.log(values); // Ausgabe: [1,2,3,4,,,,,8]
```

Datentypen: `object` (Array)

- Es gibt weitere Methoden, z.B. `shift`
  (verschiebt alle Elemente um eines nach links):

```
values = [1, 2, 3, 4];
values.shift();
console.log(values); // Ausgabe: [2, 3, 4]
```

- Die JavaScript-Referenz sowie die Code-Vervollständig in z.B. WebStorm zeigen viele weitere Methoden auf Arrays.



Datentypen: `string`

- Anführungszeichen:
  - `var name = 'Vogelsang';`
  - `var name = "Vogelsang";`
- Der Typ eines Strings ist `string`:

```
console.log(typeof name); // Ausgabe: string
```

- Jedes Zeichen eines Strings ist wiederum ein String:

```
var first = name[ 0 ];
console.log(typeof first); // Ausgabe: string
```

- Es gibt also kein einzelnes `char` wie in Java.

Datentypen: `string`

- Strings lassen sich mit += und + verbinden
  - Kann bei sehr alten Browsern (z.B. IE7) langsam sein, weil viele Zwischenobjekte erzeugt werden.
- Alternative, wenn die Strings im Array vorliegen:

```
var strings = ['a', 'b', 'c', 'd' ];

// Alle verbinden, Trennzeichen ist ,-'
var result = strings.join('-');

console.log(result); // Ausgabe: a-b-c-d
```

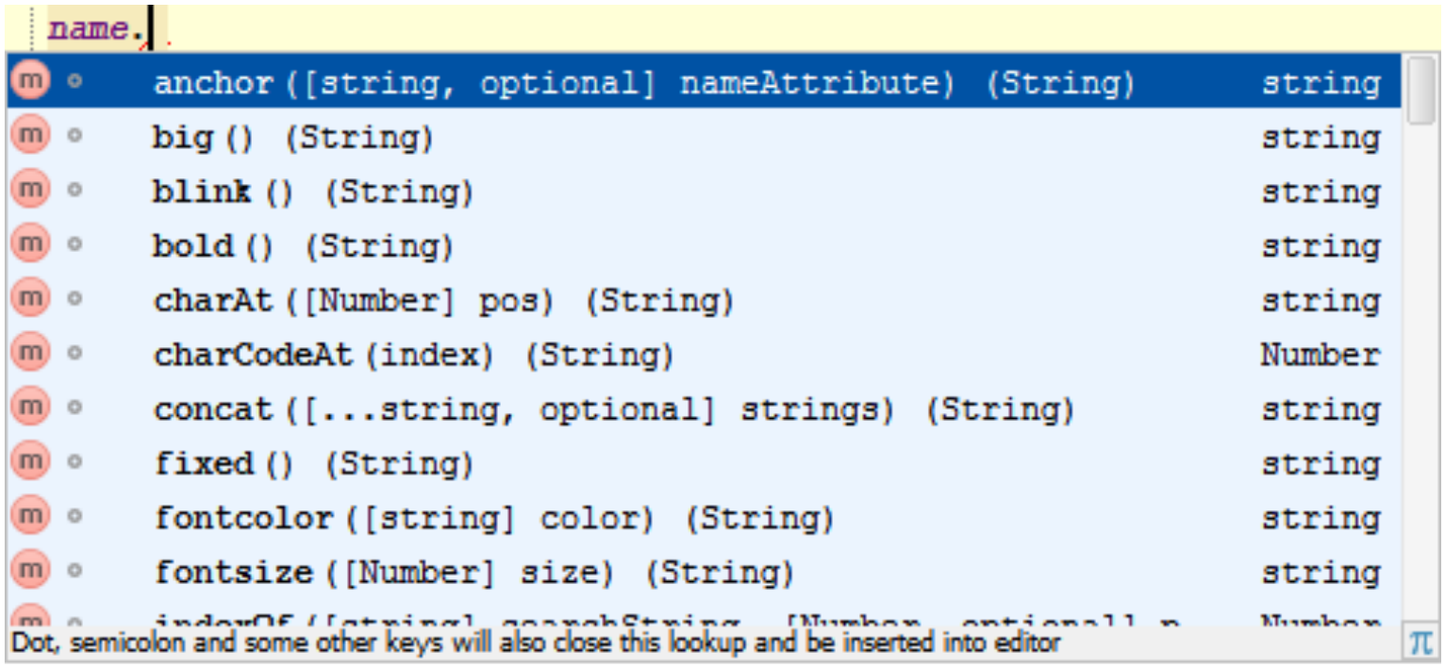
- Bei modernen Browsern ist beides schnell.

Datentypen: `object` (string)

- Zum Typ String (u.a.) existieren Wrapper Objekte, die mit Konstruktor oder automatischer Typumwandlung entstehen.

```
var school = new String("HsKA");
var faculty = "IWI";
console.log(typeof school);  // Ausgabe: object
console.log(typeof faculty); // Ausgabe: string
console.log(school.length);  // Ausgabe: 4
console.log(faculty.length); // Ausgabe: 3
```

- Die JavaScript-Referenz (oder Code-Completion in WebStorm) zeigt viele weitere Methoden auf String Wrappern.



Datentypen: `number`

- Es gibt in JavaScript mit `number` nur einen Zahlentyp.
  - Spezialfall: `BigInt` für Zahlen > 2^{53}-1
- Er entspricht im Wesentlichen `double` in Java.

```
var age = 22;
console.log(age);                  // Ausgabe: 22
console.log(typeof age);           // Ausgabe: number

var price = 42.33;
console.log(price);                // Ausgabe: 42.33
console.log(typeof price);         // Ausgabe: number

console.log(age * price);          // Ausgabe: 931.26
```

Datentypen: `number`

- Zwischen `string` und `number` kann manuell umgewandelt werden:

```
age = parseInt("42.33");
console.log(age);                // Ausgabe: 42
console.log(typeof age);         // Ausgabe: number

price = parseFloat("42.33");
console.log(price);              // Ausgabe: 42.33
console.log(typeof price);       // Ausgabe: number
```

Datentypen: `object` (number)

- Eine Fließkommazahl kann gerundet werden, dabei darf die Anzahl der Nachkommastellen angegeben werden:

```
price = 22.66;
price = price.toFixed(); // Rundung in ganze Zahl
console.log(price);      // Ausgabe: 23
```

```
price = 22.66;
price = price.toFixed(1);
 // Rundung auf eine Nachkommastelle
console.log(price);      // Ausgabe: 22.7
```

- Zahlen haben also auch (Wrapper) Objekte!
- Was passiert beim Rechnen, wenn Fehler auftreten?
  - Es gab bei der Einführung von JavaScript noch keine Exceptions.
  - Es werden Fehlerinformationen in der Variablen gespeichert.

Datentypen: `number`

- Beispiele:

```
var result = price / 0;
console.log(result);              // Ausgabe: Infinity
console.log(typeof result);       // Ausgabe: number
result *= 10;
console.log(result);              // Ausgabe: Infinity

result = 0 / 0;
console.log(result);              // NaN (Not a Number)
console.log(typeof result);       // Ausgabe: number
result /= (0 / 0);
console.log(result);              // Ausgabe: NaN
```

`Math` Operatoren

- Weitere Mathe-Operationen sind über das `Math`-Objekt verfügbar.

```
Math.
```

```
abs([number] x) (Math)                                  number
acos([number] x) (Math)                                 number
asin([number] x) (Math)                                 number
atan([number] x) (Math)                                 number
atan2([number] x, [number] y) (Math)                    number
ceil([number] x) (Math)                                 number
cos([number] x) (Math)                                  number
E Math (EcmaScript.js)                                  number
```

Strg+Unten and Strg+Oben will move caret down and up in the editor >>

Datentypen: `boolean`

- Für Wahrheitswerte gibt es den Datentyp **`boolean`**:

```
var dead = false;
console.log(dead);          // Ausgabe: false
console.log(typeof dead); // Ausgabe: boolean
```

JavaScript Grundlagen

Variablen

Typisierung von Variablen

- Erinnerung:
  - Variablen werden keine festen Datentypen zugewiesen.
  - Variablenwerte haben zur Laufzeit Typen, die sich ändern können.
- Wie kann man trotzdem halbwegs sicher programmieren?
- **Lösung**: Typ-Hinweis im Kommentar, wird von IDE ausgewertet
- Beispiel in WebStorm:

```
/**
 * @type {string}
 */
var name = 'Vogelsang';

name = 42;
```

Warnung

- Einblendung unter dem Mauszeiger:
  - Aber das ist trotzdem erlaubt und funktioniert!

```
13    name = 42;
```

Assigned expression type number is not assignable to type string

Sichtbarkeit (Scope) von Variablen

- In Java lebt eine Variable nur innerhalb des *zugehörigen Blocks* („zwischen den geschweiften Klammern“).
- In JavaScript begrenzen nur Funktionen den Scope von Variablen.

```
for (var x = 0; x < 10; x++) {
  console.log(x);
}
console.log("Nach der Schleife: " + x);
```

- `x` ist nach Beendigung der Schleife noch sichtbar und hat Wert 10.

```
function calc() {
  var x = 0;
  console.log(x);
}
```

- `x` ist außerhalb der Funktion nicht sichtbar (`var` beachten!).

Vergleiche zwischen Variablen

- Es gibt zwei Vergleichsoperatoren
- ==, != testen auf Gleichheit bzw. Ungleichheit, notfalls mit Hilfe von Konvertierungen! („equals operator“)
- ===, !==: (Un)Gleichheit ohne Konvertierung („strict equals operator“)
  - Entspricht eher dem == bzw. != in Java und ist bevorzugt zu verwenden.
- Objekte/Arrays werden über ihre Referenzen, nicht Inhalt, vergleichen.

`1 == true`	`true`
`1 == "1"`	`true`

`var obj1 = {};`
`var obj2 = {};`
`obj1 == obj2`
	`false`

`1 === true`	`false`
`1 === "1"`	`false`

`var obj1 = {};`
`var obj2 = {};`
`obj1 === obj2;`
	`false`

Initialisierung von Variablen

- In Java stellt der Compiler sicher, dass es keine Zugriffe auf nicht initialisierte Variablen gibt und dass alle Variablen, auf die zugegriffen wird, auch existieren.
- In JavaScript können Variablen beim Zugriff uninitialisiert sein.
- Beispiel:

```
var name;
console.log(name);  // Ausgabe: undefined
console.log(typeof name); // Ausgabe: undefined
```

- Beispiel für den Test auf Initialisierung:

```
var name1;
console.log(typeof name1 === 'undefined');
// Ausgabe:true
```

Initialisierung von Variablen

- Wie in Java dient `null` der gezielten Initialisierung mit „nichts“.
- Beispiel:

```
var name2 = null;
console.log(typeof name2 === 'undefined');
// Ausgabe: false
console.log(typeof name2);
// Ausgabe: object (null ist vom Typ object)
```

- Zugriff auf eine nicht vorhandene Variable:

```
console.log(name3); // Programmabbruch
```

Initialisierung von Variablen

- Zugriff auf nicht initialisierte bzw. vorhandene Objekt Member:

```
var object = {
   attr1: undefined
};

console.log(object.attr1); // Ausgabe: undefined
console.log(object.attr2); // Ausgabe: undefined
```

- Tipp: Initialisiere alle Variablen und Member bei Deklaration.
  - Es gibt keinen Compiler, der die Initialisierung sicherstellt.

JavaScript Grundlagen

Kontrollstrukturen

Kontrollstrukturen

Anweisung	Java	JavaScript
`if`	Bedingung muss ein Boole'scher Wert sein	Bedingung ist `true` für alles außer `false`, `undefined`, `null`, `0`, `NaN`, `""`
`switch`	erlaubte Typen: ganzzahlige Werte, String, Aufzählungen	wie in Java, Vergleich mit `===`
`? :`		wie in Java
`while`, `do … while`, `break, continue`		wie in Java
`for`		ähnlich zu Java, Laufvariable mit `var` ist auch außerhalb der Schleife sichtbar

Schleifen mit `for … in`

- Besondere Form der `for`-Schleife
- Iteriert über alle Member eines Arrays oder Objekts.

```
// Durchlaufen aller Attribute eines Objektes
var object = {
  attr1: 1,
  attr2: 2
};
for (var p1 in object) {
  console.log('Attribut ' + p1 + ' = ' + object[p1]);
}
```

```
// Durchlaufen aller Elemente eines Arrays
var values = [1, 2, 3, 4];
for (var p2 in values) {
  console.log('Index ' + p2 + ' = ' + values[p2]);
}
```

Ausnahmebehandlung

- Ausnahmen sehen denen in Java ähnlich (aber ohne Klassen)
- Eine Ausnahme ist ein Objekt. Beispiel:

```
try {
   throw 'Eingabefehler';
   // Oder besser: throw new Error('Eingabefehler');
}
catch (error) {
   console.log('Fehler: ' + error);
}
finally {
   console.log('Wie in Java immer aufgerufen');
}
```

JavaScript Grundlagen

Funktionen

Funktionen sind Objekte

- Funktionen sind Objekte, die Variablen zugewiesen werden können.
- Funktion kann Funktions- und Parameternamen besitzen.

```
// Definition der Funktion print.
function print() {
  console.log('Aufgerufen 1');
}
// Aufruf der Funktion print:
print();  // Ausgabe: Aufgerufen 1
```

```
// Variablen printer nimmt anonyme Funktion auf.
var printer = function() {
  console.log('Aufgerufen 2');
};
// Aufruf der anonymen Funktion mit der Variablen
printer();  // Ausgabe: Aufgerufen 2
```

Funktionsobjekte und -parameter

- Auf Funktions-Objekten kann die Methode `call` aufgerufen werden.

```
// Definition der Funktion print.
var printer = function() {
  console.log('Aufgerufen 2');
}
// Direkter Aufruf der Funktion:
printer();
// Aufruf über die call-Methode:
printer.call();
```

- Funktionsparameter / Rückgabewerte sind im Quellcode „typlos“:

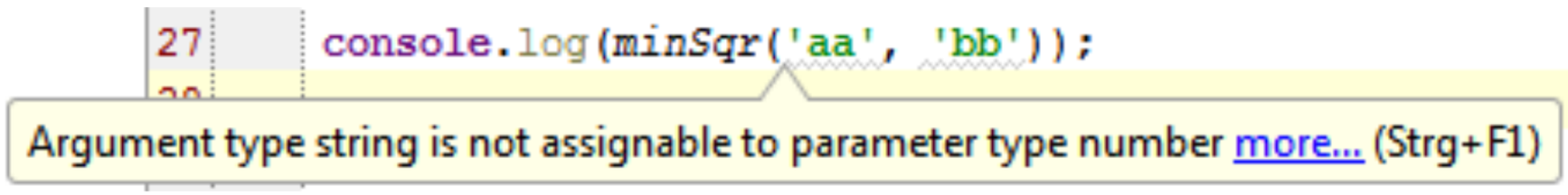
```
function minSqr(x, y) {
  return x < y ? x * x : y * y;
}
console.log(minSqr(2,3));
```

Deklaration von Parametertypen

- Typen können im JsDoc-Kommentar in geschweiften Klammern angegeben werden. IDEs können sie erkennen und prüfen.

```
/**
   * Berechnet Quadratwert des Minimums zweier Zahlen.
   * @param {number} x Erste zu vergleichende Zahl.
   * @param {number} y Zweite zu vergleichende Zahl.
   * @returns {number} Das Minimum beider Argumente.
   */
function minSqr(x, y) {
  return x < y ? x * x : y * y;
}
console.log(minSqr(2,3));
console.log(minSqr('aa', 'bb'));
```

- Optionale Parameter und variable Parameteranzahl werden hier nicht betrachtet.



Warnung von WebStorm

Funktionen höherer Ordnung

- Funktionen als Parameter an andere Funktionen übergeben
- Beispiel: Durchlaufen eines Arrays mit einer `forEach`-Schleife:

```
var values = [1, 2, 3, 4, 5];

values.forEach(function (value) {
  console.log(value);
});
```

- `forEach` erwartet Funktion als Parameter. Alternative Schreibweise:

```
var logger = function(value) {
  console.log(value);
};

values.forEach(logger);
```

- Bereits von der Stream-API in Java bekannt!

Funktionen höherer Ordnung verketten

- Wozu das Ganze? Es geht doch auch prozedural mit Index-Zugriff.
- Vorteil: Längere „Verkettung“ von Aufrufen (wie Java Stream-API)
- Beispiel: Ausgabe gerader Werte im Array `values`:

```
values.filter(function(value) {
  return value % 2 == 0;
}).forEach(function(value) {
  console.log(value);
});
```

- Die Funktion im `filter`-Argument liefert `true`, wenn der Wert im Ergebnis vorhanden sein soll.
- Die Funktion im `forEach`-Argument wird für jeden Wert, den der Filter durchgelassen hat, aufgerufen.

JavaScript Entwurfsmuster

Objektorientierung (OO) für JavaScript

- **OO** gruppiert Daten und Verhalten in Objekten einer Klasse, die einfach instanziiert und vererbt werden können.
- Kleine Programme brauchen sie oft nicht, große Projekte profitieren
- JavaScript hat Funktionen als "First Class Citizens" (nicht Klassen wie in Java)
- Eingebaute Objekte: *Strings*, *Arrays etc.* (kennen wir schon) aber auch z.B. *HTML/DOM-Knoten*
- Objekte können auf unterschiedliche Arten erstellt werden (am besten man beschränkt sich auf Eine)
- Wir erfassen diese Arten der Erstellung als Entwurfsmuster

Entwurfsmuster

" **Entwurfsmuster** sind wiederverwendbare Lösungen für häufig auftretende Probleme im Software-Design " (Addy Osmani)

- Wir folgen einem sehr praktischen Ansatz
- Es gibt viele Entwurfsmuster; wir konzentrieren uns auf **drei** (die wichtigsten für unseren Anwendungsfall)
- Entwurfsmuster entwickeln sich mit der Zeit
- Entwurfsmuster gelten oft für verschiedene Programmiersprachen

Rückblick: Objekte in JavaScript

- `new Object()` erzeugt ein leeres Objekt, das Name/Wert-Paare aufnehmen kann
  - Name: beliebiger String
  - Wert: irgendetwas (String, Array, Zahl, etc.) außer `undefined`
- Member (Attribute) werden zugegriffen durch
  - `.name` (Punktnotation)
  - `[Name]` (Klammernotation)

```
var note1 = new Object();
note1["type"] = 1;
note1["note"] = "Math homework due";
console.log(note1["type"]); /* schreibt: 1 */
console.log(note1.note); /* schreibt: "Math…" */
```

Anderer Weg: Objektliterale

```
var note2 = {
    type: 2,
    message: "Math homework due"
    /* kein Komma am Ende */
};
```

Eine Methode hinzufügen

```
var note1 = new Object();
note1["type"] = 1;
note1["note"] = "Math homework due";
note1["toString"] = function() {
    /* 'this' bezieht sich auf aktuelles Objekt */
    return "Note:" + this.note + ",type:" + this.type;
};
```

```
var note2 = {
    type: 2,
    message: "Math homework due",
    toString: function() {
        /* 'this' bezieht sich auf aktuelles Objekt */
        return "Note:" + this.note + ",type:" +
this.type;
    }
};
```

Objektliterale können komplex sein

```
var paramModule = {
    /* Parameter Objekt */
    Param: {
        minType: 1,
        maxType: 5,
        maxNoteLen: 100,
    },

    getParams: function() {
        var s = "Here all params should be listed ... ";
        return s;
    }
};
```

```
/* Using it: */
/* Wie ruft man minType und getParams() auf ? */
```

Objektliterale können komplex sein

```
var paramModule = {
    /* Parameter Objekt */
  Param : {
    minType : 1,
    maxType : 5,
    maxNoteLen : 100,
  },

  getParams : function() {
                var s = "minType: "+this.Param.minType+"\n";
                s += "maxType: "+this.Param.maxType+"\n";
                s += "maxNoteLen: "+this.Param.maxNoteLen+"\n";
                return s;
  }
};
```

```
/* Using it: */
paramModule.Param.minType; /* 1 */
paramModule.getParams();
```

Ist das genug?

```
var note = {
    type: 1,
    message: "Math homework due",
    toString: function() {
        return "Note:" + this.note + ",type:" + this.type;
    }
};
```

- Was, wenn wir 1000 Objekte dieser Art brauchen?
- Was passiert in einem großen Projekt, wenn eine Methode allen `note` Objekten hinzugefügt werden muss?

1. Entwurfsmuster

Basic Constructor

Basic Constructor

```
function Note( note, type ) {
  this.note = note;/* 'this': Referenz auf aktuelles Objekt */
  this.type = type;

  this.setType = function(t) {this.type = t;};

  this.getType = function() {return this.type;};

  this.getNote = function() {return this.note;};

  this.toString = function () {
    return "Note: "+this.note+", type: "+this.type;
  };
}
```

```
var note1 = new Note("Maths homework assignment", 1);
note1.setType(2);

var note2 = new Note("English homework due"); /* was ist mit type? */

var note3 = Note("Music homework due", 3); /* was nun? */
```

Basic Constructor

- Ein **Objektkonstruktor** ist nur eine normale Funktion
- Was macht JavaScript mit `new`?
  - Ein neues anonymes leeres Objekt wird erstellt und als `this` verwendet
  - Am Ende der Funktion wird das neue Objekt **zurückgegeben**

Basic Constructor

```
/* Erinnerung: JavaScript ist dynamisch typisiert */
var note1 = new Note("Maths homework", "IMPORTANT");
note1.type; /* "IMPORTANT" */

var note2 = new Note("English homework", 1);
note2.type; /* 1 */

note2.dueDate = "1-1-2015"; /* neues Attribut spontan ergänzt */
note1.toString(); /* "Note: Maths homework, type: IMPORTANT" */
note1.toString = function(){return this.type;};
note1.toString(); /* "IMPORTANT" */

/* Es gibt auch einige zusätzliche Funktionen */
note1.hasOwnProperty("dueDate");/* false */
note1.hasOwnProperty("type");/* true */
```

- Neue Variablen und Methoden können sofort hinzugefügt werden
- Objekte haben Standardmethoden (Prototypverkettung)

Zusammenfassung: Basic Constructor

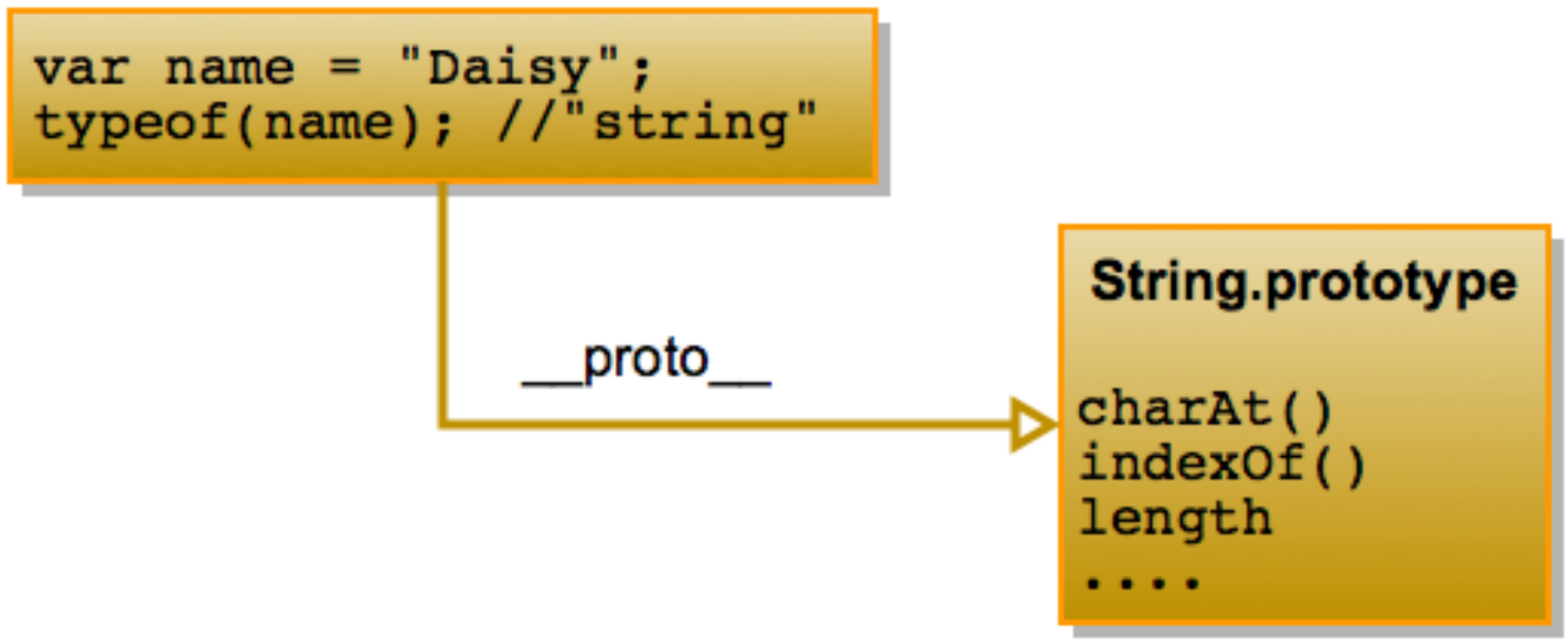
- **Vorteil**: sehr einfach zu verwenden
- **Probleme**:
  - Nicht offensichtlich wie **Vererbung** funktioniert (z.B. `NoteWithDueDate`)
  - Objekte **teilen** Funktionen nicht
    - Funktion `toString()` wird nicht zwischen `note1` und `note2` geteilt
  - Alle Mitglieder sind **öffentlich**
    - Jedes Stück Code kann die Attribute `type` und `note` abrufen/ändern/ löschen (!)

2. Entwurfsmuster

Prototype-based Constructor

Prototype Verkettung

- Jedes neue Objekt hat automatisch einen Zeiger (`__proto__`) auf ein spezielles Objekt: seinen **Prototyp.**
- Der Prototyp wird im Attribut `prototype` des Konstruktors gesetzt.
- Member des Prototyps sind auch im neuen Objekt zugänglich:
  - Ist ein Element nicht im Objekt definiert, wird es **im Prototyp gesucht.**
  - Auch Prototypen haben wieder Prototypen → dieser **Prototypkette** wird gefolgt, bis das Element gefunden wird, oder die Kette endet.



Prototype-based Constructor

```
function Note( note, type ) {
  this.note = note;/* this: Referenz auf aktuelles Objekt */
  this.type = type;
}

/* Member Methoden werden einmal im Prototyp definiert */
Note.prototype.setType = function(t) {this.type = t;};
Note.prototype.getType = function() {return this.type;};
Note.prototype.getNote = function() {return this.note;};
Note.prototype.toString = function () {
    return "Note: "+this.note+", type: "+this.type;
};

// Using it:
var note1 = new Note("Maths homework due", "IMPORTANT");
note1.getType(); /* "IMPORTANT" */
```

Prototype-based Constructor

```
function Note( note, type ) {
  this.note = note;/* this: Referenz auf aktuelles Objekt */
  this.type = type;
}

/* Member Methode setType() definiert */
Note.prototype.setType = function(t) {this.type = t;};

var note1 = new Note("Maths homework due", "IMPORTANT");
note1.setType(2); /* OK */
note1.getType(); /* TypeError: note1.getType keine Funktion*/

/* Definition der Methode können wir nachholen */
Note.prototype.getType = function() {return this.type;}

note1.getType(); /* 2 */
```

Prototypänderungen spiegeln sich auch in bestehenden Objekten wider!

Vererbung durch Prototyping

- **Basis** für die Vererbung:
  - Konstruktor Funktion mit Attributen
  - Prototype mit Funktionen

```
function Note( note, type ) {
   this.note = note;
   this.type = type;
}

Note.prototype.setType = function(t) {this.type = t;};
Note.prototype.getType = function() {return this.type;};
Note.prototype.getNote = function() {return this.note;};
Note.prototype.toString = function () {
    return "Note: "+this.note+", type: "+this.type;
};
```

- **Ziel**: Funktionen der Basis in abgeleitetem Konstruktor erben.

Vererbung durch Prototyping

- **Lösung** (eine): Objektinstanz des Basis-Konstruktors als Prototyp des abgeleiteten Konstruktors nutzen.

```
/* Konstruktor */
function NoteWithDeadline(note,type,dueDate) {
  this.note = note;
  this.type = type;
  this.dueDate = dueDate;
};

/* Umleiten des Prototype */
NoteWithDeadline.prototype = new Note();
/* Umleiten des Konstruktors */
NoteWithDeadline.prototype.constructor = NoteWithDeadline;

/* using it */
var nw =
  new NoteWithDeadline("Maths homework",1,"1-1-2015");
nw.setType(2); /* ok! setType(t) ist in Note definiert */
```

Zusammenfassung: Prototype-based Constructor

- **Vorteile**:
  - **Vererbung** ist leicht zu erreichen
  - Objekte teilen Funktionen
- **Problem**:
  - Alle Mitglieder sind **öffentlich**
    - Jedes Stück Code kann die Attribute `type` und `note` abrufen/ändern/ löschen (!)

3. Entwurfsmuster

Module

JavaScript Scoping (Sichtbarkeitsbereiche)

- Jeder JavaScript-Code kommt in denselben **Namensraum**
- JavaScript hat begrenztes **Scoping**
  - `var` in Funktion: Sichtbarkeit lokal begrenzt
  - `var` außerhalb einer Funktion: globale Sichtbarkeit
  - Kein `var`: globale Sichtbarkeit (gilt auch für Funktionsnamen)

JavaScript Scoping (Sichtbarkeitsbereiche)

```
var note1 = new Note("Maths", 1); /* global */
var note2 = new Note("English", 3); /* global */

function calcMinType(n1, n2) { /* global */
  var t1 = Number(n1.type); /* local */
  t2 = Number(n2.type);     /* global */
  return Math.min(t1,t2);
}

t1; /* ReferenceError: t1 is not defined */
t2; /* ReferenceError: t2 is not defined */

calcMinType(note1, note2); /* 1 */

t1; /* ReferenceError: t1 is not defined */
t2; /* 3 */
```

- Was ist, wenn eine andere im Projekt verwendete JavaScript-Bibliothek `note1` oder `calcMinType(a, b, c)` definiert?

Begrenzter Scope durch Module

- **Ziele**:
  - Keine globalen Variablen/Funktionen deklarieren, sofern nicht nötig
  - Private/Public Member **emulieren**
  - Member nur dann der Öffentlichkeit zeigen wenn nötig (als **API**)
- **Ergebnisse**:
  - Weniger mögliche Konflikte mit anderen JavaScript-Bibliotheken
  - **Public API** minimiert unbeabsichtigte Seiteneffekte durch falsche Verwendung der Bibliothek.

Funktionen als Basis für Module

```
var notesModule = (function () {

  /* 'private' Member */
  var noteCounter = 0;

  /* 'public' Member als Rückgabe-Objekt */
  return {
    step : 2,
    incrNoteCounter : function () {
      /*  private Member der Funktion bleiben
      auch nach dem Aufruf erhalten ('Closure') */
      noteCounter += this.step;
      return noteCounter;
    }
  };

})();/* Funktion wird direkt aufgerufen */
```

```
notesModule.incrNoteCounter(); /* 2 */
notesModule.noteCounter; /* undefined */
```

Zusammenfassung: Module

- **Vorteile**:
  - Kapselung funktioniert
  - Objekt Member sind entweder öffentlich oder privat
- **Probleme**:
  - Auf "öffentliche" und "private" Member wird unterschiedlich zugegriffen
  - Ändern von Members zwischen öffentlich/privat kostet Zeit
  - Methoden, die später hinzugefügt werden, können nicht auf "private" Member zugreifen

Literatur

- **Learning Web App Development, Kapitel 4**
- Empfehlung: Marijn Haverbeke, *"Eloquent JavaScrip"*, No Starch Press, 2018 (Online: http://eloquentjavascript.net)