# Javascript als Programmiersprache

https://iuk.one/1066-1003

Clemens H. **Cap**
ORCID: 0000-0003-3958-6136

Department of Computer Science
University of **Rostock**
Rostock,Germany
clemens.cap@uni-rostock.de

Version 2

# Übersicht

1. Multiparadigmatische Sprache
2. Imperative Sprache
3. Funktionale Sprache
4. Objekt-orientierte Sprache
5. Skript-Sprache
6. Reflexive Sprache

## 1. Multiparadigmatische Sprache

Javascript ist eine gewachsene Sprache, die Elemente vieler Paradigmen kombiniert.

# Vertretene Paradigmen

**Vertretene Paradigmen**

- **Imperativ:** Ja. Grundkonzept der Sprache.
- **Objekt-orientiert:** Ja. Prototypen-Vererbung statt Klassen-Vererbung.
- **Funktional:** Jein. Hat viele Elemente, ist aber nicht typisch funktional.
- **Scripting:** Ja.
- **Reflexiv:** Ja.

**Abgrenzung:**

- **Keine funktionale** Sprache im traditionellen Sinne (wie Haskell, SML).
- **Keine Objekt-orientierte** Sprache im traditionellen Sinne (wie C++, Java).
- **Keine logische** Sprache im traditionellen Sinne (wie Prolog).
- **Weitere Konzepte** kann man flexibel reinquetschen (Bsp: Aspect-orientation).

## 2. Imperative Sprache

Nichts wirklich aufregend Neues.

# Javascript als imperative Sprache

**Javascript hat:** Alle klassischen Konstrukte imperativer Sprachen.

- Syntax sehr nahe an Java / C.
- Variablen, einfaches Wert- und Typsystem.
- Ablaufsemantik mit Maschinenzustand und "Program Counter".
- Bedingungen (`if`, `else`, `switch`), Schleifen (`while`, `for`, `do`).
- Exceptions `throw`, `try`, `catch`, `finally`.
- Funktionen, Rekursion, Stack (eigentl. funktionale Features)

**Javascript hat nicht:** Kaum sinnvolle Abgrenzung von imperativen Sprachen möglich.

# Universeller Target

**Universeller Target**, sehr weit verbreitet (in jedem Browser).

- Bsp: **Google Web Toolkit:** Java nach Javascript kompilieren.
- Bsp: **Emscripten:** LLVM bytecode nach Javascript kompilieren.
- Bsp: C++ nach Javascript kompilieren (via LLVM).
- Bsp: TeX, OGG-Codecs uvm. von C nach JS.

Verständnis von Javascript als **Deployment Target**.

## 3. Funktionale Sprache

Javascript **ist keine** funktionale Sprache, **hat aber** viele Konstruktionskonzepte, die typisch für funktionale Sprachen sind – bzw. dort eher zu erwarten sind als in "klassischen" imperativen Sprachen.

# Einordnung

**Javascript hat:**

- Funktionen als "first class objects"
  Dh: Können als Werte gespeichert und übergeben werden
- Rekursion
- Anonyme Lambda Konstruktionen
- Dynamische Konstruktion von Funktionen aus Strings
- Variadische Funktionen
- Funktionen höherer Ordnung
- Closures
- Currying
- Konstruktionen wie Map, Reduce, Filter

# Abgrenzung

**Javascript hat aber nicht:**

- Komplexes Typsystem
  Bsp: SML; Anti-Bsp: LISP
  Aber: Typescript: Javascript-Variante mit optionalen Typ-Annotationen
- Freiheit von Seiteneffekten
- Referentielle Transparenz
- Single Assignment

# Aufgabe

**Aufgabe:**

- Erklären Sie die unteren 3 Begriffe abstrakt.
- Erklären Sie die unteren 3 Begriffe an einem Beispiel.
- Zeigen Sie an Beispielen, warum Javascript diese 3 Eigenschaft nicht hat.

**Die 3 Begriffe:**

1. Freiheit von Seiteneffekten.
2. Referentielle Transparenz.
3. Single Assignment.

# Parameter-Übergabe

**4 Varianten:**

1. Positionelle Argumente
2. Benannte Argumente.
3. `arguments` (pseudo)-Array.
4. Spread-Operator ...

```
function fct1 (a, b, c) { return a + b + c;}
function fct2 ( obj )   { return obj.a + obj.b + obj.c;}
function fct3 ()        { return arguments[0] + arguments[1] + arguments[2];}
function fct4 (x, ...y) { return x + y[0] + y[1];}

fct (2, 3, 4);  // jeweiliger Aufruf
```

**Src. 1:** Addieren von drei Zahlen in einer Funktion

# Rekursive Funktionsdeklaration

Verschiedene Mechanismen des rekursiven Rückgriffs
auf die gerade definierte Funktion.

```
function f(n) { if (n<=0)  return (1);            // Über Funktionsname
                else       return (n*f(n-1));  }
var g = function (n) { if (n<=0)  return (1);     // Über Variablenname
                       else       return (n*g(n-1));  }
var h = function (n) { if (n<=0)  return (1);     // Über arguments.callee
                       else       return ( n * arguments.callee(n-1) );}

var x = (function (n) {if (n<=0)  return (1);     // Aufruf Funktions-Ausdruck
                       else       return ( n * arguments.callee(n-1) );} ) (5);
```

**Src. 2:** Rekursive Definitionen der Fakultätsfunktion illustrieren den rekursiven Rückgriff innerhalb der Definition auf die gerade definierte Funktion.

# Ungetypte Werte und gemischte Retouren

**Beachte:** Variablen und Retouren nicht getypt (mixed types möglich).

```
function fct (x) { if (x > 0)  return Math.log (x);
                   else        return "Illegales Argument";
}
```

**Src. 3:** Gemischte Typen bei Funktions-Rückgaben sind möglich.

```
function fct (x) { if (x > 0)  return Math.log (x);
                   else        throw new Error ("Illegales Argument");
}
```

**Src. 4:** Eine bessere Lösung des obigen Beispiels arbeitet mit Exceptions und nutzt die damit verfügbare Unterbrechung des Kontrollflusses.

# Funktionen als First Class Objects

**Funktionen** sind "First Class Objects", also **normale Werte**.

- Können in Variablen gespeichert werden.
- Können als Parameter und Rückgabewert übergeben werden.

```
var fun = Math.sin; alert ("sin(pi/2) =" + fun ( Math.PI/2 ) );

var dble = function (x) {return 2*x;}
var quad = function (x) {return dble (dble (x));}

var twice = function (x, fct) {return fct(fct(x));}
console.log ( twice (dble, 4) );

var twice2 = function (fct) { return function (x) {return fct (fct (x));} }
console.log ( twice2 (dble) (3) );
```

**Src. 5:** Funktionen als First Class Objects

# Anonyme Lambda Definitionen

Funktionen erlauben anonyme (namensfreie) Lambda-Definition.
Rekursiver Rückgriff via `arguments.callee`.

```
var quad   = function (x) {return x*x;}
var addOne = function (n) {return n+1;}
```

**Src. 6:** Anonyme Lambda-Definitionen

# Closures: Definitionen mit freien Variablen

**Closures:** Funktions-Definitionen mit freien Variablen. (vgl: $\lambda y.xy$)
**Kernfrage:** Wie werden die freien Variablen bei der Auswertung bewertet?

```
var x = 7;
var f = function (y) {return x*y;};
x = 8;
f(3) = ??    // 21=3*7 oder 24=3*8
```

**Src. 7:** Die zwei sinnvoll möglichen Semantiken bei Closures.

**Mögliche** sinnvolle Antworten sind:

- 21 wenn Wert von x zum **Definitionszeitpunkt** genommen wird
  Auch: Interpretation von x als **Wert**
- 24 wenn Wert von x zum **Evaluationszeitpunkt** genommen wird
  Auch: Interpretation von x als **Referenz** bzw. Adresse

**Frage:** Welche Design-Entscheidung wählt Javascript?

# Aufgabe

```
var x = 7;
var f = function (y) {return x*y;}
x = 8;
console.log ( f(3) );  // 21 oder 24 ??

var obj = {x:7};
var f   = function (y) {return obj.x * y;}
obj.x   = 8;
console.log (f(3));    // 21 oder 24 ??

var obj = {x:7};
var f   = function (y) {   return f.val * y;}
f.val   = obj.x;       // was ist das jetzt ??
obj.x   = 8;
console.log (f(3));    // 21 oder 24 ??
```

**Src. 8:** Aufgaben zur Semantik der Closures.

# Currysieren im Beispiel

**Zur Erinnerung:**

$$\begin{array}{lll} f\colon A \times B \to C & (a,b) & \mapsto f(a,b) \\ f\colon A \to [B \to C] & a & \mapsto f(a)(\bullet) = f(a,\bullet) = \lambda x.f(a,x)\colon B \to C \end{array}$$

Eigentlich banal $f(a)(b) = f(a,b)$, in Typ-Theorie aber sehr tiefliegend.
Overloading nicht ungefährlich, da Verwechslungen ermöglicht.

```
function add(x, y) {
  if (typeof y === "undefined") {  // partial application
      return function (y) {return x + y;};   }
  else { return x + y; }           // full application
}
```

**Src. 9:** Currying als Rückgabe einer partiell ausgewerteten Funktion mit Bildung einer Closure

# Currysieren im Abstrakten

```
function CURRY (fn) {
  var storedArgs = Array.from(arguments).slice(1);  // fn entfernen, Args speichern
    return function ()  {
      var newArgs = Array.from(arguments);
      var  args   = storedArgs.concat(newArgs);     // restliche Args dran
      return fn.apply(null, args);                  // anwenden
    };
}

function add(x, y) {return x + y;}
var newadd = CURRY (add, 5);
newadd(4);
CURRY (add, 6)(7);
```

**Src. 10:** Systematisches Currysieren

# Filter

```
function filter (array, test) {
  var passed = [];
  for (var i = 0; i < array.length; i++) {
    if (test(array[i])) passed.push(array[i]);
  }
  return passed;
}

function forEach (array, action) {
  for (var i = 0; i < array.length; ++i) action(array[i]);
}
```

**Src. 11:** Filter und forEach Funktion.

# Map–Reduce

```
function map (func, array) {
  var result = [];
  forEach(array, function (elem) {result.push(func(elem));} );
  return result;
}

function reduce (combine, base, array) {
  forEach(array, function (element) { base = combine(base, element); } );
  return base;
}
```

**Src. 12:** Map-Reduce Mechanismus.

# Eingebauter Compiler

## Für **beliebige Evaluation**

```
var str = "if ( a ) { 1+1; } else { 1+2; }";
var a = true;
var b = eval(str); // 3
```

**Src. 13:** Beliebige Evaluation von Ausdrücken. Aus diversen Gründen (langsam, keine Optimierung, unsicher) weniger empfohlen.

## Für **Funktions-Körper**

```
var body = "return x * y;"
var multiply = new Function("x", "y", body);
```

**Src. 14:** String wird dynamisch in eine Funktion konvertiert. Besser als `eval`, da klares Scope Konzept.

## 4. Objekt-orientierte Sprache

Javascript ist eine **Prototypen-orientierte** OO Sprache; sie ist damit flexibler als eine **Vererbungs-orientierte** OO Sprache.

# Javascript als objekt-orientierte Sprache

JS **hat**:

- **Vererbung** (via Prototyp-Delegation, nicht via Klassen-Hierarchie)
- **Polymorphismus**

JS **hat nicht**:

- **Sichtbarkeit** (vgl: C++: `public`, `protected`, `private`)
  Aber: In funktionalen Umgebungen (scopes) nachbildbar.
- **Namensräume** (vgl: C++ `namespace`)
  Aber: Über key-value Pfade nachbildbar.
- **Modulkonzepte** (vgl: Ada `package`)
  Aber: Auf viele Arten nachbildbar durch funktionale Konzepte.
- **Type-casting** (vgl: C++ `dynamic_cast`)
  Aber: Über prototype chain teilweise nachbildbar
- **Templates** (vgl: C++ `template <class X>`)
  Aber: Über Funktionen höherer Ordnung und eingebetteten Compiler nachbildbar.

# Klassen-basierte Vererbung (1)

**Typische Beispiele:** Java, C++

**Grundkonzepte:**

- Es gibt Klassen.
- Klassen unterliegen einer Vererbungshierarchie.
- Objekte sind Instanzen von Klassen.
- Objekt-Layout erfolgt zur Compilezeit.
- Klassenhierarchie zur Compilzeit bekannt.

**Zeitpunkt** der Methoden-Bindung:

- **statisch** zur Compile-Zeit abhängig vom deklarierten Typ.
- **dynamisch** zur Laufzeit abhängig vom tatsächlichen Typ (C++: `virtual`).

# Klassen-basierte Vererbung (2)

**Technik** der Methoden-Bindung:

- Methoden-Namen zur Compile-Zeit bekannt.
- Offset-Tabelle: `Methoden-Name` $\mapsto$ `Offset`
- **Statisch:** Tabelle zur Compilezeit bekannt.
- **Dynamisch:**
  Tabelle zur Laufzeit bekannt, Typ-spezifisch (Polymorphismus).
- Schönes Tutorial zu Java

**Abstrakte** Klassen

- Unvollständige Typ-Information (Nicht-Blätter in Hierarchie)

# Prototypen-basierte Vererbung

**Grundkonzepte:**

- Jedes Objekt ist ein key-value Store.
- Methoden und Eigenschaften können zur Laufzeit hinzugefügt werden.
- Jedes Objekt hat ein Prototyp-Objekt.
- Methodenaufruf:
  - 1. Fall: Objekt besitzt Methode selber.
  - 2. Fall: Wenn nicht: delegiert weitere Suche an Prototyp-Objekt.

**Unterschiede:**

- Es gibt keine Klassen!
- Vererbungs-Hierarchie entsteht zur Laufzeit durch Prototyp-Chain.
- Vererbungs-Hierarchie kann zur Laufzeit dynamisch & mehrfach verändert werden.

**Implementierung:**

- Durch key-value Speicher (Hash-Tabelle; Baum) und Prototyp-Kette (langsam)
- Durch Patchen am Binary zur Laufzeit (V8; sehr schnell)

# Hat Javascript wirklich keine Klassen?

**Nein:** Im Sinne einer statischen Klassenhierarchie.

**Ja:** Im Sinne eines traditionallen OO Konzepts.

- Später (2015, ECMA Script 6) hinzugefügt.
- Syntaktisches Konstrukt, das auf Klassen-freies JS abbildbar ist.
- Keine neues semantisches Sprachkonzept, nur “syntactic sugar”.
- Schließt nur scheinbar auf zu traditionellen OO Sprachen.

# Kontext

Jede Funktionsauswertung läuft in einem Kontext (`this`).

Dieser **Kontext** kann sein

1. **Globales** Objekt
2. **Kontextuelles** Objekt (dynamisch und statisch)
3. **Gebundenes** Objekt
4. **Verbotenes** Objekt

**Beachte:** Funktionen und **Arrow-Funktionen** lösen das `this` verschieden auf!
Davon später mehr.

# Globales Objekt

Ist vom System vorgegeben. Browser: `window`

```
window.meintest=42;
function show () {console.log (this.meintest);}
show ();      // 42
```

**Src. 15:** `this` bei seiner Auflösung als globales Objekt.

# Kontextuelles Objekt (statisch)

```
var obj = {
  "typ": "Katze",
  "show": function () {console.log (this.typ);}
};
obj.show();  // Katze
```

**Src. 16:** Funktion greift das `this` im eigenen Kontext auf.

# Kontextuelles Objekt (dynamisch)

```
function Prof (name) {  // Uppercase Konvention
  this.name  = name;
  this.show  = function () { console.log (this.name); };
  this.check = function () { console.log (this);      };   }
var dozent   = new Prof ("Gero");  // Konstruktor-Verwendung der Funktion
dozent.show();   // Gero
dozent.check();  // Komplettes Objekt, soweit es console.log zeigt

// Hausaufgabe - alle Details con console.log (this) in console ansehen
```

**Src. 17:** `this` im dynamisch, durch `new` neu erzeugten Kontext.

# Gebundenes Objekt: call und apply

**bind**, **call** und **apply** verändern die Bindung.

```
...                         // Code von zuerst
var dozent2 = new Person ("Andreas");
window.name = "Fenster";
var fct = dozent.show;
fct();                      // Fenster  -  normale Bindung
fct.apply (dozent2);        // Andreas  -  veränderte Bindung
```

**Src. 18:** Veränderung der Bindung durch `apply`.

```
fun.apply (thisArg, argsArray);
fun.call  (thisArg, arg1, arg2, arg3, ...);
var numbers = [5, 6, 2, 3, 7];
var max = Math.max.apply(null, numbers);
```

**Src. 19:** Unterschied von `apply` und `call` liegt in den weiteren Argumenten (Array oder gelistete Werte).

# Gebundenes Objekt: bind

```
var dozent  = new Person ("Gero");
var dozent2 = new Person ("Andreas");
var fct = dozent.show;           // Funktion stammt von Gero
var fct2 = fct.bind (dozent2);   // bindet Andreas fest an Funktion
fct2();                          // Funktion liefert Andreas
```

**Src. 20:** Wirkung von bind

# Verbotenes Objekt

**Strikter Modus**: Veränderte, sicherere Semantik.

```
"use strict";      // aktiviert den strikten Modus
function fct () {console.log (this);}
fct();  // undefined
```

**Src. 21:** Im strikten Modus wird der Rückgriff auf das globale Objekt verboten.

```
function fct () {console.log (this);}
fct();  // window Objekt
```

**Src. 22:** Bekannter Rückgriff auf das globale Objekt.

# Typsysteme und Klassenhierarchien

**Mögliche Ziele:**

- Auffinden von Programmfehlern zu Compile / Binde / Laufzeit
- Schaffen mentaler Ordnung im Kopf des Programmierers
- Wiederverwendung von Code durch Vererbung
- Überschreiben wiederverwendbaren Codes
- Garantien von Code-Verfügbarkeit (`abstracte` Klassen)
- Anwendbarkeit von Operationen
- Auflösen überladener Bezeichner
- ... (viele spezifische Ansätze und Theorien)

**Tradeoff:** Flexibilität – Sicherheit

# Beispiel zum Setzen des Prototypen

```
let Kfz   =  {farbe: "rot"};
let Taxi  =  {paxe: 4};
Object.setPrototypeOf (Taxi, Kfz);
console.log (Taxi.farbe);  // rot
```

**Src. 23:** Setzen des Prototypen.

# Generieren von Objekten

```
function Hund () {}
Hund.prototype.sagt = function () {console.log (" wuff ");}

function Dackel () {}
Dackel.prototype = new Hund ();

var Wau = new Dackel ();

Wau.sagt();
console.log (Wau instanceof Dackel);   // true
console.log (Wau instanceof Hund);     // true
```

**Src. 24:** Generieren von Objekten mit new.

# Generieren von Objekten

```
function Hund (name) {this.name = name;}
Hund.prototype.sagt = function () {console.log (this.name + ": wuff ");}

function Dackel () {}
Dackel.prototype = new Hund ();

var Wau = new Dackel ();
Wau.sagt();                 // nicht ganz was wir wollen
                            // Alle Dackel heißen gleich und undefined
```

**Src. 25:** Achtung auf korrekten Aufbau der Klassen.

# Aufgabe

Spielen Sie eine klassisches Vererbungs-Beispiel mit JS durch.

- Mehrstufige Vererbungshierarchie.
- Vererben und Überschreiben von Methoden und Eigenschaften.
- Polymorphismus.
- Jeweils in der statischen Objekt-Variante und in der dynamischen `new` Variante.
- Abstrakte Objekte.

Nutzen Sie dazu einmal JS ohne `class` und einmal mit `class` und `extends`.

Was passiert, wenn die Prototype Chain zyklisch wird?

# Beispiele (1)

```c
typedef struct complex {float re; float im;} complex;

complex add(complex n1, complex n2) {
  complex tmp;
  tmp.re   = n1.re   + n2.re;
  tmp.imag = n1.imag + n2.imag;    // Schreibfehler
  return (tmp);
}

...
add (7, a);  // Typfehler im 1. Parameter
```

**Src. 26:** Hilfestellung durch das Typsystem

# Beispiele (2)

**Auflösen überladener Bezeichner**

- 3 + 4 = 7
- "hallo " + "welt" = "hallo welt"
- { a, b } + { a, c } = {a, b, c}
- [a, b] + [a, c] = [a, b, a, c]

**Anwendbarkeit** von Operationen (und Auflösung von Überladung)

- 5 / 4 = 1
- 5.0 / 4.0 = 1.25
- 4.0 / 6.0 = 0.6666666 oder 0.6666667 oder (2,3)
- "hallo " / "welt" = Compilezeitfehler / Laufzeitfehler / Exception / Coredump

# Aufgabe: Beispiele für Typ Konversion

```
function sum (a,b,c) {return a+b+c;}

console.log (sum ("hallo ","schöne ", "Welt"));
console.log (sum ("hallo ","Welt "));
console.log (sum (2, 3, 4));
console.log (sum (2, 3));
console.log (sum ("hallo ","Welt ", 23));
console.log (sum ("71",44, 23));
console.log (sum (44, 23, "88"));
console.log (sum (3, 2, 1));
console.log (sum (3, 2, true));
```

**Src. 27:** Was passiert jeweils genau? Beachte die Bedeutung der Reihenfolge der Auswertung!

# Quiz: Typ Konversion

Sei `obj` Teil eines Dokuments mit Font Size 12.
Was ist dann der Ausdruck `2 + obj.style.fontSize`?

1. 14 — $2 + 12 = 14$
2. "14px" — $2 + 12 +$ "px" von links nach rechts
3. "212px" — $2 +$ "12px" mit Konvertierung und Konkatenation
4. "2100%" — Interne Darstellung als 100% zu `baseSize`
5. Was anderes? — Name it!

Sei `obj` ein `<span>` Element mit aktueller Breite 30.
Was ist der Ausdruck `10 + obj.clientWidth`?

1. 40 — $10 + 30 = 40$
2. "1030" — $10 +$ "30" = "1030" Konvertierung und Konkatenation
3. "40px" — $10 + 30 +$ "px"
4. "1030px" — $10 +$ "30px"
5. Was anderes? — Name it!

# Typing in Javascript

**Duck-Typing:** If it knows how to "quack()" it's a duck.

**typeof Operator:** Unterscheidet primitive Werte und Objekte.

**instanceof Operator:** Erkennt ob eine Funktion in der Prototyp-Kette ist.

**Keine** Typkontrolle bei Variablen!

**Bewertung:** Weniger Garantien, Code wird unsicherer.

- Weniger erkannte Compilezeit- und Laufzeit-Fehler
- Bsp: Ist Zahl + String geplant oder Fehler?
- **Ausweg 1:** Testen mit hoher oder vollständiger Coverage.
- **Ausweg 2:** Typ Annotationen und statische Analyse.
- **Ausweg 3:** Typescript benutzen.

## 5. Skript-Sprache

Eine etwas unklare Kategorie...

# Einordnung ist schwierig

**Schwierig:** Was ist die *genaue* formale Definition einer "Skript Sprache"?

**Beispiele:** VB**Script**, J**Script**, Apple**S**cript, Java**s**cript

**Scherz:** Sprache ist Skriptsprache, wenn sie "Script" im Namen hat???

**Beispiele:** Shell, Lua, Guile, Tcl

**Daher:** Nur einige eher willkürliche Argumente.

# Flexibilität

**Wenig Zwang, viel Flexibilität.**

- Sprachdesign übt wenig "Druck" aus.
- Ganz anders bei "pure functional" oder strikten Typsystemen.
- Programmierer kann Stil (eher imperative, funktional oder OO) selber wählen.

**Beispiel: Typhierarchien**

- Bsp: Java: Falsche Typisierung erfordert Umbau der Hierarchie.
- Bsp: C: Ausweg über das abenteuerliche `(void*)`
- Bsp: C++: Ausweg über das abenteuerliche `dynamic_cast <>`
- Bsp: JS: Freie Wahl ob Klasse oder Key-Value Store mit Methoden-Lookup

## 6. Reflexive Sprache

Eine etwas eigenwillige Kategorie...

# Was ist eine reflexive Sprache?

Kann eine Sprache **über sich selber** "nachdenken"?

**Beispiele:**

- Finde die Namen aller Methoden einer Klasse.
- Finde den Namen der Klasse eines Objekts.
- Analysiere die Vererbungs-Hierarchie eines Objekts.
- Verändere die Vererbungs-Hierarchie zur Laufzeit.
- Füge neue Methoden zur Laufzeit hinzu oder verändere bestehende.
- Kombiniere Funktionen einer Klasse mit Funktionen einer zweiten Klasse zur Laufzeit.

Alles und mehr geht in JS.

# Reflexive Eigenschaften von Programmiersprachen

**C:** Sehr eingeschränkt, stark abhängig von Compiler und Binär-Architektur.

**C++:** Eingeschränkt; via Compiler-Option und zusätzliche Runtime.

```
const type_info& tp = typeid(*obj);        // ermittle Type Info zu Objekt
cout << "type = " << tp.name() << endl;    // gib Name der Klasse aus
```

**Src. 28:** Beispiel für reflexives Programmieren in C++

**Java:** Teilweise; durch Analyse des eigenen Class-Files
Übersicht über die Reflection API von Java

**PHP:** Übersicht über die Reflection API von PHP

# Reflexive Eigenschaften von Javascript

**Primär** durch die Konstruktion der Sprache selber. Hier:

- Objekt konzeptuell als key-value Store angelegt.
- Klassenname = Name der erzeugenden Funktion.
- Vererbung über die Prototyp Chain.

**Sekundär** durch schrittweise hinzugefügte Erweiterungen.

- Funktionszusammenfassungen oder syntaktische Verschönerungen.
  Auf bisheriges Javascript transpilierbar.
- Grundlegende Erweiterungen in der Semantik.
  Nicht auf bisheriges JS transpilierbar.

# Schnittstellen für reflexive Programmierung in Javascript

**Drei wichtige APIs.**

- MDN: Object API
- MDN: Proxy Objekt
- MDN: Reflect Objekt

**Wichtiger Übersichten:**

- MDN: Vergleich: Objekt und Reflect
- MDN: Übersicht über Meta-Programmierung in Javascript

# Anhang

# Übersicht

# Programmquellenverzeichnis (1/3)

# Rechtliche Hinweise (1)

Die hier angebotenen Inhalte unterliegen deutschem Urheberrecht. Inhalte Dritter werden unter Nennung der Rechtsgrundlage ihrer Nutzung und der geltenden Lizenzbestimmungen hier angeführt. Auf das Literaturverzeichnis wird verwiesen. Das Zitatrecht in dem für wissenschaftliche Werke üblichen Ausmaß wird beansprucht. Wenn Sie eine Urheberrechtsverletzung erkennen, so bitten wir um Hinweis an den auf der Titelseite genannten Autor und werden entsprechende Inhalte sofort entfernen oder fehlende Rechtsnennungen nachholen. Bei Produkt- und Firmennamen können Markenrechte Dritter bestehen. Verweise und Verlinkungen wurden zum Zeitpunkt des Setzens der Verweise überprüft; sie dienen der Information des Lesers. Der Autor macht sich die Inhalte, auch in der Form, wie sie zum Zeitpunkt des Setzens des Verweises vorlagen, nicht zu eigen und kann diese nicht laufend auf Veränderungen überprüfen.

Alle sonstigen, hier nicht angeführten Inhalte unterliegen dem Copyright des Autors, Prof. Dr. Clemens Cap, ©2020. Wenn Sie diese Inhalte nützlich finden, können Sie darauf verlinken oder sie zitieren. Jede weitere Verbreitung, Speicherung, Vervielfältigung oder sonstige Verwertung außerhalb der Grenzen des Urheberrechts bedarf der schriftlichen Zustimmung des Rechteinhabers. Dieses dient der Sicherung der Aktualität der Inhalte und soll dem Autor auch die Einhaltung urheberrechtlicher Einschränkungen wie beispielsweise Par 60a UrhG ermöglichen.

Die Bereitstellung der Inhalte erfolgt hier zur persönlichen Information des Lesers. Eine Haftung für mittelbare oder unmittelbare Schäden wird im maximal rechtlich zulässigen Ausmaß ausgeschlossen, mit Ausnahme von Vorsatz und grober Fahrlässigkeit. Eine Garantie für den Fortbestand dieses Informationsangebots wird nicht gegeben.

Die Anfertigung einer persönlichen Sicherungskopie für die private, nicht gewerbliche und nicht öffentliche Nutzung ist zulässig, sofern sie nicht von einer offensichtlich rechtswidrig hergestellten oder zugänglich gemachten Vorlage stammt.

**Use of Logos and Trademark Symbols:** The logos and trademark symbols used here are the property of their respective owners. The YouTube logo is used according to brand request 2-9753000030769 granted on November 30, 2020. The GitHub logo is property of GitHub Inc. and is used in accordance to the GitHub logo usage conditions https://github.com/logos to link to a GitHub account. The Tweedback logo is property of Tweedback GmbH and here is used in accordance to a cooperation contract.

# Rechtliche Hinweise (2)

**Disclaimer:** Die sich immer wieder ändernde Rechtslage für digitale Urheberrechte erzeugt für mich ein nicht unerhebliches Risiko bei der Einbindung von Materialien, deren Status ich nicht oder nur mit unverhältnismäßig hohem Aufwand abklären kann. Ebenso kann ich den Rechteinhabern nicht auf sinnvolle oder einfache Weise ein Honorar zukommen lassen, obwohl ich – und in letzter Konsequenz Sie als Leser – ihre Leistungen nutzen.

Daher binde ich gelegentlich Inhalte nur als Link und nicht durch Framing ein. Lt EuGH Urteil 13.02.2014, C-466/12 ist das unbedenklich, da die benutzten Links ohne Umgehung technischer Sperren auf im Internet frei verfügbare Inhalte verweisen.

Wenn Sie diese Rechtslage stört, dann setzen Sie sich für eine Modernisierung des völlig veralteten Vergütungssystems für urheberrechtliche Leistungen ein. Bis dahin klicken Sie bitte auf die angegebenen Links und denken Sie darüber nach, warum wir keine für das digitale Zeitalter sinnvoll angepaßte Vergütungssysteme digital erbrachter Leistungen haben.

Zu Risiken und Nebenwirkungen fragen Sie Ihren Rechtsanwalt oder Gesetzgeber.

Weitere Hinweise finden Sie im Netz hier und hier oder hier.

# Zitierweise dieses Dokuments

Wenn Sie Inhalte aus diesem Werk nutzen oder darauf verweisen wollen,
zitieren Sie es bitte wie folgt:

Clemens H. Cap: Javascript als Programmiersprache. Electronic document. https://iuk.one/1066-1003 24. 4. 2021.

**Bibtex Information:** https://iuk.one/1066-1003.bib

```
@misc{doc:1066-1003,
   author       = {Clemens H. Cap},
   title        = {Javascript als Programmiersprache},
   year         = {2021},
   month        = {4},
   howpublished = {Electronic document},
   url          = {https://iuk.one/1066-1003}
}
```

**Typographic Information:**
Typeset on April 24, 2021
This is pdfTeX, Version 3.14159265-2.6-1.40.21 (TeX Live 2020) kpathsea version 6.3.2
This is pgf in version 3.1.5b
This is preamble-slides.tex myFormat©C.H.Cap

# Verzeichnis aller Folien

Legende:
Fortsetzungsseite
Seite ohne Überschrift
Bildseite