




# ECMAScript

Grundlagen der Programmierung

← ⓘ | [www.ecma-international.org/publications/standards/Ecma-262.htm](http://www.ecma-international.org/publications/standards/Ecma-262.htm)

 **Standards**

 **Contact Ecma**  
Rue du Rhône 114 · CH-1204 Geneva  
T: +41 22 849 6000 F: +41 22 849 6001

 [What is Ecma](#) | [Activities](#) | [News](#) | [Standards](#)

[Standards Index](#)  
[Standards List](#)  
[Withdrawn Standards](#)  
[Tech. Reports Index](#)  
[Tech. Reports List](#)  
[Withdrawn Tech. Reports](#)  
[Mementos](#)

[Printer Friendly Version](#)  
[« Back](#)

## Standard ECMA-262

### ECMAScript® 2016 Language Specification

7<sup>th</sup> edition (June 2016)

This Standard defines the ECMAScript 2016 general purpose programming language.

The following files can be freely downloaded:

File name	Size (Bytes)	Content
<a href="#">ECMA-262.pdf</a>	13 714 458	Acrobat (r) PDF file
<a href="#">ECMA-262 edition 7</a>		Browsable HTML

This 7<sup>th</sup> edition has been prepared under the Ecma RF patent policy.

The latest drafts are available at: <https://tc39.github.io/ecma262/>. Reporters should generally only file bugs if the bug is still present in the latest drafts.

Please find hereafter the place to file bugs: <https://github.com/tc39/ecma262#ecmascript>.

The previous replaced "historical" editions of this Ecma Standard are available [here](#).

[« Back](#)



## Standard ECMA-262

7<sup>th</sup> Edition / June 2016

# ECMAScript® 2016 Language Specification

# Standard

Rue du Rhône 114 · CH-1204 Geneva · T: +41 22 849 6000 · F: +41 22 849 6001

- Die in diesem Seminar verwendete Werkzeuge und Frameworks sind Open Source
  - LPGL Lizenzmodell
- Dies ist ein Programmier-Seminar
  - Damit werden die Inhalte durch Übungen vertieft und verinnerlicht
  - Musterbeispiele werden zur Verfügung gestellt
  - Diese können am Ende des Seminars als ZIP-Datei kopiert werden
    - USB-Stick oder ähnliches
- Dokumentation und Ressourcen stehen auch im Internet zur Verfügung

© Javacream

Javacream

Dr. Rainer Sawitzki

Alois-Gilg-Weg 6

81373 München

**Alle Rechte, einschließlich derjenigen des auszugsweisen Abdrucks, der fotomechanischen und elektronischen Wiedergabe vorbehalten.**

Grundlagen	1
Variablen und Operatoren	2
Kontrollstrukturen	3
Funktionen	4
Objekte und Eigenschaften	5
Referenzen und this	6
Objektorientierte Programmierung	7

# 1 ÜBERSICHT

## 1.1

# **EINORDNUNG VON ECMASCRIPT**



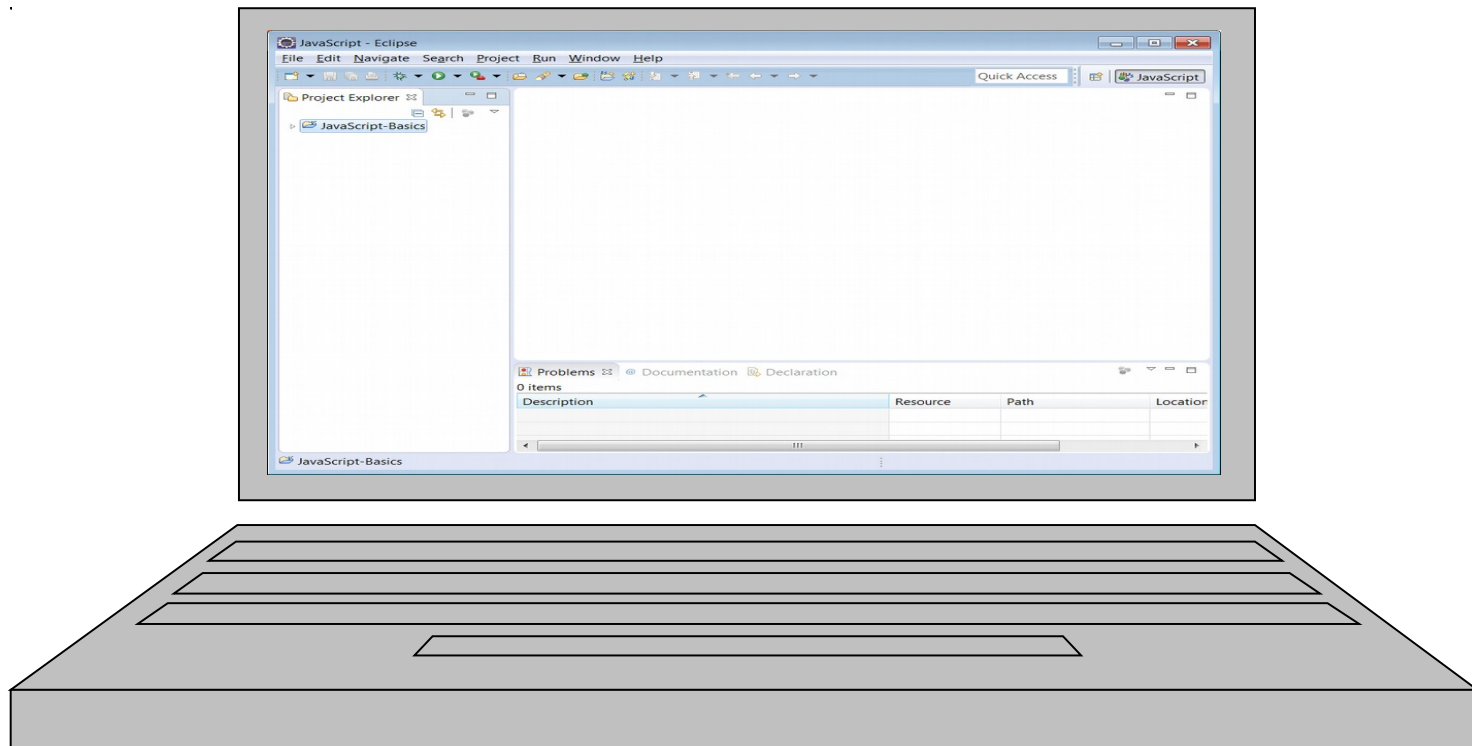


- ECMAScript
  - Eine von der „European Computer Manufacturers Association“ spezifizierte Script-Sprache
  - Enthält elementare Syntax und Sprachkonstrukte
- JavaScript
  - Ursprünglich eine nicht-standardisierte Programmiersprache für den Browser
    - Die einzelnen Browser-Hersteller fügen ihren Engines neue, nicht-standardisierte Funktionen hinzu
    - Grad der Standardisierung aber auch hier bereits hoch
  - JavaScript ist ein Superset von ECMAScript
    - Eigentlich genau anders: ECMAScript vereinheitlichte die verschiedenen JavaScript-Dialekte
- Siehe <http://en.wikipedia.org/wiki/ECMAScript>

1.2

## **WERKZEUGE**

- Starten der Umgebung mit `eclipse.exe`
- Das Projekt `ECMAScript-Basics` ist bereits eingerichtet und geöffnet
  - Simplex ECMAScript-Beispiel



- Im Rahmen des Kurses werden ECMAScript-Programme entwickelt
  - Browser sind als Laufzeitumgebung, Konsole und Debugger geeignet
    - Firefox und Chrome enthalten hervorragende Werkzeuge hierfür
  - Reines ECMAScript läuft auch in ECMAScript-Engines ohne Browser-Anbindung
    - Google V8
    - Java-basierte Implementierungen wie Rhino oder Nashorn
  - Node.JS
    - Die Quasi-Standard-Laufzeitumgebung

2

# **GRUNDLEGENDE SPRACHELEMENTE**

## 2.1

# LAUFZEITUMGEBUNG

- Mit Hilfe des Script-Tags werden externe ECMAScript-Dateien eingebunden
  - Mit dem Attribut `src` wird auf die Datei, in der die Kommandos stehen, verwiesen

```
<script type="text/ECMAScript" src="script.js"></script>
```

- Die ECMAScript-Anweisungen werden beim Laden der Seite sofort ausgeführt
- Programmausgaben erfolgen auf der Konsole des Browsers
  - `console.log("Output");`
- Hinweis
  - Eine Interaktion mit der dargestellten Web-Seite ist nicht vorgesehen!
  - Und kann mit ECMAScript auch nicht umgesetzt werden!

- NodeJS ist ein Standalone-Interpreter für ECMAScript
  - Installation der frei erhältlichen Distribution notwendig
- Anschließend steht der Node-Interpreter zur Verfügung
  - Ausführung in einer Eingabeaufforderung
    - Interaktive Shell mit `nodejs`
    - Ausführen eines Programms mit `nodejs <programmdatei.js>`



- Ab der Version 8 ist ein Java-basierter JavaScript-Interpreter Bestandteil der Java-Installation
  - Beruht auf der Nashorn-Engine
- Ausführen mit `jjs`
  - Interaktive Shell
  - Ausführen der Liste der angegebenen Skript-Dateien

## 2.2

# **ERSTE BEFEHLE**

- Folgende Kommentarmöglichkeiten sind möglich

- `//`Einzeiliger Kommentar

- `/*`  
    Mehrzeiliger Kommentar  
    `*/`

- Beispiele für Anweisungen
  - Wert-Zuweisungen
  - Ausführung einer Operation
  - Funktionsaufrufe
- Eine Anweisung in ECMAScript wird durch einen Strichpunkt ";" oder einem Zeilenumbruch abgeschlossen

- Identifizier dienen zur eindeutigen Benennung von
  - Variablen
  - Funktionen
  - Klassen (ab ES2015)
- Bei selbstvergebenen Namen gelten Regeln:
  - Buchstaben, Ziffern und der Unterstrich
    - das erste Zeichen darf keine Ziffer sein darf
    - Groß- und Kleinschreibung wird unterschieden
    - keine Leerzeichen
  - sie dürfen nicht mit einem reservierten Wort identisch sein
    - Dies sind die Schlüsselwörter von ECMAScript wie `if`, `for` ...

- Ein Literal erzeugt Daten eines bestimmten Typs
- Zeichenketten
  - `"45678"`
- Ganzzahlen
  - `42`
- Kommazahlen
  - `47.11`
- Logische Werte
  - `true` bzw. `false`
- Reguläre Ausdrücke
  - `/[a..z]/`
- Arrays
  - `[element1, element2, element3]`

- Numerische Werte sind Ganzzahlig oder Komma-Zahlen
  - Der Unterschied wird automatisch erkannt
  - Punkt zur Trennung des Nachkomma-Anteils
- Exponentialschreibweise mit  $e$  oder  $E$

## 2.3

# **VARIABLE**



- Deklaration
  - `var my_variable;`
- Deklaration und Zuweisung
  - `var my_variable = Wert;`
- Zuweisungen an Variablen können beliebig oft erfolgen
  - `var my_variable = "Hello";`
  - `my_variable = "Hello World!";`

- Eine Variable hat keinen statischen Typ
  - Dies ist ein großer Unterschied zu statisch typisierten Sprachen wie Java oder C#
  - Also ist möglich:
    - `var my_variable = "Hello";`
    - `my_variable = 42;`
- Variablen können an beliebigen Stellen im Code deklariert werden
  - Es ist nicht notwendig, sie alle am Anfang des Skripts zu sammeln
- Namenskonventionen
  - Camel-Case
    - Identifier bestehen nur aus Buchstaben
    - zur besseren Lesbarkeit werden Großbuchstaben eingestreut
  - Snake-Case
    - Nur Kleinbuchstaben, die mit Unterstrichen strukturiert werden

- Eine Deklaration ohne das Schlüsselwort `var` ist möglich, ist aber zu vermeiden
  - Damit wird etwas völlig anderes gemacht, nämlich ein Hinzufügen einer Eigenschaft zum globalen Objekt
  - Aktuell völlig unverständlich und in den allermeisten Fällen so auch nicht beabsichtigt, sondern ein Programmier-Fehler
- Obwohl Variablen an beliebiger Stelle deklariert werden können, sammelt der ECMAScript-Interpreter vor Ausführung alle Variablendeklarationen und stellt diese an den Anfang des Skripts
  - "Hoisting"
- ES2015 führt das neue Schlüsselwort `let` ein, das die Gültigkeit einer Variable beschränkt
  - Und zwar auf den deklarierenden Block

- Welchen Typ eine Variable besitzt, entscheidet sich bei der ersten Wertzuweisung:

```
var x = "45678";    //String
var y = 45678;      //Zahl
var z = 13.2345;    //Zahl
```

  - Hier enthält die Variable `x` den String (Buchstaben- und Ziffernfolge) 45678, während die Variable `y` die Zahl 45678 und die Variable `z` die Zahl 13,2345 enthält.
- Variable haben einen Typ, der zur Laufzeit festgelegt wird
  - `var x = "45678"; //String`
  - `var x = 45678; //Zahl`

2.4

## **KONTROLLSTRUKTUREN**

```
■ if (Bedingung)
{
  Anweisung1;
  Anweisung2;
  ...
}
else
{
  Anweisung7;
  Anweisung8;
  ...
}
```

```
switch (Ausdruck) {  
  case FallA: {  
    Anweisung1; break;  
  }  
  case FallB: {  
    Anweisung2; break;  
  }  
  default: {  
    Anweisung4;  
  }  
}
```

- `while (Bedingung)`  
    {  
    Anweisung;  
    Anweisung;  
    }
  
- `do`  
    {  
    Anweisung;  
    }  
    `while (Bedingung)`



- `for(Initialisierung; Bedingung; Zähleranweisung) {  
    Anweisungen;  
}`
- `break`
  - Veranlasst das sofortige Verlassen der aktuellen Schleife
- `continue`
  - Springt zum nächsten Schleifendurchlauf

- Mit dieser Schleife kann man durch alle Eigenschaften eines Objektes und alle Elemente eines Arrays laufen
  - Diese Variante ist nicht ECMA-Standard, wird aber von den meisten Browsern unterstützt

```
for each (a in object | array) {  
    //Anweisungen  
}
```

2.5

## **OPERATOREN**

<b>+</b>	<b>Addition</b>
<b>-</b>	<b>Subtraktion</b>
<b>*</b>	<b>Multiplikation</b>
<b>/</b>	<b>Division</b>
<b>%</b>	<b>Modulo (Teilungsrest bei einer Division)</b>
<b>++</b>	<b>um 1 erhöhen (Inkrement)</b>
<b>--</b>	<b>um 1 vermindern (Dekrement)</b>

<b>==, ===</b>	<b>Gleichheit, mit oder ohne Konvertierung</b>
<b>&gt;</b>	<b>Größer</b>
<b>&lt;</b>	<b>Kleiner</b>
<b>&gt;=</b>	<b>Größer gleich</b>
<b>&lt;=</b>	<b>Kleiner gleich</b>
<b>!=, !==</b>	<b>ungleich</b>

<b>  </b>	<b>OR</b>
<b>&amp;&amp;</b>	<b>AND</b>
<b>!</b>	<b>Negation</b>

2.6

## **FUNKTIONEN**

- **Allgemeine Form**

```
function name([var1][,var2][,var3]){  
    Anweisung1;  
    Anweisung2;  
    ...  
    [return ausdruck;]  
}
```

- **Dabei bedeuten:**

- `name()`
  - der Name der Funktion
- `var1, var2, var3`
- **Beliebig lange Listen von Parametern**
  - Diese Parameter sind innerhalb der Funktion als lokale Variable bekannt.
- `return ausdruck;`
  - optionaler Rückgabewert der Funktion



- Statt der allgemeinen Form der Funktions-Deklaration kann das Funktions-Literal benutzt werden

```
var function_name = function(var1, var2, var3) {  
    //...  
}
```

- Diese Möglichkeit wird für die Objekt-Orientierte Programmierung in ECMAScript sehr wichtig werden

- Die Anzahl der deklarierten Parameter in der Funktions-Definition und im Funktions-Aufruf ist vollkommen gleichgültig
  - Überschüssige Parameter werden ignoriert
  - Ein „überladen“ von Funktionen, das heißt eine Unterscheidung gleich benannter Funktionen über eine Parameter-Liste ist in ECMAScript nicht möglich
    - Die jeweils letzte Funktion wird effektiv benutzt und ersetzt alle vorherigen Definitionen.
- Nicht vorhandene Parameter bekommen den internen Wert `undefined` zugeordnet
- Alle Parameter stehen innerhalb der Funktion in dem Array `arguments` zur Verfügung

3

# **OBJEKTORIENTIERTE PROGRAMMIERUNG**

## 3.1

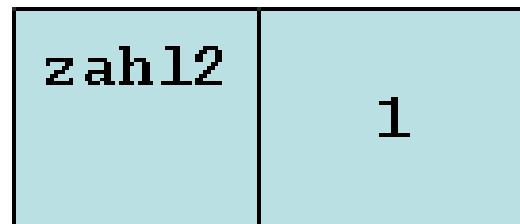
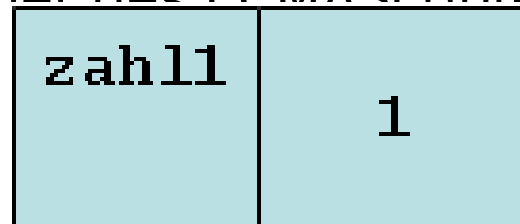
# **SPEICHERVERWALTUNG UND REFERENZEN**

- Gehen wir zurück auf die bisher bekannten Typen, beispielsweise eine Zahl:

```
var zahl1 = 1;
```

```
var zahl2 = 1;
```

- `zahl1` und `zahl2` sind benannte Variable
- Obwohl beide Variablen denselben Wert aufweisen, existiert dieser Wert im Speicher des ECMAScript-Interpreters doppelt:



- Wird beispielsweise `zahl2` inkrementiert, so hat diese selbstverständlich nur Auswirkungen auf einen Speicherbereich:

```
zahl2++;
```

<b>zahl1</b>	<b>1</b>
--------------	----------

<b>zahl2</b>	<b>2</b>
--------------	----------

- Was passiert aber nun bei einer Zuweisung an eine weitere Variable:  
`var zahl3 = zahl1;`
- Das ist nun nicht automatisch klar, denn es gibt prinzipiell zwei Möglichkeiten

- Der Speicherbereich bekommt einen weiteren Namen:

<b>zahl1</b> <b>zahl3</b>	<b>1</b>
------------------------------	----------

<b>zahl2</b>	<b>2</b>
--------------	----------



# Alternative 2: Kopie des Wertes

- Der Wert wird in einen neuen Speicherbereich kopiert:

zahl1	1
-------	---

zahl2	1
-------	---

zahl3	1
-------	---

- Welcher der beiden Varianten korrekt ist bestimmt ein einfacher Test:
- Wir erhöhen die Variable `zahl3` und schauen nach, was in `zahl1` steht:

```
zahl3++;  
console.log(zahl1)
```

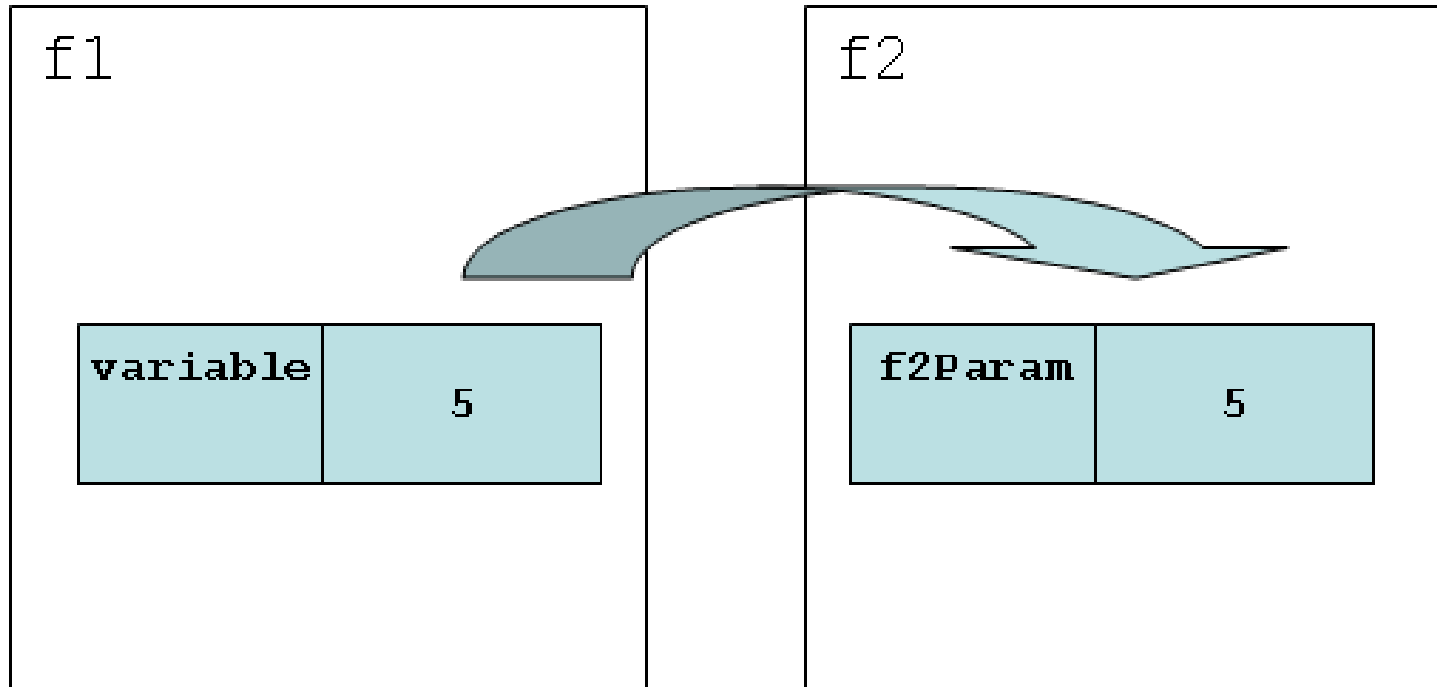
- Die Ausgabe ist „1“ und somit ist klar, dass bei der Zuweisung an Variablen der Wert an einen neuen Speicherbereich kopiert wird

- Was hier am Beispiel von Variablen gezeigt wurde ist auch bei einem Funktionsaufruf relevant
- Auch hier wird die übergebene Variable in den benannten Funktionsparameter kopiert

```
function f1(){  
  var variable = 5;  
  f2(variable);  
}  
function f2(f2Param){  
  //aktionen mit f2Param durchführen  
  f2Param = 7;  
}
```

- Sämtliche Änderungen, die innerhalb von `f2` mit `f2Param` passieren, sind lokal innerhalb von `f2` und haben keine Auswirkung auf die Variable `variable` in `f1`

# Parameterübergabe an eine Funktion



- Arrays sind Variablencontainer, die mehrere Variablen enthalten können
- Auf einzelne Variable greift man über den Variablennamen und eine Nummer zu
- Als Literal-Kennzeichen für Arrays wird eine eckige Klammer benutzt, die Werte sind durch Kommas getrennt:  

```
var theBeatles = ["John", "Paul", "George", "Ringo"];
```
- Der Index, über den auf ein Array-Element zugegriffen wird, steht in eckigen Klammern:  

```
var john = theBeatles[0];
```
- Ein Array kann jederzeit mit weiteren Werten befüllt werden bzw. die vorhandenen Einträge geändert werden
  - Der Zugriff erfolgt ebenfalls über den Index:  

```
theBeatles[4] = "George Martin";
```

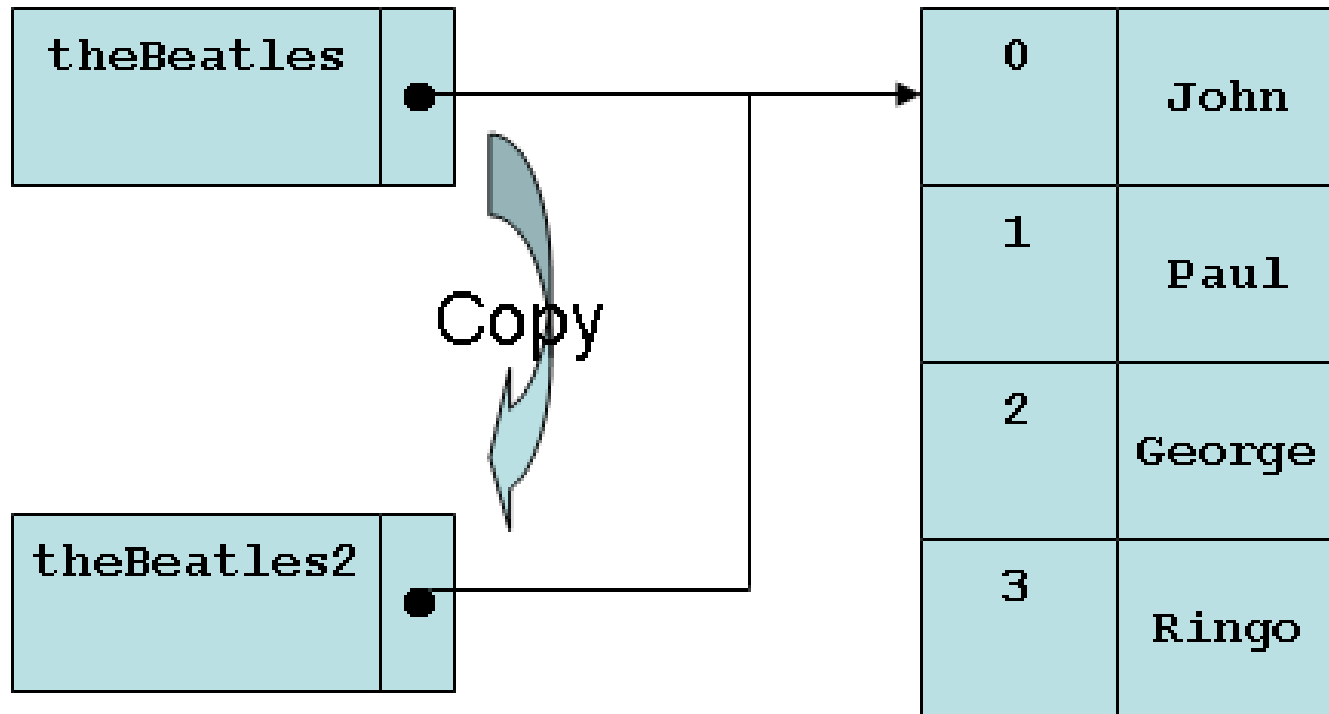
- Nun gehen wir aber zurück auf das Array-Beispiel

- Dies verhält sich nämlich grundsätzlich anders

```
var theBeatles = ["John", "Paul", "George", "Ringo"];  
var theBeatles2 = theBeatles;  
theBeatles[4] = "George Martin";  
console.log(theBeatles2[4])
```

- Hier wird nämlich sehr wohl der „5. Beatle“ George Martin ausgegeben!

- Eine grafische Darstellung arbeitet am Besten mit einem Pfeilsymbol, den sogenannten Referenzen:
  - Technisch gesehen ist eine Referenz eine interne Speicheradresse
- Auch hier wird bei der Zuweisung bzw. bei einer Parameterübergabe an eine Funktion ein Wert kopiert
  - Es ist hier allerdings der Wert der Referenz!
- Wird somit ein Array einer Funktion als Parameter übergeben, so schlagen Änderungen, die innerhalb der aufgerufenen Funktion durchgeführt werden, sehr wohl auf die Variablen der aufrufenden Funktion durch
  - Ein fundamentaler und sehr wichtiger Unterschied im Vergleich zu den bisher benutzten Variablen!





3.2

## **VON DATENSTRUKTUREN ZU OBJEKTEN**

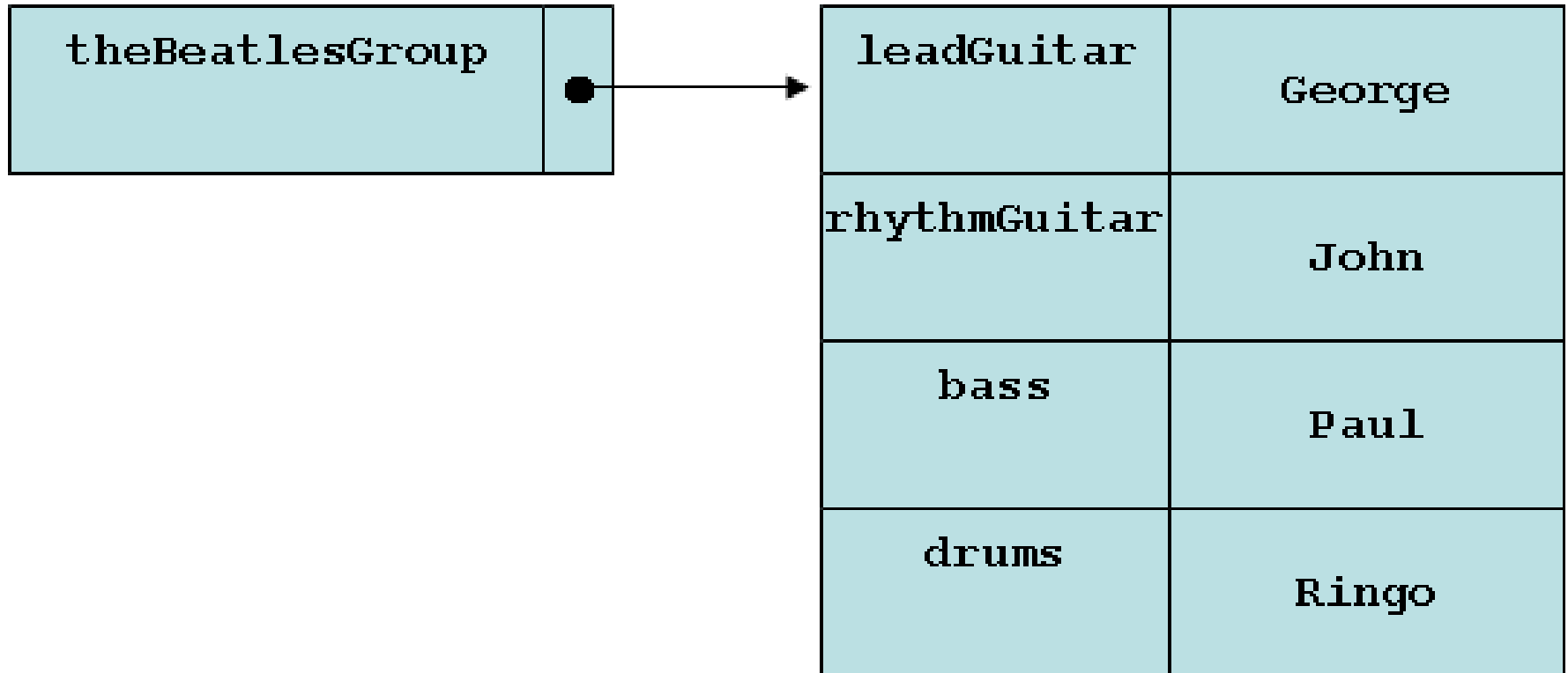
- Eine Datenstruktur kann auch mit Schlüssel-Werte-Paaren definiert werden:

```
var theBeatlesGroup = {leadGuitar: "George",  
  rhythmGuitar: "John", bass: "Paul", drums: "Ringo"};
```

- Im Unterschied zu den Arrays verwenden wir hier ein paar geschweifter Klammern
  - Weiterhin müssen wir natürlich die Schlüssel-Namen bei der Definition des Hashes mit angegeben werden
- Der Zugriff erfolgt nun über den Schlüssel, ebenfalls angegeben in eckigen Klammern:

```
console.log(theBeatlesGroup["drums"])  
//-> "Ringo"
```

- Auch Hashes werden über Referenzen angesprochen



- Statt der eckigen Klammern kann für den Zugriff auch der Punkt-Operator benutzt werden:

```
theBeatlesGroup.drums
```

- Die Bedeutung dieses Punkts ist für den ECMAScript-Laufzeitumgebung einfach zu interpretieren:
  - Der Punkt-Operator heißt: „Verfolge die Referenz“
- Diese erst einmal einfach scheinende Änderung hat begrifflich weit reichende Folgen
  - Statt eines „Assoziativen Arrays“ mit Schlüsseln sprechen wir von einem "Objekt mit Eigenschaften"
  - Rein technisch gesehen unterscheidet ECMAScript nicht zwischen Hashes und Objekten
  - oder noch genauer: (Fast) alles in ECMAScript sind Objekte

- Was nun extrem wichtig für den Objekt-Begriff ist, ist die Tatsache, dass eine Eigenschaft eines Objekts auch eine Funktion sein kann!
- Oder anders formuliert:
  - „Ein Objekt hält Daten-Informationen (die Eigenschaften oder Attribute) und Verhalten (die Funktionen oder Methoden)“
  - Dies ist eine grundlegende Definition innerhalb der Objekt-orientierten Programmierung
- ECMAScript interpretiert den Objekt-Begriff dynamisch:
  - Jedem Objekt können zur Laufzeit beliebige Eigenschaften neu zugeordnet werden

- **Simple Zuweisung:**

```
var theBeatles = {leadGuitar: "George", rhythmGuitar:
"John", bass: "Paul", drums: "Ringo"};
theBeatles.home = "Liverpool";
```

- **Als Eigenschaften können natürlich auch wieder komplexe Referenz-Typen benutzt werden, also zum Beispiel ein Array:**

```
theBeatles.albums = ["Please please me", /*usw*/, "Let
it be"];
```

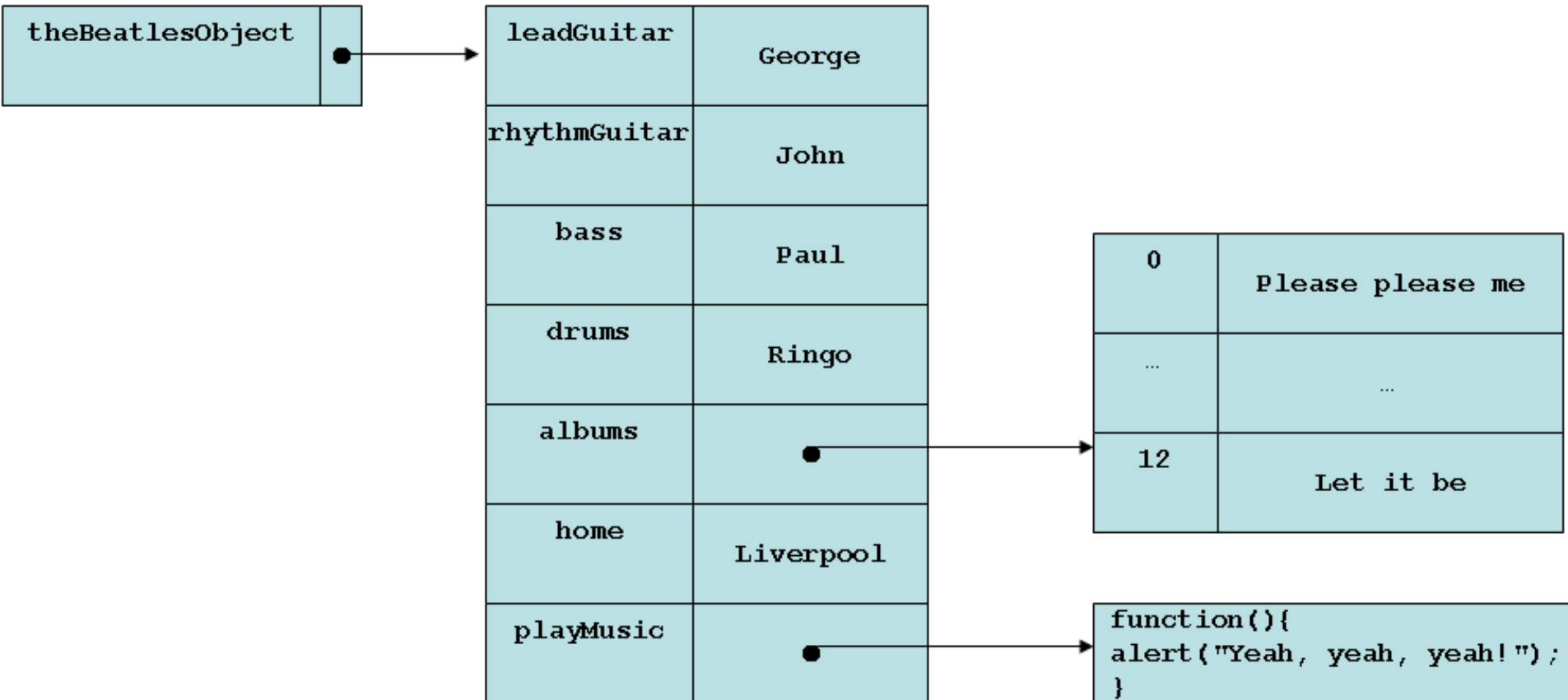
- **ein anderes Objekt:**

```
theBeatles.lifetime = {start: 1962, end: 1970};
```

- **oder eben eine Funktion:**

```
theBeatles.playMusic = function() {console.log("Yeah,
yeah, yeah!");};
```

# Ein Blick in den Speicher



### 3.3

## **EXKURS: KONSTRUKTOREN UND KLASSEN**



- Statt Objekte über ein `{}`-Literal zu definieren kann auch eine Konstruktor-Funktion benutzt werden
- Der Konstruktor bekommt für jede Eigenschaft einen Parameter
- Innerhalb der Konstruktor-Funktion werden diese Parameter den Eigenschaften des neuen Objekts zugewiesen

```
function Person(lastname, firstname){  
    this.lastname = lastname;  
    this.firstname = firtsname;  
}
```

- Das Erzeugen von Instanzen kann nun an anderer Stelle im ECMAScript-Programm mit dem Operator `new` geschehen  
`var person = new Person('Lennon', 'John');`

- Die Konstruktor-Funktionen und der `new`-Operator sind in ECMAScript notwendig, da es keine Klassen-Definitionen gibt
  - Eine Klasse ist ein abstraktes Template, aus dem Objekte erzeugt, besser: instanziiert werden
  - Jede Instanz einer Klasse hat damit einen durch die Klassen-Definition Satz von Eigenschaften
- Klassen sind in anderen Programmiersprachen wie Java und C# weit verbreitet
  - und sind bei Entwicklern sehr beliebt
- ECMAScript und Klassen:
  - Workarounds sind möglich
    - Das "Module-Pattern" ist ein Beispiel hierfür
  - Die Sprache Typescript führt Klassen ein
    - Aus Typescript wird ECMAScript generiert
  - ES2015 führt Klassen ein

3.4

## **SPEZIELLE REFERENZEN**

- Das eben eingeführte `this`-Schlüsselwort in ECMAScript muss noch genauer beschrieben werden
- `this` ist die Referenz auf den aktuellen Aufruf-Kontext
  - Ohne weitere Aktionen ist dies das „Globale Objekt“, siehe unten, das für jedes ECMAScript-Programm definiert ist

- Jeder normale Funktionsaufruf findet im diesem Kontext statt. Wird jedoch der Aufruf mit dem Schlüsselwort new durchgeführt passiert folgendes:
  - Es wird ein neues Objekt im Speicher erzeugt
  - Die this-Referenz deutet auf dieses neu erzeugte Objekt
    - Nicht mehr auf das globale Objekt!
  - Die constructor-Eigenschaft dieses neuen Objekts wird mit der aufgerufenen Funktion belegt
- Sämtliche Aktionen innerhalb der Konstruktor-Funktion beziehen sich somit auf das neu erzeugte Objekt

- Objekte werden, wie wir gesehen haben, mit Hilfe von `new` erzeugt
- Können Objekte auch durch eine Anweisung gelöscht werden?
  - Prinzipiell „Ja“, dafür gibt es die Anweisung `delete`
  - Allerdings ist die Anwendung dieser Anweisung optional,
    - statt dessen kontrolliert die ECMAScript-Laufzeitumgebung selber, welche Objekte vom Programm noch benutzt werden können
    - Alle anderen werden automatisch entfernt
- Dieser ständig laufende Hintergrundprozess wird „Garbage Collection“ genannt
- Damit muss sich der Programmierer nicht um die Speicherbereinigung kümmern

- `undefined` ist etwas, was noch nicht definiert ist
  - Wird beispielsweise auf eine Eigenschaft eines Objektes verwiesen, die nicht existiert, so wird der Wert `undefined` benutzt.
- `null` hingegen ist zwar vorhanden, soll aber aus Sicht der Anwendung „nichts“ beinhalten
- Hinweis:
  - In Abfragen werden implizit beide Objekt-Referenzen als `false` evaluiert

3.5

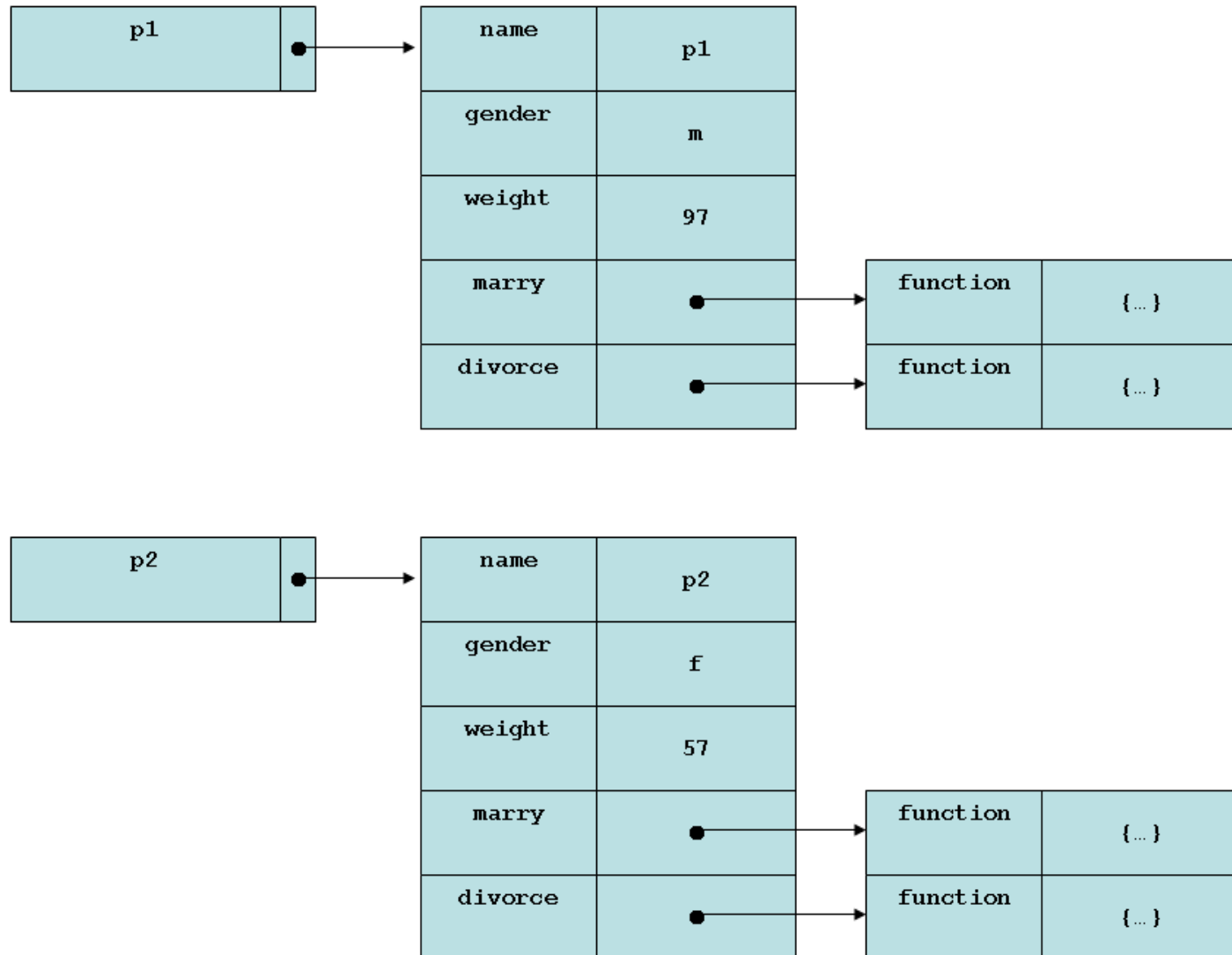
## **EXPERTENWISSEN: DAS PROTOTYPE-OBJEKT**



- Werden in einer Konstruktor-Funktion dem Objekt Funktionen zugewiesen, so wird für jedes Objekt ein eigenes Funktionsobjekt neu erzeugt

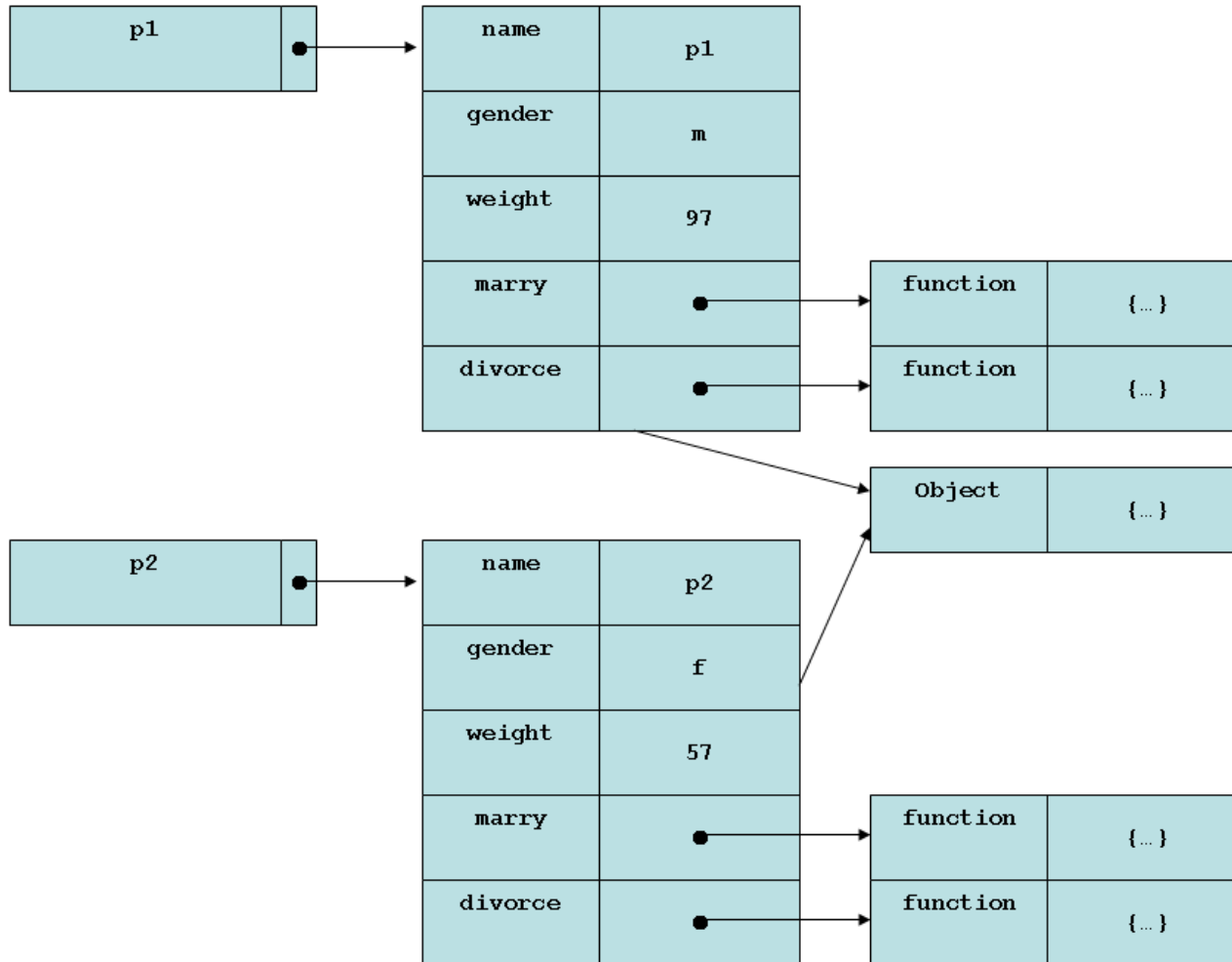
```
function Person(lastname, firstname, age){  
  this.lastname = lastname;  
  this.firstname = firtsname;  
  this.age = age;  
  this.marry = function(){//...}  
  this.divorce = function(){//...}  
  
}
```

```
var p1 = new Person();  
var p2 = new Person();
```

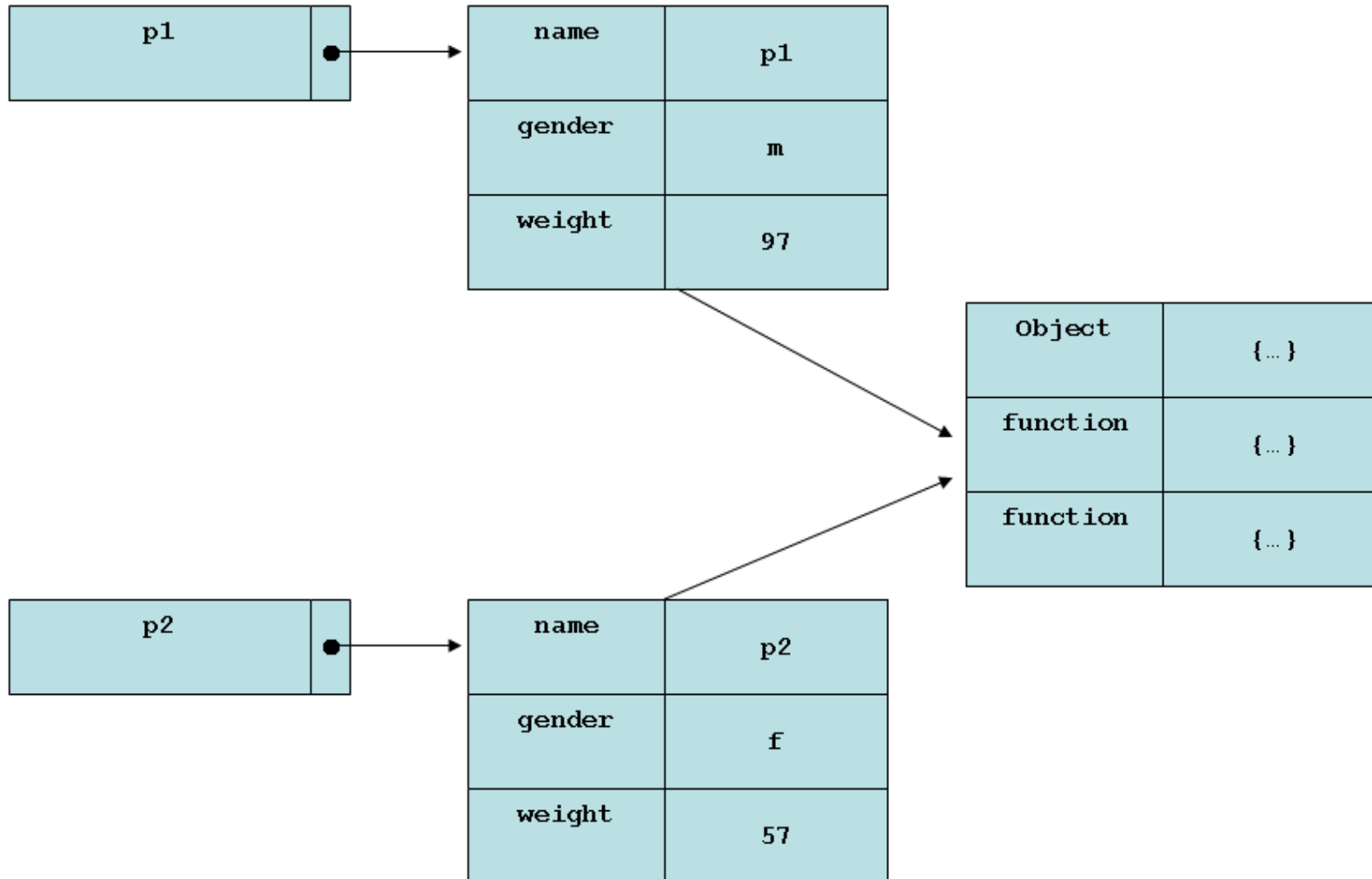


- Diese Verschwendung können wir unterbinden:
  - Jedes Objekt besitzt eine interne weitere Referenz auf sein Prototyp-Objekt
    - Dieses ist ohne weitere Aktionen nichts anderes als ein normales Objekt ohne besondere Eigenschaften und Methoden
- Der ECMAScript-Interpreter berücksichtigt beim Auflösen von Eigenschaften auch das prototype-Objekt
  - "Wenn du die Eigenschaft nicht im Objekt findest, so gehe zum Prototypen"

# Referenz auf das prototype-Objekt



- Der ECMAScript-Interpreter berücksichtigt bei der Suche nach Funktionen jedoch das Prototype-Objekt
  - Der Aufruf `object.myFunction()` ist also erfolgreich wenn
    - entweder `object` unter dem Namen `myFunction` ein Funktionsobjekt besitzt oder
    - `object.prototype` unter dem Namen `myFunction` ein Funktionsobjekt besitzt



## 3.6

# FUNKTIONEN IM DETAIL

- Jedem ECMAScript-Programm wird eine globale Objekt-Instanz zur Verfügung gestellt
  - Dieses ist abhängig von der Laufzeitumgebung, in der das Skript aufgerufen wird
  - In der Praxis ist diese Umgebung in den allermeisten Fällen der Browser
  - Das globale Objekt ist das aktuelle Browser-Fenster repräsentiert durch das `window`-Objekt.



<code>decodeURI ()</code>	<b>Kodierten URI dekodieren</b>
<code>decodeURIComponent ()</code>	<b>Kodierten URI dekodieren - II</b>
<code>encodeURI ()</code>	<b>URI kodieren</b>
<code>encodeURIComponent ()</code>	<b>URI kodieren - II</b>
<code>eval ()</code>	<b>ECMAScript-Code interpretieren</b>
<code>escape ()</code>	<b>Zeichen in Zahlencodes umwandeln</b>
<code>isFinite ()</code>	<b>Auf numerischen Wertebereich prüfen</b>
<code>isNaN ()</code>	<b>Auf numerischen Wert prüfen</b>
<code>parseFloat ()</code>	<b>Text in Kommazahl umwandeln</b>
<code>parseInt ()</code>	<b>Text in Ganzzahl umwandeln</b>
<code>Number ()</code>	<b>auf numerischen Wert prüfen</b>
<code>String ()</code>	<b>in Zeichenkette umwandeln</b>
<code>unescape ()</code>	<b>Zahlencodes in Text umwandeln</b>

- Eine Funktions-Definition der Form
  - `function myFunction() {...}`
- ist in Wahrheit (fast) völlig identisch zu:
  - `"global".myFunction = function() {...}`
- Funktionen werden so dem globalen Objekt zugeordnet.
  - Einem Aufruf einer Funktion wird somit implizit die (nicht existierende) `"global"`-Referenz vorangestellt.
- Dieses Prinzip ist auch für Variablen gültig:
  - Ohne `var` wird implizit bei der Deklaration ein `"global"` vorangestellt
- Können Funktionen auch anderen Objekten zugeordnet werden?
  - Selbstverständlich
  - genau das passiert ja bei der Zuweisung einer Funktion an ein anderes Objekt im Rahmen der Objekt-orientierten Programmierung

- Funktionen sind auch wiederum Objekte, die so wie alle anderen Objekte auch als Variable oder als Parameter benutzt werden können
- Zum Aufruf eines Funktions-Objekts existieren die beiden Methoden `apply` und `call`
  - Beide Funktionen bekommen als ersten Parameter die Referenz, die innerhalb der Methode als `this` benutzt werden soll
  - Die Funktionen unterscheiden sich nur in den weiteren Parametern
    - `apply` verlangt ein Array der Aufrufparameter
    - `call` benutzt eine beliebig lange Parameterliste

- Funktionen können auch innerhalb anderer Funktionen definiert werden
  - Dies haben wir auch bereits bei den Konstruktor-Funktionen gesehen
- Variablen der umhüllenden Funktion bleiben so lange gültig wie die innere Funktion
  - Dies nennt man allgemein „Closures“

- Ein potenziell gefährlicher Effekt von Closures darf nicht verschwiegen werden:
  - Durch die Verlängerung des Gültigkeitsbereiches kann die Garbage Collection die Objekte nicht mehr am Ende des Aufrufs sofort bereinigen
  - Insgesamt wird somit mehr Speicher benötigt
  - durch fehlerhafte Verwendung von Closures kann es sogar dazu führen, dass der Interpreter seine Speicher-Ressourcen erschöpft
    - es kommt zu einem Speicherleck

## 3.7

# FEHLERBEHANDLUNG

- Jeder Aufruf von Programmlogik kann potenziell zu Fehlersituationen führen
- Die Auswertung einer Fehlersituation wird durch einen `try-catch`-Block erfolgen
  - Im `try`-Block steht die normale Anwendungs-Sequenz
  - Im `catch`-Block erfolgt die Fehlerbehandlung mit Zugriff auf ein Fehler-Objekt
    - Das Fehler-Objekt hat die Eigenschaften `message` und `name`, die eine genauere Analyse des Fehlers ermöglichen
- Durch den `try-catch`-Block wird eine saubere Unterteilung zwischen erwartetem Programmablauf und Fehler-Behandlung erreicht.

- Fehler können an allen Stellen des Programms auftreten.
  - So kann beispielsweise eine Division durch 0 erfolgen oder aber ein Objekt benutzt werden, das noch gar nicht initialisiert worden ist.
  - Solche Fehler sind in der Regel jedoch Programmierfehler und werden intern von der ECMAScript-Engine des Browsers erzeugt.
- Möchte eine eigene Anwendung sich ebenfalls in diesen Fehler-Mechanismus einklinken, so erfolgt dies mit Hilfe der Erzeugung eines Error-Objekts sowie dem Werfen dieses Objekts mit `throw`



- Sollen bestimmte Aufgaben sowohl für die normale Ausführung als auch für die Fehlerbehandlung ausgeführt werden kann der `try-catch-Block` durch einen `finally-Block` ergänzt werden

```
function testExceptions() {  
    try {  
        throwException();  
    } catch (error) {  
        console.log("Caught error: message=" + error.message + ",  
name=" + error.name);  
    }  
    finally{  
        console.log("Always");  
    }  
}
```

# Ein komplettes Beispiel

```
function testExceptions() {  
    try {  
        throwException();  
    } catch (error) {  
        console.log("Caught error: message=" +  
error.message + ", name=" +  
        error.name);  
    }  
    finally{  
        console.log("Always");  
    }  
}
```

3.8

## **ECMAScript STANDARD OBJEKTE**

- Array
- Date
- Math
- Reguläre Ausdrücke
- ...
- Seit ES2015 auch Listen und Maps (Dictionaries)

- Die ECMAScript-Bibliothek enthält einen überschaubaren Satz von Fehlern, die im Wesentlichen von der ECMAScript-Engine benutzt werden.
- Diese Laufzeitfehler sind:
  - `EvalError`
  - `ReferenceError`
  - `RaiseError`
  - `RangeError`
  - `SyntaxError`
  - `TypeError`
  - `URIError`

- Die Standard-Objekte sind für den Programmierer eine enorme Erleichterung
  - Zeichenkettenverarbeitung
    - Auch mit komplexen regulären Ausdrücken
  - Rechnen mit Datumswerten
  - Typ-Konversionen
  - Komplexe mathematische Operatoren
  - Listen- und Mengenverarbeitung
    - Bis ES2015 fehlen jedoch noch Maps
      - andere Namen: "Dictionary" oder "Assoziative Arrays"

- Einfach Online zu finden
  - HTML-basiert
- Beschrieben werden alle Funktionen und andere Eigenschaften
- Beispielprogramme, Code-Fragmente und Tutorials stehen ebenfalls zur Verfügung
  - Natürlich auch die Beispiele der elektronischen Musterlösung

## JavaScript Strings

A JavaScript string stores a series of characters like "John Doe".

A string can be any text inside double or single quotes:

```
var carname = "Volvo XC60";  
var carname = 'Volvo XC60';
```

String indexes are zero-based: The first character is in position 0, the second in 1, and so on.

For a tutorial about Strings, read our [JavaScript String Tutorial](#).

## String Properties and Methods

Primitive values, like "John Doe", cannot have properties or methods (because they are not objects).

But with JavaScript, methods and properties are also available to primitive values, because JavaScript treats primitive values as objects when executing methods and properties.

## String Properties

Property	Description
<a href="#">constructor</a>	Returns the string's constructor function
<a href="#">length</a>	Returns the length of a string
<a href="#">prototype</a>	Allows you to add properties and methods to an object

## String Methods

Method	Description
<a href="#">charAt()</a>	Returns the character at the specified index (position)
<a href="#">charCodeAt()</a>	Returns the Unicode of the character at the specified index
<a href="#">concat()</a>	Joins two or more strings, and returns a new joined strings



## Math Object

The Math object allows you to perform mathematical tasks.

Math is not a constructor. All properties/methods of Math can be called by using Math as an object, without creating it.

## Syntax

```
var x = Math.PI;           // Returns PI
var y = Math.sqrt(16);     // Returns the square root of 16
```

For a tutorial about the Math object, read our [JavaScript Math Tutorial](#).

## Math Object Properties

Property	Description
<u>E</u>	Returns Euler's number (approx. 2.718)
<u>LN2</u>	Returns the natural logarithm of 2 (approx. 0.693)
<u>LN10</u>	Returns the natural logarithm of 10 (approx. 2.302)
<u>LOG2E</u>	Returns the base-2 logarithm of E (approx. 1.442)
<u>LOG10E</u>	Returns the base-10 logarithm of E (approx. 0.434)
<u>PI</u>	Returns PI (approx. 3.14)
<u>SQRT1_2</u>	Returns the square root of 1/2 (approx. 0.707)
<u>SQRT2</u>	Returns the square root of 2 (approx. 1.414)

- ECMAScript-Editoren können zumindest eine rudimentäre Code-Vervollständigung anbieten
  - Aber kein Vergleich zu statisch typisierten Sprachen wie Java oder C#
  - Der Editor kann den konkreten Datentyp und damit die anzuzeigenden Möglichkeiten nur "erraten"

# Beispiel: Code-Vervollständigung in Eclipse

```
var message = "Hello World!";  
message.
```

- **charAt(Number position) : String - String**
- charCodeAt(Number position) : Number - String
- concat(String value) : String - String
- <sup>s</sup>fromCharCode(Number charCode) : String - String
- hasOwnProperty(String name) : Boolean - Object
- indexOf(String searchString, Number startPosition) : Number - String
- isPrototypeOf(Object o) : Boolean - Object
- lastIndexOf(String searchString, Number startPosition) : Number - String
- localeCompare(String otherString) : Number - String
- match(RegExp regexp) : any[] - String
- propertyIsEnumerable(Object name) : Boolean - Object

Press 'Ctrl+Space' to show Template Proposals

# 4

## **ECMA2016**

## 4.1

# NEUERUNGEN

- Klassen und Vererbung
- Generator Functions und Proxies
- Scoped Variables und Konstanten
- Neue Collection-Datentypen
  - Set
  - Map
- Vereinfachte Funktionslitterale mit Arrow Functions
- Promise-API

## ECMAScript 6 — New Features: Overview & Comparison

[Tweet](#) [Star](#) 1,652 [Fork me on GitHub](#)

### Constants

- Constants
- Scoping
  - Block-Scoped Variables
  - Block-Scoped Functions
- Arrow Functions
  - Expression Bodies
  - Statement Bodies
  - Lexical this
- Extended Parameter Handling
  - Default Parameter Values
  - Rest Parameter
  - Spread Operator
- Template Literals
  - String Interpolation
  - Custom Interpolation
  - Raw String Access
- Extended Literals
  - Binary & Octal Literal
  - Unicode String & RegExp Literal
- Enhanced Regular Expression
  - Regular Expression Sticky Matching
- Enhanced Object Properties
  - Property Shorthand
  - Computed Property Names

## Constants

### Constants

Support for constants (also known as "immutable variables"), i.e., variables which cannot be re-assigned new content. Notice: this only makes the variable itself immutable, not its assigned content (for instance, in case the content is an object, this means the object itself can still be altered).

#### ECMAScript 6 — syntactic sugar: reduced | traditional

```
const PI = 3.141593
PI > 3.0
```

#### ECMAScript 5 — syntactic sugar: reduced | traditional

```
// only in ES5 through the help of object properties
// and only in global context and not in a block scope
Object.defineProperty(typeof global === "object" ? global : window, "PI", {
  value: 3.141593,
  enumerable: true,
  writable: false,
  configurable: false
})
```

## 4.2

# **BROWSER-KOMPATIBILITÄT**



- Praktisch kein aktueller Browser unterstützt den kompletten ECMAScript-Standard
  - Anwendungen, die für das freie Internet konzipiert sind, müssen auch noch alte Versionen unterstützen
- Allerdings können einige Features von ECMA2016 auch in klassischem JavaScript abgebildet werden
  - Allerdings deutlich umständlicher
  - Beispiel: Klasse
    - Eine Klassendefinition kann durch das "Module-Pattern" umgesetzt werden

- Der Vorgang des Umsetzens von einer Script-Sprache in eine andere wird als "Transpilation" bezeichnet
- Im JavaScript-Umfeld existieren verschiedene Transpiler-Ansätze:
  - TypeScript
    - Eine eigene Programmiersprache mit ECMAScript-ähnlicher Syntax aber mit der Möglichkeit einer statischen Typisierung
      - Entwicklungsumgebungen können somit verbesserte Code-Assists anbieten
    - Zur Ausführung wird dann daraus JavaScript transpiliert
- Babel
  - Transpiler nach JavaScript für verschiedene Sprachen
  - Benutzung
    - Im Build-Prozess wird JavaScript generiert und vom Server bereitgestellt
    - Einbinden des Babel-Scripts in der HTML-Seite des Browsers und Transpilation "on the fly"

## 4.3

# KLASSEN

```
class Book{
    constructor(isbn, title) {
        this.title = title;
        this.isbn = isbn;
    }
    get isbn() {
        return this.isbn;
    }
    get title() {
        return this.title;
    }
    set title(value) {
        this.title = value;
    }
    info() {
        return "Book: isbn=" + isbn + ", title=" + title;
    }
}
```

```
class SchoolBook extends Book{  
  constructor(isbn, title, topic){  
    super(isbn, title);  
    this.topic = topic;  
  }  
  
  info(){  
    return super.info + ", topic=" + topic;  
  }  
}
```

4.5

## **SCOPED VARIABLES UND KONSTANTEN**

- `let` beschränkt den Gültigkeitsbereich einer Variable auf den deklarierenden Scope
  - Also beispielsweise einem Block einer Schleife
- `const` deklariert eine Konstante

4.6

## **COLLECTIONS**



- Eine Map besteht aus key-value-Paaren
  - In anderen Sprachen als Dictionary oder assoziatives Array bezeichnet

```
map = new Map(); //oder mit Vorbelegung
map = new Map(['key1', 'value1'], ['key2', 'value2']);
map.set('key', 'value');
map.get('key');
map.size;
map.clear();
```

- Iteration

```
for (let key of map.keys()) {}
for (let value of map.values()) {}
```

- Eine Set besteht aus Unikaten
  - In anderen Sprachen als Dictionary oder assoziatives Array bezeichnet

```
var set = new Set();  
set.add("Hugo")  
set.add("Emil")  
set.add("Hugo")  
set.has("Hugo")  
set.size; // -> 2
```

4.7

## **VEREINFACHTE FUNKTIONSDEKLARATION**

- Eine vereinfachte Schreibweise für Funktions-Definitionen
  - beispielsweise für Parameter-Übergabe

```
(res) => console.log(res + " at " + new Date())
```

## 4.4

# GENERATORS UND PROXIES

- Generators sind spezielle Funktionen, die den Kontrollfluss an die aufrufende Funktion zurück delegieren
  - Dafür wird die `yield`-Funktion eingeführt

```
function* sampleGenerator() {  
    print('First');  
    yield("Hugo");  
    print('Second');  
};  
  
//...  
  
let gen = sampleGenerator();  
print(gen.next());  
print(gen.next());
```

- Proxies erweitern ("dekorieren") bereits vorhandene Funktionen und Objekte
- Dieses Design-Pattern ist in untypisierten Sprachen sehr einfach umzusetzen

```
var handler = {  
    get: function (target, name)  
        return Reflect.get(target, name) },  
    apply: function (receiver, ...args) {  
print("applying...") }  
};  
//...  
obj = new Proxy(obj, handler);
```

4.8

## **PROMISES**



- Das Promise-API ordnet verschachtelte Callback-Funktionen in eine Sequenz von Funktionsaufrufen

```
var p = new Promise(function(resolve, reject)
{   setTimeout(() => resolve(4), 2000);});
p.then((res) => {   res += 2;   console.log(res + " at "
+ new Date());});
p.then((res) => console.log(res + " at " + new
Date()))
```