Web-Entwicklung

JavaScript

Inhalte der Vorlesung

- JavaScript
  - Kommentare
  - Variablen und Datentypen
  - Operatoren
  - Funktionen
  - Kontrollstrukturen
  - Fehlerbehandlung

Kommentare

```
// Das ist ein einzeiliger Kommentar.


/* Das
ist
ein
mehrzeiliger
Kommentar. */
```

Variablen und Datentypen

Dynamische Typisierung

- Statische Typisierung (z.B. in Java oder C#): Datentypen aller Ausdrücke sind bereits zur Compile-Zeit bekannt
  - Überprüfungen im Zusammenhang mit dem Datentyp können bereits zur Compile-Zeit durchgeführt werden, z.B. Existenz von Methoden, die aufgerufen werden sollen
- Dynamische Typisierung (z.B. in JavaScript): Datentypen von Ausdrücken sind erst zur Laufzeit bekannt
  - Werte haben einen Typ, aber nicht Variablen!
  - Eine Variable kann im Verlauf auf Werte unterschiedlichen Typs zeigen

Variablen

- Variablendeklaration mithilfe des Schlüsselworts `var`
- Variablenbezeichner
  - beginnen mit einem Buchstaben, Unterstrich oder Dollar-Zeichen, anschließend folgen Buchstaben, Ziffern und Unterstriche
  - sind case-sensitive
- Globale Variablen
  - Lässt man das Schlüsselwort `var` weg, wird die Variable zu einer globalen Variable
  - Eigenschaft des sog. global object
  - Überschreibt ggf. bereits vorhandene, globale Variable!

```
var value = 42;
value = "Zweiundvierzig";   // bad idea


globalVar = "...";          // bad idea
```

Datentypen

- Primitive Datentypen
  - `Number`
  - `String`
  - `Boolean`
- Spezielle primitive Datentypen
  - `Undefined`
  - `Null`
- Referenztypen
  - `Object` (Assoziative Arrays)
    - `Array` (Arrays)
    - `Function` (Funktionen)
    - …

typeof

- `typeof`-Operator liefert Datentyp des Operanden als String
- Weitergehende Möglichkeiten zur Typüberprüfung später

Typ	**Rückgabewert**
Number	`"number"`
String	`"string"`
Boolean	`"boolean"`
Undefined	`"undefined"`
Null	`"object" (!)`
Function	`"function"`
Alle anderen Objekte	`"object"`

Number

- Nur ein Datentyp für ganze und nicht-ganze Zahlen
- `Number` repräsentiert eine 64 Bit-Gleitkommazahl gemäß IEEE 754
  - Entspricht dem Datentyp `double` in Java
  - Kein Datentyp für 32 Bit-Fließkommazahlen (`float`)
- Wertebereich zwischen `Number.MIN_VALUE` und `Number.MAX_VALUE`

```
var n = 1024;
console.log(typeof n);        // number

var r = 3.141;
console.log(typeof r);        // number

var hex = 0xff;
console.log(hex);             // ?
```

Number

- Besondere Werte:
  - `Infinity`

    Repräsentiert die Unendlichkeit, z.B. bei Übertreten des Wertebereichs oder bei Division durch 0
  - `NaN`

    Für andere ungültige Zahlenwerte nimmt die Variable den Wert `NaN` (für *Not a Number*) an

```
var x = Number.MAX_VALUE * 2;
console.log(x);     // Infinity

var y = 1 / 0;
console.log(y);     // Infinity


var z = Math.sqrt(-1);
console.log(z);               // NaN
console.log(isNaN(z));        // true
```

Number

- Globale Funktionen zur Umwandlung von Zeichenketten in `Number`-Objekte
  - `parseInt(string, radix)`
  - `parseFloat(string)`
- Funktionen parsen maximal-langen, gültigen Präfix
  - Teil-Zeichenkette ab erstem ungültigen Zeichen werden ignoriert
  - Whitespace-Präfix wird ignoriert

```
parseInt(" 0xF", 16);
parseInt("F", 16);
parseInt("17", 8);
parseInt("015", 10)
parseInt(15.99, 10);
parseInt("15,123", 10);

parseFloat("3.14");
parseFloat("314e-2");
parseFloat("0.0314E+2");
parseFloat("3.14more non-digit characters");
```

String

- UTF-16-kodierte Unicode-Zeichenketten
  - `length` liefert Anzahl Bytes zurück!
- Unveränderlich (*immutable*)
- Einfache oder doppelte Anführungszeichen
  - Erlaubt Verwendung des jeweils anderen Zeichens ohne Escape-Sequenz innerhalb der Zeichenkette
- Kein Datentyp für Einzelzeichen
  - `charAt()` bzw. `[]`-Operator liefern String-Objekt
- Methoden: https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/String

```
var tamil = "This is தமிழ்";

var greeting = 'I ❤️ "JavaScript"';

console.log(typeof tamil.charAt(0));  // string

console.log("Hello".length);   // 5
console.log(tamil.length);     // 13!
```

Boolean

- Wahrheitswert `true/false`
- Alle Ausdrücke können vom Interpreter zu einem Wahrheitswert evaluiert werden
  - `false`
    - Leere Strings
    - `0`
    - `null`
    - `undefined`
    - `NaN`
  - `true`: Alle anderen Ausdrücke
  - Erlaubt häufig stark verkürzte Bedingungen

Undefined - Null

- Datentypen mit jeweils nur einem möglichen Wert
- Beide Werte drücken aus, dass eine Variable nicht belegt ist
- `undefined`
  - Nicht initialisierte Variablen
  - Nicht existente Objekteigenschaften
  - Nicht übergebene Funktionsargumente
  - Rückgabe von Funktionsaufrufen ohne Rückgabewert
- `null`
  - Beabsichtigte Nichtbelegung einer Variablen durch den Entwickler

```
var u;
console.log(u, typeof u);   // undefined 'undefined'

var v = 500;
console.log(v.nonExistentProperty, typeof
   v.nonExistentProperty);  // undefined 'undefined'


var w = null;
console.log(w, typeof w);   // null 'object'
```

Arrays

- Container-Datentyp für Objekte beliebigen Typs
- Numerischer, 0-basierter Schlüssel
- Erzeugung über Literal-Kurzschreibweise oder sog. Konstruktorfunktion
  - Ein `Number`-Argument: Array-Länge, Werte: `undefined`
  - Mehrere Argumente: Array-Inhalte
- Lesender und schreibender Zugriff über Index-Operator

```
var names = [“Tim", “Struppi"];

var otherNames = new Array();
otherNames[0] = “Max Moritz";
otherNames[1] = "Vincent";
otherNames[2] = 0;

var tenItems = new Array(10);
var threeItems = new Array(1, 2, 3);
```

Arrays

- Gängige Methoden
  - `concat()`
  - `filter()`
  - `join()`
  - `map()`
  - `pop()`
  - `push()`
  - `reverse()`
  - `shift()`
  - `slice()`
  - `splice()`
  - `sort()`
  - …
- https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Global_Objects/Array

Objekte – Assoziative Arrays

- Objekte in JavaScript sind assoziative Arrays, d.h. Container für Schlüssel-Wert-Paare
  - Schlüssel sind im Gegensatz zum Array nicht numerisch-aufsteigend, sondern alphanumerisch
  - Werte sind beliebige Objekte
- Ein Schlüssel-Wert-Paar nennt man auch
  - Eigenschaft (*Property*)
  - Methode, falls Wert eine Funktion ist
- Erzeugung von Objekten mithilfe des sog. Objekt-Literals
  - Weitere Erzeugungsmöglichkeiten später
- Lesender und schreibender Zugriff auf Eigenschaften oder Methoden über Punkt- oder Index-Operator

Objekt-Literal

```
var phonebookEntry = {
   firstName: "Hans",
   lastName: "Wurst",
   number: "0123-456789"
};

console.log(phonebookEntry.number);              // 0123-456789
console.log(phonebookEntry["number"]);           // 0123-456789


phonebookEntry.firstName = "John";               // { firstName: 'John',
phonebookEntry["lastName"] = "Doe";              //   lastName: 'Doe',
                                                 //   number: '0123-456789' }
console.dir(phonebookEntry);
```

Objekt-Literal

```
var phonebookEntry2 = {
   firstName: "Hans",
   lastName: "Wurst",
   number: "0123-456789",
   print: function() {
      console.log("%s, %s: %s",
         this.firstName,
         this.lastName,
         this.number);
   }
};

phonebookEntry2.print();   // Hans, Wurst: 0123-456789
```

Enumerationen

- Assoziative Arrays können auch zur Emulation von Enumerationen genutzt werden

```
var Color = {
   RED: 0,
   GREEN: 1,
   BLUE: 2
};
console.log(Color.RED);

var color = Color.BLUE;
console.log(color === Color.GREEN);
```

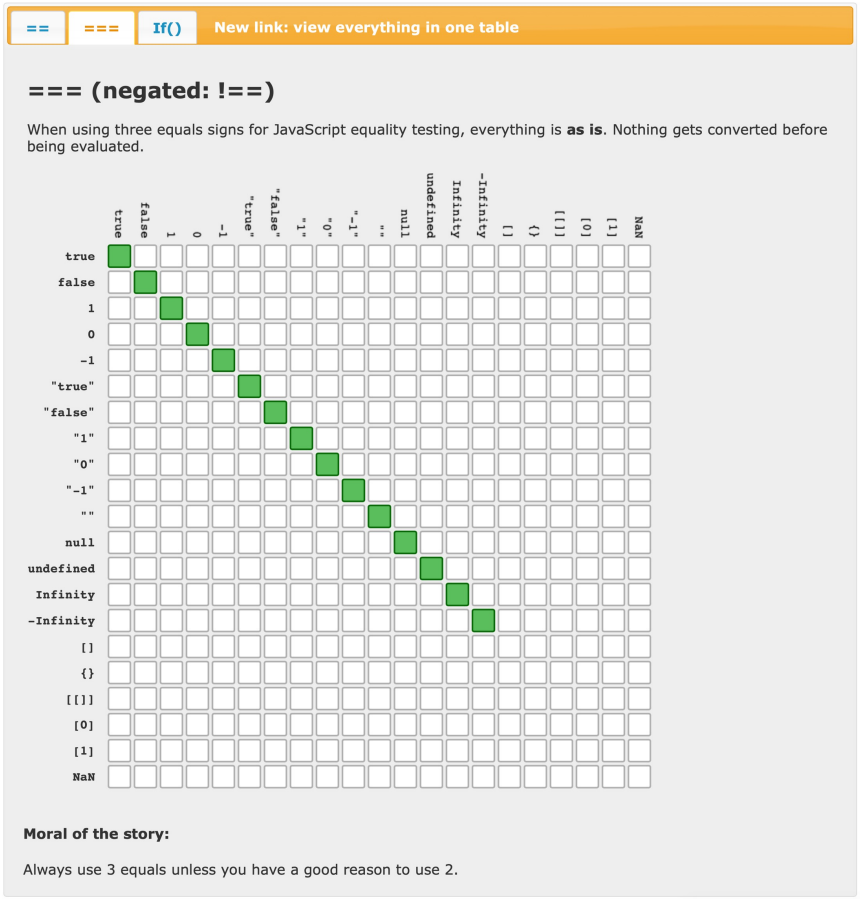
```
var Color2 = {
   RED: "RED",
   GREEN: "GREEN",
   BLUE: "BLUE"
};
console.log(Color2.RED);

var Color3 = {
   RED: 0xFF0000,
   GREEN: 0x00FF00,
   BLUE: 0x0000FF
};
console.log(Color3.RED);
```

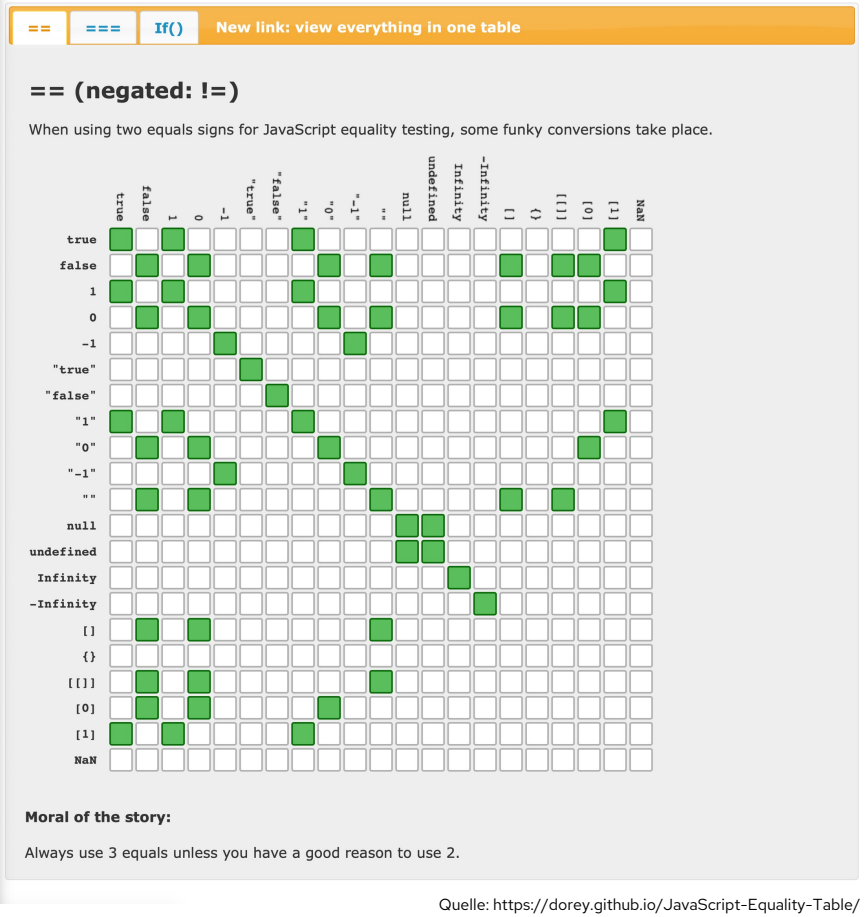
Operatoren

Vergleichsoperatoren

Operation	Operator	Beschreibung
Strikte Gleichheit	===	true, falls Operanden gleich sind und Datentypen übereinstimmen (weil eine implizite Typumwandlung nicht erlaubt ist)
Strikte Ungleichheit	!==	
Gleichheit	==	true, falls Operanden gleich sind (Implizite Typumwandlung erlaubt)
Ungleichheit	!=	
Größer	>	
Größer oder Gleich	>=	
Kleiner	<	
Kleiner oder Gleich	<=	

Vergleichsoperatoren





Quelle: https://dorey.github.io/JavaScript-Equality-Table/

Arithmetische Operatoren

Operation	Operator	Beschreibung
Modulo	%	Rest der ganzzahligen Division der Operanden
Inkrement	++ (Präfix oder Postfix)	Erhöht den Wert des Operanden um 1
Dekrement	-- (Präfix oder Postfix)	Senkt den Wert des Operanden um 1

Logische Operatoren

Operation	Operator	Beschreibung
Log. UND	&&	Liefert ersten Operanden zurück, falls dieser zu false evaluiert wird, sonst den zweiten Operanden
Log. ODER	||	Liefert ersten Operanden zurück, falls dieser zu true evaluiert wird, sonst den zweiten Operanden
Log. NICHT	!	Negiert den zu einen Wahrheitswert evaluierten Operanden

Bitweise Operatoren

Operation	Operator
Bitw. UND	&
Bitw. ODER	|
Bitw. XOR	^
Bitw. NICHT	~
Bitw. Linksverschiebung	<<
Bitw. Rechtsverschiebung	>>
Bitw. Rechtsverschiebung ohne Berücksichtigung des Vorzeichens	>>>

Weitere Operatoren

Operation	Operator	Beschreibung
Ternärer Bedingungsoperator	`<Bedingung>?<Wert1>:<Wert2>`	Liefert <Wert1>, falls Bedingung zu true evaluiert wird, sonst <Wert2>
Löschoperator	`delete`	Löscht Objekt-eigenschaften oder Array-Elemente
Eigenschaftsexistenz-operator	`<Eigenschaft> in <Objekt>`	true, falls <Objekt> die <Eigenschaft> besitzt (traversiert Prototypenkette)
Typüberprüfung	`<Objekt> instanceof <Typ>`	true, falls <Objekt> vom <Typ> ist
Typbestimmung	`typeof <Operand>`	Liefert Typ des Operanden (String)

Funktionen

Funktionsanweisung

- Deklaration von Funktionen mithilfe des Schlüsselworts `function`
- Keine Typangaben bei Argumenten oder Rückgabe
  - Keine Typsicherheit

```
function add(arg1, arg2) {
   return arg1 + arg2;
}

console.log(add(1, 2));                    // 3
console.log(add("Hello", "World"));        // HelloWorld
```

Funktionsanweisung

- Typsicherheit kann erst zur Laufzeit hergestellt werden

```
function add2(arg1, arg2) {
   if (typeof(arg1) !== "number" || typeof(arg2) !== "number") {
      throw new TypeError('Arguments of "add" must be of type "number"');
   }
   return arg1 + arg2;
}

console.log(add2("Hello", "World"));                // TypeError
```

Funktionsausdruck

- Alternative: Erstellung einer (anonymen) Funktion und Zuweisung zu einer Variablen
- Im Gegensatz zu Funktionsanweisungen stehen solche Funktion erst nach der Zuweisung zur Verfügung

```
var add3 = function(arg1, arg2) {
   return arg1 + arg2;
};

console.log(add3(1, 2));          // 3
```

Function-Objekte

- Funktionen werden durch Objekte repräsentiert
- Eigenschaften
  - `name`
  - `length`
  - …
- Funktionsanweisung erzeugt Instanz und gleichnamige Variable

```
function add(arg1, arg2) {
   return arg1 + arg2;
}
```

Funktionen als First Class Objects

- Funktionen sind First Class Objects, d.h. sie können wie Objekte primitiver Datentypen
  - einer Variablen zugewiesen werden

```
function add(arg1, arg2) {
   return arg1 + arg2;
}

var plus = add;
console.log(add(1, 2));
console.log(plus(1, 2));
```

Funktionen als First Class Objects

- Funktionen sind First Class Objects, d.h. sie können wie Objekte primitiver Datentypen
  - als Rückgabe einer Funktionen dienen
- Erstellung und Rückgabe einer anonymen Funktion

```
function createPow(exponent) {
   return function (x) {
      var result = x;
      for (var i = 1; i < exponent;
         i++) {
         result *= x;
      }
      return result;
   };
}

console.log(createPow(10)(2));   // ?
console.log(createPow(3).name);  // ?
```

Funktionen als First Class Objects

- Funktionen sind First Class Objects, d.h. sie können wie Objekte primitiver Datentypen
  - als Argument einer Funktionen dienen
- z.B. in Array-Methoden
  - `forEach()`
  - `map()`
  - `filter()`
  - `every()`
  - `some()`
  - `reduce()`
  - …
- Später: Callback-Entwurfsmuster, Events

```
function isEven(element, index, array) {
   return element % 2 === 0;
}

var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

console.log(numbers.filter(isEven));
```

Funktionsargumente

- Beim Aufruf von Funktionen kann die Anzahl der aktuellen Argumente von der Anzahl der formalen Parameter abweichen
- Implizites Array-ähnliches Objekt `arguments` enthält innerhalb von Funktionen alle übergebenen Argumente
  - `length`-Eigenschaft
  - Index-Operator

```
function sum() {
   var result = 0;
   for (var i=0;i<arguments.length;i++) {
      result += arguments[i];
   }
   return result;
}

console.log(sum(1, 2, 3));       // 6
console.log(sum(1, 2, 3, 4, 5)); // 15
```

Funktionsargumente

- Überladen von Funktionen ist nicht möglich
- Realisierung optionaler Argumente möglich durch
  - Weglassen der optionalen Argumente bei
    - Aufruf
    - Deklaration
  - Konfigurationsobjekt

Funktionsargumente

- Angabe optionaler Argumente nach nicht-optionalen Argumenten
- Aufruf mit oder ohne optionale Argumente möglich

- Nachteile:
  - Umständlich, falls mehrere optionale Argumente existieren, aber nicht alle beim Aufruf benötigt werden

```
function add4(a, b, log) {
   var result = a + b;
   if (log) {
      console.log(result);
   }
   return result;
}

add4(5, 2, true);         // 7
add4(5, 2, false);        //
add4(5, 2);               //
```

Funktionsargumente

- Weglassen der optionalen Argumente bei Deklaration
- Optionale Argumente aus `arguments`-Objekt auslesen

- Nachteile:
  - Wie zuvor
  - Funktionsdeklaration lässt nicht erkennen, dass evtl. weitere Parameter erwartet werden

```
function add5(a, b) {
   var result = a + b;
   if (arguments[2]) {
      console.log(result);
   }
   return result;
}

add5(5, 2);              //
add5(5, 2, true);        // 7
add5(5, 2, false);       //
```

Funktionsargumente

- Konfigurationsobjekt
  - *best practice*
  - Zusammenfassung ggf. vorhandener optionaler Argumente in einem assoziativen Array
  - Übergabe des Arrays als letztes, aber einziges optionales Argument der Funktion
    - Nur benötigte Parameter werden explizit gesetzt

```
function add6(a, b, config) {
   var result = a + b;
   if (config && config.log) {
      console.log(result);
   }
   return result;
}

add6(5, 2, { log: true });  // 7
add6(5, 2, { log: false }); //
add6(5, 2);                 //
```

Ausführungskontext

- Innerhalb einer Funktion verweist `this` auf ein Objekt, den sog. Ausführungskontext
- `this` wird dynamisch beim Funktionsaufruf ausgesetzt
  - Aufruf als Funktion
  - Aufruf als Methode
  - Aufruf mit explizitem Kontext
  - Aufruf als Konstruktor
    - Siehe nächste Vorlesung

```
function getName() {
   return this.name;
}

var heinz = {
   name: "Heinz",
   getName: getName
};

var anna = {
   name: "Anna",
   getName: getName
};

console.log(getName());         // ?
name = "Michael";               // globale Variable
console.log(getName());         // ?
console.log(heinz.getName());   // ?
console.log(anna.getName());    // ?
```

Ausführungskontext

- Häufige Fehlerquelle, da Kontext nicht dem erwarteten entspricht
  - z.B. bei Callbacks/Event Listener

- Hier:
  - Kontext der Funktion `onClick()` ist das Objekt `button`, d.h. `this` verweist innerhalb der Funktion `onClick()` auf das Objekt `button`
  - `button.name` ist undefiniert

```
var button = {
   onClick: null,
   click: function () {
      if (typeof this.onClick === "function") {
         this.onClick();
      }
   }
};

var handler = {
   name: "MyClickHandler",
   onClick: function () {
      console.log("Click handled by: %s", this.name);
   }
};

button.onClick = handler.onClick;
button.click();  // Click handled by: undefined
```

Ausführungskontext

- Aufruf von `bind()` auf einer Funktion erzeugt neues Funktionsobjekt mit identischer Funktionalität, aber explizit vorgegebenem Kontext
  - Kontext wird als Argument übergeben
  - Funktion wird nicht aufgerufen

```
var button = {
   onClick: null,
   click: function () {
      if (typeof this.onClick === "function") {
         this.onClick();
      }
   }
};

var handler = {
   name: "MyClickHandler",
   onClick: function () {
      console.log("Click handled by: %s", this.name);
   }
};

button.onClick = handler.onClick.bind(handler);
button.click();  // Click handled by: MyClickHandler
```

Ausführungskontext

- Aufruf von `call()` auf einer Funktion führt diese Funktion unmittelbar aus
  - Kontext wird als erstes Argument übergeben
  - Parameter der Funktion können ggf. als folgende Argumente übergeben werden

- Aufruf von `apply()` auf einer Funktion führt diese Funktion unmittelbar aus
  - Kontext wird als erstes Argument übergeben
  - Parameter der Funktion können ggf. in einem Array zusammengefasst als zweites Argument übergeben werden

Gültigkeitsbereiche

- Die Gültigkeit einer Variablen wird nicht durch Blöcke begrenzt (*block scope*), sondern durch Funktionen (*function level scope*)
- Innerhalb einer Funktion definierte Variablen sind in der gesamten Funktion sichtbar
  - Auch in inneren Anweisungsblöcken definierte Variablen
  - Auch vor der Deklaration
    - *Hoisting*: Implizites "Anheben" der Deklaration aller Variablen an den Beginn des Funktionsrumpfes
- In inneren Funktion deklarierte Variablen sind in der äußeren Funktion nicht sichtbar (eigener Gültigkeitsbereich der inneren Funktion)

Gültigkeitsbereiche

```
function testScope() {
   if (true) {
      var m = 1337;
   }
   for (var i = 0; i < 1024; i++) {
      // do something
   }
   console.log(m);                // 1337
   console.log(i);                // 1024
}
```

Gültigkeitsbereiche

```
function testScope() {
   …
   console.log(k);                  // undefined
   var k = 5503;                    // Hoisting
   console.log(k);                  // 5503

   console.log(j);                  // ReferenceError
}
```

Gültigkeitsbereiche

```
function testScope() {
   …
   var n = 2048;
   var innerFunction = function() {
      console.log(n);
      var innerVar        = true;
   };
   innerFunction();       // 2048
   console.log(innerVar); // ReferenceError
}
```

Kontrollstrukturen

Sequenz

- Anweisungen werden durch Semikola getrennt
  - Anschließender Zeilenumbruch i.d.R. empfehlenswert
- *Automatic Semicolon Insertion*
  - Führt gelegentlich zu überraschenden Effekten
  - Laut Spezifikation eine Maßnahme zur Fehlerkorrektur
    - Erhöhung der Toleranz ggü. Programmierfehlern(!)
  - https://brendaneich.com/2012/04/the-infernal-semicolon/

```
console.log("Hello, World!")
console.log("Hello, World!");
console.log("Hello, "); console.log("World!")
```

Bedingte Verzweigung: if

```
var h = new Date().getHours();
console.log("Stunde: %d", h);

if (h < 9) {
    console.log("Morgen");
}
else if (h < 12) {
    console.log("Vormittag");
}
else {
    console.log("Zeit zum Aufstehen");
}
```

Bedingte Verzweigung: if

- Erinnerung: Alle Werte können zu Wahrheitswerten evaluiert werden
  - Ermöglicht oft kompaktere Schreibweise

```
function log(msg) {
   if (msg !== undefined && msg !== null && msg.length !== 0) { console.log(msg); }
}

function log2(msg) {
   if (msg && msg.length) { console.log(msg); }
}

log2(undefined); log2(null); log2("");           //
log2("Hello");                                   // Hello
```

Bedingte Verzweigung: switch-case

```
var month = new Date().getMonth() + 1;

switch (month)  {
   case 3:
   case 4:
   case 5:
      console.log("Frühling");
      break;
   case 6:
   case 7:
   case 8:
      console.log("Sommer");
      break;
   case 9:
   case 10:
   case 11:
      console.log("Herbst");
      break;
   case 12:
   case 1:
   case 2:
      console.log("Winter");
      break;
   default:
      console.error("WTF?!");
}
```

Schleifen: while, do-while

```
var i = 1;

while (i <= 50) {
   console.log(i);
   i++;
}
```

Ausgabe:

```
1
2
3
...
50
```

```
var i = 1;

do {
   console.log(i);
   i++;
} while (i <= 50);
```

Ausgabe:

```
1
2
3
...
50
```

Schleifen: for

```
for (var i = 1; i <= 50; i++) {
   console.log(i);
}
```

Ausgabe:

1

2

3

...

50

```
for (var i = 1; i <= 10; i++) {
   var line = "";
   for (var j = 1; j <= 10; j++) {
      line += ((j*i) + "\t");
   }
   console.log(line);
}
```

Ausgabe?

Schleifen: for

- Enthält das Abbruchkriterium den Zugriff auf eine Objekteigenschaft sollte dessen Wert in der Schleifeninitialisierung zwischengespeichert werden
- Erhöht Performanz, z.B. beim Iterieren über Objekte vom Typ
  - `Array`
  - `HTMLCollection`

```
var a = ["a", "b", "c", "d", "e", ...];

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

Schleifen: for..in

- `for..in`-Schleife iteriert über alle aufzählbaren Eigenschaftsschlüssel eines Objekts
  - Iteriert in willkürlicher Reihenfolge, i.d.R. also nicht für Arrays geeignet
  - Iteriert auch über „geerbte“ Eigenschaften (genauer: Eigenschaften der Prototypen)
- Spätere Vorlesungseinheit: `for..of`-Schleife

```
var pbEntry = {
   firstName: "Hans",
   lastName: "Wurst",
   number: "0123-456789",
   print: function() {}
};

for (var key in pbEntry) {
   console.log("%s: %s (%s)", key, pbEntry[key], typeof pbEntry[key]);
}
```

Schleifen: continue

- Schlüsselwort `continue` springt unmittelbar zur nächsten Iteration

```
for (var i = 1900; i <= 2000; i++) {
   if (((i % 4 === 0) && (i % 100 !== 0)) || (i % 400 === 0)) {
      continue;
   }
   console.log(i);
}
```

Ausgabe?

Schleifen: break

- Schlüsselwort `break` springt unmittelbar hinter die aktuelle Schleife

```
function search(haystack, needle) {
   var found = false;
   for (var i = 0; i < haystack.length; i++) {
      if (haystack[i] === needle) {
         found = true;
         break;
      }
   }
   return found;
}

console.log(search([1, 2, 3, 4, 5], 3));
```

Fehlerbehandlung

Fehler werfen: throw

- Laufzeitfehler können mithilfe des Schlüsselworts `throw` geworfen werden

```
throw new Error("Error message");
```

- Objekt hinter `throw` sollte von einem Standardfehlertyp oder von einem eigens davon abgeleiteten Typ sein
  - Grundsätzlich können sogar beliebige Objekte geworfen werden

- Standardfehlertypen
  - `Error`
    - `EvalError`
    - `SyntaxError`
    - `RangeError`
    - `TypeError`
    - `ReferenceError`
    - `URIError`

Fehler abfangen: try-catch-finally

- Fehlerabhang mithilfe eines `try-catch-finally`-Konstrukts
- Nur ein `catch`-Block
- `finally`-Block ist optional und wird immer ausgeführt, unabhängig davon, ob innerhalb des `try`-Blocks ein Fehler geworfen wurde oder nicht

```
try {
  // do something that may
  // throw an error, e.g.
  var i = 42;
  console.log(i.toUpperCase());
}
catch (error) {
  console.error("Unable to print i:", error.message);
}
finally {
  console.log("Done.");
}
```

Fragen?