WBE: JAVASCRIPT

OBJEKTE UND ARRAYS

ÜBERSICHT

- Objekte
- Spezielle Objekte: Arrays
- Werte- und Referenztypen
- Vordefinierte Objekte
- JSON

ÜBERSICHT

- Objekte
- Spezielle Objekte: Arrays
- Werte- und Referenztypen
- Vordefinierte Objekte
- JSON

OBJEKTE UND ARRAYS

- Objekte: Werte zu Einheiten zusammenfassen
- Arrays: Objekte mit speziellen Eigenschaften

Was	Objekt	Array
Art	Attribut-Wert-Paare	Sequenz von Werten
Literalnotation	werte = { a: 1, b: 2 }	liste = [1, 2, 3]
Ohne Inhalt	werte = { }	liste = [ ]
Elementzugriff	werte["a"] oder werte.a	liste[0]

OBJEKTLITERALE

```
1 let person = {
2   name: "John Baker",
3   age: 23,
4   "exam results": [5.5, 5.0, 5.0, 6.0, 4.5]
5 }
```

- Sammlung von Attributen und Werten
- Attributname und Wert durch Doppelpunkt getrennt
- Attribut-Wert-Paare durch Kommas getrennt
- Attributname als String, wenn es kein gültiger Name ist

Speaker notes

Objektliterale werden mit geschweiften Klammern `{...}` geschrieben. Geschweifte Klammern haben also zwei Bedeutungen in JavaScript: einerseits umfassen sie Anweisungsblöcke, andererseits bilden sie Objekte.

Attributnamen *können* als String geschrieben werden. Sie *müssen* als String geschrieben werden, wenn es keine gültigen JavaScript-Namen sind.

Als Attributwerte sind alle Datentypen zulässig, Strings, Zahlen, Wahrheitswerte, aber auch Arrays (wie im Beispiel) oder Objekte. Auch Funktionen können Attributwerte sein. In diesem Fall spricht man von *Methoden*.

Nach dem letzten Attribut-Wert-Paar muss kein Komma stehen. Man kann dieses aber einfügen, um zu vermeiden, dass beim Einfügen eines weiteren Attributs ins Objektliteral das Komma vor dem neuen Attribut vergessen geht:

```
let person = {
  name: "John Baker",
  age: 23,
  "exam results": [5.5, 5.0, 5.0, 6.0, 4.5],
  /* Objekt wird ggf. noch ergänzt */
}
```

ZUGRIFF AUF ATTRIBUTE

```
1 let person = {
2   name: "John Baker",
3   age: 23,
4   "exam results": [5.5, 5.0, 5.0, 6.0, 4.5]
5 }
6
7 console.log(person.name)       /* → John Baker  */
8 console.log(person["age"])     /* → 23          */
9 console.log(person["exam"])    /* → undefined   */
```

- Punkt- oder Klammernotation zum Zugriff
- Punktnotation: Attribut muss gültiger Name sein
- Zugriff auf nicht vorhandenes Attribut liefert `undefined`

Speaker notes

Wenn die Notation mit eckigen Klammern verwendet wird, kann ein beliebiger String oder eine Zahl als Attribut verwendet werden. In diesem Fall kann auch ein Ausdruck eingesetzt werden, der ausgewertet wird und das Attribut liefert.

`person.x` greift auf auf das Attribut "x" von person zu, `person[x]` wertet zunächst `x` aus und nimmt das Ergebnis als Attributname.

OPTIONAL CHAINING

```
1 const adventurer = {
2   name: 'Alice',
3   cat: { name: 'Dinah' }
4 }
5
6 console.log(adventurer.dog.name)                    /* → TypeError */
7 console.log(adventurer.dog && adventurer.dog.name)  /* → undefined */
8 console.log(adventurer.dog?.name)                   /* → undefined */
```

- Eingeführt mit ECMAScript 2020
- Verschiedene weitere Möglichkeiten
  (s. Optional Chaining)

ATTRIBUTE HINZUFÜGEN

- Objekte sind dynamische Datenstrukturen
- Sie können jederzeit erweitert werden

```
> let obj = { message: "not yet implemented" }
> obj.ready = false

> obj
{ message: 'not yet implemented', ready: false }

> obj.attr
undefined
```

ATTRIBUTE ENTFERNEN

- Objekte können jederzeit verkleinert werden
- Mit `delete` wird ein Attribut entfernt
- Mit `in` kann überprüft werden, ob ein Attribut existiert

```
> let obj = { message: "ready", ready: true, tasks: 3 }
> delete obj.message
> obj.tasks = undefined

> obj
{ ready: true, tasks: undefined }

> "message" in obj
false
> "tasks" in obj
true
```

Speaker notes

Es ist also ein Unterschied, ob ein Attribut auf `undefined` gesetzt oder gelöscht wird. Im ersten Fall ist es immer noch vorhanden, im zweiten Fall nicht mehr.

Noch ein kleiner Ausblick: `attr in obj` liefert auch `true`, wenn `attr` geerbt wird, das heisst irgendwo in der Prototypenkette von `obj` vorkommt. Das wird zwar erst beim Thema *Prototypen von Objekten* behandelt, aber wenn Sie bereits einen Blick darauf werfen wollen, sehen Sie sich dieses Beispiel an:

```
> let protoObj = {attr:15}
> let obj = Object.create(protoObj)
> obj
{}
> obj.attr
15
> "attr" in obj
true
```

METHODEN

- Attribute, deren Werte Funktionen sind, werden Methoden genannt
- Sie werden über das Objekt aufgerufen

```
> let cat = { type: "cat", sayHello: () => "Meow" }

> cat.sayHello
[Function: sayHello]

> cat.sayHello()
'Meow'
```

METHODEN

```
1 let player = {
2   sayHello: function () { return  "Hello" },  /* ausführlich    */
3   sayHi() { return  "Hi" },                   /* abgekürzt      */
4   sayBye: () => "Bye",                        /* Pfeilnotation  */
5 }
```

- Verschiedene Notationen für Methoden
- Abgekürzte Variante seit ECMAScript 2015 möglich

METHODEN

```
1 let cat = {
2   type: "cat",
3   say1() { return  "Meow from " + this.type },
4   say2: () => "Meow from " + this.type,
5 }
6 console.log( cat.say1() )    /* → Meow from cat          */
7 console.log( cat.say2() )    /* → Meow from undefined    */
```

- `this` ist das Objekt, über das die Methode aufgerufen wird
- Das gilt nicht für Funktionen in Pfeilnotation
- Mehr dazu in einer anderen Lektion

Speaker notes

Bei Funktionen in Pfeilnotation wird `this` aus der Umgebung der Funktionsdefinition übernommen. Da im Beispiel nichts anderes spezifiziert wurde, ist `this` in der Node.js REPL das globale Objekt, für das offenbar kein `type`-Attribut definiert ist.

Das globale Objekt ist unter Node.js `global`, im Browser ist es `window`.

OBJEKT ANALYSIEREN

- Methode `keys` von `Object`
- Liefert Array aller Attributnamen
- Analog liefert `values` alle Werte

```
> let obj = {a: 1, b: 2}

> Object.keys(obj)
[ 'a', 'b' ]

> Object.values(obj)
[ 1, 2 ]
```

OBJEKTE ZUSAMMENFÜHREN

- Methode `assign` von `Object`
- Erstes Argument ist das Zielobjekt
- Attribute der weiteren Argumente ins Zielobjekt kopiert
- Referenz auf Ergebnis (erstes Arg.) zurückgegeben

```
> let objectA = {a: 1, b: 2}

> Object.assign(objectA, {b: 3, c: 4})
{ a: 1, b: 3, c: 4 }

> Object.assign(objectA, {m: 10}, {n: 11})
{ a: 1, b: 3, c: 4, m: 10, n: 11 }
```

Speaker notes

Durch Kopieren der Attribute in ein leeres Objekt kann eine Top-Level-Kopie (s. später Thema Referenzen) eines Objekts erstellt werden:

```
> let obj = {a: 1, b: 2};
> let objCopy = Object.assign({}, obj);
> objCopy
{ a: 1, b: 2 }
> objCopy == obj
false
```

In diesem Fall muss das Ergebnis von `Object.assign` aber zugewiesen werden, da wir noch keine Referenz auf das Zielobjekt haben. `Object.assign` modifiziert also das Objekt, welches als erstes Argument angegeben wird und gibt ausserdem eine Referenz auf dieses Objekt zurück.

Die Methoden `assign`, `keys` und `values` sind Methoden von `Object`, aber nicht von einzelnen Objekten:

```
> let obj = {a:1, b:2}
> obj.values()
Uncaught TypeError: obj.values is not a function
> Object.values(obj)
[ 1, 2 ]
```

Dagegen ist `toString` eine Methode einzelner Objekte (auch wenn die Ausgabe hier nicht besonders aufschlussreich ist):

```
obj.toString()
'[object Object]'
```

Diese Unterscheidung entspricht etwa der Unterscheidung von Klassen- und Instanzenmethoden in Java. In JavaScript wird das mit Prototypen umgesetzt, was wir in einer späteren Lektion genauer ansehen werden.

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

SPREAD-SYNTAX

```
> let objectA = { a: 1, b: 2 }
> let objectB = { c: 100, d: 200 }

> {...objectA, ...objectB, c: 3}
{ a: 1, b: 2, c: 3, d: 200 }

> {...objectA}
{ a: 1, b: 2 }

> {...objectA} == objectA
false
```

- Inhalte eines Objekts in ein anderes Objekt einfügen
- Spread-Operator `...`

OBJEKTE DESTRUKTURIEREN

```
1 let bar = 87
2 let obj = { foo: 12, bar, baz: 43 }
3
4 let {foo, baz} = obj
5 console.log(foo)                    /* → 12 */
```

- Teile aus (möglicherweise grossen) Objekten extrahieren
- Auch in Funktionsparametern möglich (spätere Lektion)

Speaker notes

Im Beispiel sieht man, dass man in einem Objektliteral anstelle von `bar: bar` auch abgekürzt `bar` schreiben kann. Es wird dann ein Attribut mit diesem Namen und dem Wert der Variablen eingefügt.

ÜBERSICHT

- Objekte
- Spezielle Objekte: Arrays
- Werte- und Referenztypen
- Vordefinierte Objekte
- JSON

ARRAYS

- Sequenzen von Werten
- Zugriff über Index (erstes Element hat Index 0)
- Nicht jede Position muss besetzt sein
- Nicht besetzte Positionen liefern `undefined`

```
1 let a = [1, 2, 3]
2 a[10] = 99
3
4 console.log( a )          /* → [ 1, 2, 3, <7 empty items>, 99 ]  */
5 console.log( a.length )   /* → 11                                */
6 console.log( a[1000] )    /* → undefined                         */
```

Speaker notes

Wie in Java und C ist `a[1]` also nicht das erste sondern das zweite Element des Arrays. Oder allgemein: beim Zugriff auf `a[n]` werden vom Beginn des Arrays n Elemente übersprungen.

Es kann problemlos auch auf eine Position ausserhalb des bestehenden Arrays zugewiesen werden. Die Länge des Arrays ist um 1 grösser als der grösste Index.

ARRAYS

- Array-Elemente können von beliebigem Typ sein
- Typen können problemlos gemischt werden
- Hier ist das letzte Element des Arrays eine Funktion:

```
> let data = [41, 3.14, "pi", [1, 2, 3], n => 2*n]
undefined

> data[4](3)
6
```

ARRAYS

- Arrays sind Objekte mit speziellen Eigenschaften
- Sie haben Attribute und Methoden
- Test auf Array: `Array.isArray()`

```
> let data = [1, 2, 3]

> typeof(data)
'object'
> Array.isArray(data)
true
> data.length
3
```

ARRAY-METHODEN

- Für Arrays stehen zahlreiche Methoden zur Verfügung
- Zum Beispiel `push` und `pop`

```
> let data = [1, 2, 3]

> data.push(10)
4
> data.push(11, 12)
6
> data.pop()
12

> data
[ 1, 2, 3, 10, 11 ]
```

Speaker notes

`push` hängt ein oder mehrere Elemente ans Ende eines Arrays an und gibt die Anzahl der Elemente im Array zurück.

`pop` entfernt das letzte Element aus dem Array und gibt es zurück.

ARRAY-METHODEN

- `shift`, `unshift`: Einfügen und Entfernen am Array-Anfang
- `indexOf`, `lastIndexOf`: Element im Array finden
- `slice`: Bereich eines Arrays ausschneiden
- `concat`: Arrays zusammenhängen
- `at`: Zugriff auf Index (ECMAScript 2022)

```
> let data = [ 1, 2, 3, 10, 11 ]

> data.slice(1, 3)
[ 2, 3 ]

> data.concat([100, 101])
[ 1, 2, 3, 10, 11, 100, 101 ]
```

Speaker notes

Die Methoden `slice` und `concat` erzeugen neue Arrays (s. gleich: Referenz-Datentypen):

```
> data.slice(0)
[ 1, 2, 3, 10, 11 ]
> data.slice(0) == data
false
```

Die neue Methode `at` (ECMAScript 2022) wird von allen Indextypen (Strings, Arrays, typisierte Arrays wie Uint8Array) unterstützt und erlaubt den Zugriff über einen bestimmten Index, der auch negativ sein kann (-1 greift auf das letzte, -2 auf das vorletzte Element zu usw.):

```
> [1, 2, 3, 4].at(1)
2
> [1, 2, 3, 4].at(-1)
4
```

Weitere Methoden:

- https://www.w3schools.com/js/js_array_methods.asp
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

SCHLEIFEN ÜBER ARRAYS

```
1 /* Standard for-Schleife */
2 for (let i = 0; i < myArray.length; i++) {
3   doSomethingWith(myArray[i])
4 }
5
6 /* einfachere Variante für Arrays */
7 for (let entry of myArray) {
8   doSomethingWith(entry)
9 }
```

Speaker notes

Zum Verarbeiten von Arrays gibt es auch eine Reihe vordefinierter Funktionen höherer Ordnung (s. später).

Übrigens kann man auch über Strings mit einer for..of-Schleife iterieren.

SPREAD-SYNTAX

```
> let parts = ['shoulders', 'knees']
> ['head', ...parts, 'and', 'toes']
["head", "shoulders", "knees", "and", "toes"]

> [...parts]
['shoulders', 'knees']

> [...parts] == parts
false
```

- Inhalte eines Arrays in ein anderes Array einfügen
- Spread-Operator `...`

ARRAYS DESTRUKTURIEREN

- Mehrere Parameter oder Variablen aus einem Array zuweisen
- Vermeidet das spätere Zugreifen über den Array-Index

```
1 let numbers = [1, 2, 3]
2 let [a, b, c] = numbers
3 console.log(c)                          /* → 3     */
```

Speaker notes

Wenn die Anzahl der Variablen/Werte bei der Zuweisung oder Initialisierung nicht übereinstimmt, werden überzählige Werte ignoriert oder überzählige Variablen auf `undefined` gesetzt.

Das letzte Element auf der linken Seite kann auch ein Rest-Element sein:

```
> let [a, ...b] = [1,2,3,4,5,6,7]
undefined
> a
1
> b
[ 2, 3, 4, 5, 6, 7 ]
```

Das Destrukturieren von Arrays ist oft nützlich. So können auf einfache Weise Variableninhalte vertauscht oder mehrere Werte aus Funktionen zurückgegeben werden:

```
> [a, b] = [1, 2]
> [a, b] = [b, a]
[2, 1]
> const web = () => [404, 'not found']
> web()
[404, 'not found']
> let [code, msg] = web()
```

ÜBERSICHT

- Objekte
- Spezielle Objekte: Arrays
- Werte- und Referenztypen
- Vordefinierte Objekte
- JSON

WERTE-DATENTYPEN

- Zahlen, Strings und Wahrheitswerte sind *Wertetypen*
- Sie sind unveränderlich
- Zuweisung kann wie Kopieren behandelt werden

```
1 let msg = "Hello developers!"
2 let greeting = msg
3 greeting += "!!"
4
5 console.log(greeting)     /* → 'Hello developers!!!' */
6 console.log(msg)          /* → 'Hello developers!'   */
```

Speaker notes

String-Anpassungen erfolgen also nicht an Ort und Stelle. Es wird ein neuer String erzeugt und dieser muss wieder zugewiesen werden. Nach

```
let msg = "Hello developers!"
```

kann man also sicher sein, dass sich der Inhalt von `msg` nicht ändert, solange der Variablen nichts neues zugewiesen wird. Das gilt auch für die Parameterübergabe: da ein übergebener String unveränderbar ist, kann angenommen werden, dass die Funktion mit einer Kopie der Zeichenkette arbeitet.

REFERENZ-DATENTYPEN

- Objekte und Arrays sind *Referenz-Datentypen*
- Sie sind jederzeit veränderbar
- Es werden Referenzen zugewiesen

```
1 let obj = { message: "loading..." }
2 let anotherObj = obj
3 anotherObj.message = "ready"
4
5 console.log(obj)          /* → { message: 'ready' } */
```

Speaker notes

Hier werden also Referenzen zugewiesen. Bei der Übergabe von Objekten an eine Funktion muss damit gerechnet werden, dass diese in der Funktion geändert werden.

In vielen Situationen möchte man auch Objekte als Werte betrachten können. Das setzt voraus, dass man Objekte nicht verändert bzw. bei Änderungen immer neue Objekte erzeugt. Das hat zahlreiche Vorteile, geht jedoch auf Kosten von Speicherverbrauch und Rechenzeit. Unveränderbare Datenstrukturen sind das Ziel der Bibliothek immutable.js von FaceBook:

https://immutable-js.github.io/immutable-js/

In ECMAScript 2023 wurden einige Array-Methoden hinzugefügt, welche das ursprüngliche Array nicht verändern und ein neues Array liefern:

- `toSorted` liefert ein neues, sortiertes Array
- `toReversed` kehrt ein Array um und liefert das Ergebnis als neues Array
- `toSpliced` liefert ein neues Array mit geänderter Teilsequenz
- `with` liefert ein neues Array mit einer überschriebenen Position

```
const languages = ["JavaScript", "TypeScript", "CoffeeScript"]
const updated = languages.with(2, "WebAssembly")
```

https://www.sonarsource.com/blog/es2023-new-array-copying-methods-javascript/

WERTE- UND REFERENZTYPEN

- Objekt- und Array-Literale legen neue Objekte an
- `==` und `===` vergleichen die Referenzen
- `const` heisst: Referenz kann nicht geändert werden

```
> const a = [1, 2, 3]
> const b = [1, 2, 3]
> const c = a

> a == b
false
> a == c
true
> c[0] = 99
> a
[ 99, 2, 3 ]
```

```
> const obj = { message: "loading..." }

> obj.message = "ready"
'ready'

> obj = {}
Uncaught TypeError: Assignment to
constant variable.
```

Speaker notes

Mit `const` kann die Referenz als Konstante definiert werden. Das Ziel ist dadurch immer noch veränderbar, nicht aber die Referenz selber.

WERTE- UND REFERENZTYPEN

- Objekte sind also Referenztypen
- Das gilt auch für Arrays und Funktionen
- Referenzen vs. Werte vergleichen:

```
> [ []==[], {}=={}, (()=>{})==(()=>{}) ]
[ false, false, false ]

> [ 3.5==3.5, "abc"=="abc", false==false ]
[ true, true, true ]
```

Speaker notes

Konsequenz: Strings können in JavaScript jederzeit mit == oder === verglichen und generell wie Werte behandelt werden.

Sowohl Funktionen als auch Arrays erben von `Object.prototype`. Mehr dazu in einer späteren Lektion.

ATTRIBUTE

- Wie Objekte und Arrays können auch Werte in JavaScript Attribute (bzw. Methoden) haben
- Ausnahmen: `null`, `undefined`

```
> "Zeichenkette".length
12
> "Zeichenkette"["length"]
12
> "Zeichenkette".toUpperCase()
'ZEICHENKETTE'
```

```
> 1.5.toString()
'1.5'
> (1/3).toPrecision(4)
'0.3333'
```

Speaker notes

Methoden werden wohl selten auf Zahlenliteralen aufgerufen wie im Beispiel. Wenn das trotzdem geschieht, muss man darauf achten, dass es nicht zu Verwechslung mit dem Dezimalpunkt kommt:

```
> 1.5.toString()
'1.5'
> 1.toString()
Uncaught SyntaxError: Invalid or unexpected token
> 1..toString()
'1'
> (1).toString()
'1'
```

ATTRIBUTE

- Attribute von Wertetypen sind unveränderlich
- Zuweisung neuer Attribute zu Wertetypen wird ignoriert
- Objekten (auch Arrays, Funktionen) können aber jederzeit Attribute zugewiesen werden

```
> const square = n => n*n

> square.doc = "Quadratfunktion"
'Quadratfunktion'

> square(3)
9

> square.doc
'Quadratfunktion'
```

Speaker notes

Die Zuweisung neuer Attribute zu Wertetypen ist zwar zulässig, bewirkt aber nichts. Beispiel:

```
> let name = "John Johnson"
undefined
> name.doc = "Das ist sein Name"
'Das ist sein Name'
> name.doc
undefined
```

ARRAYS UND ATTRIBUTE

- Die (normalen) Attribute eines Arrays sind ganze Zahlen >=0
- Wird etwas anderes als Index angegeben, wird ein Attribut hinzugefügt

```
> let arr = [1, 2, 3]

> arr[-1] = 4
> arr['key'] = 'value'

> arr
[ 1, 2, 3 ]
> arr.key
'value'
```

Speaker notes

Je nach Implementierung werden die Attribute '-1' und 'key' mit dem Array ausgegeben. Unter bestimmten Node.js-Versionen zum Beispiel:

```
> arr
[ 1, 2, 3, '-1': 4, key: 'value' ]
```

Tatsächlich kann man alle Attribute, numerische Array-Attribute und andere Attribute, in ein *normales* Objekt überführen:

```
> Object.assign({}, arr)
{ '0': 1, '1': 2, '2': 3, '-1': 4, key: 'value' }
```

Auch lässt sich der reine Array-Inhalt extrahieren:

```
> Array.from(arr)
[1, 2, 3]
```

ÜBERSICHT

- Objekte
- Spezielle Objekte: Arrays
- Werte- und Referenztypen
- Vordefinierte Objekte
- JSON

String

- Strings sind in JavaScript ein primitiver Datentyp
- Erzeugt durch String-Literale `"..."`, `'...'`, `` `...` ``
- String-Methoden sind in `String.prototype` definiert (mehr zu Prototypen später)

```
> String.prototype.slice
[Function: slice]

> "Hello World".slice(0, 5)
'Hello'
```

Speaker notes

Tatsächlich gibt es auch String-Objekte, da String auch ein Konstruktor ist. In den meisten Fällen können primitive Strings und String-Objekte gleich verwendet werden. String-Objekte sind allerdings Referenztypen und können um Attribute ergänzt werden. Mit der `valueOf`-Methode kann aus einem String-Objekt ein primitiver String gewonnen werden.

```
> let sPrim = 'foo'
> let sObj = new String(sPrim)

> console.log(typeof sPrim)
"string"
> console.log(typeof sObj)
"object"

> sObj
[String: 'foo']
> sObj.valueOf()
'foo'
```

Man könnte sich fragen, warum `String.prototype.slice` eine Funktion liefert, `String.prototype` jedoch ein leeres Objekt. Das liegt daran, dass nicht alle Properties von Objekten *enumerable* sind. Sie können jedoch durch `Object.getOwnPropertyNames` in ein Array ausgegeben werden:

```
> String.prototype.slice
[Function: slice]
> String.prototype
{}
> Object.getOwnPropertyNames(String.prototype)
[
  'length',      'constructor',       'anchor',
  'big',         'blink',             'bold',
...
]
```

STRING-METHODEN

- `slice`: Ausschnitt aus einem String (vgl. Arrays)
- `indexOf`: Position eines Substrings (vgl. Arrays)
- `trim`: Whitespace am Anfang und Ende entfernen
- `padStart`: vorne Auffüllen bis zu bestimmter Länge
- `split`: Auftrennen, liefert Array von Strings
- `join`: Array von Strings zusammenfügen
- `repeat`: String mehrfach wiederholen

MDN: String

Speaker notes

Auf einzelne Zeichen eines Strings kann auch über den Index zugegriffen werden:

```
> let msg = 'Hello'
> msg[0]
'H'
> Object.keys(msg)
[ '0', '1', '2', '3', '4' ]
```

Seit ECMAScript 2022 ist ein Zugriff auch mit der `at`-Methode möglich. >Im Gegensatz zur []-Notation sind auch negative Werte möglich:

```
> let msg = 'Hello'
> msg.at(0)
'H'
> msg.at(-1)
'o'
```

Number

- Methoden und Konstanten von `Number`
- Methoden von `Number.prototype` können auf einzelne Zahlen angewendet werden

```
> Number.MAX_VALUE
1.7976931348623157e+308

> Number.isInteger(1.5)
false

> 3500.75.toLocaleString('de-DE')
'3.500,75'
```

MDN: Number

Speaker notes

Für `Number` gilt ebenso wie für `String`: es kann auch als Konstruktor verwendet werden, um Objekte zu erstellen, mit `valueOf` kann man den primitiven Typ erhalten. Das wird man aber kaum je brauchen. `Number` kann auch verwendet werden, um zu Zahlen zu konvertieren:

```
let n = Number("12")
```

Math

- Methoden und Konstanten als Attribute des `Math`-Objekts
- Objekt als Namensraum für Methoden und Konstanten
- Gleich wie `String` und `Number` ist `Math` built-in

```
> Math.E
2.718281828459045

> Math.exp(1)
2.718281828459045

> Math.min(5, 2, 7, -4, 12)
-4
```

```
> Math.sin(0.5)
0.479425538604203

> Math.round(3.7)
4

> Math.random()
0.04802545627381716
```

MDN: Math

Speaker notes

Im Gegensatz zu `Number` und `String` kann `Math` nicht als Konstruktor verwendet werden. `Math` ist ein vordefiniertes Objekt mit einigen Attributen und Methoden.

Um von einer Zahl mit Nachkommastellen zu einer Ganzzahl zu kommen, gibt es verschiedene Methoden von `Math`:

- `Math.floor`: nächst kleinere Ganzzahl
- `Math.ceiling`: nächst grössere Ganzzahl
- `Math.round`: Runden zur nächsten Ganzzahl

Weitere Methoden:

- `Math.abs`: positiver Wert
- `Math.log`: natürlicher Logarithmus
- `Math.max`: grösstes der Argumente
- `Math.min`: kleinstes der Argumente
- `Math.pow`: Zahl hoch Exponent
- `Math.sin`: Sinus
- `Math.sqrt`: Quadratwurzel

Weitere Konstanten:

- `LN2`: natürlicher Logarithmus von 2
- `LN10`: natürlicher Logarithmus von 10
- `LOG2E`: Logarithmus von 2
- `LOG10E`: Logarithmus von 10
- `PI`: Konstante PI
- `SQRT1_2`: Quadratwurzel aus 0.5
- `SQRT2`: Quadratwurzel aus 2

Date

```
let now = new Date()

console.log(now)                    /* → 2021-10-09T15:43:21.753Z */
console.log(now.getHours())         /* → 17                       */
console.log(now.getUTCHours())      /* → 15                       */
console.log(now.getTime())          /* → 1633794201753            */

now.setFullYear(now.getFullYear()+1)
console.log(now.toString())
// 'Sun Oct 09 2022 17:43:21 GMT+0200 (Mitteleuropäische Sommerzeit)'
```

MDN: Date

Speaker notes

Es gibt ein paar Merkwürdigkeiten des JavaScript `Date`-Konstruktors. Zum Beispiel liefert `getYear()` die Anzahl Jahre seit 1900, für 1995 also 95. Erstaunlich, dass man nicht daran gedacht hat, dass das Jahr 2000 kommen wird und das Jahr dann 100 liefert. Einige Webseiten haben dann auch am 1.1.2000 das Datum als 1.1.19100 angegeben. Schliesslich war der Fix dann, die Methode `getFullYear()` einzuführen.

Ein zweiter Fallstrick ist der Monat. Hier wird mit 0 begonnen zu zählen, der Januar ist also Monat 0, Dezember ist Monat 11.

Da `Date` immer wieder Probleme macht, gibt es einen Vorschlag, ECMAScript mit `Temporal` eine neue API hinzuzufügen. Aus dem Vorschlag:

„Date has been a long-standing pain point in ECMAScript. This is a proposal for Temporal, a global Object that acts as a top-level namespace (like Math), that brings a modern date/time API to the ECMAScript language.”
https://tc39.es/proposal-temporal/docs/index.html

WEITERE VORDEFINIERTE OBJEKTE

- `Map`, `Set`: Schlüssel/Wert Zuordnung bzw. Menge
- `RegExp`: reguläre Ausdrücke
- `Object`: allgemein Objekte

Zahlreiche weitere vordefinierte Objekte...

MDN: Standard built-in objects

ÜBERSICHT

- Objekte
- Spezielle Objekte: Arrays
- Werte- und Referenztypen
- Vordefinierte Objekte
- JSON

JSON

- JavaScript Object Notation
- Daten-Austauschformat, nicht nur für JavaScript
- Orientiert an Notation für JavaScript-Objektliterale

```
> JSON.stringify({ type: "cat", name: "Mimi", age: 3})
'{"type":"cat","name":"Mimi","age":3}'

> JSON.parse('{"type":"cat","name":"Mimi","age":3}')
{ type: 'cat', name: 'Mimi', age: 3 }
```

https://www.json.org/json-en.html

QUELLEN

- Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
  https://eloquentjavascript.net/

LESESTOFF

Geeignet zur Ergänzung und Vertiefung

- Kapitel 4 von:
  Marijn Haverbeke: Eloquent JavaScript, 3rd Edition
  https://eloquentjavascript.net/