Einführung in die Programmierung mit JavaScript

Teil 2

Lexikalische Struktur

- Case sensitiv
- Leerzeichen werden überlesen
- Zeilenenden werden ausserhalb von Literalen überlesen
- Zeilenenden werden bei fehlenden Semikolons berücksichtigt

Lexikalische Struktur: Kommentare

- Es gibt „End of Line“ und „Block“ Kommentare
- // this is an End of Line comment
- /* This is a block comment
  it can contain anything.
  even comments // this is just text
  */
- Es gibt keinen Standard für „Dokumentations-Kommentare“ wie JavaDoc. Allerdings gibt es JSDoc als Add-On-Tool [15]

Lexikalische Struktur: Identifier

- Namen in JavaScript (Identifier) sind sehr frei wählbar mit folgenden Einschränkungen
  - Muss mit einem Buchstaben, Dollar oder Unterstrich anfangen.
  - Darf nicht mit einer Ziffer beginnen.
  - Darf kein reserviertes Wort sein
- Reservierte Worte sind die Schlüsselworte der Programmiersprache

Lexikalische Struktur: Schlüsselworte

- Die Schlüsselworte der Sprache dürfen im Allgemeinen nicht als Namen benutzt werden.
-

```
as        const       export      get         null      target    void
async     continue    extends     if          of        this
while
await     debugger    false       import      return    throw     with
break     default     finally     in          set       true
yield
case      delete      for         instanceof  static    try
catch     do          from        let         super     typeof
class     else        function    new         switch    var
```

- Als Eigenschaften-Namen sind sie größtenteils erlaubt. Ist aber nicht zu empfehlen.

Lexikalische Struktur: Reservierte Worte

- Um die zukünftige Erweiterung der Sprache nicht zu gefährden, wurden Worte reserviert.
- Die Begriffe sind heute noch nicht Teil der Sprache, es ist jedoch geplant.
- `enum` `implements` `interface` `package` `private` `protected` `public`
- `arguments` und `eval` sind nicht mehr Teil der Sprache aber weiterhin reserviert.

Lexikalische Struktur: Zeichensatz

- Unicode (UTF-16) ist als Source-Code Zeichensatz vorgeschrieben
- Der in Dateien gespeicherte Zeichensatz kann davon abweichen
- JS Runtime (Browser, Node) muss mit Source-Encoding umgehen können

Lexikalische Struktur: Zeichensatz

- Unicode Zeichen-Kodierung im Source-Code
  - `\uXXXX – erlaubt Unicode bis FFFF`
  - `\u{XXXX} – erlaubt auch Codepoints >FFFF`
  - Ein beliebiges Unicode Zeichen
- XXXX ist eine 4-stellige HEX Zahl mit Großbuchstaben als Hex-Ziffern A-F.

Lexikalische Struktur: Zeichensatz

- Unicode sollte im Source-Code-Speicherformat vermieden werden
  - ASCII-7 oder UTF-8
  - Support von ByteOrderMark (BOM) [16] sicherstellen, besser: vermeiden
- Identifier und Kommentare auf ASCII-7 beschränken
- Literale mit \u-Notation schreiben
- Auf jeden Fall einheitlichen Zeichensatz nutzen
- Prüfen, ob die Dateien auf einem anderen Rechner fehlerfrei geladen werden können

Lexikalische Struktur: Das Semikolon

- Moderne JavaScript Interpreter kommen auch ohne explizites Semikolon aus
- Lässt man das Semikolon weg, kann das zu unerwarteten Kommandofolgen führen
- JavaScript ergänzt Semikolons am Zeilenende, wenn die folgende Zeile ohne Semikolon keinen Sinn ergibt
- ```
  let a
  a
  =
  3
  console.log(a)
  ```
- `let a; a = 3; console.log(a); // Interpretiertes Ergebnis`
  ```

Lexikalische Struktur: Das Semikolon

- Das Ergebnis ohne Semikolon ist nicht immer eindeutig:
- `let y = x + f`
  `(a+b).toString()`
- `let y = x + f(a+b).toString();`
- existiert keine Funktion `f()`, wird ein schwer zu findender Fehler ausgeworfen. Ansonsten wird unerwartet `f()` aufgerufen
- Defensives Semikolon bei Ausdrücken, die mit `(`, `[`, `/`, `+`, `-` starten:
  ```
  let y = x + f
  ;(a+b).toString()
  ```

Lexikalische Struktur: Das Semikolon

- Nach `return, break, continue` wird ein Zeilenende immer als Semikolon gewertet
- Bei Arrow-Funktionen muss der Pfeil immer mit der Parameterliste stehen
  - ```
    let itB = bee =>
      "Beatles"
    ```
- Pre-Dekrement/-Inkrement und Post-Dekrement/-Inkrement Operatoren dürfen nicht getrennt werden und sind in der selben Zeile wie die Variable zu schreiben
  - ```
    ++a
    a++
    ```

Aufgabe: Benutze optionale Semikolons

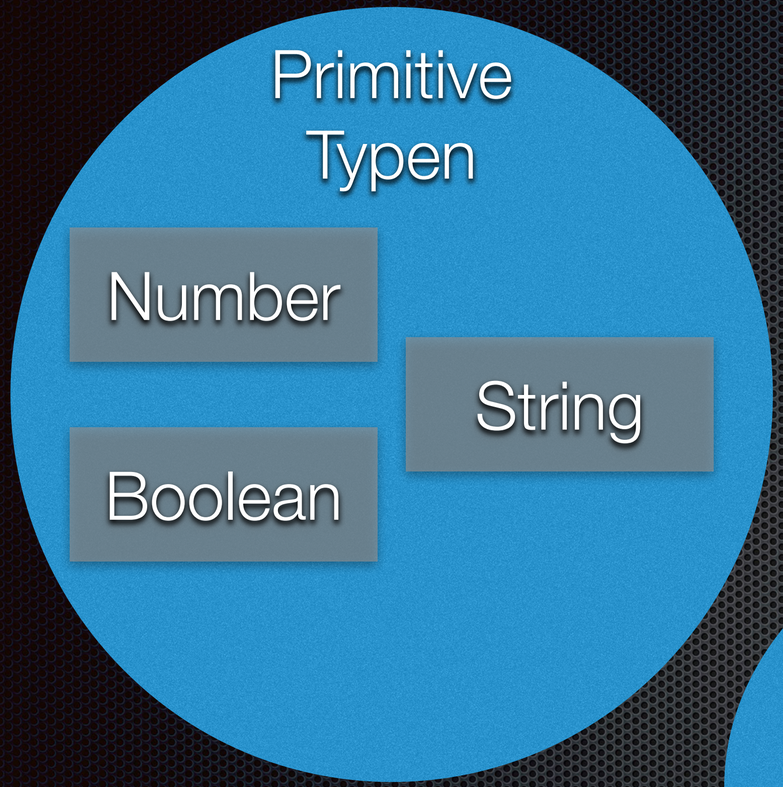
- Schreibe deine Implementierung des Weihnachtsbaums und des Zahlenfinders so um, dass sie möglichst wenig Semikolons benutzen

Arbeitsphase

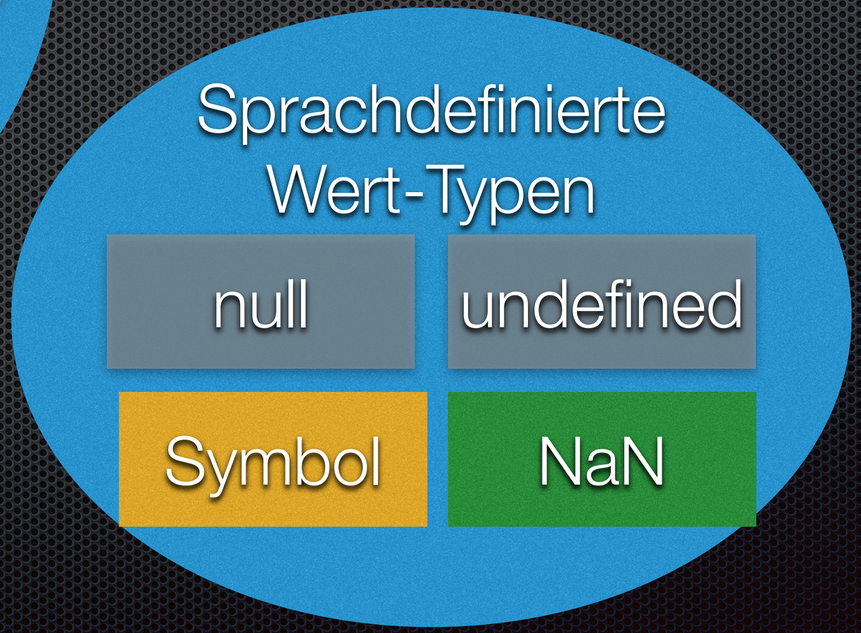
— — —

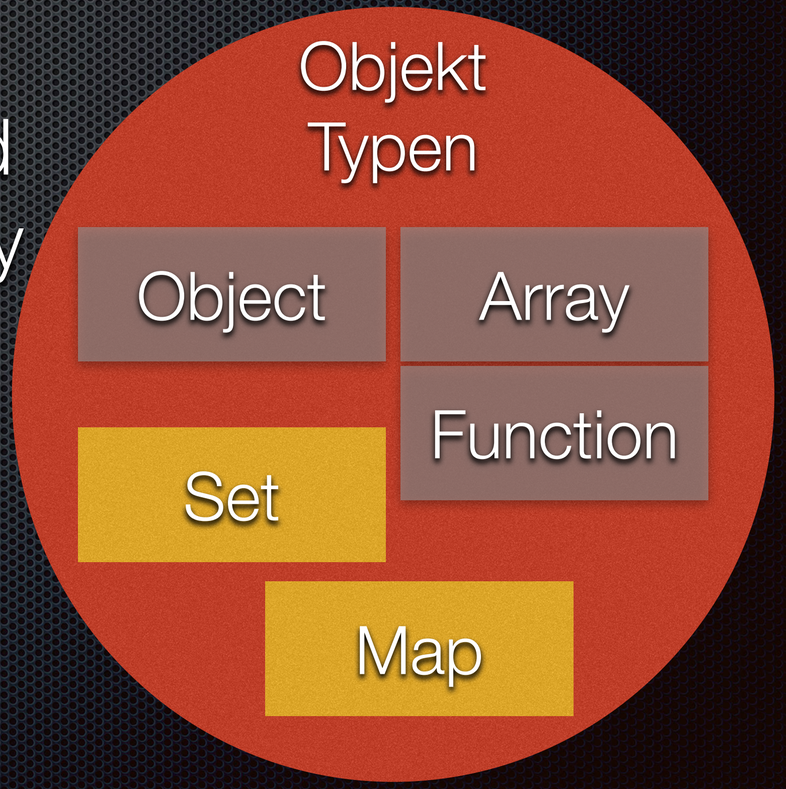
10 Minuten

Das JavaScript Typen-System



Object und Array sind Teil der Sprache
Set, Map, Date, … sind Teil der Standard-Library





Primitive Typen

- Wert-Typen: Enthalten immer nur einen Wert und werden als Wert übergeben
- Immutable: Werte und Eigenschaften können nicht geändert werden
- Jeder Wert, der keine number, string, boolean, Symbol, null oder undefined ist, ist vom Typ Object

Objekt Typen

- Array-Objekt: Nach Index geordnete Liste von Werten. Der Index ist der Name des Wertes.
- Objekt-Typen: unsortierte Sammlung von Name-Wert-Paaren
- Function-Objekte: Objekt-Typen, die ausführbaren Programm-Code enthalten.
- Mutable: Die Eigenschaften von Objekt-Typen können geändert werden. Das gilt auch für Functions.
- By Reference: Objekte werden nur als Referenzen übergeben, nicht kopiert

Speicher-Management

- JavaScript Runtimes tracker den Lebenszyklus von Objekten automatisch
- Tote Objekte werden mittels Garbage Collection entfernt

Manipulation von Objekten und Primitiven Typen

- Die Objekt-Notation von JavaScript ist auch aus anderen Sprachen bekannt
  ```
  <Objekt-Instanz>.<Eigenschaft>
  <Objekt-Instanz>.<Methode>(<parameter>)
  <Objekt-Instanz>[<Eigenschaft>]
  <Objekt-Instanz>[<Methode>](<parameter>)
  ```
- Obwohl Primitive Typen keine Objekte sind, haben sie Methoden und Eigenschaften, die über die Objektnotation angesprochen werden:
  - `”Hello”.length –> 5`
  - `5[’toExponential’]() –> '5e+0'`

Primitive Typen: Number

- Der Zahlentyp von JavaScript
  - Integer
    - `0`
      `100000000`
      `1_000_000_000`

Primitive Typen: Number

- Der Zahlentyp von JavaScript
  - Floating Point in IEEE 754 Darstellung
    - `[digits][.digits][(E|e)[(+|-)]digits]`

```
0.7
.88
6.02e23          // 6.02 × 10²³
6.02E23          // 6.02 × 10²³
1.4738223e-32    // 1.4738223 × 10⁻³²
1.4738223E-32    // 1.4738223 × 10⁻³²
```

Primitive Typen: Number

- Zahlenraum:
- 64Bit Floating Point:
  $\pm 5 \times 10^{-324}$ bis
  $\pm 1.79769313486231 57 \times 10^{308}$
- Integer:
  `-9.007.199.254.740.992` (-2^{53}) bis
  `9.007.199.254.740.992` (2^{53})
- Bit-Operatoren (Shift, AND, OR, …) und Array Indexes:
  32Bit (`0 - 4.294.967.296`)

Primitive Typen: Number

- Zahlenraum: 64Bit Floating Point: $\pm 1.7976931348623157 \times 10^{308}$ bis $\pm 5 \times 10^{-324}$
  Integer: –9.007.199.254.740.992 (-2^{53}) and 9.007.199.254.740.992 (2^{53})
  Array Indexes: 32Bit (0 - 4.294.967.296)

- Zahlensysteme: Die Basis wird durch einen Prefix bestimmt

  - Hex: 0x oder 0X :
    `0x0123456789ABCDEF  // case für A-F ist egal`

  - Binär: 0b oder 0B :
    `0x0100_0010`

  - Oktal: 0o oder 0O:
    `0o12345670`

Primitive Typen: BigInteger

- Fließkomma Zahlen decken nicht alle Zahlen im Zahlenraum ab, es gibt zu wenige
- Allgemeines Problem der Arithmetik mit Computern
- Beispiel:

```
let x = .3 - .2;     // —> 0.09999999999999998
let y = .2 - .1;     // —> 0.1
x === y              // => false: nicht identisch!
x === .1             // => false: .3-.2 nicht gleich .1
y === .1             // => true: .2-.1 ist gleich .1
```

- Genaue Berechnung mit beliebig großen Ganzzahlen durchführen: BigInteger

Primitive Typen: BigInteger

- Beispiel:

```
let x = .3 - .2;      // —> 0.09999999999999998
let y = .2 - .1;      // —> 0.1
x === y               // => false: nicht identisch!
x === .1              // => false: .3-.2 nicht gleich .1
y === .1              // => true: .2-.1 ist gleich .1
```

Primitive Typen: BigInteger

- Floating Point Auf INT abbilden:

```
let x100 = 30n - 20n;     // —> 10 div 100 -> 0.1f
let y100 = 20n - 10n;     // —> 10 div 100 -> 0.1f
x100 === y100
x100 === 10n
y100 === 10n
```

- Ergebnisse in der Ausgabe formatieren oder durch geschicktes Runden abbilden

```
let xIStr = x100.toString();
console.log(`${xIStr.slice(0,-2)},${xIStr.slice(-2)}`);
   // -> ,10
```

Primitive Typen: Arithmetik

- Das Math.* Modul stellt wichtige Rechenoperationen zur Verfügung, die über die Operatoren +, -, *, /, ** hinaus gehen
- Konstanten für Pi und E
- Max / Min
- Random
- Trigonometrie
- Runden für Integer und Float

Primitive Typen: Number Konstanten

- NaN - Not a Number
  - Spezieller Wert, der mit nichts identisch ist, auch nicht zu sich selbst.
  - Ergebnis von 0 / 0, Infinity / Infinity, Negativer Wurzel, Arithmetik mit Strings
  - API Funktionen können NaN als Fehler zurückgeben

Primitive Typen: Number Konstanten

- Infinity / -Infinity - Unendlich
  - Spezieller Wert, der für „Out of Bounds“ Berechnungen
  - X / 0 === Infinity für X != 0
- +0 und -0

Aufgabe: NaN Detektor

- Schreibe ein Programm, dass für eine Division von zwei Zahlen identifiziert, ob das Ergebnis eine Zahl, oder
  - Infinity
  - NaN
- Finde Beispiele für jedes Ergebnis!
- Nutze das Number.* Objekt und überlege wie man NaN ohne Number Unterstützung erkennen kann.

Arbeitsphase

— — —

10 Minuten

Lösung: NaN Detektor

- Schreibe ein Programm, dass für eine Division von zwei Zahlen identifiziert, ob das Ergebnis eine Zahl, oder
  - 10/0 === Infinity // -> true
    Number.isFinite(10/0)
  - X === X // —> False ? X = NaN !
    Number.isNaN(X)

Lösung: NaN Detektor

- Finde Beispiele für jedes Ergebnis!
  - 10/10 -> 1
    10/0 -> Infinity
    -10/0 -> -Infinity
    0/0 -> NaN
    100/’zwei’ -> NaN

Primitive Typen: Number-Methoden

- isNaN(x) - True, wenn x Not a Number ist
- isFinite(x) - False, wenn x positiv/negativ Unendlich ist

Primitive Typen: Number() Konstanten

- Größter Wert für Number: Number.MAX_VALUE
  Kleinste Zahl > 1: Number.EPSILON
  Kleinste Zahl > 0: Number.MIN_VALUE

Primitive Typen: Ungenaues Rechnen

- Umgang mit Floating Point Fehlern
- Scaled Integer
- BigInt vs. IEEE Floating Point

Aufgabe: BigInt Class

- Erstelle eine Klasse, die mit BigInt Zahlen Fließkommaberechnungen durchführt: addiere, multipliziere, dividiere
- Gehe von einer festen Anzahl Nachkommastellen aus
- Beachte, dass Operationen die Anzahl Nachkommastellen verändern können (Multiplikation, Potenzieren)
- Das Ergebnis der Operation soll nach jeder Rechnung ausgegeben werden
- Ergebnisrundung kann vernachlässigt werden

Arbeitsphase

— — —

30 Minuten