JavaScript

Introduction

Sebastian Mader
18.10.2017
JavaScript Overview

- programming language of the internet
  - interpreted directly in the browser, without the need of installing a plugin
- is a standard defined by Ecma International called ECMAScript
  - each browser vendor implements its own version of the specification
  - i.e., Safari uses another implementation as Chrome and so on.

- dynamically typed, i.e., types are inferred at run time
- weakly-typed, i.e., types are implicitly converted if necessary
- multi-paradigm programming language
  - imperative
  - object-oriented
  - functional
JavaScript Short History

- Initially envisioned by Netscape which wanted **a more dynamic and interactive internet**
  - A first prototype was implemented by Brendan Eich in ten days in May 1995
  - First (beta) release in September 1995
- Was not an open standard: Microsoft **reverse-engineers JavaScript** and builds their own version called JScript for Internet Explorer
- In 1997 NetScape **tasks the Ecma International with the standardization process**: ECMAScript is born
- Many internal and external quarrels with subsequent versions of the standard
- Currently: a working standardization process with a new version every twelve months
ECMAScript Standardization Process

- each new proposed language feature passes through five stages
  - Stage 0: Strawman
    - just an idea, looking for input and collaborators
  - Stage 1: Proposal
    - identified which problem is solved by the proposed feature
    - includes a polyfill implementation
  - Stage 2: Draft
    - includes an initial draft of the specification text
  - Stage 3: Candidate
    - specification text is finished
    - includes a specification-compliant implementation
  - Stage 4: Finished
- each step is voted on by the committee
- from stage 2 on, it’s expected for the feature to land sometime in the standard
- every March: every stage 4 proposal is incorporated into the standard
Problem: Browser vendors have to implement the updated specification and release an updated version of the browser

- not always as fast as programmers wish it to be
- users are not always upgrading or are locked to legacy browsers

Babel takes JavaScript of a newer version and transforms it into JavaScript of an “earlier” (and more widely supported) version of JavaScript.

```
x => 2 * x
```

Babel

```
function(x) {
  return 2 * x;
}
```

Babel allows transpiling of features from all stages, as long as a suitable plugin exists.
People build whole programming languages that transpile into JavaScript

- **TypeScript**
  - a superset of ECMAScript 6
  - introduces static typing, interfaces, generics, ...

```typescript
interface Person {
  firstName: string;
}

function greeter(person: Person) {
  return "Hello, " + person.firstName;
}
```

- **PureScript**
  - “strongly-typed functional programming language”
  - mimics Haskell syntax

```haskell
greet :: String -> String

let name = "Hello, " ++ name ++ "!
```

- **Elm, Scala.js, CoffeeScript, ...**
Executing JavaScript

- directly into the website
  ```html
  <script type="text/javascript">
  // JavaScript
  </script>
  ```

- as external files
  ```html
  <script src="filename.js" type="text/javascript"></script>
  ```

- inside Chrome’s Developer Tools
**Variables, Constants, and Scopes**

### let/const

- **const**
  - identifier declared with const cannot be reassigned
  - block-scoped

```javascript
const x = 3;
x = 4;
TypeError: Assignment to constant variable.
```

- **let**
  - identifier declared with let can be reassigned
  - block-scoped

```javascript
let x = 3;
x = 4;
```
Variables, Constants, and Scopes

- **var**
  - behaves (nearly) the same as `let`
  - function-scoped

```javascript
function f() {
    {
        const inBlock = 3;
    }
    console.log(inBlock);
}

> f()
=> ReferenceError: inBlock is not defined

function f() {
    {
        var inBlock = 3;
    }
    console.log(inBlock);
}

> f()
=> 3
```
Variables, Constants, and Scopes  

**Hoisting**

- Identifier declared with `var` are hoisted to function-level and initialized with `undefined`
- Identifier declared with `const/let` are hoisted to block-level and not initialized

```javascript
function g() {
    console.log(x);
    var x = 3;
}

> g()
=> undefined
```

```javascript
function g() {
    console.log(x);
    const x = 3;
}

> g()
=> ReferenceError: x is not defined
```
JavaScript Control Structures

- JavaScript offers control structures known from other (imperative) programming languages
  - for-loops
    ```javascript
    for(let i = 0; i < list.length; i++) {
        if(list[i] % 2 == 0) {
            console.log(`${list[i]} is even.`);
        }
    }
    ```
  - while-loops
    ```javascript
    while(i < list.length) {
        if(i % 2 == 0) {
            console.log(`${list[i]} is even.`);
        }
        i++;
    }
    ```
Functions can be defined in various ways in JavaScript

- using the `function` keyword

```javascript
function square(x) {
    return x * x;
}
```

- as a function expression (optionally assigned to an identifier)

```javascript
const square = function(x) {
    return x * x;
}
```

- using arrow functions (and optionally assigned to an identifier)

```javascript
const square = x => x * x;
```
Functional JavaScript  
Functions as First-Class Citizens

- Functions in JavaScript can be **passed as arguments**
  ```javascript
  > [1, 2, 3].map(x => x * x)
  => [1, 4, 9]
  ```

- Functions in JavaScript can **return functions**
  ```javascript
  function add(x) {
    return function(y) {
      return x + y;
    }
  }

  > const addOne = add(1)
  > addOne(2)
  => 3
  > add(3)(4)
  => 7
  ```
Partial Function Application

- `bind(thisArg, ...args)`
  - creates a new function
  - can be used to pass arguments into the function, without actually calling the function
  - arguments are set in order in which they appear in the argument list
  - first argument is execution context, see later.

```javascript
function map(f, list) {
    const mappedList = [];
    for(let i = 0; i < list.length; i++) {
        mappedList.push(f(list[i]));
    }
    return mappedList;
}

const squareList = map.bind(null, x => x * x);
const plusOneList = map.bind(null, x => x + 1);
```
Objects and Classes

- "is a collection of related data and/or functionality"
- can consist of variables, functions, and other objects

```javascript
const point = {
  x: 3,
  y: 4,
  print: function() {
    return `(\${this.x}, $\{this.y\})`;
  }
};

point.x = 3;
point.y = 4;
point.print = function() {
  return `(\${this.x}, $\{this.y\})`;
};

> point.x
=> 3

> point.print()
=> (3, 4)
```
Objects and Classes Prototypal Inheritance

- JavaScript has inheritance
  - based on prototypal inheritance, not class inheritance as, e.g., Java
  - means, that every object is either created from nothing or based on another object
- ES2015 introduced a syntax for classes, that are not really classes, but based on prototypal inheritance
- No class fields yet without transpiling (currently stage 3)

```javascript
class Point {
  constructor(x, y) {
    this.x = x;
    this.y = y;
  }

  print() {
    return `($this.x}, $this.y)`;
  }
}
```
“this” in JavaScript

- does not work at all as in other programming languages
- is determined by the way a function is called
  - if function was called on an object, the object is passed as this
  - if called without an object, the surrounding object is passed as this
  - when in doubt: use bind!
- arrow expressions always have the surrounding object as context!

```javascript
const object = {
  f: function() {
    console.log(this);
  }
};

> object.f()
=> yields object

> const y = object.f;
> y();
=> yields global object
```
“this” in JavaScript **bind to the rescue**

- `bind(thisArg, ...args)`
  - the first argument can be used to specify the context in which a function is executed

```javascript
const object = {
  f: function() {
    console.log(this);
  }
};

> const y = object.f.bind(object);
> y();
=> yields object
```
Comparison and Types  == and !=

- When using `==` and `!=` for comparison, operands are implicitly type converted ("coercion"), if they cannot be compared directly.
Comparison and Types

• When using `===` and `!==` values are not converted, i.e., both type and value are compared.