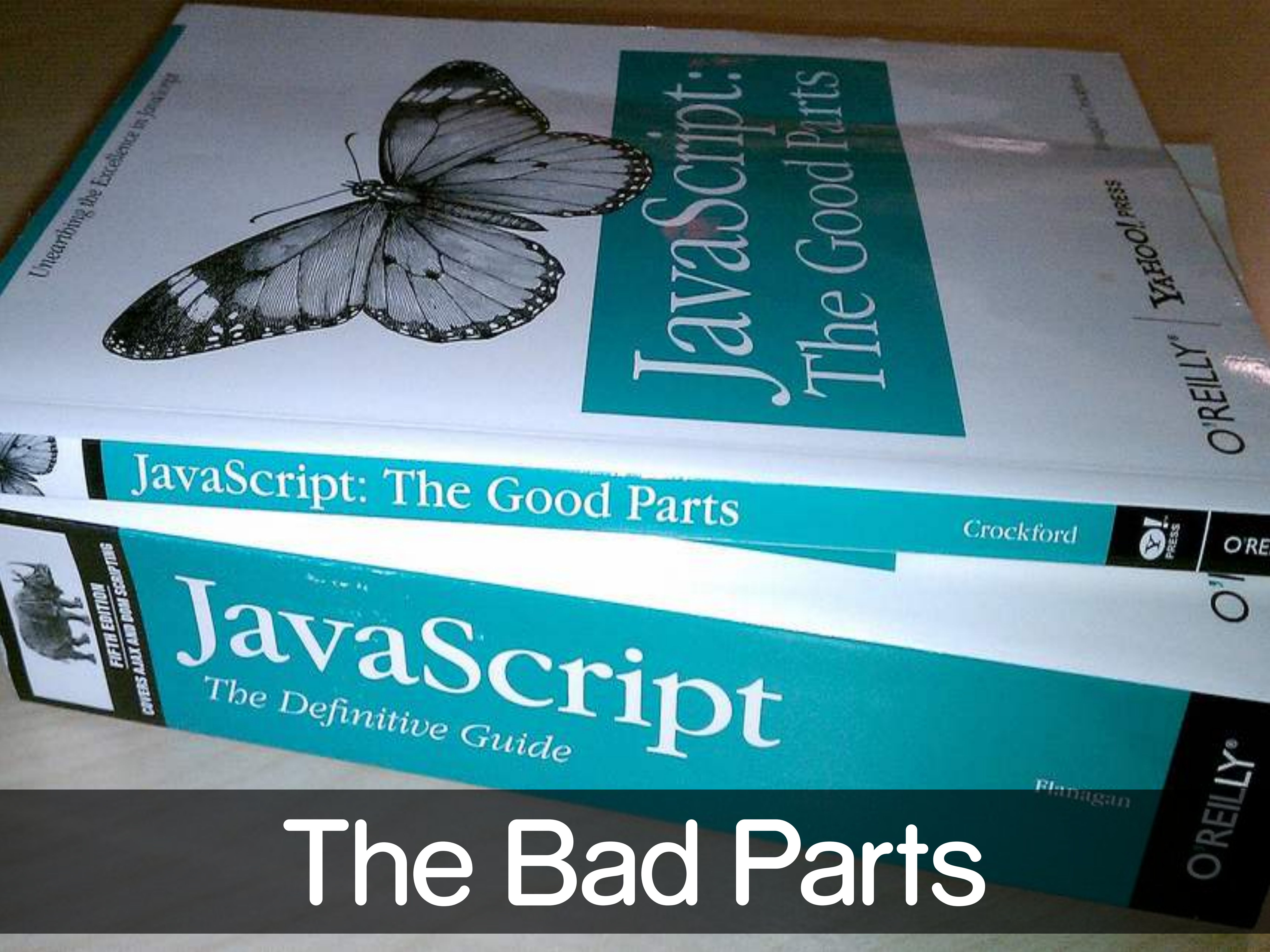
Modern JavaScript Fundamentals


Unearthing the Excellence in JavaScript
JavaScript: The Good Parts
O'REILLY
YAHOO! PRESS
JavaScript: The Good Parts
Crockford
JavaScript
The Definitive Guide
FIFTH EDITION
COVERS AJAX AND DOM SCRIPTING
Flanagan
O'REILLY

The Bad Parts

Agenda: Modern JavaScript

Functions

Closures

Classes

Modules

Selection of modern Syntax

Async / Await

$$f(x)$$

Functions

Functions

There are 3 ways to create a function:

- function declaration

```
function doIt(foo, bar) {
    return "result";
}
```

- function expression

```
let doIt = function(foo, bar) {
    return "result";
};
```

- named function expression

```
let doIt = function myFunc(foo, bar) {
    return "result";
};
```

not commonly used today!

Calling the function:

```
console.log(doIt());
```

Function declaration vs. function expression:

- All function declarations are *hoisted* to the begin of the scope and can be used before they are declared.
- Function expressions are not hoisted. Also named function expressions are not hoisted.

Named function expressions:

- Can improve debugging /tracing experience.
- The function name is only visible inside the function.

Arrow Functions: ( )=>{...}

ES2015 introduced a compact syntax for function expressions.

ES5 JS

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

var add = function(a, b){
   console.log('Adding ...');
   return a + b;
}

var pi = function(){
   return 3.1415;
}

var obj = function(){
   return {props: '!!'};
}
```

2015 ES

```
const square = x => x * x;
const add = (a, b) => a + b;
const pi = () => 3.1415;

const add2 = (a, b) => {
   console.log('Adding ...');
   return a + b;
}

const obj = () => ({prop: '!!'});
```

Differences between regular functions and arrow functions:

- `this` is used from the *lexical scope*, they do not have their own binding to `this`
- `call` and `apply` do not bind `this`
- have no `arguments` and no `super`
- calling with `new` is not allowed
- no `prototype` property

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
https://javascript.info/arrow-functions

Function Arguments

Functions can declare named arguments:

```
function sayHello(name, date) {…}
```

The caller has to pass these arguments in the the specified order:

```
const name1 = "John";
const date1 = new Date();
sayHello(name1, date1);
```

You can pass fewer arguments as declared.
Remaining arguments are undefined.

```
sayHello(name1);
```

You can pass more arguments as declared. All the arguments are accessible via the `arguments` object in the function.

Function Arguments

"options object pattern"

It is a common pattern to pass several arguments as a single object to functions.

Example: fetch API:

```
fetch(resource)
```

```
fetch(resource, options)
```

```
fetch('https://jsonplaceholder.typicode.com/todos');
```

```
const data = await fetch('https://jsonplaceholder.typicode.com/todos', {
   method: 'POST',
   body: JSON.stringify(...),
   headers: {'Content-type': 'application/json'}
});
```

https://developer.mozilla.org/en-US/docs/Web/API/Window/fetch
https://developer.mozilla.org/en-US/docs/Web/API/RequestInit

Functions are Objects

Functions are regular objects with the additional capability of being callable.

```
function foo(){};
typeof foo; // -> 'function'
foo instanceof Object; // -> true
```

Functions can have properties:

```
function foo(){};
foo.myProp = 'Hello World';
console.log(foo.myProp); //-> 'Hello World'
```

(Functions have a prototype … more about that later)

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures

Higher Order Functions

- Higher order functions operate on functions
  - functions as parameters
  - functions as return values
- Callback Functions
- Partial Application

```
$("#myBtn").on('click', function() {
  alert("Button Clicked!");
});
```

```
function arrayForEach(array, func) {
  for (let i = 0; i < array.length; i++) {
    func(array[i]);
  }
};
arrayForEach([1,2,3,4,5],
     function(msg){console.log(msg)});
```

```
function add(first, second) {
    return first + second;
}

function defineFirstArg(first, func){
    return function(second){
        return func(first, second);
    }
}

const addOne = defineFirstArg(1, add);
addOne(22); // -> 23
```

Higher Order Functions

More Examples

Wrapping another function:

```
function add(x,y){
  return x + y;
}

function wrapWithLog(fun){
  return function(...args) {
    console.log('Arguments:', args);
    const returnValue = fun(...args);
    console.log('Return Value', returnValue);
    return returnValue;
  }
}

const wrappedAdd = wrapWithLog(add);

console.log(wrappedAdd(41, 1));
```

Composing other functions:

```
function add2(x){ return x + 2; }
function multiplyBy4(x){ return x * 4; }
function divideBy5(x){ return x / 5; }

function pipe(...funs) {
  return function(x){
    const value = x;
    for (const fun of funs){
      value = fun(value);
    }
    return value;
  }
}

cosnt calc = pipe(add2, multiplyBy4, divideBy5);
console.log(calc(3));
```

Note: these examples use the *rest paramater* syntax (`...args`) of ES2015.

`call()` and `apply()`

Functions are objects that can be (re)used in different contexts.

When a function is executed it is bound to a context. The context is represented by the `this` keyword in the function body.

Every function has two properties: `call` and `apply`. These are functions which execute a function with a given context:

```
const context = {prop: 'Hello'};
const doIt = function(arg1, arg2){...}
doIt.call(context, "foo", "bar");
doIt.apply(context, ["foo", "bar"]);
```

Closures

Inner Functions

- Function can be nested inside functions
- Nested functions can access variables in their parent function's scope

```
function outerFunction() {
    const a = 1;
    function innerFunction() {
        return a + 1;
    }
    return innerFunction();
}
const x = outerFunction();
console.log(x);
```

calling the inner function while the outer function is still processing => no closure involved

```
function outerFunction() {
    const a = 1;
    function innerFunction() {
        return a + 1;
    }
    return innerFunction;
}
const x = outerFunction();
console.log(x);
console.log(x());
```

the inner function "closes over" the scope of the outer function

the scope of the outer function is available, even though the outer function has completed

Closures

- Inner functions contain the scope of parent functions even if the parent function has returned.
- A closure is a special kind of object that combines two things: a function, and the environment in which that function was created. The environment consists of any local variables that were in-scope at the time that the closure was created.

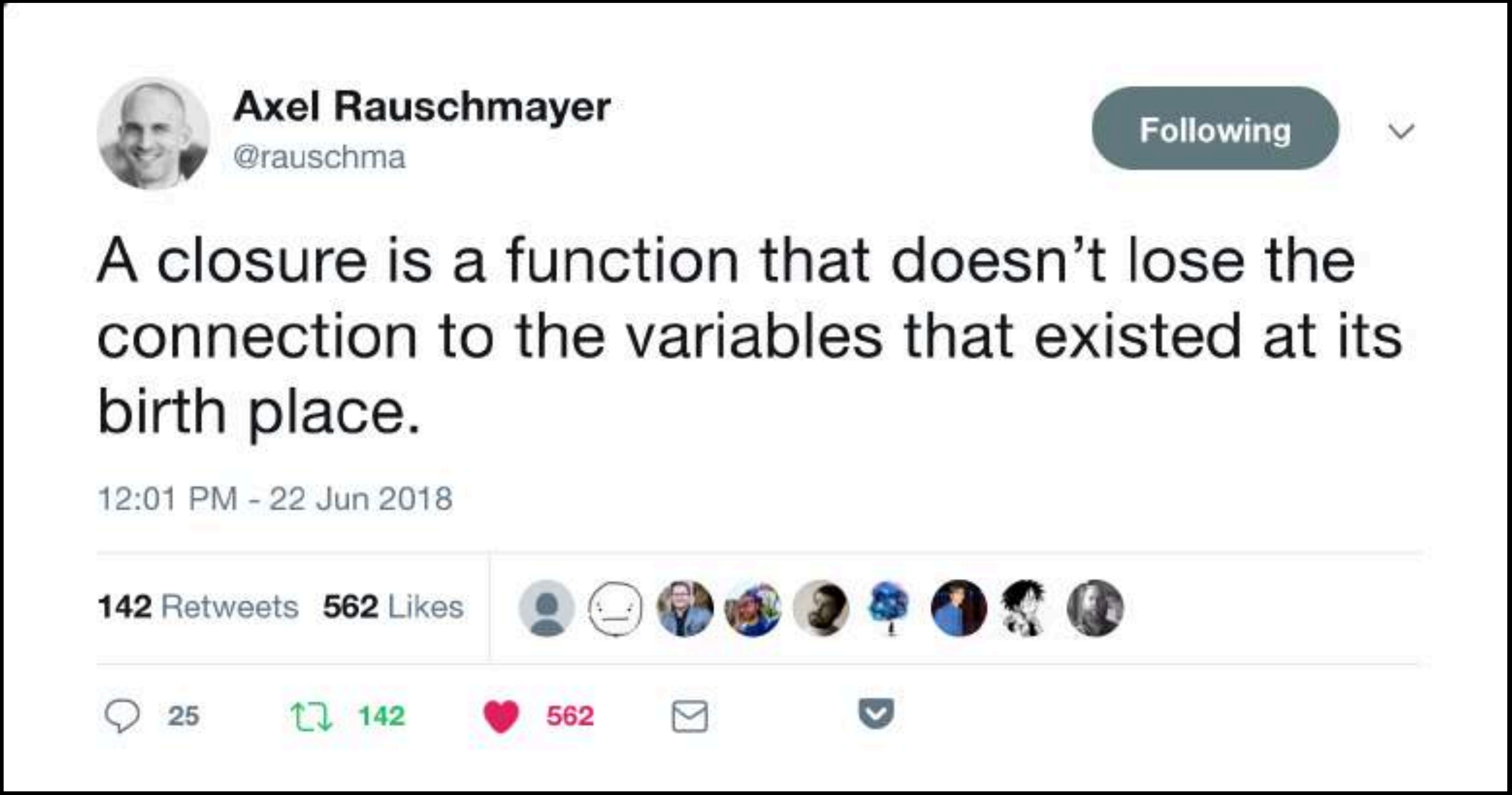
```
const name = 'Bob';
function sayHello() {
     alert(name);
}

name = 'Tim';
sayHello(); // --> Tim
```

```
var name = 'Bob';
function getGreetingClosure() {
    const text = 'Hello ' + name;
    return function() {
              console.log(text);
           };
}
const sayHello = getGreetingClosure();

name = 'Tim';
sayHello(); // --> Hello Bob
```

inner function
+ context
= closure

EXERCISES

Exercise - Scope


Axel Rauschmayer
@rauschma

A closure is a function that doesn’t lose the connection to the variables that existed at its birth place.

12:01 PM - 22 Jun 2018

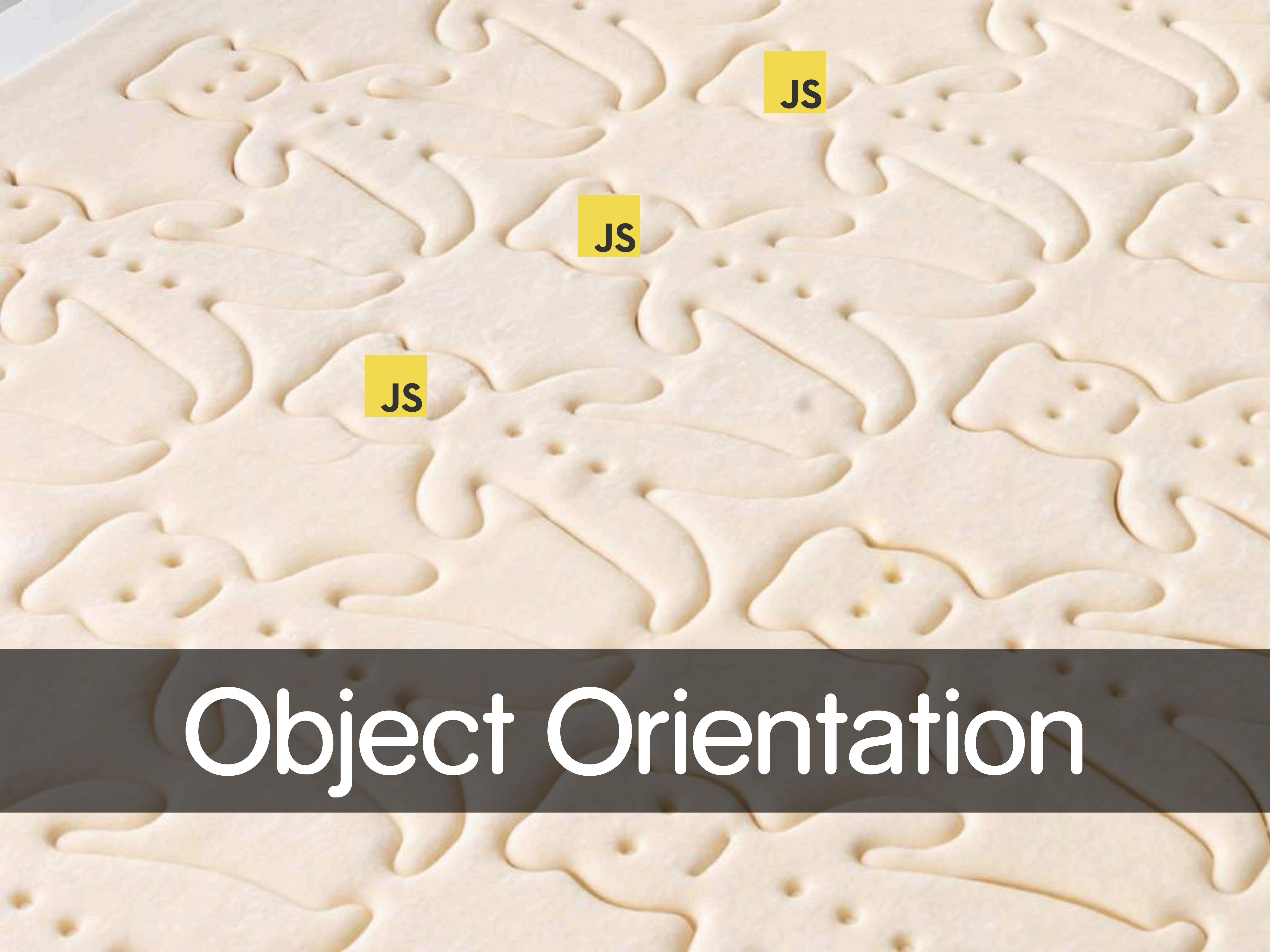
142 Retweets 562 Likes

25 142 562

https://twitter.com/rauschma/status/1010160848491425792

We can think of closures as of “ghosts” or “memories” of the past function calls.

- Dan Abramov, whatthefuck.is/closure



Object Orientation

Creating Objects

Object Literal:

```
const obj = {}
```

Constructor Function:

```
const obj = new Object()
```

ES2015: Class Instance:

```
class Person {}

const obj = new Person()
```

Adding Behaviour to Objects

Functions are "first class values", as a consequence we can assign functions to object properties.

```
var firstName = 'Jonas';
var lastName = 'Bandi';
var greet = function(){
        console.log('Hello!');}

var o1 = {
    firstName: firstName,
    lastName: lastName,
    greet: greet
};
```

ES5 JS

2015 ES

```
const firstName = 'Jonas';
const lastName = 'Bandi';
const greet = function(){
        console.log('Hello!');}
const propName = 'myprop';

var o1 = {
    firstName,
    lastName,
    greet,
    say(){console.log('Hi!')};
    [propName]: 'gugus'
};
```

Idiomatic Classes

aka: "How you should use classes today"

```
class Counter {

    count = 0;

    constructor(initialValue){
        this.count = initialValue;
    }

    increase = () => {
        this.count += 1;
    };

    print(){
      console.log(`Count is: ${this.count}`);
    }

    get description(){
      return `Count is: ${this.count}`;
    }
}
```

class fields

method

getter
(setter also
possible)

```
const pers = new Counter(42);
console.log(person.description);
console.log(`Count is: ${person.count}`);
pers.print();

setTimeout(pers.increase, 1000);
setTimeout(() => pers.print(), 2000);
setTimeout(pers.print, 3000);
```

ensuring 'this'
binding with
arrow function

'this' is not
bound!

Classes & Inheritance

```
class Person {

    firstName = '';
    lastName = '';

    constructor(firstName, lastName){
        this.firstName = firstName;
        this.lastName = lastName;
    }

    fullName() {
        return this.firstName + ' ' + this.lastName
    };
}

const pers = new Person('John', 'Doe');
```

```
class Employee extends Person {
    company = '';
    constructor(firstName, company){
        super(firstName, lastName);
        this.company = company;
    }
}

const empl = new Employee('John', 'Doe', 'EvilCorp');
```

Note:

- classes can have static methods
- fields do not have to be declared - but since ES2022 (and with TypeScript) they can be declared
- typically everything is public - but ES2022 introduced private fields (and TypeScript also has "compile-time" private fields)

ES2022: Private Fields & Methods

```
class Car {
  #milesDriven = 0
  drive(distance) {
    this.#milesDriven += distance
  }
  getMilesDriven() {
    return this.#milesDriven
  }
}

const tesla = new Car()
tesla.drive(10)
tesla.getMilesDriven() // 10
tesla.#milesDriven // Invalid
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes/Private_class_fields

Private fields and methods are part of ECMAScript 2022.
Implemented in all modern browsers and in TypeScript and other transpilers.

Note: classes with private fields cannot be proxied!
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy#no_private_property_forwarding
https://lea.verou.me/2023/04/private-fields-considered-harmful/

The `this` Keyword

For regular functions the `this` keyword references the context of that function. The value of this is determined by how the function is called:

1. If the function is called as a `constructor` (either a class constructor or a function called with `new`) => `this` is a new object
2. If the method is called via `call()` or `apply()` => this is explicitly passed
3. If the function is called as a *method* of an object => this points to that object
4. If none of the above are used => `this` is the global object or undefined in *strict mode*.

Especially for callbacks it might be tricky to find out the value of this.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/this
http://2ality.com/2017/12/alternate-this.html

Arrow Functions

Arrow function do not explicitly bind **`this`** (aka. *lexical* binding for **`this`**):
The value of **`this`** inside the function body is resolved along the scope chain.
This means that the **`this`** value of the enclosing execution context is used.

```
class Controller {
  constructor() {
    this.count = 0;
  }
  increment() {
    this.count++;
    console.log(this.count);
  }
}

let controller = new Controller();
setTimeout(controller.increment, 0); //-> NaN
```

class method

passed as function

```
class Controller {
  constructor() {
    this.count = 0;
    this.increment = () => {
      this.count++;
      console.log(this.count);
    };
  }
}

let controller = new Controller();
setTimeout(controller.increment, 0); -> 1
```

class field

arrow function

passed as arrow function

Note: functions assigned to *class fields* are not on the prototype and can't be accessed via the **`super`** keyword from a base class!

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions
Real World Use Case: https://reactjs.org/docs/handling-events.html

EcmaScript Modules

EcmaScript Modules (ESM)

In ES5 sharing constructs between files was only possible via the global namespace. Module- and Namespace Patterns helped to restrict what is exposed. Dependencies were managed implicitly.

`library.js`

```
window.doSomething
            = function(){…};
```

`program.js`

```
window.doSomething();
```

ES2015 introduces modules to declare dependencies explicitly.

```
export function doSomething(){
  …
};
```

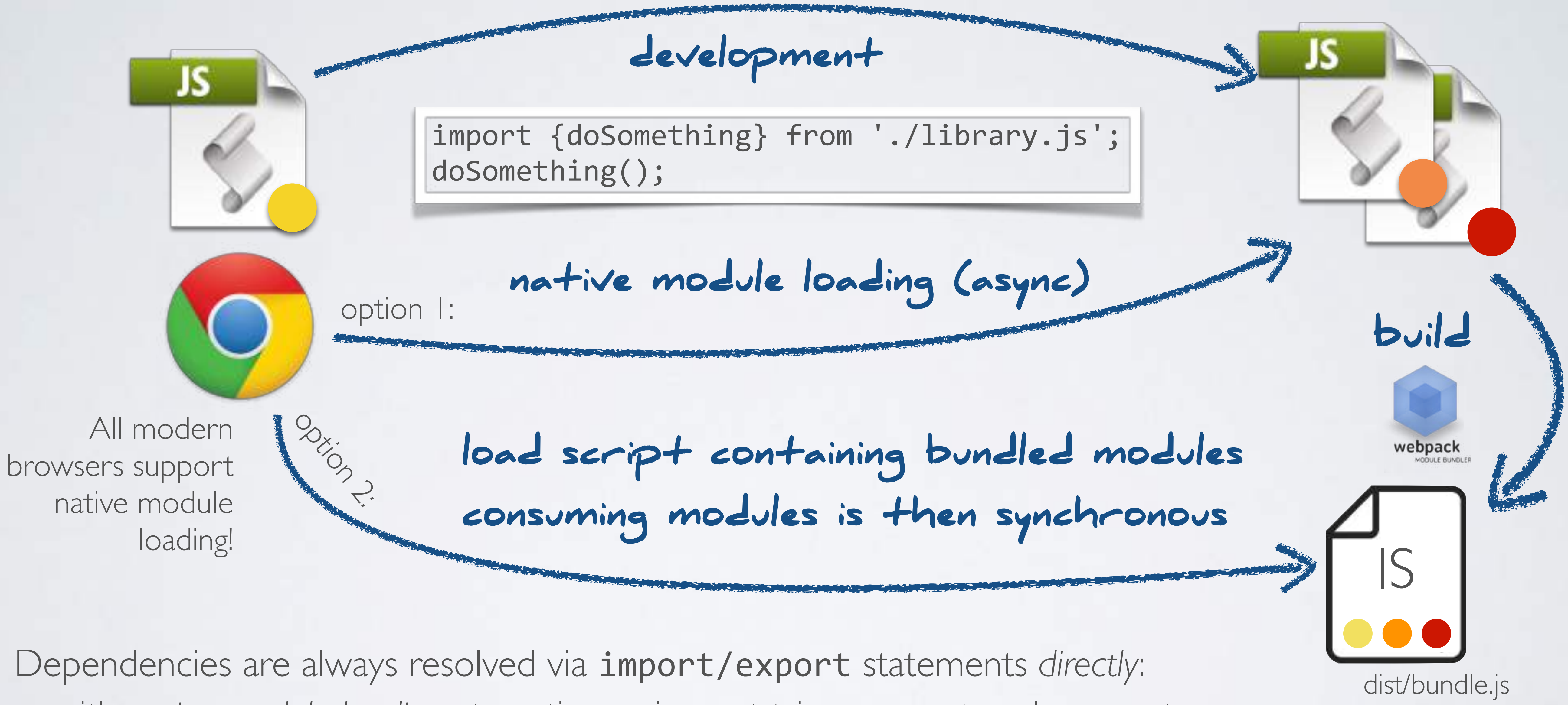
```
import {doSomething}
       from './library.js';

doSomething();
```

At development time each module is a separate file.
At build time modules can (optinally) be bundled with a module bundler (webpack, rollup ...).
At runtime modules can be loaded as separate file (no bundling) or loaded in one or several bundles.

https://hacks.mozilla.org/2018/03/es-modules-a-cartoon-deep-dive/

ES2015 Modules

Special challenge: in the browser scripts are loaded asynchronously!



development

```
import {doSomething} from './library.js';
doSomething();
```

option 1: native module loading (async)

All modern browsers support native module loading!

option 2: load script containing bundled modules consuming modules is then synchronous

build

dist/bundle.js

Dependencies are always resolved via **`import/export`** statements *directly*:

- with *native module loading* at runtime: a import triggers a network request
- with *bundling* at build time: a import defines that the module is included in the bundle (dependency resolution via bundler, typically using node resolution (i.e 3rd party modules are in node_modules installed via npm & package.json)

https://caniuse.com/#search=module

ES2015 Modules Semantics

- Modules are executed when imported
- A modules is only executed once (even when imported several times)
- Modules are always executed in strict mode
- Modules have a top-level scope that is not the global scope
- Modules can `export` bindings to variables, functions or classes
- Modules can `import` bindings from other modules
- `import` bindings are readonly
- `import` statements are hoisted and can't be dynamic

EXERCISES

Exercise - Modules

Modules: Syntax

There are different ways to export & import

`library.js`

```
export function
      doSomething(){...};

export let dataObj = {...};

export default () => {...};

export {fun1 as fun2, obj};
```

`program.js`

```
import './library1.js';

import {doSomething, dataObj}
      from './library2.js';

import {doSomething as doIt}
      from './library3.js';

import * as lib3
      from './library4.js';

import defaultExport
      from './library5.js';
```

Opinion: Avoid `export default`: https://basarat.gitbook.io/typescript/main-1/defaultisbad

Default Export vs. Named Export

```
export default class Person {}
```

```
import MyPerson from './person'
```

```
export class Person {}
```

```
import {Person} from './person'
```

Technical difference:
A *named export* is a "live" binding while a *default export* is not.
With a "live" binding the client "sees" later changes by the provider.

Argumentation for default export (howeveer this style guide is not "state of the art"):
https://github.com/airbnb/javascript#modules--prefer-default-export

Argumentation for named export:
https://basarat.gitbook.io/typescript/main-1/defaultisbad

The default export can be error prone (different behavior depending on environment):
https://esbuild.github.io/content-types/#default-interop

Some frameworks require default exports for certain constructs:
React.lazy, route definitions in Remix and Next.js, Angular lazy loading

Personally, I prefer the "TypeScript Way" of Named Exports.

Modules at Runtime

Module *loading* is not part of the ECMA specification!
It took a while until browser supported modules natively.

Today all modern browsers support *static* module imports:
https://caniuse.com/#feat=es6-module

```
<script type="module" src="app.js"></script>
```

(module scripts are executed after the html parsing has finished, therefore are ideally placed in the <head>
https://html.spec.whatwg.org/multipage/scripting.html#attr-script-defer)

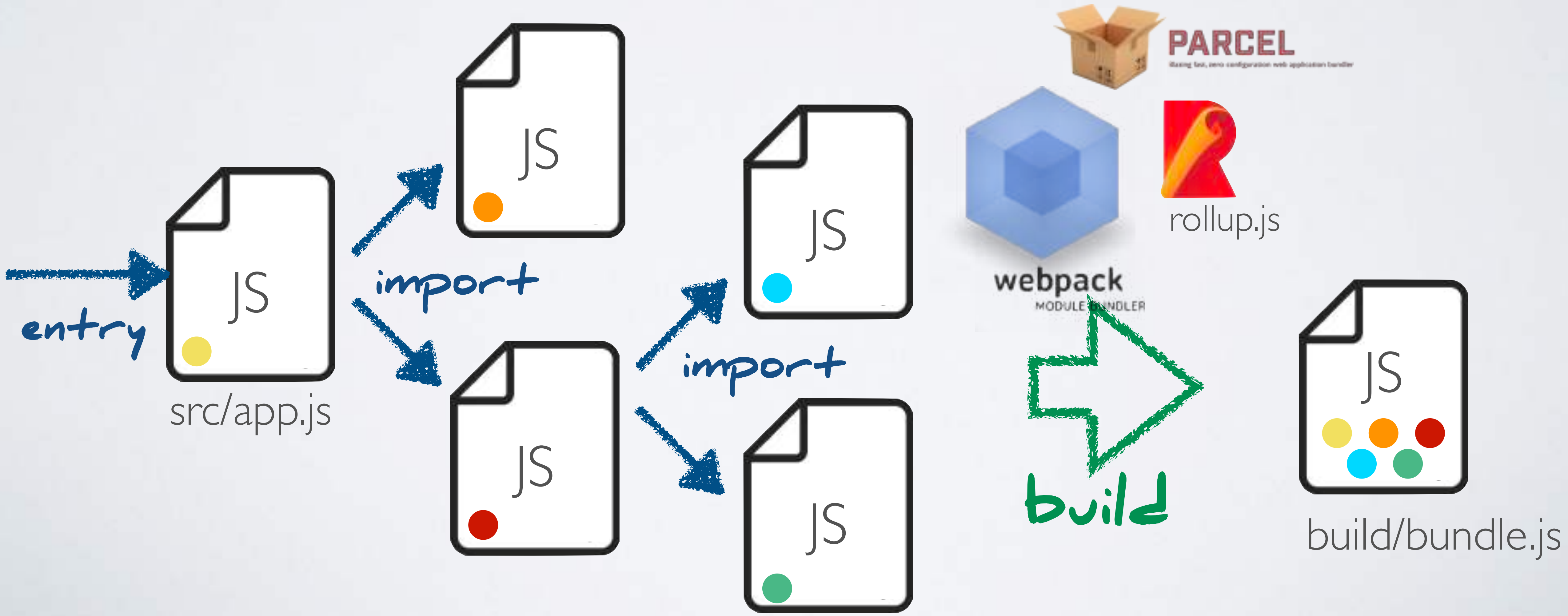
For better performance non-trivial web apps should use a bundler and not native module loading (even with http/2).
https://developers.google.com/web/fundamentals/primers/modules

ECMAScript modules are supported in Node.js v16 and later
https://nodejs.org/docs/latest/api/esm.html

Modules at Build Time

The primary use-case for modules today is build-tooling.
Modern *bundlers* (WebPack, Rollup, Parcel ...) work with modules:

- build a dependency tree based on fine grained ecma script modules (import/export)
- create coarse grained JavaScript bundles which are optimized to be loaded by a browser.



Dynamic Module Loading

Dynamic imports are part of ES2020:

```
import('./second.js')
      .then(m => m.sayHello());
```

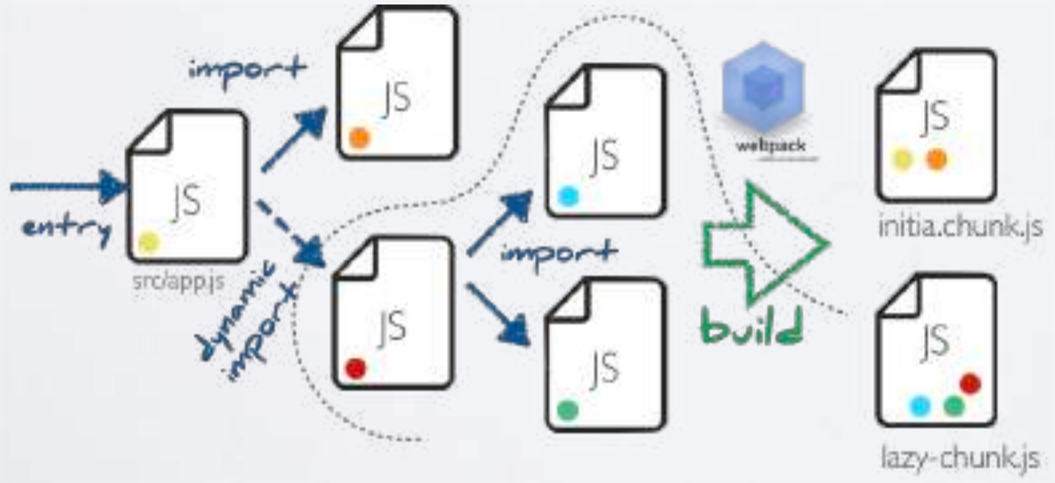
function call

Runtime: supported in all modern browsers.

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import#Dynamic_Imports
https://caniuse.com/#feat=es6-module-dynamic-import

Build-Time / Bundling:

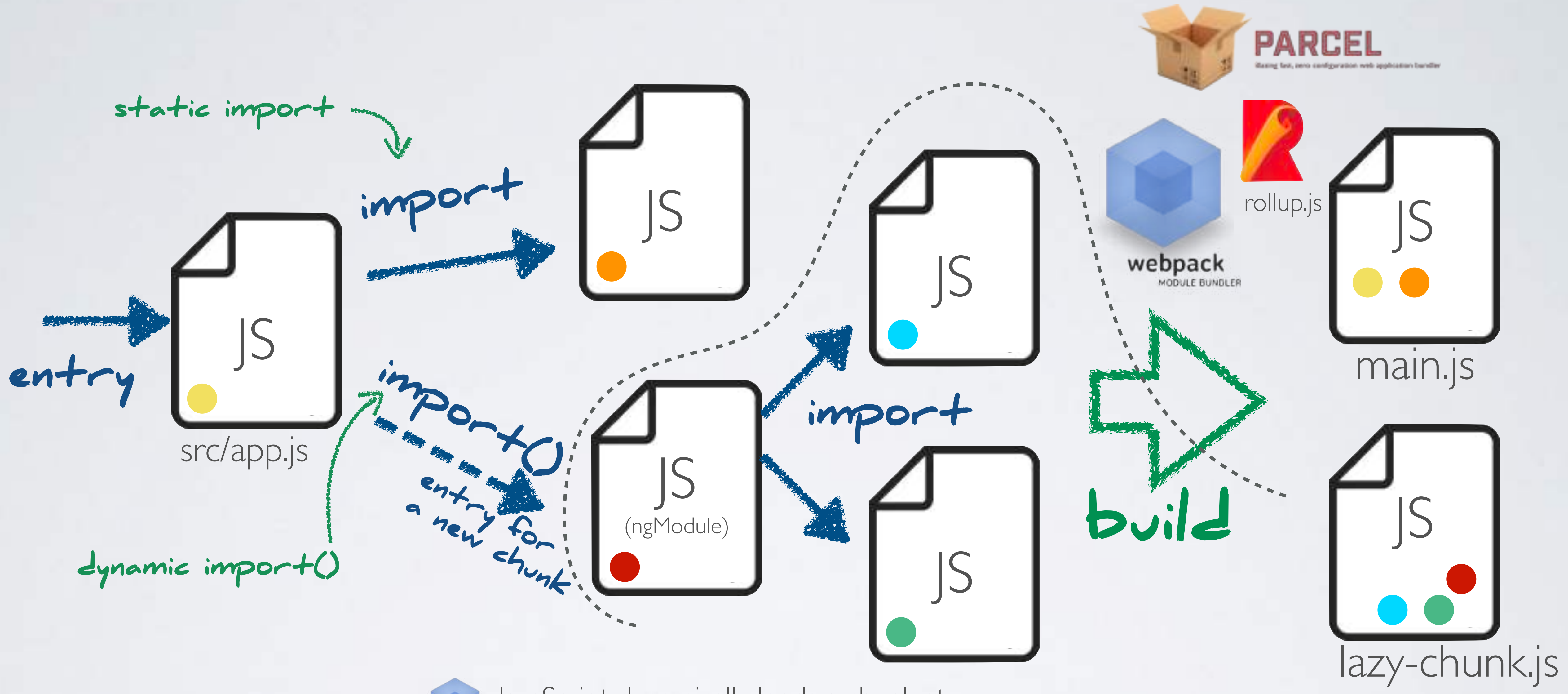
Dynamic imports are supported by modern bundlers (webpack, rollup, parcel):
Based on `import()` a separate JavaScript bundle is created at build time, that can be asynchronously loaded by the browser when needed.
This is typically used to keep the initial bundle small. This is also called: *code-splitting or lazy-loading* of *chunks*.



https://webpack.js.org/guides/code-splitting/
https://parceljs.org/code_splitting.html
https://rollupjs.org/guide/en#experimental-code-splitting

traditional

Lazy Loading



JavaScript dynamically loads a chunk at *runtime*. The code is (partially) generated by the bundler.
Frameworks have additional abstractions (Angular Router, React.lazy ...)

All the application parts (source, npm-packages ...) must be available at build time.

One build creates a *single* deployment artifact consisting of several chunks.

Import Maps

https://developer.mozilla.org/en-US/docs/Web/HTML/Reference/Elements/script/type/importmap

Html:

```
<script type="importmap">
    {
      "imports": {
        "chalk": "https://cdn.skypack.dev/chalk-web"
      }
    }
</script>
```

JS (directly loaded in the browser, no bundler):

```
import chalk from "chalk";
```

Enables decoupling of source code from URL/Version.
Makes code portable between Browser and Node/Bundlers.

$f(x)$

Modern JS Syntax Elements

Strings

JavaScript Strings:

```
const aString = "First";
const anotherString = 'Second';
const oneMore = `Third`; // this is a template string
```

Template Strings:

```
const formatted = `Third
can contain
  multi lines
    and white spaces
`;
```

```
const val = 42;
const template = `Can interpolate ${val}`;
```

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

nullish coalescing & optional chaining

```
const size = settings.size ?? 42;
```

size or 42

checks for null or undefined (not for falsy values)

```
const txtName = document.getElementById("txtName");
const name = txtName?.value;
```

value or undefined

checks for null or undefined

```
const customerCity = invoice?.customer?.address?.city;
```

```
const userName = user?.["name"];
```

```
const fullName = user.getFullName?.();
```

introduced in ES2020: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for-await...of

Spread

Expand an iterable or object to its elements.
Spread syntax enables elegant copying and merging of arrays and objects.

Spread for Arrays (specified in ES2015):

```
const sourceArr1 = [1,2,3,4];
const sourceArr2 = [5,6,7,8];
const mergedArray = [...sourceArr1, ...sourceArr2, 9];
console.log(mergedArray);
```

Another usage secenario:

```
function addThreeThings(a, b, c) {
    return a + b + c;
}
const result = addThreeThings(...[1,2,3]));
```

Spread for Objects (specified in ES2018):

```
const sourceObj1 = {a: 1, b:2, c:3};
const sourceObj2 = {c: 5, d:6, e:7};
const mergedObject = {...sourceObj1, ...sourceObj2, f:9};
console.log(mergedObject);
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

Destructuring

Break-up an object into variables

```
const obj = {three: 3, four: 4};
const {three, four, other = 42} = obj;
const {three:five, four:six} = obj; // renaming
console.log(three);
console.log(six);
```

Break-up an array into variables

```
const array = [1,2, 'a'];
const [one, two, three = '33'] = array;
console.log(one);
console.log(two);
```

Destructuring Examples

Selecting values from a parameter object:

```
function doSomething({param1, param3 = 42}){
     console.log('params: ', param1, param3);
}

const params = {param1: 1, param2: 2, param3: 3};
doSomething(params);
```

Selecting values from a return object:

```
function getValue(){
     return {val1: 1, val2: 2, val3: 3};
}

const {val1, val3 = 42} = getValue();
console.log('values: ', val1, val3);
```

Multiple return values:

```
function useElement(){
  let element = 42;
  const getElement = () => element;
  const setElement = e => element = e;
  return [getElement, setElement];
}
const [getValue, setValue] = useElement();
```

Nested destructuring:

```
const {val2: {b = 42}} = {val1: 1, val2: {a:8, b:9}, val3: 3};
console.log(b);
```

Object Literal Syntax

```
var firstName = 'Jonas';
var lastName = 'Bandi';
var greet = function(){
       console.log('Hello!');}

var o1 = {
    firstName: firstName,
    lastName: lastName,
    greet: greet
};
```

ES5 JS

2015 ES

```
const firstName = 'Jonas';
const lastName = 'Bandi';
const greet = function(){
       console.log('Hello!');}
const propName = 'myprop';

var o1 = {
    firstName,
    lastName,
    greet,
    say(){console.log('Hi!')};
    [propName]: 'gugus'
};
```

property shortcut

method shortcut

dynamic property

Default Parameters

```
function greet(message, name = 'Joe'){
    console.log(message + ', ' + name);
}

greet('Hello'); // -> Hello, Joe
greet('Goodbye', 'Jeff'); // -> Goodbye, Jeff
```

Rest Paramaters

The rest parameter syntax allows a function to accept an indefinite number of arguments as an array.

```
function multiGreet(message, ...names){
    names.forEach((name) =>
        console.log(message + ' ' + name));
}

multiGreet('Hi', 'John', 'Jane', 'Alice');
```

array

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/rest_parameters

Iterators

In ES2015 the iteration protocol is standardized.
Many built in constructs are iterable.
Custom iterators can be implemented using **`Symbol.iterator`**.

```
const a = [1,2,3,4,5];

const iter = a[Symbol.iterator]();

console.log(iter.next());
// ->  {value: 1, done: false}
```

- The `for...of` loop works with iterables.
- The spread operator (`...args`) works with iterables.
- *Generators* can be used to create custom iterators.

```
const fibonacci = {
  [Symbol.iterator]() {
    let pre = 0, cur = 1;
    return {
      next() {
        [pre, cur] = [cur, pre + cur];
        return { done: false, value: cur }
      }
    }
  }
}

for (const n of fibonacci) {
  // truncate the sequence at 1000
  if (n > 1000)
    break;
  console.log(n);
}
```

Set & Map

Efficient data structures for common algorithms.

In ES5 objects could be used as "hash maps" (key-value stores).

`Set`, `Map`, `WeakSet`, `WeakMap` are built in objects in ES2015.

```
const s = new Set();
s.add("hello").add("goodbye").add("hello");
s.size === 2;
s.has("hello") === true;
```

```
const m = new Map();
m.set("hello", 42);
m.set(s, 34);
m.get(s) == 34;
```

Set, Map:
- values (and keys) can be primitive types and objects.
- are iterable

WeakSet, WeakMap:
- values (and keys) can objects.
- are not iterable
- values are garbage collected if not referenced from somewhere else

Symbol

Symbols are guaranteed unique values.

`symbol` is a new primitive data type in ES2015.

A JavaScript runtime has built in ("well-known") symbols.

The `Symbol()` function creates a new value of type symbol.

Built in symbols expose language behaviors:

```
const a = [1,2,3,4,5];

const iter = a[Symbol.iterator]();

console.log(iter.next());
// ->  {value: 1, done: false}
```

New symbols can be used to create unique values:

```
Symbol('foo') === Symbol('foo'); // false
```

```
const key = Symbol("hidden");

const obj = {
  name: 'Jonas',
  [key]: 'Not visible!'
};

console.log(obj.hidden); // undefined
console.log(obj[key]); // "Not visible!"
```

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

Proxies

Proxies can wrap an object and "trap" (intercept) interactions with the wrapped object.

```
const target = {};
const handler = {
  get: function (receiver, name) {
    return `Hello, ${name}!`;
  }
};

const p = new Proxy(target, handler);
console.log(p.world); // Hello, world!
```

```
const target = function () {
  return "Hello, world!"
};
const handler = {
  apply: function (receiver, ...args) {
    return "Proxy: " + receiver();
  }
};

const p = new Proxy(target, handler);
console.log(p('world')); // Proxy: Hello, world!
```

Proxies can not be transpiled nor polyfilled!

Private class fields cannot be proxied!

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy#no_private_property_forwarding
https://lea.verou.me/2023/04/private-fields-considered-harmful/

Generators

A generator is a special type of function that creates an iterator. It can **yield** the control flow and maintain it's state.

```
function* idMaker(){
  const index = 0;
  while(true){
   console.log('returning next val ...');
    yield index++;
  }
}

const iter = idMaker();
console.log(iter.next().value); // 0
console.log(iter.next().value); // 1
```

https://developer.mozilla.org/en/docs/Web/JavaScript/Guide/Iterators_and_Generators

Promises & async / await

ES2018: Asynchronous Iteration

`for-await-of` and async generators

```
const peopleUrls = [
  'https://swapi.co/api/people/1/',
  'https://swapi.co/api/people/2/'
];

async function* createAsyncIterable() {
  for (const url of peopleUrls) {
    yield axios.get(url);
  }
}
```

```
async function run() {
  const asyncIterable = createAsyncIterable()
  const asyncIterator =
            asyncIterable[Symbol.asyncIterator]();

  for await (const response of asyncIterator) {
    console.log(response);
  }
}
run();
```

introduced in ES2018: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/for-await...of

Future JavaScript

Decorators

May 2025: Stage 3

```
@Component({
   selector: 'app-counter',
   templateUrl: 'counter.component.html'
})
class CounterComponent {
  constructor(){
   this.count = 0;
  }
  increase(){
    this.count++;
  }
}
```

Decorators can be used to attach metadata to classes. Frameworks can then use this metadata.

The specification for class fields is currently at stage 3.
Browsers do not yet supports decorators.
Decorators are supported in TypeScript and Babel.

https://github.com/tc39/proposal-decorators

using Declarations & Explicit Resource Management

May 2025: Stage 3
Supported in TypeScript

```
class TempFile implements Disposable {
    #path: string;
    #handle: number;

    constructor(path: string) {
        this.#path = path;
        this.#handle = fs.openSync(path, "w+");
    }

    // other methods

    [Symbol.dispose]() {
        // Close the file and delete it.
        fs.closeSync(this.#handle);
        fs.unlinkSync(this.#path);
    }
}
```

optional TypeScript interface

built-in Symbol

```
export function doSomeWork() {
    using file = new TempFile(".some_temp_file");

    // use file...

    if (someCondition()) {
        // do some more work...
        return;
    }
}
```

dispose is called automatically at the end of the scope

There is also `Symbol.asyncDispose` and `await using` for asynchronous resource disposal.

`DisposabeStack` and `AsyncDisposableStack` are new built-in objects to wrap cleanup logic.

https://github.com/tc39/proposal-explicit-resource-management
https://devblogs.microsoft.com/typescript/announcing-typescript-5-2-beta/#using-declarations-and-explicit-resource-management

Signals

Modeling Reactivity with a build in construct.

May 2025: Stage 1

```
const counter = new Signal.State(0);
const isEven = new Signal.Computed(() => (counter.get() & 1) == 0);
const parity = new Signal.Computed(() => isEven.get() ? "even" : "odd");

// A library or framework defines effects based on other Signal primitives
declare function effect(cb: () => void): (() => void);

effect(() => element.innerText = parity.get());

// Simulate external updates to counter...
setInterval(() => counter.set(counter.get() + 1), 1000)
```

Many frameworks implement Signals for *fine grained reactivity:*
Angular (v16), Solid, Vue (Composition Api), Preact, Svelte (v5)

Futur JavaScript might standardize Signals an bring them into the platform.

https://github.com/tc39/proposal-signals

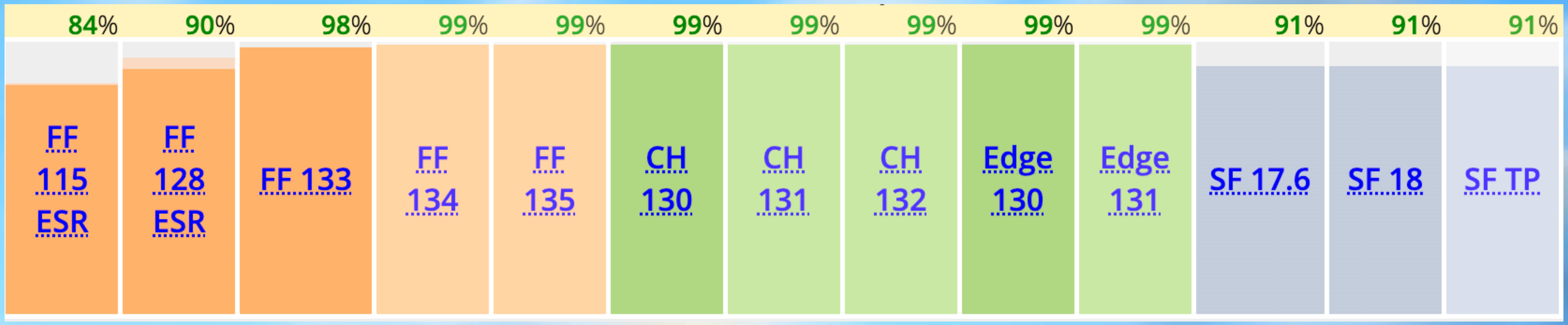
JavaScript Enums: Stage 0
https://github.com/tc39/proposal-enum

Temporal: Stage 3
https://github.com/tc39/proposal-temporal

Composites: Stage 1 (Replacement for Records and Tuples)
https://github.com/tc39/proposal-composites

Web Platform Observables:
https://github.com/wicg/observable

ECMAScript browser support:

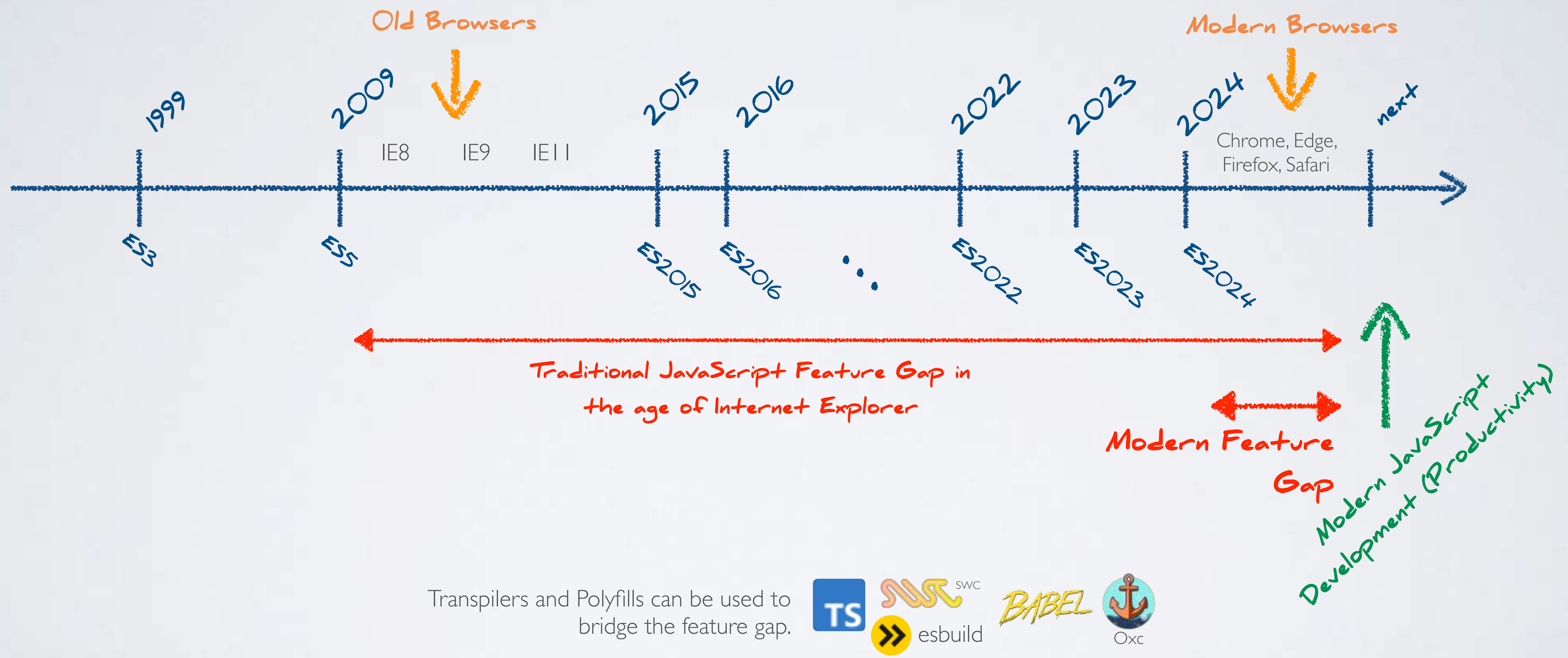


84%	90%	98%	99%	99%	99%	99%	99%	99%	99%	91%	91%	91%
FF 115 ESR	FF 128 ESR	FF 133	FF 134	FF 135	CH 130	CH 131	CH 132	Edge 130	Edge 131	SF 17.6	SF 18	SF TP

https://compat-table.github.io/compat-table/es6/

The Feature Gap

In the Age of IE and Today



Transpilers and Polyfills can be used to bridge the feature gap.

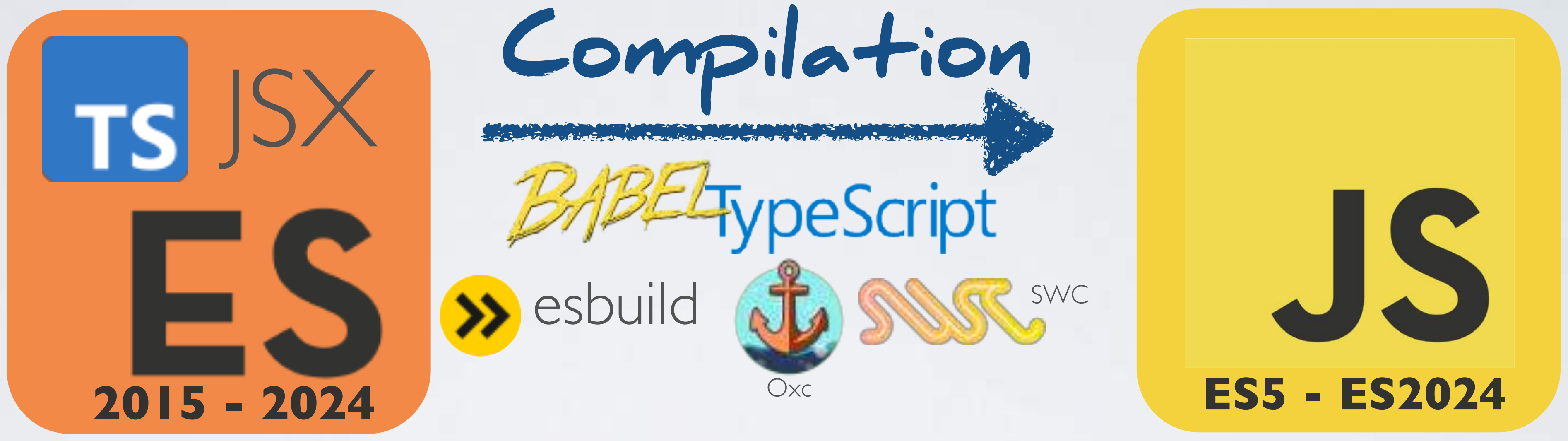
Modern frameworks dropped IE support:

- Angular v12 (2021)
- React 18 (2022)
- Vue 3 (2020)

https://compat-table.github.io/compat-table/es6/

Transpiler / Compiler / Transformer

A transpiler is a scource-code to source-code compiler.



Most modern and even future ECMAScript syntax can be compiled to older versions of ECMAScript or even ES5.

```
class Person {
  private name: string;
  constructor(name: string){
    this.name = name;
  }
}
const pers: Person = new Person('John');
```

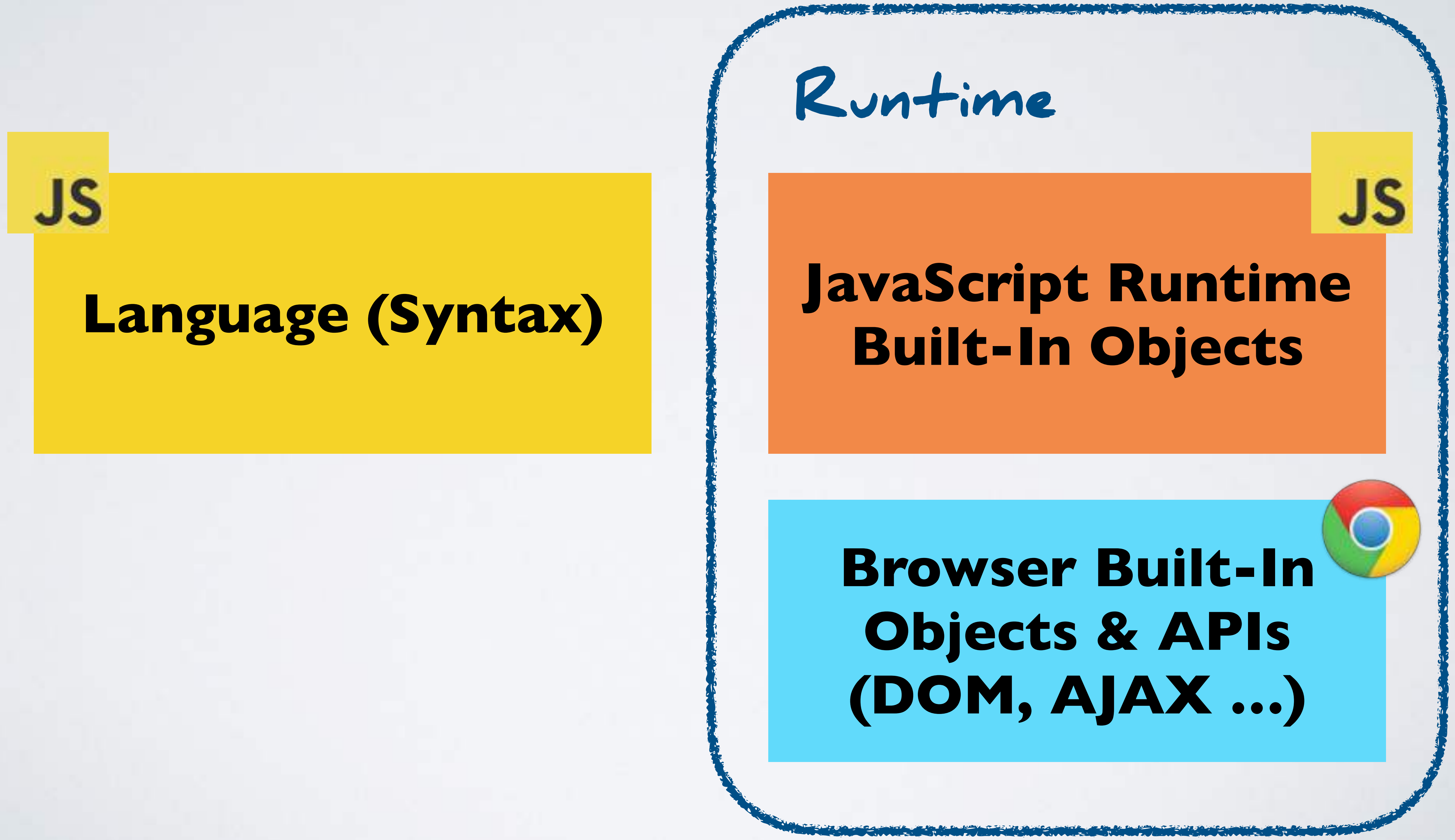
```
var Person = (function () {
    function Person(name) {
        this.name = name;
    }
    return Person;
}());
var pers = new Person();
```

In the age of evergreen modern browser compiling ECMAScript language features is not as relevant as in the past, when IE support was still needed.
But compiling *TypeScript* and *JSX* to plain ECMAScript is essential in most modern projects.

https://babeljs.io/repl/
https://www.typescriptlang.org/play
https://playground.oxc.rs/

https://esbuild.github.io/
https://swc.rs/
https://oxc.rs/

JavaScript


JS
Language (Syntax)
Runtime
JS
JavaScript Runtime Built-In Objects
Browser Built-In Objects & APIs (DOM, AJAX ...)

Modern ES Support: Transpiler & Polyfills

New Language Features (Syntax)

transpiling new syntax to supported syntax

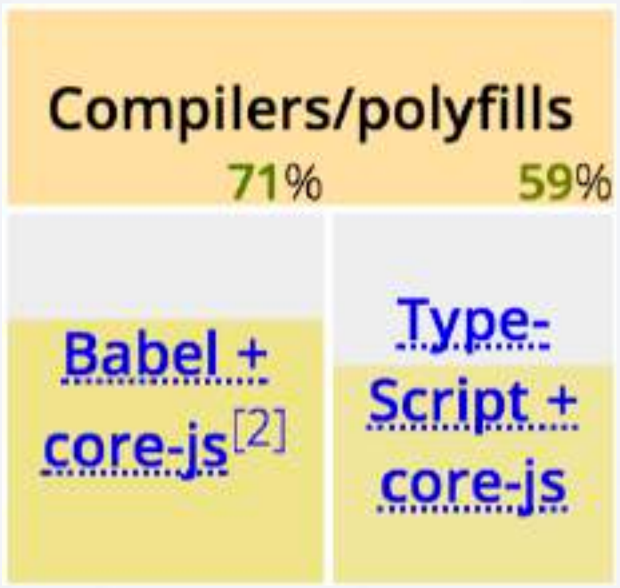
TypeScript

BABEL

esbuild

SWC

Oxc



Compilers/polyfills	
71%	59%
Babel + core-js[2]	Type-Script + core-js

https://kangax.github.io/compat-table/es6/

New Built-In Functionality (Runtime Objects)

polyfills for new built in features

A polyfill is a a JavaScript library that implements a missing browser feature.

core-js

https://github.com/zloirock/core-js
https://cdnjs.cloudflare.com/polyfill/