Session 02

JavaScript

CIT 2025, section 3

Morten Rhiger

2025-11-17 09:01:38

In this session

The JavaScript language.

(With focus on the parts important for implementing UIs in React.)

Running JavaScript

- Using **Node.js**. (Type node in a terminal.)
  Advantage: Straightforward.

- In **the browser**. (In Chrome, Ctrl-Shift-I, then select tab Console.)
  Advantage: dynamic completion of object fields and methods.

- In the **source code** of a React **app**.
  Advantage: has access to the props and variables of the application.

Some tools on your development machine are implemented in JavaScript:

- `npm create vite` *`APP`*
- `npm install` *`PACKAGE`*
- `babel` (e.g., the JSX translator)

Other than that, JavaScript is **not needed** on development machine (but it is required in the app running in the browser).

Values

Type		Example values	typeof
Numbers	No distinction between integers and floating-point numbers	`0`, `1`, `-7.0005`, …	`"number"`
Strings	(There is no char type)	`""`, `"hello"`, `'Bob'`, …	`"string"`
Booleans		`true`, `false`	`"boolean"`
Null		`null`	`"object"`
Undefined		`undefined`	`"undefined"`
NaN	"Not a number"	`NaN`	`"number"`
Objects	mapping keys to values	`{ name:"Bob", age:23 }`, `{}`, …	`"object"`
Arrays	of values	`[1, "Bob"]`, `[]`, …	`"object"`
Functions		`(x) => x + 1`, …	`"function"`

Types

- In JavaScript, **values have types**.
  JavaScript is **dynamically** typed.
  JavaScript performs **type checking at runtime**:

```
let x = 42;   // variable
x = true;     // boolean value
x = "Bob";    // string value
```

vs

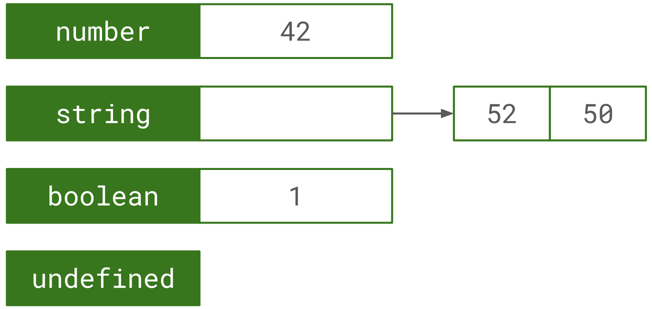
- In C# and Java, **variables have types**.
  C# and Java are **statically** typed.
  C# and Java perform **type checking at compile time**:

```
var x = 42;   // variable of type int
x = true;     // type error!
x = "Bob";    // type error!
```

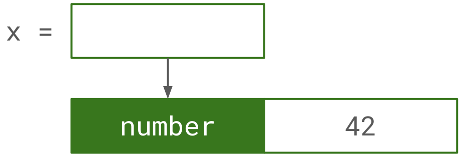
Type tags

- All values carry **type tags**:

```
typeof 42        ⟶ "number"
typeof "42"      ⟶ "string"
typeof true      ⟶ "boolean"
typeof undefined ⟶ "undefined"
```



```
let x;
typeof x         ⟶ "undefined"
x = 42;
typeof x         ⟶ "number"
```



Values : Tips, tricks, pitfalls

- Automatic **conversions**:

```
5 * "4.07"     →  20.35
5 * true       →  5
false - 5      →  -5
```

- **Overloaded** operators:

```
5 + 6          →  11
"U" + "2"      →  "U2"
```

- **Comparisons** (== vs ===):

```
5 == "5"         →  true   (tags differ, converts)
true == "true"   →  false  (no conversion)
0 == false       →  true   (tags differ, converts)
5 === "5"        →  false  (tags differ, hence false)
5 === 6          →  false  (content differ)
5 !== "5"        →  true   (tags differ)
```

Numbers in JavaScript

What you would expect:

```
5 + 4.07              → 9.07

5 * 4.07              → 20.35

parseInt("12")        → 12
parseInt("ff", 16)    → 255
parseFloat("4.07")    → 4.07

(4.07).toString()     → "4.07"
(255).toString(16)    → "ff"

"" + 5                → "5"
```

... but with some peculiarities:

```
5 * "4.07"            → 20.35

5 + "4.07"            → "54.07"
```

Booleans in JavaScript

Falsy values are the values that count as **false** in tests:

- `false`
- `undefined`
- `null`
- `0`
- `""` (the empty string)
- (plus s few more).

Truthy values are the values that count as **true** in tests (all other values):

- `true`
- All non-zero numbers: `1`, `-1`, `42`, `0.001`, …
- All non-empty strings: `"0"`, `"false"`, …
- All arrays: `[]`, `[1,2]`, …
- (plus many more).

Boolean operators

(Conditional expression) `A ? B : C` $\longrightarrow \begin{cases} B\text{, if } A \text{ is truthy} \\ C\text{, otherwise (} A \text{ is falsy)} \end{cases}$

(Logical "and") `A && B` $\longrightarrow \begin{cases} B\text{, if } A \text{ is truthy} \\ A\text{, otherwise (} A \text{ is falsy)} \end{cases}$

(Logical "or") `A || B` $\longrightarrow \begin{cases} A\text{, if } A \text{ is truthy} \\ B\text{, otherwise (} A \text{ is falsy)} \end{cases}$

"Short-circuit" and left-to-right:
Only evaluate B (and C) if necessary.

Arrays and objects

- **Arrays** are base-zero **indexed collections** of values:

  ```
  let a = ["one", "two", "three"];
  a[1]                   → "two"
  ```

- **Objects** are **associative arrays** (or attribute–value pairs): Mappings from names to values:

  ```
  let user = {
    name:    "Bob",
    age:     23,
    isAdmin: false
  };
  user.name              → "Bob"
  ```

- One array may contain elements of different types (a **heterogeneous** array), but try to avoid this situation.

- Objects need **not be declared** by a class (as in OOP), but try to stick to identical structure.

Destructuring values

- The **destructuring** binding syntax **extracts** values from deep inside arrays or objects and **binds** them to variables (both local variables and function parameters).

```
let user = {
  name:    "Bob",
  age:     23,
  isAdmin: false
};

let {name, isAdmin} = user;
  name      → "Bob"
  isAdmin   → false
```

- The **“rest syntax”** binds what is not otherwise matched:

```
let {name, ...rest} = user;
  name     → "Bob"
  rest     → { age: 23,
               isAdmin: false }


let a = ["one", "two", "three"];
let [first, ...others] = a;
  first    → "one"
  others   → ["two", "three"]
```

The spread operator

- The **spread** expression operator (`...x`) "**unfolds**" or "**inserts**" an array or an object (x) into another array or object:

```
let rest = {
  age: 23,
  isAdmin: false
};
{ name: "Arthur", ...rest }
                  → { name: "Arthur",
                      age: 23,
                      isAdmin: false }

let a = [2, 3, 4];
[1, ...a, 5]    → [1, 2, 3, 4, 5]
```

- Arguments to a function can be a "spread array:"

```
function add(x, y) { return x + y; }
let values = [5, 4.07];
add(...values)              → 9.07
```

- Properties to a component can be a "spread object." Here component `User` receives props `name`, `age`, and `isAdmin`:

```
let bob = { name: "Bob",
            age: 23,
            isAdmin: false};
return <User {...bob} />;
```

JSON

Arrays and objects can be **nested**. Typical for representing **data structures**.

JSON (JavaScript Object Notation)

- A **text-based encoding** of data structures.
- A subset of JavaScript itself.
- A file format, widely uses for **transmitting** or **storing** data:
  - `text = JSON.stringify(data)`
  - `data = JSON.parse(text)`

```
{
  "results": [
    {
      "gender": "male",
      "name": {
        "title": "Mr",
        "first": "Lado",
        "last": "Tomashevskiy"
      },
      "location": {
        "street": {
          "number": 9170,
          "name": "Yaroslavska"
        },
        "city": "Lipovec",
        "state": "Ternopilska",
        "country": "Ukraine",
        "postcode": 28336,
      },
      "email": "lado.tomashevskiy@example.com",
      "picture": {
        "large": "https://randomuser.me/api/portraits/men/32.jpg",
        "medium": "https://randomuser.me/api/portraits/med/men/32.jpg",
        "thumbnail": "https://randomuser.me/api/portraits/thumb/men/32.jpg"
      },
    }
  ]
}
```

`let` vs `const`

- `let` introduces variables that we may modify later.

```
let x = 5;
x = x + 1;
x                           → 6
```

- `const` introduces constants: “variables” that cannot be changed.

```
const y = 5;
y = y - 1;                    error!
```

- Pitfall: a const may point to an array or object which **can** be modified:

```
const user = {
               name: "Bob",
               age: 23
             };
user.name = "Arthur";
user       → {
               name: "Arthur",
               age: 23
             }
```

Functions

- Functions are **values**.

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

add          ⟶ [Function: add]
typeof add ⟶ "function"
```

- Functions are **first-class:** they can be passed as **arguments**, returned as **results**, and **stored** in data structures (like any other value).

- A **function expression** (lambda expression, “arrow notation”) produces a function value:

3 × the same

```
function (x, y) {
  return x + y;
}                 ⟶ [Function]
(x, y) => { return x + y; }
                  ⟶ [Function]
(x, y) => x + y   ⟶ [Function]
```

These are **anonymous functions**. (They don’t have a name.)

Defining and using functions

- **Defining** (declaring) functions:

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

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

const add = (x, y) => {
  return x + y;
};

const add = (x, y) => x + y;
```

4 × the same

- **Using** (calling) functions:

$\texttt{add (5, 4.07)} \longrightarrow \texttt{9.07}$

$\underbrace{\texttt{((x, y) => x + y)}}_{\text{A function, and ...}}\ \underbrace{\texttt{(5, 4.07)}}_{\text{its arguments}} \longrightarrow \texttt{9.07}$

Local functions

- Defining a **local function:**

```
function triple(x) {
  const double = (y) => 2 * y;
  return x + double(x);
}

triple(4)                ⟶ 12
double(4)                  error!
```

- Returning a local function:

```
const add = x => y => x + y;
add(5)                   ⟶ [Function]

const addFive = add(5);
addFive(4.07)            ⟶ 9.07
addFive(2)               ⟶ 7
```

- A **closure** represents a local function that has "escaped"; it contains the "missing" (free) variables:

addFive =	function	y => x+y	x=5

Default and “rest” arguments

- Function with **default argument:**

```
function increment(x, y=1) {
  return x + y;
}
increment(4)                 → 5
increment(4, 3)              → 7
```

- Components with default arguments:

```
function Button({label, col="#fff"}) {
  return …;
}

<Button label="Start"/>
<Button label="Stop" col="red" />
```

- Function with a **variable number of arguments,** bound to a “rest” parameter:

```
function sep(s, arg, ...args) {
  let res = arg;
  for (const arg of args)
    res += s + arg;
  return res;
}

sep(" and ", "Bob", "Mary", "Arthur")
     → "Bob and Mary and Arthur"
```

Array methods

Method	Call	Result
`every`	`a.every(f)`	`true` if `f(x)` is `true`, for every `x` in *a*, else `false`.
`some`	`a.some(f)`	`true` if `f(x)` is `true`, for some `x` in *a*, else `false`.
`find`	`a.find(f)`	The first `x` in *a* for which `f(x)` is `true`.
`filter`	`a.filter(f)`	An array of all `x` in *a* for which `f(x)` is `true`.
`map`	`a.map(f)`	An array $[f(x_1), \ldots, f(x_n)]$ when $a = [x_1, \ldots, x_n]$.
`reduce`	`a.reduce(⊕, init)`	$\mathrm{init} \oplus x_1 \oplus \ldots \oplus x_n$, when $a = [x_1, \ldots, x_n]$.
`slice`	`a.slice(i, j)`	An array $[x_i, \ldots, x_{j-1}]$, when $a = [x_1, \ldots, x_n]$.
`toSorted`	`a.toSorted()`	An array of the elements in *a*, sorted.

every

- Are **all of** the first eight primes **odd**?

```
let primes =
    [2, 3, 5, 7, 11, 13, 17, 19];

function isOdd(n) {
  return n % 2;
}

primes.every(isOdd)        → false
```

- `every()` is **higher order:** it takes a function as argument.

- Better: Pass a function expression.

```
primes.every(n => n % 2)  → false
```

- `every()` is **pure:** it leaves the array unmodified.

```
primes
   → [2, 3, 5, 7, 11, 13, 17, 19]
```

`some`, `find`, `filter`, and `map`

- Is **one of** the first eight primes **odd**?

  ```
  primes.some(n => n % 2)      → true
  ```

- What is then the **first** odd prime?

  ```
  primes.find(n => n % 2)      → 3
  ```

- Give me an **array of all** odd primes.

  ```
  primes.filter(n => n % 2)
      → [3, 5, 7, 11, 13, 17, 19]
  ```

- What is the parity (i.e., the last bit) of these primes?

  ```
  primes.map(n => n % 2)
       → [0, 1, 1, 1, 1, 1, 1, 1]
  ```

- Spell out the evenness or oddness of these primes.

  ```
  primes.map(n => n % 2 ? "odd" : "even")
       → ["even", "odd", "odd", "odd",
          "odd", "odd", "odd", "odd"]
  ```

`reduce`

- What is **the sum** of the first eight primes?

  ```
  primes.reduce((r, n) => r + n, 0)
                      → 77
  ```

- The product?

  ```
  primes.reduce((r, n) => r * n, 1)
                      → 9699690
  ```

- The last?

  ```
  primes.reduce((r, n) => n, "none")
                      → 19
  ```

- `reduce` computes a **functional for loop**:

  ```
  let r = 0;
  for (n in primes) {
    r = r + n;
  }
  return r;
  ```

- In general, `a.reduce(f, init)` behaves as

  ```
  let r = init;
  for (x in a) r = f(r, x);
  return r;
  ```

Array methods, loop with index

- In all of

  ```
  a.every(f)
  a.some(f)
  a.find(f)
  a.filter(f)
  a.map(f)
  a.reduce(f, init)
  ```

  function `f` may take the **index** of the current element as an extra argument. (In `reduce`, `f` thus takes three arguments.)

- Primes with their index:

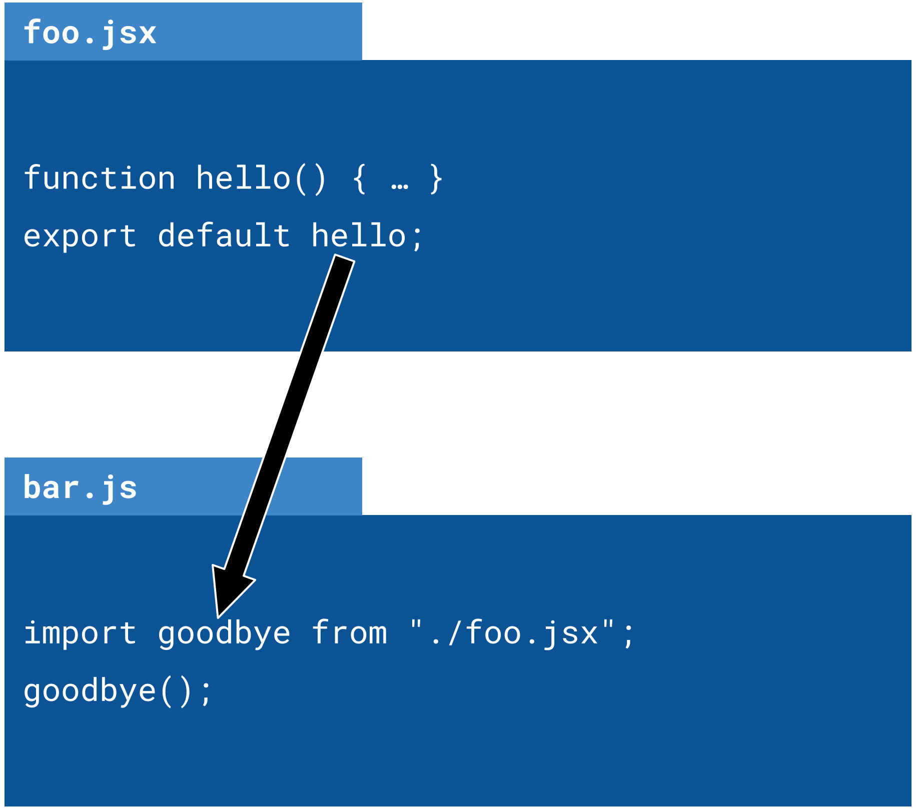
  ```
  primes.map((p, i) => i + ":" + p)
      → [
           "0:2",  "1:3",
           "2:5",  "3:7",
           "4:11", "5:13",
           "6:17", "7:19"
        ]
  ```

&

Chaining method calls

```
primes.filter(n => n % 2)              // find odd primes
      .map(n => n - 1)                 // decrement them by 1
      .map(n => n / 2)                 // divide by 2
      .reduce((r, n) => r + n, 0)      // compute the sum
```

`import` and `export default`



foo.jsx

```
function hello() { … }
export default hello;
```

bar.js

```
import goodbye from "./foo.jsx";
goodbye();
```

- Export a “main” function from file `foo.jsx`:

  ```
  export default hello;
  ```

  or equivalently

  ```
  export default function hello() {
    …
  }
  ```

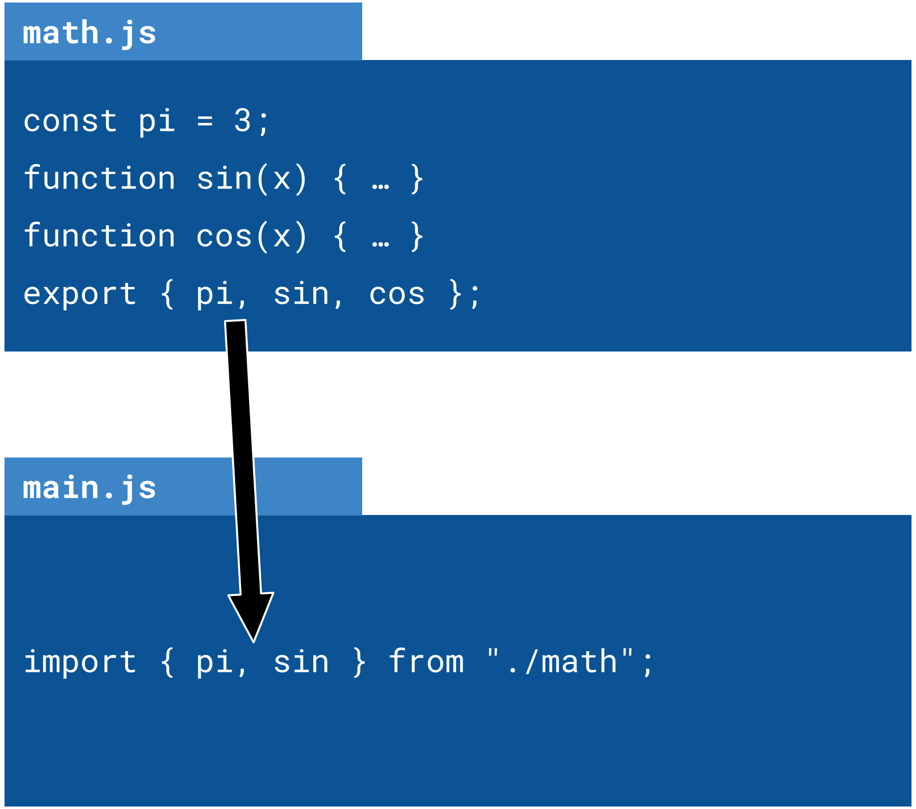
- Importing this function, perhaps using some other **local name**:

  ```
  import goodbye from "./foo.jsx";
  ```

  or equivalently

  ```
  import goodbye from "./foo";
  ```

import and export (named)



math.js

```
const pi = 3;
function sin(x) { … }
function cos(x) { … }
export { pi, sin, cos };
```

main.js

```
import { pi, sin } from "./math";
```

- Export one or more constants, variables, or functions from file `math.js`:

  ```
  export { pi, sin };
  ```

  or equivalently

  ```
  export const pi = 3;
  export function sin() { … }
  ```

- Importing these constants, variables, or functions, using the **names given in the defining file:**

  ```
  import { pi, sin } from "./math";
  ```

`import` and `export`, rules and conventions

Button.jsx

```
import { message } from "./messages";

const version = "1.1.4";
const Button = () =>
  <button>{message}</button>;

export { version };
export default Button;
```

- Import path must start with "`./`" for own modules! (Otherwise an import is treated as a library found in directory `node_modules`.)
- One component per file: make it a "default export."
- File extension = "`.jsx`" if file contains JSX.
- Filename = the name of default export.
- Leave out file extension in import path.
- Put import declarations at the beginning.
- Put export declarations at the end.

Summary

Today, we have covered the following. (Most pages refer to mdn web docs at developer.mozilla.org.)

- JavaScript is a programming language that runs in the browser. (See the overview of JavaScript and the JavaScript Guide at MDN.)
- JavaScript is **dynamically typed:** the value of an expression is not known until runtime (unlike in C# or Java). (See the sections on Data types and Data type conversion.)
- JavaScript has values that are **numbers**, **strings**, **booleans**, **arrays**, **objects**, **functions**, the special **null** and **undefined**, plus a few more. (See JavaScript data types and data structures.) Values carry a **type tag** that can be extracted using `typeof`.
- JavaScript implicitly **converts (coerces)** other values to a number, “when needed.” (See the section on Number coercion.)
- **Variables** are declared by keyword `let` and **immutable constant** by or `const`. (See the section on Declarations.)
- **Conditional statements** (`if`, `while`) and **expressions** (&&, `||`, `_?_:_`) branch according to the **“truthiness”** (see Truthy) or **“falsiness”** (see Falsy) of a value (rather than just on true and false). (See Logical operators and the Conditional operator.)
- In JavaScript, **functions** are **first-class values**. The expression

  ```
  (x) => … x …
  ```

  produces an (anonymous) function that takes x as input at returns the value of

  ```
  … x …
  ```

  as its output. Function definitions can be **nested** and they can be **recursively** defined. Function values are represented by **closures.** (See Functions and Arrow function expressions.)
- We define functions using the shortest possible form, using const and a function expression. For example

  ```
  const increment = (x) => x + 1;
  ```

- A function definition may specify default values for some ot ats parameters. (See Default parameters.) For example

  ```
  const increment = (x, y=1) => x + y;
  ```

- Functions may be defined to take an arbitrary number of arguments, wrapped up as an array:

  ```
  const f = (x, y, ...rest) => …;
  ```

Summary, continued

- In JavaScript, **arrays** and **objects** are (heterogeneous) indexed collections. (See Array and Working with objects.)

```
let bob = {name:"Bob", age: 23};
let users = [bob, {name: "Mary", age: 8}];
bob.age > users[1].age          → true
```

- **JSON** is a **textual representation of data** built out of primitive values, arrays, and objects. JSON is widely used for storing and transmission data. (See Working with JSON and JSON.)

```
typeof users                    → "object"
let json = JSON.stringify(users);
typeof json                     → "string"
json                            →
                  '[{"name":"Bob","age":23},
                   {"name":"Mary","age":8}]'
JSON.parse(json)                →
  [ {
      name: "Bob",
      age:  23
    },
    {
      name: "Mary",
      age:  8
    }
  ]
```

- In the context of React, arrays are often manipulated using **pure array methods**. (These are functions that don't modify the input array.) The most important is

```
a.map(f)
```

but other useful methods are

```
a.every(f)
a.some(f)
a.find(f)
a.filter(f)
a.reduce(g, init)
a.slice(i, j)
a.toSorted()
```

where a is an array, `f` is a function of one or two arguments, g is a function of two or three arguments, and `i` and `j` are integers. (See Array.)