Note Web Application

Brendon Mendicino

April 14, 2023

Contents

1 Introduction

JS is backward compatible, to be able to use the previous features is use the directive:

```
"use strict";
```

JS has primitive types and non-primitive types, JS is also and strongly typed language, the primitive types are: string, number, boolean, null, undefined. The non-primitive are the objects, which can be: array, function, user-defined.

The all possible false values in JS: `0, -0, NaN, undefined, null, ''`, in JS there are two main comparison operators:

```
a == b    // equal, convert types and compare
a === b   // strict equal, inhibits automatic type conversion
```

In JS you can create variable with:

```
// modern
let a = 10;    // can be changed
const b = 'a'; // constant

// old
var k = 9;
j = 30;
```

The difference between null and undefined, is that variable with null they old a value which is null, on the other way if a variable is declared and nothing is associated with it the value olds by default undefined.

A scope is defined by a **block**, which is created with ...

There two kinds of `foreach` in JS, using `in` allows iterating over objects, while `of` allows iterating over iterable objects:

```
for (let a in object) {
  ...
}

for (let b of iterable) {
  ...
}
```

Using arrays:

```
let a = [1, 2, 'ok', false];
let b = Array.of(1, 2, true);
a.push(5);      // append an element
b.unshift(2);   // insert at the beginning

let copy = Array.from(a);  // shallow copy, it does not deep copy
```

The **destructuring assignment** can be done, it extracts the values from the mast left-hand side:

```
let [x, y] = [1, 2];
[x, y] = [y, x]     // swap
```

The **spread operator** (. . .) expands on iterable object into it's values:

```
let [x, ...y] = [1, 2, 3, 4];      // y == [2, 3, 4]

const a = [1, 2];
const b = [0, ...a, 3]; // [0, 1, 2, 3]
```

Spreading can be from the left or from the right, usually the spread operator is used for copying array:

```
const a = [1, 2];
const b = [...a];
```

A **string** is JS is an immutable type (like python) encoded in Unicode. The **template literals** can be done with the **tick** operator ' ' (expression like Kotlin):

```
let name = 'Bre';
let sur = 'Mend';
// Template literal
let fullName = `${name} ${sur}`;
```

1.1 Objects

JS is **prototype based language**, which means that there are no declarations of classes. In JS property names must be strings and can be modified, the value of the property can be any other type of type or object. To create and object in JS you use curly braces and the defined properties:

```
const movie = {
  title: 'Inception',
  genre: 'sci-fi',
  duration: 180
}

console.log(movie)
console.log(movie['title'])
console.log(movie.title)
```

It is also possible to add a property by simple assigning a new name to a type, it is also possible to delete a property with the keyword `delete`. There are two helper functions:

- `Object.key(object)`: return only the key;
- `Object.entries(object)`: return an array with the key and value;

To copy an object it is possible to use:

```
const copied = Object.assign({}, original)
const withSpread = {...original}    // it also possbible to use the spread
    operator

```

```
// assign can also be used to merge objects
const merged = Object.assign({}, copied, {something: 'test'})
```

1.2 Functions

In JS functions are objects, so it is possible to assign a function to a property or use it in a parameter in another function. There three possible ways to define a function:

```
// 1. Function
function do(a, b = 1) {
  ...
}

// parameters can also hava a deafult value
function some(par1, par2, ...variable) { // ... is the 'rest' operator, like
    varargs, rest parameters can be iterated
  ...
}

// 2. Function Expression
const fn = function(params) { }

// 3. Arrow Function
const func = (params) => { }
```

In JS **Closure** can be created, with closure it is possible to use parameters of the scope where the function is defined, even if that scope does not exist any more.

```
function greeter(name) {
  const myname = name;

  const hello = () => {
    return "Hello " + myname;
  }

  return hello;
}

const helloTest = greeter('test');

console.log(helloTest());   // 'Hello test'
```

To create an object there are also **constructor functions**:

```
function Movie(title, director, duration) {
  this.title = title;
  this.director = director;
  this.duration = duration;
  this.isLong = () => this.duration > 120;
}

```

```
cosnt movie = new Movie('Inception', 'Nolan', 180);
console.log(movie.isLong);  // true
```

1.3 Dates

We use `dayjs()` objects in JS to build a data, it is an external library. The return of `dayjs()` fetches the time from the locale time, other than that it can create a data from ISO8601 strings, 8 digit dates, etc. To install this library: `$ npm install dayjs`. The string value of the standard format is in ISO9601 in UTC time, it's important to remember that the days and the month inside the object start *counting from 0*. Other than that the library has some methods to compare different `dayjs` objects, also by choosing the level of *granularity* (year, month, day, ...).

```
const date = dayjs('2023-03-15');
const now = dayjs();

now.isAfter(date, 'day');  // comparing 'now' and 'date' by day
```

1.4 Asynchronous Programming

In JS when passing functions to other functions it's called a **callback**, this functions can be *synchronous* or *asynchronous*.

```
function logQuote(quote) {
  console.log(quote);
}

funtion createQuoute(quote, callback) {
  const myQuote = `Like I always say, ${quote}`;
  callback(quote);
}

createQuote('sium', logQuote);
```

In order to have functional features in language there some need properties:

- *functions as first class citizen*;
- *higher-order functions*;
- *function composition*;
- *call chaining*;

In JS arrays have functional methods, for example:

```
a.forEach(item => ...);   // action on every element of the array
a.every(x => x > 10);     // return true if all elements satisfy the condition,
    false otherwise
a.some(x => x < 10);      // return true if at least one element satisfy the
    condition
a.map(x => `${x}`);       // return a new array with every element mapped to a new
     one
a.filter(x => x === 0);   // return a new array with all elements that satisfy the
     condition
a.reduce((x, y) => x + y, 0);   // return a reduced value
```

Even though JS is executed on a single thread it is possible to create concurrent code, for example a function that allows to excute a callback after a certain amount of time is the `setTimeout()` function:

```
const f = (task) => {
  // do something
};

setTimeout(f, 2000, task);
```

This is possible because JS runs in the **Event Loop**, which periodically checks if there are some part of the code that needs to be executed.

There is a function that allows asynchronous callback after a timeout:

```
const onesec = setTimeOut(() \implies {
  console.log('1 second has passed');
}, 1000);
```

There is also the `setInterval()` function that periodically runs:

```
const period = setInterval(() => {}, 2000);
clearInterval(period);
```

1.4.1 Database Access (SQLite)

The module for `sqlite3` allows calling sql queries via his APIs, first there needs to be an open with the database, to open a connection use:

```
const sqlite = require('sqlite3');

const db = new sqlite.Database('exams.sqlite',
  (err) => { if (err) throw err; });
```

Example of query:

```
let result = [];
let sql = "SELECT * FROM course LEFT JOIN score ON course.code=score.coursecode";
db.all(sql, (err, row) => {
  if (err) throw err;
  for (let row of rows
});
```

The problem with execution this queries is that they are *all asynchronous*, and they can cause race conditions. The solution to this problem are `Promise`, which helps simplyfing asynchronous programming. A `Promise` handles a `resolve` and a `reject` which needs to be called when the callback fails or succedes. The values passed to `resolve` can be accessed by in the `then` method, which gets called when the `Promise` is completed.

```
function waitPromise(duration) {
  return new Promise((resolve, reject) => {
    if (duration < 0) {
      reject(new Error('...'));
    } else {
      setTimeout(resolve, duration);
    }
  }
}

waitPromise(1000).then((result) => {
  colsole.log('Success :', result);
}).catch((error) => {
  colsole.log('Error :', error);
});
```

A promise has 3 main methods: `then`, `catch`, `finally`, which are similar behaviour to the try catch block in Java. Promises can also work concurrently with `Promise.all()` or `Promise.race()`.

1.4.2 Await/Async

The keywords `async` and `await` allows to convert pieces of code to a `Promise`:

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling');
  const result = await resolveAfter2Seconds();
  console.log(result);
}

asyncCall();  // this returns is a promise
```

In fact a function marked as `async` returns a promise.

This method can be combined with the database queries:

```javascript
async function main() {
}

main();
```

2 HTML/CSS

CSS has different mesuraments units:

- `em`: unit size relative to the font size present in the current element;
- `rem`: unit relative to the font size of the root element;
- `vw`: relative to 1% of the width of the viewport;
- `vh`: relative to 1% of the height of the viewport;

CSS has aldo **pseudo selector** which represent changes based on the state of an element.

```
a:visited { color: green; }
```

In CSS there 4 position schemes: **static**, **relative**, **absolute**, **fixe**.

In CSS the flex schema allows for direct control over the element of the page, it allows modifying: direction, sizes, alignment, position, spacing, ...

In CSS the ~ selector is called **subsequent sibling combinator**, the element represented by the first sequence precedes (not necessarily immediately) the element presented by the second one.

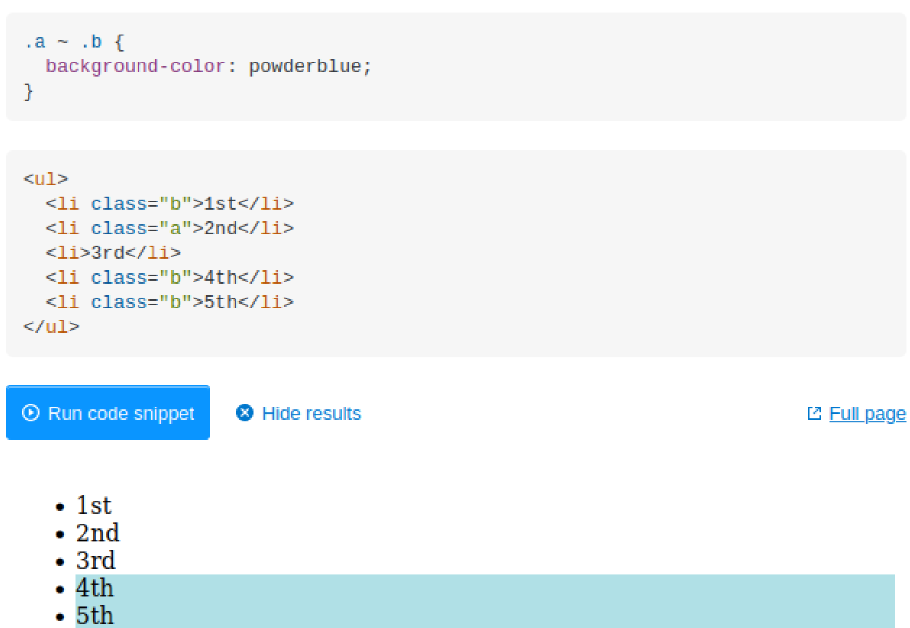

Figure 1: Subsequent Sibling Operator

2.1 Responsive

It's possible to achieve a responsive layout by using `media query`:

```
@media(min-width:900) { }
```

3 JS inside HTML

The perferred way to include javascript code inside the html is:

```
<script async src="script.js"></script>
// or better
<script defer src="script.js"></script>
```

Where does the code run?

The main objects of the browser:

- **DOM**: Document Object Model
- **BOM**: Browser Object Model, non-standard

The BOM has a `window` object, which contains: `console, document, history, location, localStorage, sessionStorage`.

The DOM is

The DOM can be accessed like a squence of Nodes. To find a node there are vaarius methods, like:

- `document.getElementById(value)`
- `document.getElementsByTagName(value)`
- `document.getElementsByClassName(value)`
- `document.querySelector(css)`
- `document.querySelectorAll(css)`

From each node there are many mothod to access all the neigbhor nodes.

3.1 Event Handling

...

4 React

React is a framework that allows DOM manipulation with a level of abstractions, and while using it won't be necessary to touch the DOM directly.

React has a functional approach, which allows bulding a web page in a declarative approach. On any change that is acted on a component all the other components are rerendered. Fot this reason React has a **virtual DOM** which is built on top of the DOM which will eventually push his changes to the original DOM.

The basic information that are shared between components are the `state` and `props` (properties), which are passed to functions inside the component.

On re-rendering when the virtual DOM is stabilized, the difference between the virtual DOM and the DOM are computed and only then the changes are moved on the actual DOM, this is why this re-rendering is not so heavy.

There are event that are normalized across the browser, this are called **synthetic events**.

If we want to write a minimal React application:

```
const container = document.getElementById('root');

const root = createRoot(container);
root.render(<h1>Hello, world!</h1>);
```

React uses `jsx` that are translated into *react elements*, this `jsx` will be then be translated to plain javascript by React. To define a component in React we do:

```
const BlogPostExcerpt = (props) => {
  return (
    <div>
      <h1>Title</h1>
      <p>{props.content}</p>
    </div>
  )
}
```

There are two types of component:

- **presentation component** don't manage the state;
- **container component** manages the state of all his children;

`props` can only be passed from a parent to his children, if the changes shoul be performed from the a children to a parent then, a callback needs to be passed.

React flow: view $\Longrightarrow$ actions $\Longrightarrow$ state $\Longrightarrow$ view $\Longrightarrow$ ...

- A **state** is always **owned be one compoment**.
- Changing state on a childred should not affect the state of a parent.

To create a React application

```
$ npm create vite@latest my-app
// Choose React, javascript + SWC
$ cd my-app
$ npm install
$ npm run dev
```

In `jsx` the attributes of a component (`props`) are translated into js objects

```
color="red" -> {color: "red"}
tone={2} -> {tone: 2}
// it's possible compute exressions
color={active ? 'green' : 'red'}
```

Some attributes in HTML do not a value like `selected disabled`, but in `jsx` they can have a value: `selected={true}`

In React when using a list it's mandatory to use give a key to each element of the list (the key is unique in the list, not globally), this is done because React internally can recognise if the list has changed, if this is not done there are undefined behaviur in the page.

```
<li key={todo.id}>{todo.text}</li>
```

When returning a gruop of components they should always be packed in a root component, when returing a group of components there is *React Fragment* which makes it more easy.

4.1 State

Ad we have seen using pure functions to create Components in react, this is very simple but in react the various components need a state, to use this feature **Hooks** have been proposed which intruduce a way to indroduce a state. One of them is `useState`. While **props** are passed from father to child, **state** is private to the component and hold his data, the **context** is a global variabl eavailable to every component.

To create a hook, we use the `useState` and it gives back the current value and a callback to cahnge that satate.

```
function ShortText(props) {
  const [hidden, setHidden] = useState(true);
  return (
    <span>
      {hidden ? "Hidden" : "Not hidden"}
      <a onClick={() => setHidden(hidden => !hidden)}>hide</a>
    </span>
  );
}
```

All the modification of the state need to be done through `setVariable`. The value can be set by passing a value or with a function.

Can a child mutate the state of the parent? This simplest way is to pass a the `setVaraible` to the child inside a callback through the `props`

```
function App() {
  const [hidden, setHidded] = useState(tru);

  function hide() {
    setHidden(hidden => !hidden);
  }

  return (
    <>
      <span>
        {hidden ? "Hidden" : "Not hidden"}
      </span>
      <Change hide={hide} />
    </>
  );
}

function Change(props) {
  return (
    <a onClick={() => props.hide()}>Hide</a>
  );
}
```

4.2 Forms