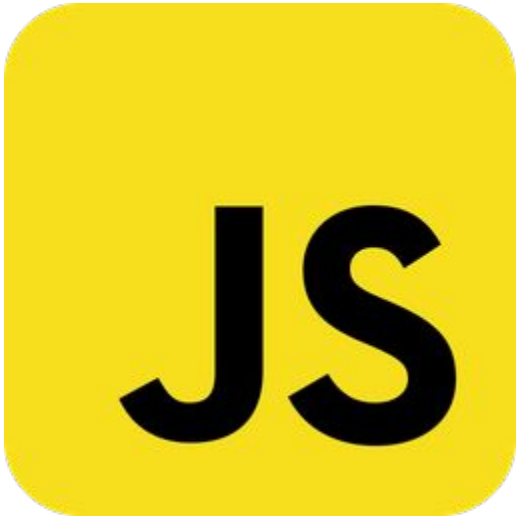


Javascript Recap

KdG Karel de Grote Hogeschool


KEEP
CALM
AND
LETS
RECAP

CONST vs LET vs VAR

ES6 Conventions:

1. Use `const` by default.
2. Use `let` if you have to rebind a variable.
3. Use `var` to signal untouched legacy code

Source: https://twitter.com/raganwald/status/564792624934961152

JS

https://dev.to/sarah_chima/var-let-and-const--whats-the-difference-69e

Object literals

```
const bart = {
    firstName: 'Bart',
    lastName: 'Vochten',
    age: 49,
    adress: {
        street: 'Baker street'
    },
    printName () {
        console.log(this.firstName + ' ' + this.lastName)
    },
    printAdress () {
        console.log(this.firstName + ' lives in ' + this.adress.street)
    },
    get isOld () {
        return this.age > 40
    }
}

console.log(bart.firstName) // Bart
console.log(bart.isOld)     // true
bart.age = 30
console.log(bart.isOld)     // false
bart.printName()            // Bart Vochten
```

Since ES6 the word function can be omitted

```
function printName () {
    ....
}
```

Classes

```
class Person {
    constructor(firstName, lastName, age) {
      this.firstName = firstName;
      this.lastName = lastName;
      this.age = age;
    }
    printName () {
        console.log(this.firstName + ' ' + this.lastName)
    }
    get isOld () {
        return this.age > 40
    }
  }

const bart = new Person('Bart', 'Vochten', 49)
console.log(bart.firstName)
console.log(bart.isOld)
bart.age = 30
console.log(bart.isOld)
bart.printName()
```

Shallow vs Deep copy

Objects in Javascript are passed by reference

```
const bart = {
    firstName: 'Bart',
    lastName: 'Vochten',
    address: {
        street: 'Baker street'
    },
    printName () {
        console.log(this.firstName + ' ' + this.lastName)
    },
    printAddress () {
        console.log(this.firstName + ' lives in ' + this.address.street)
    }
}

const john = bart
john.firstName = "John"
john.printName() //  "John Vochten"
bart.printName() //  "John Vochten"
```

Shallow vs Deep copy

Object.assign or spreading (see further) makes a shallow copy
All properties are copied into a new object, but sub objects (like address in this example) are copied by reference

```
const bart = {
    firstName: 'Bart',
    lastName: 'Vochten',
    address: {
        street: 'Baker street'
    },
    printName () {
        console.log(this.firstName + ' ' + this.lastName)
    },
    printAddress () {
        console.log(this.firstName + ' lives in ' + this.address.street)
    }
}

const john = Object.assign({}, bart);

john.firstName = 'John'
john.address.street = 'James street'

bart.printName()   // Bart Vochten
bart.printAddress() // lives in James street

john.printName()    // John Vochten
john.printAddress()  // lives in James street
```

Shallow vs Deep copy

To make a real deep copy you need a library (like LoDash) or use JSON.parse and JSON.stringify

```
const bart = {
    firstName: 'Bart',
    lastName: 'Vochten',
    address: {
        street: 'Baker street'
    },
    printName () {
        console.log(this.firstName + ' ' + this.lastName)
    },
    printAddress () {
        console.log(this.firstName + ' lives in ' + this.address.street)
    }
}

const john = JSON.parse(JSON.stringify(bart))
john.firstName = 'John'
john.address.street = 'James street'

bart.printName()    // Bart Vochten
bart.printAddress()  // lives in Baker street

// these will give an error, since you lose the functions with JSON conversion
// john.printName()
// john.printAdress()

console.log(john.address.street) // James street
```

JSON vs Object literals

JSON is a wire-format widely used to communicate data between a back-end and a frond-end (eg. as the result of an HTTP GET)

```
[
  {
    "id": 1,
    "name": "Java programming 1",
    "description": "...",
    "skillLevels": [
      {
        "skillId": 1,
        "level": 2
      },
      {
        "skillId": 1,
        "level": 2
      }
    ]
  },
  {
    "id": 2,
    "name": "Java programming 2",
    "description": "...",
    "skillLevels": [
      {
        "skillId": 1,
        "level": 3
      },
      {
        "skillId": 2,
        "level": 2
      }
    ]
  },
  {
    "id": 3,
    "name": "Web development with Vue",
    "description": "...",
    "skillLevels": [
```

JSON looks like an object literal but it is not the same.
Eg. You can not represent a method in JSON

JSON can very easily be converted to a Javascript object and the other way around

`JSON.parse()`

Parse a string as JSON, optionally transform the produced value and its properties, and return the value.

`JSON.stringify()`

Return a JSON string corresponding to the specified value, optionally including only certain properties or replacing property values in a user-defined manner.

Arrow functions

A more compact syntax for functions

frequently used when applying operations on other objects (eg arrays)

```
var elements = [
  'Hydrogen',
  'Helium',
  'Lithium',
  'Beryllium'
];

elements.map(function(element) {
  return element.length;
}); // this statement returns the array: [8, 6, 7, 9]

// The regular function above can be written as the arrow function below
elements.map((element) => {
  return element.length;
}); // [8, 6, 7, 9]

// When there is only one parameter, we can remove the surrounding parenthesies:
elements.map(element => {
  return element.length;
}); // [8, 6, 7, 9]

// When the only statement in an arrow function is `return`, we can remove `return` and remove
// the surrounding curly brackets
elements.map(element => element.length); // [8, 6, 7, 9]

```

spread

Expand the properties of an array or object

```
var parts = ['shoulders', 'knees'];
var lyrics = ['head', ...parts, 'and', 'toes'];
// ["head", "shoulders", "knees", "and", "toes"]
```

```
var obj1 = { foo: 'bar', x: 42 };
var obj2 = { foo: 'baz', y: 13 };

var clonedObj = { ...obj1 };
// Object { foo: "bar", x: 42 }

var mergedObj = { ...obj1, ...obj2 };
// Object { foo: "baz", x: 42, y: 13 }
```

spread

```
const bart = {
    firstName: 'Bart',
    lastName: 'Vochten',
    address: {
        street: 'Baker street'
    },
    printName () {
        console.log(this.firstName + ' ' + this.lastName)
    },
    printAddress () {
        console.log(this.firstName + ' lives in ' + this.address.street)
    }
}

// shallow copy (like Object.assign)
const john = { ...bart } // = create a new object with all the properties of bart

john.firstName = 'John'
john.address.street = 'James street'

bart.printName()   // Bart Vochten
bart.printAddress() // lives in James street

john.printName()    // John Vochten
john.printAddress()  // lives in James street

const extraInfo = {
    gender: 'M',
    vegetarian: false
}

// combine the properties of 2 (or more) objects into a new object
const bartInfo = { ...bart, ...extraInfo }

console.log(bartInfo) // bartInfo has firstName, lastName, addrress, printName, printAdress, gender, vegetarian

// merge the properties of an object with the supplied properties into a new object
const leo = { ...bart, firstName: 'Leo' }

bart.printName()   // Bart Vochten
leo.printName()    // Leo Vochten
```

destructuring assignment

Assign parts of an object or array to one or more variables in one line of code

```
// OBJECT DESTRUCTURING
const anObject = {
  property_one: "a value",
  property_two: "a second value",
}

// "classic" way of getting the values off an object
const property_one = anObject.property_one;
const property_two = anObject.property_two;
console.log("1:", property_one);
console.log("2:", property_two);

// With object destructuring:
// (names of destructured properties have to match the name of the
field on the     // object)
const { property_one, property_two } = anObject;
console.log("1:", property_one); // prints “1: a value”
console.log("2:", property_two); // prints “2: a second value”
```

destructuring assignment

Assign parts of an object or array to one or more variables in one line of code

```
// ARRAY DESTRUCTURING
const anArray = ["first element", "second element"];

// "classic" way of getting the values of an array
const element_one = anArray[0];
const element_two = anArray[1];
console.log("1:", element_one);
console.log("2:", element_two);

// With array destructuring:
// (We define two new variables on one line here, and assign
to them the values in  // the matching position of the
array! Order is important!)
const [element_one, element_two] = anArray;
console.log("1:", element_one); // Prints “1: first element”
console.log("2:", element_two); // Prints “2: second
element”
```

export

Export and import are used to split up code in different files and bring them together when needed

```
// Exporting individual features
export let name1, name2, …, nameN; // also var, const
export let name1 = …, name2 = …, …, nameN; // also var, const
export function functionName(){...}
export class ClassName {...}

// Export list
export { name1, name2, …, nameN };

// Renaming exports
export { variable1 as name1, variable2 as name2, …, nameN };

// Default exports
export default expression;
export default function (…) { … } // also class, function*
export default function name1(…) { … } // also class, function*
export { name1 as default, … };

// Aggregating modules
export * from …;
export { name1, name2, …, nameN } from …;
export { import1 as name1, import2 as name2, …, nameN } from …;
export { default } from …;
```

import

```
// Import individual features
import { export } from "module-name";
import { export1 , export2 } from "module-name";

// Import default export
import defaultExport from "module-name";

// Import all of a module's exports as a module object
import * as name from "module-name";

// Import renamed exports
import { export as alias } from "module-name";

// Import a module for side effects only
import "module-name";

// Combinations
import { export1 , export2 as alias2 , [...] } from "module-name";
import defaultExport, { export [ , [...] ] } from "module-name";
import defaultExport, * as name from "module-name";
```

{} syntax: module must export something with the name export 1 and export 2

for the default export you can choose any name you like

Template literals

Using backticks (`) you can embed expressions in a string (using ${}

Without template literals: ugly code

```
let a = 5;
let b = 10;
console.log('Fifteen is ' + (a + b) + ' and\nnot ' + (2 * a + b) + '.');
// "Fifteen is 15 and
// not 20."
```

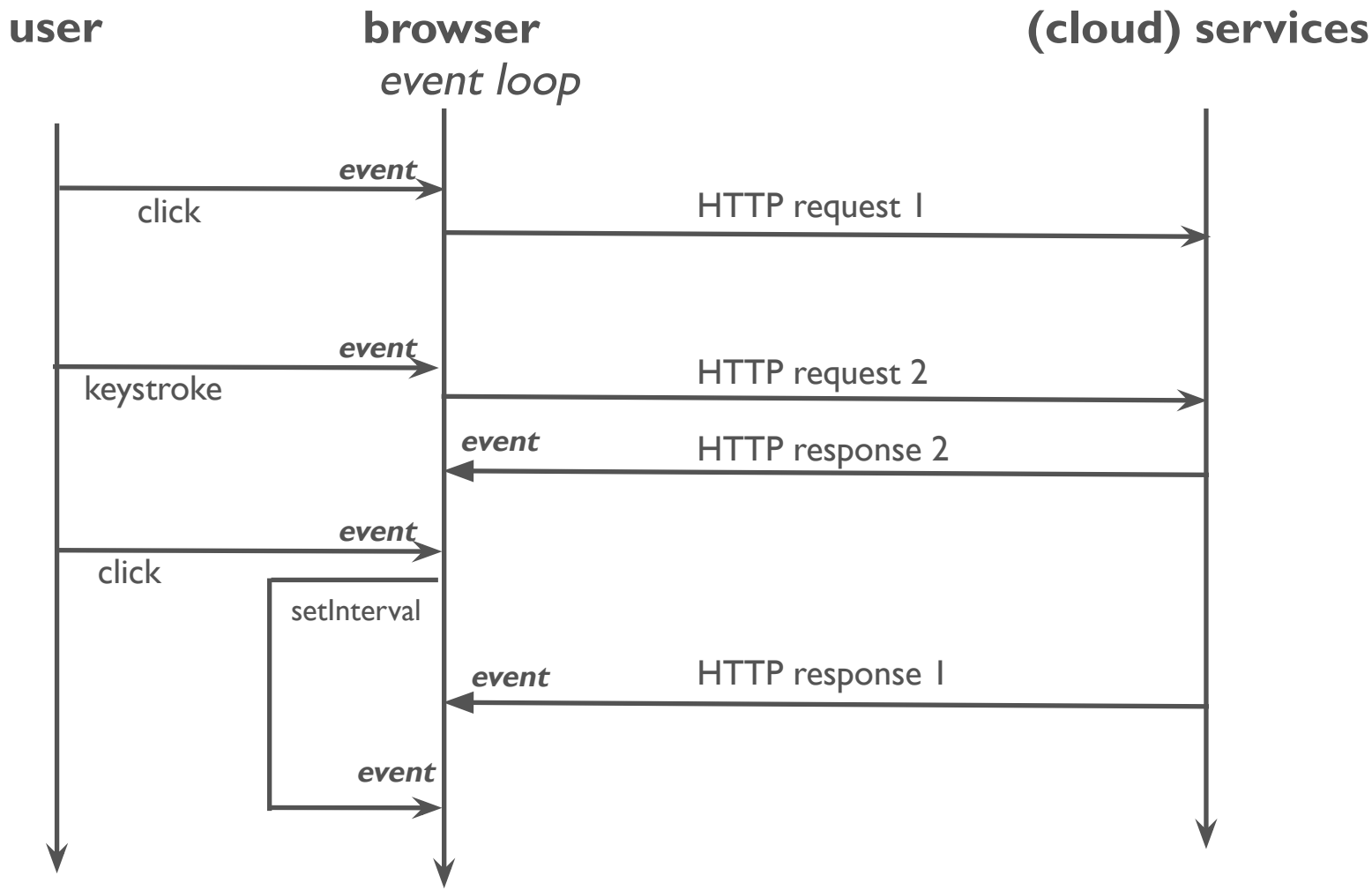
With template literals: clean code

```
let a = 5;
let b = 10;
console.log(`Fifteen is ${a + b} and
not ${2 * a + b}.`);
// "Fifteen is 15 and
// not 20."
```

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

promises

Browser runs **event loop** for each page (tab) = **UI thread**



Events are handled asynchronously in **callbacks**. This can lead to messy code (callback in callbacks…). Promises handle this in an elegant way

promises

A promise represents the "future" result of a job and can be in 3 states.

unfulfilled
the async action is still busy

fulfilled
the action has succeeded

failed
the action has failed

'settled'
fulfilled or failed

Operations

- **resolve** (*unfulfilled to fulfilled) => succes callback is called*
- **reject** *(unfulfilled to failed) => error callback is called*
- **then** register a succes (and optional error) callback
- **catch** register an error callback

promises

Creation

```
const getUser = function (login) {
  return new Promise(function (resolve, reject) {
    // async stuff, like fetching users from server
    if (response.status === 200) {
      resolve(response.data);
    } else {
      reject('No user');
    }
  });
};
```

you rarely make a promise yourself, this is what fetch and axios do behind the scenes when an HTTP response is received

Usage

```
getUser(login)
  .then(function (user) {
    console.log(user);
  })
```

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

promises

Without promises (nesting)

```
getUser(login, function (user) {
  getRights(user, function (rights) {
    updateMenu(rights);
  });
});
```

With promises (chaining)

```
getUser(login)
  .then(function (user) {
    return getRights(user);
  })
  .then(function (rights) {
    updateMenu(rights);
  })
```

The first then callback returns a promise so that you can call 'then' on it again.

If you do not return a promise in your then code (eg a value), it will be automatically converted to a promise that is immediately resolved (so that the next then is called with this value as input)

promises

Bundled error handling

```
getUser(login)
  .then(function (user) {
    return getRights(user);
  })
  .then(function (rights) {
    return updateMenu(rights);
  })
  .catch(function (error) {
    console.log(error); // will be called if getUser or getRights fails
  })
```

all

```
var p1 = Promise.resolve(3);
var p2 = 1337;
var p3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, "foo");
});

Promise.all([p1, p2, p3]).then(values => {
  console.log(values); // [3, 1337, "foo"]
});
```

'then' is only called when all promises have been resolved
(eg to wait for the result of different HTTP calls)

https://developer.mozilla.org/nl/docs/Web/JavaScript/Reference/Global_Objects/Promise/all

promises

The standard Fetch API and Axios work with promises

```
function apiGetAll () {
  // console.log("Fetching stuff")
  fetch(URL)
    .then(function(response) {
        console.log (response.json())
  })
 }
```

Fetch - LS

Global 78.35% + 0.06% = 78.41%

A modern replacement for XMLHttpRequest.

Current aligned | Usage relative | Date relative | Show all

IE	Edge *	Firefox	Chrome	Safari	iOS Safari *	Opera Mini *	Chrome for Android	UC Browser for Android	Samsung Internet
		52	49						
		55	60		10.2				4
	15	56	61	10.1	10.3				5
11	16	57	62	11	11.1	all	62	11.4	6.2
	17	58	63	TP					
		59	64						
		60	65						

promises

await allows you to wait until a promise is resolved.

```
async function getFirstUser() {
    let users = await getUsers();
    return users[0].name;
}
```

this code is only executed when the getUsers call is finished

The keyword await may only be used in a function marked with the keyword **async**
Putting async in front of a function automatically causes it to return a promise, which is resolved when the function is ready.

In the example above, when calling getFirstUser you will immediately receive a Promise that will later on be resolved with users [0] .name. Inside the function the code waits on the call to getUsers

https://tutorialzine.com/2017/07/javascript-async-await-explained

https://medium.com/@bluepnume/learn-about-promises-before-you-start-using-async-await-eb148164a9c8

promises

asyn/await is only syntax sugar for promises, used to make the code and error handling look like it is synchronous code

The following is equivalent

```
async loadTodos () {
  try {
    this.todos = await this.todoService.get()
  } catch (error) {
    ErrorService.showAlert('Loading todos failed', error)
  }
},
loadTodosSameAsAbove () {
  this.todoService.get()
    .then(result => { this.todos = result })
    .catch((error) => { ErrorService.showAlert('Loading todos failed', error) })
}
```


That's all Folks!