JavaScript

Développer des applications complexes pour le web

Guillaume Lelasseur

Introduction

Découverte du langage

Javascript ?

- Un langage de scripting
- Conçu par **Brendan Eich** (Netscape) en 1995
- Inspiré par de nombreux langages, tels que **Python**

Présentation

LiveScript

- Au départ un langage coté serveur appelé

- Ensuite, une version cliente nommée **JavaScript**
  - Partenariat entre Sun Microsystems et Netscape concernant le nom
  - « JavaScript » était une marque déposée de **Sun Microsyste** maintenant de **Oracle Corporation**

Sun

ORACLE

Client-Side

- Interprété par le navigateur
- Ajoute de l'interaction aux pages faites en HTML et CSS
  - Interaction utilisateur
  - Animations
  - Etc..

Déclaration

- Le Javascript peut-être défini :
  - Dans le code HTML
  - Dans un fichier séparé (.js)

Le DOM – Document Object Model

- Le javascript est basé sur des **events**:
  - load
  - focus
  - click
  - dblclick
  - abort
  - error
  - mouseover

- Associé aux objets du DOM:

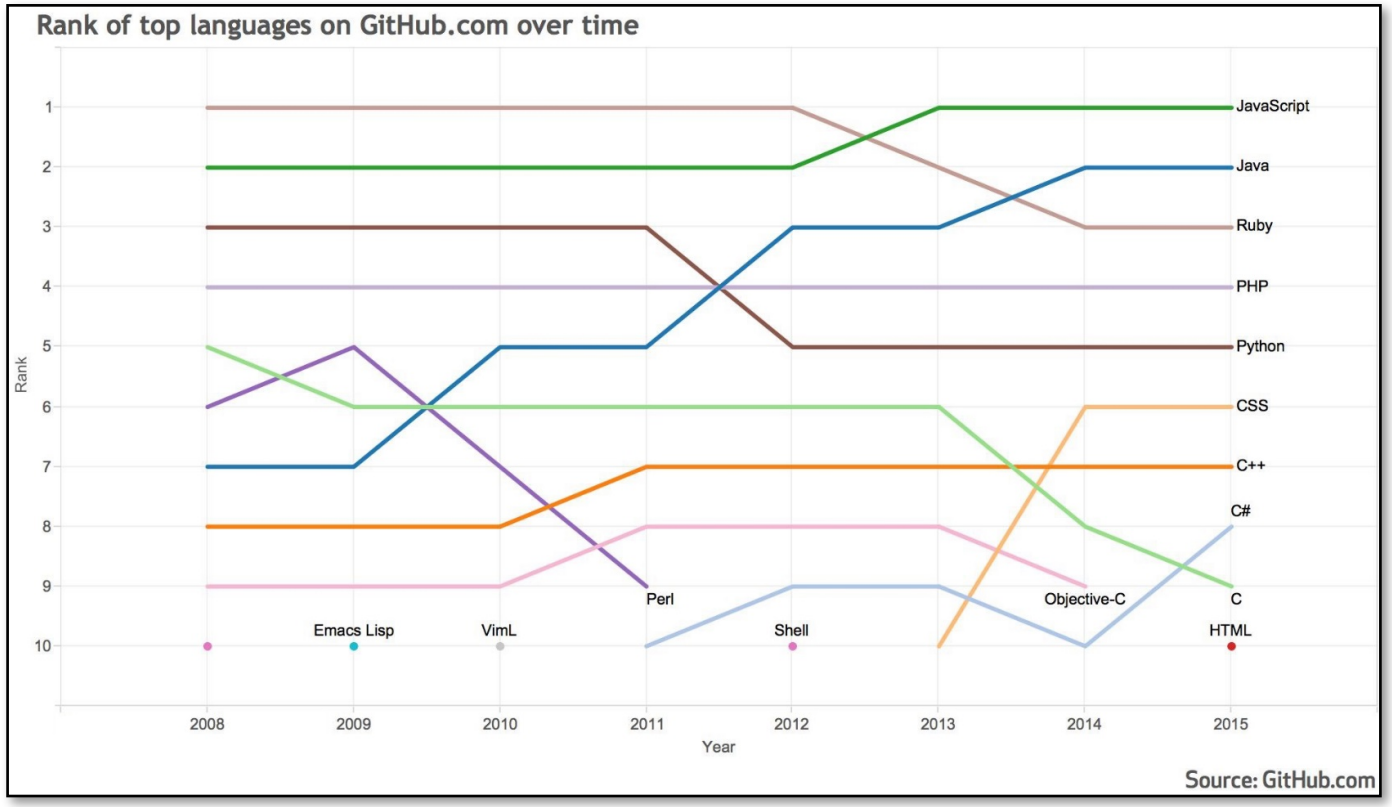
window, document, forms, ...

Un langage objet ?

- Pas totalement
  - Prototype-based
- Pas de concept de classe
  - Des « Pseudo-classes » peuvent être définies comme des collections de paires clé/valeurs
- Intègre la plupart des fonctionnalités des modèles OOP

Les librairies Javascript

Une communauté conséquente



Premiers pas

JavaScript

Déclaration

```
<script>
    const name = "Hodor";
    let variable = 12;
</script>
```

```
<script src="script.js"></script>
```

Hello world

1. Faire un fichier script.js
2. Lier le fichier script.js
3. Dans le JS :

```
console.log('Bonjour le monde !');
```

4. Regarder le résultat dans la console du navigateur

Syntaxe

- Une instruction se termine avec un point virgule

```
<script>
      Instruction_1;
</script>
```

- Une ligne de commentaire commence par //; une section de commentaires est entourée par /* et */

```
<script>
   // Single line comment
   /*  Multi line
       comment  */
</script>
```

Exemples

Fonction	
console.log()	Afficher du texte sur la console
document.write()	Afficher du texte sur une page
window.alert(message)	Utiliser une boite de dialogue

Variables

- Case sensitive:

myvariable ≠ myVariable

```
<script>
    const name = "ESTIAM";
</script>
```

```
<script>
    let name = "Doug";
</script>
```

Variables

- Concatenation: Combiner des valeurs

```
let max_age = 18;
let message = "Not allowed under " + max_age + "
   years old";
```

Variables

- Typage faible
- Type des variables définie par le format du contenu
- Obtenir le type d'une variable: **typeof**

```
const myVar1 = "I am a string !";
let myVar2 = "Am I really a string ?";
myVar2 = 100;

document.write(typeof myVar1); // Will display "string"
document.write(typeof myVar2); // Will display "number"
```

Variables cast

- parseXXX: Parser d'une type vers un autre
  - parseInt
  - parseFloat

```
const number = "11";
const parsed = parseInt(number);

document.write(parsed + 1); // Will display 12
document.write(number + 1); // Will display 111
```

Operateurs

Symbole	Exemple	Utilisation
=	let salary = 2800;	Affectation
+	salary = salary + 2800	Operation ou Concatenation
-	salary = salary – 2800	Soustraction
*	salary = salary * 2800	Multiplication
/	salary = salary / 2800	Division
%	salary = salary % 2800	Modulo

Operateurs

Symbole	Exemple	Retour	Signification
==	salary == 2800 salary == "2800"	*true* *true*	Egal
===	salary === "2800" salary === 2800	*false* *true*	Exactement égal (valeur et type)
!=	salary != 2800	*false*	Non égal
!==	salary !== "2800"	*true*	Non Exactement égal (valeur et type)

Operateurs

Symbol	Example	Explanation
&&	age == 18 && salary > 2800	AND
||	age == 18 || salary > 2800	OR
^	age == 18 ^ salary > 2800	Exclusive OR
>>	salary >> age	Bitwise shift right
<<	salary << age	Bitwise shift left

Operateurs

Symbol	Example	Explanation
+=	age += 18;	Addition (number) or concatenation (string)
new	const array = new Array();	Object instanciation
delete	delete array;	Object destruction

Structures conditionnelles

if … else if … else

```
if( expression1 ) {
     // If "expression1" is evaluated to true, then this
     // block is executed
} else if ( expression2 ) {
     // Otherwise, if "expression2" is evaluated to true,
     // this block is executed
} else {
     // Otherwise, this code block is executed
}
```

Structures conditionnelles

- **switch**

```
switch(myVar) {
    case "case1":
       // if(myVar === "case1")
       break;
    case "case2":
       // if(myVar === "case2")
       break;
    default:
       // else - Default code to execute
       break;
}
```

Arrays

- Contiennent plusieurs séquences de données
- Deux méthodes de création
  - Avec un objet **Array**
  - En utilisant des []

```
const fruitBasket1 = new Array("Apples", "Bananas", "Pears");
const fruitBasket2 = [ "Oranges", "Bananas", "Strawberries"];
const fruitBasket3 = [];

const apple = fruitBasket1[0];
fruitBasket3.push(apple);
```

Les boucles

- **while**

```
// It loops 40 times
let myVariable = 40;
while( myVariable > 0 ) {
   myVariable = myVariable - 1;
}
```

Les boucles

- **do … while**

```
let myVariable = 0;
// Loop will execute once even if the test returns
   false
do {
    myVariable -= 1;
} while (myVariable > 0);
```

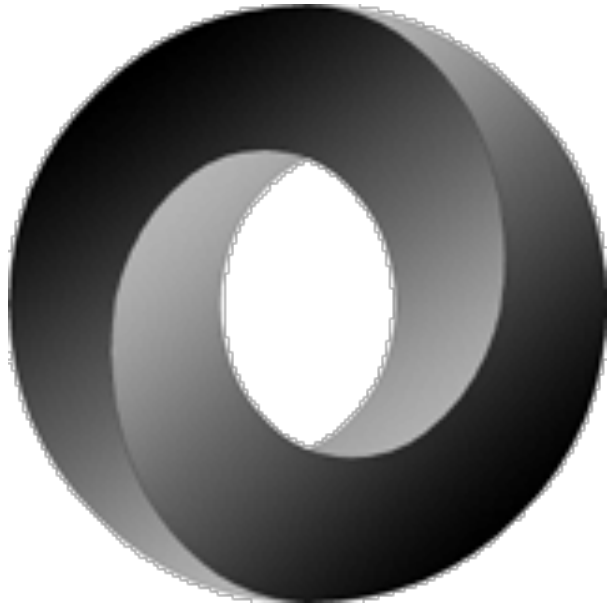
Les boucles

- **for**

```
let a;
for (a = 0; a < 100; a += 1){
    // Loop will display Blabla 100 times.
    document.write("<p>Blabla</p>");
}
```

Les fonctions

JavaScript



Les fonctions

- Instruction unit
- Déclaré avec **function**
- Peuvent prendre des **arguments**

```
function myFunction(myParam1, myParam2) {
    // Some code to execute
}
```

Les fonctions

- Appelées par leur nom avec des parenthèses

functionName();

functionName(arguments);

- Peuvent retourner des valeurs avec le mot-clé **return**

Exemple

```
function howOld(year){
    const currentYear = new Date().getFullYear();
    return currentYear - year;
}

// someValue will contain 22 (if current year is 2012)
const someValue = howOld(1990);

// Will display Bryan is 42 years old (if 2012)
console.log("Bryan is " + howOld(1970) + " years old");
```

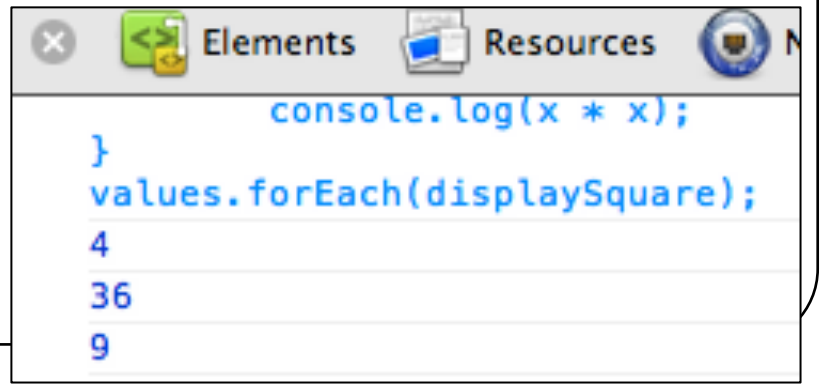
Portée des variables

- Local:
  - Uniquement dans la fonction
- Global:
  - Dans tout le document
- Portée
  - Déclaration explicite dans la fonction : local (var)
  - Déclaration implicite globale

Expressions

- JavaScript supporte des fonctions d'expressions

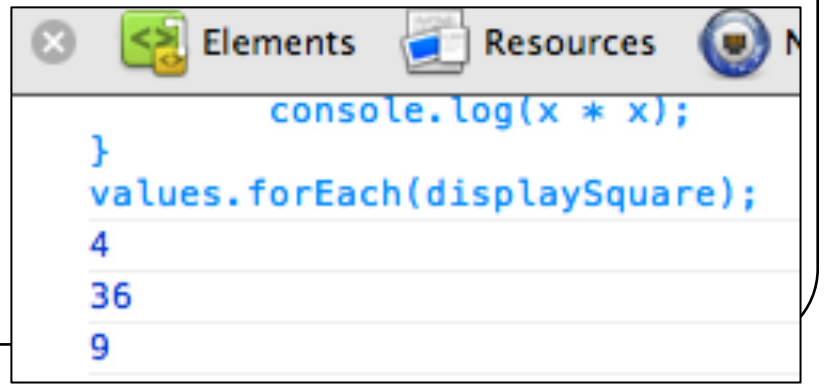
```
const values = [2, 6, 3];
const displaySquare = function(x) {
   console.log(x * x);
}
values.forEach(displaySquare);
```



Expressions

- JavaScript supporte des fonctions d'expressions

```
const values = [2, 6, 3];
const displaySquare = (x) => {
   console.log(x * x);
}
values.forEach(displaySquare);
```



Fonction comme paramètre - Exemple

- Exécuter une action par élément

```
const myArray = ["Apple", "Strawberry"];
myArray.forEach( function(element) {
   console.log(element + "/");
});
```

Fn expression VS Fn declaration

- Les déclarations de fonction sont évaluées avant n’importe quelle instruction dans le même contexte
- Les expressions sont évaluées après les instructions qui les précèdent

Exemple

```
function declaration() {
   console.log("I'm a function declaration");
}
const expression = function() {
   console.log("I'm a function expression");
}
declaration();
expression();
```

```
I'm a function declaration
I'm a function expression
>
```

Exemple

```
declaration();
expression();
function declaration() {
   console.log("I'm a function declaration");
}
const expression = function() {
   console.log("I'm a function expression");
}
```

```
I'm a function declaration
▶Uncaught TypeError: expression is not a function
>
```

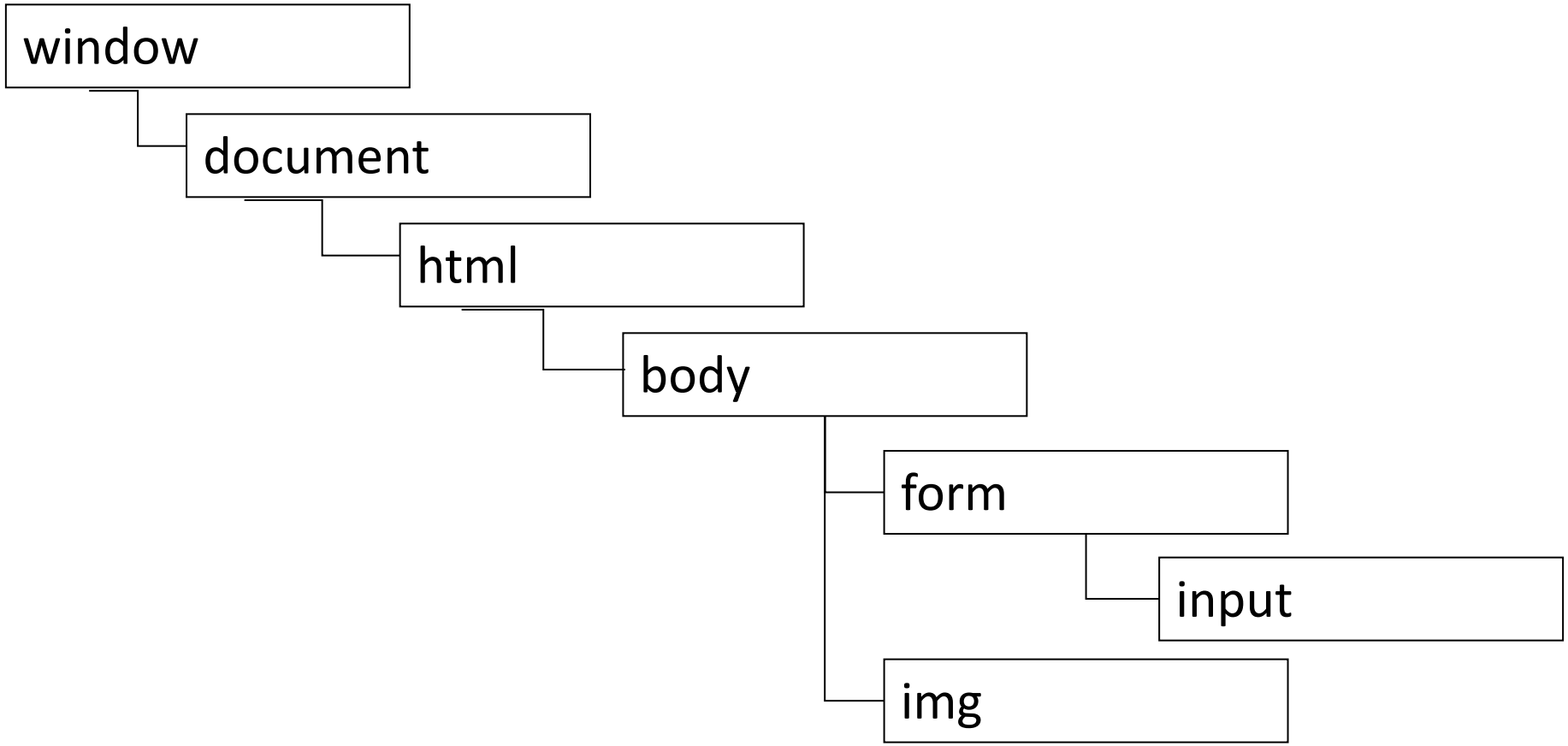
Interactions DoM

Javascript

Introduction

- Document Object Model
- W3C Standard
- Indépendent du language ou de la plateforme

Arborescence



Accéder aux éléments

- Accès à toute la structure d'une page HTML

- Capacité d'accéder dynamiquement
  - A des éléments HTML
  - Lire, Modifier et supprimer des attributs et des valeurs
  - Créer, Modifier et supprimer des éléments
  - Organiser des éléments dans une hiérarchie

Accéder aux éléments

- Accéder à un simple élément :

```
document.querySelector("#myElement");
```

- Accéder à plusieurs éléments avec un sélecteur CSS:
  - Retourne un tableau contenant tous les homonymes

```
document.querySelectorAll("p");
```

Accéder aux éléments

- Accéder aux éléments par leur tag name:

```
document.getElementsByTagName(tagName);
```

Exemple

```
<img id="img1" name="theImage" src="test.png" />
<script type="text/javascript">
  const img = document.querySelector("#img1");
  const elements = document.getElementsByName("theImage");
  console.log(elements[0] === img); // true
  elements = document.querySelectorAll("img");
  console.log(elements[0] === img); // true
</script>
```

Accéder aux éléments

- Tous les enfants d'un élément

```
element.childNodes;
```

- Accéder au parent d'un élément

```
element.parentNode;
```

Manipuler des attributs

- Obtenir les attributs d'un élément

```
element.getAttribute("attribute");
```

- Modifier les attributs d'un éléments

```
element.setAttribute("attribute", "value");
```

Manipuler les valeurs

- Accéder au texte d'un élément

```
element.firstChild.nodeValue;
```

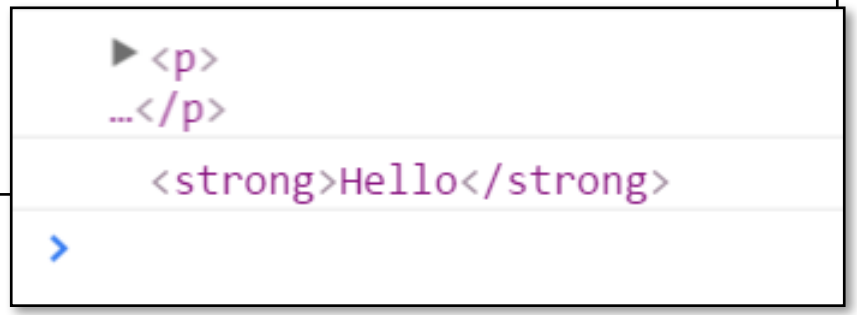
- Modifier le texte d'un élément

```
element.firstChild.nodeValue = "text";
```

Accès en cascade

- Sélectionner un élément depuis un autre

```
<p>
  <strong>Hello</strong> world!
</p>
<script>
  const p = document.querySelector("p");
  const strong = p.querySelector("strong");
  console.log(p);
  console.log(strong);
</script>
```



DOM – Propriétés des éléments

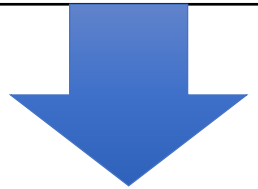
Propriété	
textContent	Accéder au texte dans un élément
innerHTML	Accéder au contenu HTML d'un élément
classList	Accès aux classes CSS
style	Accès aux styles CSS

DOM – Propriétés des éléments

- Ajouter un élément avant un autre

```
<p></p>
```

```
const p = document.querySelector("p");
p.textContent = "Hello World";
```



```
<p>Hello World</p>
```

Les évènements

Javascript

Presentation

- Arrivent
  - Interaction utilisateur
  - Action dans un contexte d'exécution
- Les propriétés des objets dépendent de ces derniers
- Peuvent appeler des fonctions

Attributs standards

- Attacher un évènement au click

```
<input type="button" class="my-button" value="Push me!" />
```

```
const button = document.querySelector(".my-button");
button.addEventListener("click", function() {
    console.log("You clicked the button!");
});
```

Attributs standards

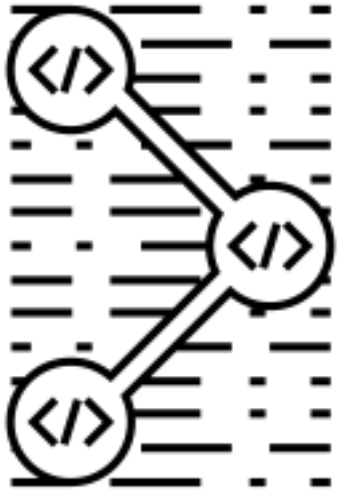
Event	Description
DOMContentLoaded	Le DOM est construit (CSS, JS & Images ne sont pas encore chargés)
load	Élement totalement chargé
unload	Le navigateur quitte la page
click	L'utilisateur click un élément
dblclick	L'utilisateur double click un élément
mouseover	La souris passe sur un élément
mouseout	La souris quitte un élément

Attributs standards

Event	Description
focus	Un champ input prend le focus
blur	Un champ input perd le focus
change	L'utilisateur modifie le contenu d'un champ input
select	L'utilisateur sélectionne le contenu d'un champ input
submit	L'utilisateur soumet un formulaire

Callback

Javascript



Callback ?

- Tout ce qui appelle une fonction passée en paramètre
  - C’est un modèle de design pas une fonctionnalité du langage

```
const button = document.querySelector(".my-button");
button.addEventListener("click", function() {
    console.log("You clicked the button!");
});
```

Exemple

```
function forEach(array, callback) { // Contains the function
  for(var i = 0; i < array.length; i++) {
    callback(i, array[i]); // Execute the function
  }
}
```

```
const fruits = ["Apple", "Orange", "Strawberry"];
forEach(fruits, function(index, value) {
   console.log("Element at index "
     + index + " is " + value);
});
```

```
Element at index 0 is Apple
Element at index 1 is Orange
Element at index 2 is Strawberry
>
```

Raison des callbacks

- Utiles pour
  - Overrider une logique
    - Par exemple, rien ne se produit si l'on clique sur un élément HTML.
    - addEventListener vous permet de définir votre propre logique avec un callback
  - Faire quelque chose après une action
    - Après un appel asynchrone ou un délai, le callback permet de déclencher une action.

SetTimeout

- Permet de décaler une action dans le temps

```
setTimeout(function() {
   alert("Hello");
}, 1000); // 1000ms = 1 second
alert("world");
```

SetInterval

- Permet de faire une action périodiquement

```
setInterval(function() {
   alert("Hello");
}, 1000);
// Displays Hello every second, which is quite annoying
```

Clear Timeout & Interval

- Permet de retirer une action setTimeout

```
const timeoutID = setTimeout(function() {
   alert("Hello");
}, 1000);
clearTimeout(timeoutID); // Won't display the alert
```

createElement

- Il est possible de créer des éléments :

```
let paragraphe = document.createElement("p");
paragraphe.innerText = "Contenu du paragraphe";
```

Ajouter un élément

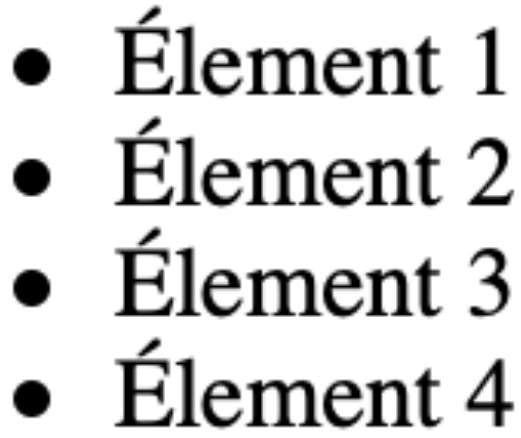
- Il est possible d'ajouter des éléments à un élément :

```
let paragraphe = document.createElement("p");
paragraphe.innerText = "Contenu du paragraphe";

document.body.appendChild(paragraphe);
```

Exercice : créer une liste

- Créer la liste suivante en utilisant createElement :



Récupérer la valeur d'un champ

- Pour récupérer la valeur d'un champ il suffit de récupérer la propriété *value* :

```
const champTexte = document.querySelector('[name="message"]');
console.log(champTexte.value); // Affiche le contenu du champ message
```

L’interface Event

- « L'interface **Event** représente tout événement qui a lieu dans le DOM » *MDN*

- L’objet émanant de cette interface lors d’un événement est récupérable de cette façon :

```
window.addEventListener("keyup", (evenement) => {
    console.log(evenement); // L’objet émanant de l’interface Event
});
```

Empêcher qu'une action se déroule lors d'un événement

- Il est possible d'empêcher le déroulement par défaut d'une action lorsqu'un événement se produit.

  Par exemple lors du clic sur un lien, on est redirigé vers une autre page. Il est possible de l'empêcher en JavaScript :

```
const lien = document.querySelector("a");
lien.addEventListener("click", (evenement) => {
    evenement.preventDefault();
});
```

Exercice

1. Créer un formulaire dans la page HTML comportant un champ de texte et un champ d'envoi.

2. Créer un paragraphe dans la page HTML.

3. Afficher le contenu du champ dans le paragraphe lorsque l'on envoie le formulaire.

4. Le faire en temps réel

Créer un objet à partir du JSON

- On utilise l'objet JSON à partir d'une chaîne avec la méthode parse() :

```
const exempleJSON = '{"nom":"Julie","ville":"Papeete"}';

const personne = JSON.parse(exempleJSON);
console.log(personne.ville); // Affiche Papeete
```

Créer du JSON à partir d'un objet

- À l'inverse, il est possible de créer du JSON à partir d'un objet :

```
const personne = {
    "nom": "Jean",
    "ville": "Nairobi"
};
const exempleJSON = JSON.stringify(personne);
console.log(exempleJSON);
// Affiche {"nom":"Jean","ville":"Nairobi"}
```

AJAX

- Il est possible de faire des requêtes HTTP en JavaScript avec l'objet : XMLHttpRequest

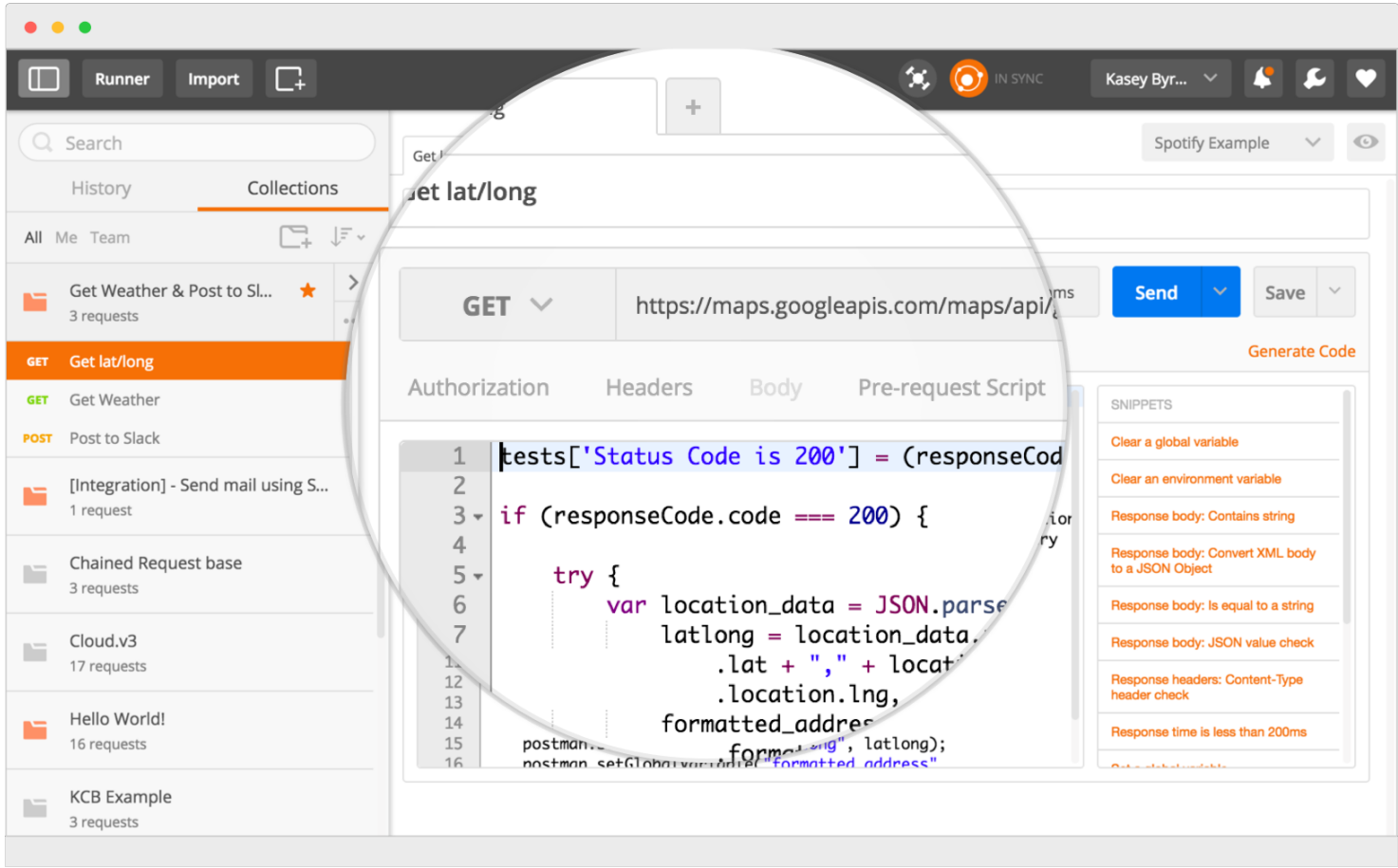
```
const req = new XMLHttpRequest();

req.onreadystatechange = function(event) {
    // XMLHttpRequest.DONE === 4
    if (this.readyState === XMLHttpRequest.DONE) {
        if (this.status === 200) {
            console.log("Réponse reçue: %s", this.responseText);
        } else {
            console.log("Status de la réponse: %d (%s)", this.status, this.statusText);
        }
    }
};

req.open('GET', 'http://www.mozilla.org/', true);
req.send(null);
```

Outils pour les APIS

- curl
- Postman



Exercice API - Image du jour NASA

- Afficher dans la page l'image du jour de la NASA avec l'API APOD

- https://api.nasa.gov

Exercice API – Qualité de l'air

- Créer un formulaire avec un champ de texte qui permet à l'utilisateur de rentrer le nom d'une ville
- Afficher le nom de la ville et les initiales du pays
- Afficher un visage souriant si la qualité est bonne, un visage neutre si elle ne l'est pas, un visage fâché si la qualité est mauvaise.
- Utiliser https://api.openaq.org/ et thenounproject pour les icônes
- Utiliser un framework CSS (Semantic UI par exemple)
- Déployer l'application

Nouvelles normes ES6+

ECMAScript

- « Core Javascript » ou « ECMAScript » désigne le « pur » JavaScript

- En opposition à ce qui est ajouté par l'environnement dans lequel JavaScript est utilisé

Déclaration de variables

- Il est désormais recommandé d'utiliser à la place de *var* dans l'ordre de préférence suivante :
  - const : déclarer une constante
  - let : déclarer une variable

Convention de nommage

- Attention aux mots réservés
- Éviter les noms sans signification (sauf pour les variables tampon)
- Utiliser la notation « camelCase »
- L'usage de la majuscule indique généralement un Constructeur d'objet
- Les constantes sont généralement en majuscules

La syntaxe « back-tick »

- Cette nouvelle syntaxe implémentée par ES6 permet d'introduire des « templates » à l'intérieur d'une chaîne de caractères classique.

```
const classeCSS = "active";
const html = `<p class="${ classeCSS }">
    Le contenu d'un paragraphe
</p>`;
```

Importer des modules

- Il est possible de créer des modules et de les importer
- Pour cela on utilisera le mot-clé export au sein du fichier comportant le module
- Ce fichier pourra être importé depuis un autre fichier avec le mot-clé Import

Créer un module

```
1  function getJSON(url, rappel) {
2    let xhr = new XMLHttpRequest();
3    xhr.onload = function () {
4      rappel(this.responseText)
5    };
6    xhr.open('GET', url, true);
7    xhr.send();
8  }
9
10 export function recupererContenuUtile(url, rappel) {
11   getJSON(url, donnees => rappel(JSON.parse(donnees)));
12 }
```

Importer un module depuis un autre fichier

```
1 | import { recupererContenuUtile } from '/modules/fichier.js';
2 |
3 | recupererContenuUtile('http://www.example.com',
4 |     donnees => { faireQuelqueChoseDUtile(donnees); });
```

Objets et Prototypes

Basé sur les prototypes

- **JavaScript dispose d'une approche basée sur les « prototypes » et non sur les « classes ».**

Concepts ou entités définies ?

- Doit-on créer une représentation générique d'un objet ? Avec une « classe » qui sert à créer des instances particulières ?

- Doit-on partir d'un objet existant comme représentant ? Avec un « prototype » qui sert à créer des objets qui lui ressemble.

L’objet littéral en JavaScript

- Un objet JavaScript est le conteneur d’une collection de valeurs nommées.

Accès aux propriétés d'un objet

- L'accès peut se faire de deux manières suivantes :
  - La notation point « . », suivi du nom de propriété sans les guillemets de programmation
  - La notation crochets « [] » comportant le nom de propriété avec les guillemets de programmation

Créer un objet avec Object.create(proto)

- Cette méthode crée un objet à partir d'un objet proto utilisé comme prototype.

```
const person = {
    isHuman: false,
    printIntroduction: function () {
        console.log(`My name is ${this.name}. Am I human? ${this.isHuman}`);
    }
};

const me = Object.create(person);

me.name = "Matthew"; // "name" is a property set on "me", but not on "person"
me.isHuman = true; // inherited properties can be overwritten

me.printIntroduction();
```

Connaitre le prototype d'un objet

- Object.getPrototypeOf(obj)

Geler un objet

- On peut empêcher toute modification ultérieure d'un objet avec : Object.freeze(obj)

- On peut savoir si un objet est gelé avec : Object.isFrozen(obj)

Sceller un objet

- On peut empêcher l'ajout de nouvelles propriétés à un objet avec : Object.seal(obj)

- On peut savoir si un objet est scellé avec : Object.isSealed(obj)

Principes de l'approche « prototypale » en JavaScript

- Le littéral objet, permet de créer des objets individuels en JavaScript

- La notation JSON permet d'archiver ces objets, ou des collections de ces objets et de les réutiliser y compris avec d'autres technologies

L’héritage par délégation

- Lorsque l’on fait appel à une propriété d’un objet, l’interpréteur JavaScript commente par la chercher au sein des propriétés propres de l’objet, ou, à défaut, parmi les propriétés propres du prototype, ou à défaut, remonte la chaîne des prototype jusqu’y la racine.

- En cas d’échec, la valeur undefined est retournée.

Recommandations

- Utiliser const pour déclarer un objet, qui correspond à un conteneur mais pas son contenu qui lui pourra être quand même modifié.

Utiliser :

```
const objet = {}
```

ou

```
const objet = Object.create(proto)
```

Exercice sur les objets

- Créer un objet Personne
  - Avec les propriétés nom, prenom, dateNaissance
  - Avec une méthode qui calcule l'âge et une méthode qui donne le nom complet
- Créer des objets qui héritent de Personne, changer leur propriétés
- Exécuter les méthodes pour chaque objet

Créer un objet avec une classe

- Attention : « Cette syntaxe n'introduit pas un nouveau modèle d'héritage dans JavaScript ! Elle fournit uniquement une syntaxe plus simple pour créer des objets et manipuler l'héritage. » MDN

```
1 class Rectangle {
2   constructor(hauteur, largeur) {
3     this.hauteur = hauteur;
4     this.largeur = largeur;
5   }
6 }
```

Les expressions de classe

```
1  // anonyme
2  let Rectangle = class {
3    constructor(hauteur, largeur) {
4      this.hauteur = hauteur;
5      this.largeur = largeur;
6    }
7  };
8
9  // nommée
10 let Rectangle = class Rectangle {
11   constructor(hauteur, largeur) {
12     this.hauteur = hauteur;
13     this.largeur = largeur;
14   }
15 };
```

Méthodes statiques

```
1  class Point {
2    constructor(x, y) {
3      this.x = x;
4      this.y = y;
5    }
6
7    static distance(a, b) {
8      const dx = a.x - b.x;
9      const dy = a.y - b.y;
10     return Math.hypot(dx, dy);
11   }
12 }
13
14 const p1 = new Point(5, 5);
15 const p2 = new Point(10, 10);
16
17 console.log(Point.distance(p1, p2));
```

Utilisation depuis une autre méthode statique

- On peut utiliser *this*

```
class StaticMethodCall {
  static staticMethod() {
    return 'Méthode statique appelée';
  }
  static anotherStaticMethod() {
    return this.staticMethod() + ' depuis une autre statique';
  }
}
StaticMethodCall.staticMethod();
// 'Méthode statique appelée'
StaticMethodCall.anotherStaticMethod();
// 'Méthode statique appelée depuis une autre statique'
```

Utilisation depuis un construteur et les autres méthodes

Les méthodes statiques ne sont pas directement accessibles via le mot-clé `this`. Il faut les appeler avec le nom de la classe qui préfixe le nom de la méthode statique `NomDeClasse.MéthodeStatique()` (comme pour les autres appels en dehors de la classe) ou avec la propriété `constructor` : `this.constructor.MéthodeStatique()`.

```
1  class StaticMethodCall{
2    constructor(){
3      console.log(StaticMethodCall.staticMethod());
4      // 'appel de la méthode statique'
5
6      console.log(this.constructor.staticMethod());
7      // 'appel de la méthode statique'
8    }
9
10   static  staticMethod(){
11     return 'appel de la méthode statique.';
12   }
13 }
```

Créer une sous classe

```
1  class Animal {
2    constructor(nom) {
3      this.nom = nom;
4    }
5
6    parle() {
7      console.log(this.nom + ' fait du bruit.');
8    }
9  }
10
11 class Chien extends Animal {
12   constructor(nom) {
13     super(nom); // appelle le constructeur parent avec le paramètre
14   }
15   parle() {
16     console.log(this.nom + ' aboie.');
17   }
18 }
```

Appeler des méthodes de la classe parente

Le mot-clé `super` est utilisé pour appeler les fonctions rattachées à un objet parent.

```
class Chat {
  constructor(nom) {
    this.nom = nom;
  }

  parler() {
    console.log(this.nom + ' fait du bruit.');
  }
}

class Lion extends Chat {
  parler() {
    super.parler();
    console.log(this.nom + ' rugit.');
  }
}
```

Exercice : classes

- Créer une classe Publication avec les propriétés auteur, année et une méthode description qui affiche l'auteur et l'année.

- Créer une sous-classe Film avec une propriété réalisateur en plus.

- Créer un objet à partir de la classe Film, et afficher la description.

Promesses

Objet Promise

- Une « promesse » est un objet Promise qui représente le retour (succès ou échec) d'une opération asynchrone : c'est à la fois un conteneur de l'information sur l'état de l'opération et un notificateur du résultat (émetteur d'événement). On attache des *callbacks* à cet objet, plutôt que de les passer dans une fonction.

Exemple de promesse

JavaScript Demo: Promise Constructor

```
1  var promise1 = new Promise(function(resolve, reject) {
2    setTimeout(function() {
3      resolve('foo');
4    }, 300);
5  });
6
7  promise1.then(function(value) {
8    console.log(value);
9    // expected output: "foo"
10 });
11
12 console.log(promise1);
13 // expected output: [object Promise]
```

Exercice

- Faire un XMLHttpRequest par le biais d'une promesse
- En faire deux à la suite

Exercice – Pays du monde

- Afficher l'ensemble des drapeaux du monde
- Lorsque l'on clique sur un drapeau, des informations sur le pays s'affichent : son nom, sa capitale (au minimum).
- Utiliser l'API restcountries.eu pour obtenir les données

Éléments personnalisés

Pour créer un élément personnalisé

1. Créer un objet de classe qui étend HTMLElement (ou celui d'un autre élément) définissant le comportement de l'élément

2. Enregistrer l'élément dans le CustomElement Registry

3. Utiliser l'élément depuis le HTML ou avec createElement

Exemple d'élément personnalisé

```
class TexteMaison extends HTMLElement {
    constructor() {
        // Toujours appeler « super » d'abord dans le constructeur
        super();

        // Création d'une racine fantôme
        const shadow = this.attachShadow({ mode: 'open' });

        // Prendre le contenu de l'attribut
        const attributTexte = this.getAttribute('texte');

        // Ajouter du contenu
        const paragraphe = document.createElement('p');
        paragraphe.textContent = attributTexte;
        shadow.appendChild(paragraphe);
    }
}

customElements.define('texte-maison', TexteMaison);
```

Exercice

- Créer un élément personnalisé pour afficher les informations du pays.