

Design Patterns :

Développement Web

Julien Provillard

PRÉSENTATION DU COURS

Ce que l'on trouve dans ce cours

- ❑ Introduction au langage Javascript
- ❑ Scripts dans une page Web
 - Manipulation du DOM
 - Programmation événementielle
 - Programmation asynchrone
- ❑ Architecture client-serveur basique
 - Introduction à Node.js
 - Requêtes HTML

Ce que l'on ne trouve pas dans ce cours

- ❑ Pas de framework avancé pour le frontend (Vue, Angular, React, ...)
- ❑ Pas de BDD
- ❑ Pas de sécurité (authentification, cryptage, ...)
- ❑ Peu de bibliothèques externes
- ❑ Apprentissage autonome en projet

L'organisation du cours

- ❑ Une moitié du cours de Design Pattern
- ❑ 12h CM, 6h TP, 6h TD
- ❑ Un projet noté

Les prérequis

- ☐ Connaître les bases de HTML / CSS (pages Web statiques)
- ☐ Être à l'aise en programmation impérative
- ☐ Avoir de bonnes notions en POO

Les outils

❑ Pour le début du cours :

- Un navigateur internet récent
- Un éditeur de texte

❑ Très vite :

- Un environnement de développement intégré
- Les outils en ligne de commande `nvm`, `node`, `npm`, `curl`

❑ À terme :

- Des outils plus spécifiques pour votre projet : `mongodb`, `postman`, `frontend`, ...

LES BASES DE JAVASCRIPT

Présentation

- ❑ Le but initial de Javascript était de rendre les pages Web dynamiques.
- ❑ Aujourd'hui, on le trouve aussi bien côté client que serveur ou même dans des applications de bureau.
- ❑ Les programmes (appelés scripts) sont interprétés. Il suffit d'avoir un interpréteur pour les exécuter. C'est le cas de tous les navigateurs.
- ❑ Le langage a été proposé en 1995 et normé une première fois en 1997 : standard ECMA-262 (ECMAScript édition 1 abrégé en ES1).
- ❑ Nous en sommes à ES14 (2023)... avec tous les problèmes de compatibilité qui en résultent (<https://kangax.github.io/compat-table/es6/>).

Scripts

- ❑ Javascript s'exécute à l'intérieur d'une page Web.
- ❑ Le code doit donc apparaître dans un document HTML.

```
<!-- hello-world.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Hello World</title>
  </head>

  <body>
    <p>Hello from html!</p>
    <script>
      alert("Hello from javascript!");
    </script>
  </body>
</html>
```



Scripts

❏ Il peut être externalisé.

```
<!-- hello-world2.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Hello World</title>
  </head>

  <body>
    <p>Hello from html!</p>
    <script src="hello.js"></script>
  </body>
</html>

/* hello.js */
alert("Hello from javascript!");
```



Scripts

❏ Il peut être externalisé.

```
<!-- hello-world3.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Hello World</title>
  </head>

  <body>
    <p>Hello from html!</p>
    <script src="http://my.domain.com/somewhere/script.js"></script>
  </body>
</html>
```

Scripts

- ❑ Il peut y avoir plusieurs scripts.
- ❑ Ils s'exécutent alors successivement dans le même environnement.

```
<!-- two-scripts.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Hello World</title>
  </head>

  <body>
    <script>
      let name = prompt("What's your name?", "John Doe");
    </script>
    <script>
      alert(`Hello ${name}!`);
    </script>
  </body>
</html>
```



Scripts

❑ Si un script a une source, son contenu est ignoré.

```
<!-- hello-world4.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Hello World</title>
  </head>

  <body>
    <p>Hello from html!</p>
    <script src="hello.js">
      alert("I don't show!"); <!-- ignoré -->
    </script>
  </body>
</html>

/* hello.js */
alert("Hello from javascript!");
```



Scripts

- ☐ À partir de maintenant, on ne montrera plus la structure du document HTML si elle n'est pas nécessaire.
- ☐ Sauf indications contraires, les scripts sont supposés apparaître à la fin du corps du document.
- ☐ On reparlera de ce dernier point plus tard dans le cours.

LES BASES DU LANGAGE

Instructions

- ❑ Javascript est un langage impératif : le code est une suite d'instructions.
- ❑ Une instruction se termine par un point-virgule usuellement suivi d'un saut de ligne.

```
alert("Hello world!");  
prompt("How are you?", "fine/not good/don't bother me");
```

- ❑ En présence de sauts de ligne, les points-virgules sont (souvent) optionnels.



```
alert(1)  
[2, 3].forEach(alert)
```

- ❑ Gardez un style propre : une instruction par ligne suivi d'un point-virgule.

Mode strict

- ❑ Javascript est un langage qui évolue depuis sa création.
- ❑ Les évolutions sont normalement compatibles avec les spécifications précédentes... sauf pour ES5.
- ❑ Pour garder la sémantique des anciens scripts, ces changements sont désactivés par défaut.
- ❑ On peut les activer :
 - Explicitement, en commençant un script (ou une fonction) par l'instruction `"use strict";`
 - Implicitement, en utilisant certaines fonctionnalités (classes, modules, ...)
- ❑ Tout nouveau script devrait l'utiliser! (debugging et sécurité)

Variables

- ❑ On déclare une variable avec le mot clé `let` suivi d'un nom.

```
let myVariable;
```

- ❑ On peut assigner une valeur à une variable déclarée avec l'opérateur `=`.

```
myVariable = 21;
```

- ❑ On peut combiner les deux opérations.

```
let myOtherVariable = 3 * myVariable + 1;
```

- ❑ Dans les anciens scripts, on peut trouver `var` à la place de `let`. Les deux mots-clés sont *presque* équivalents. L'usage de `var` est déprécié.

Variables

- ❑ Une variable introduite par `let` a une portée lexicale : elle existe de sa déclaration à la fin du bloc où elle est déclarée.
- ❑ Une variable ne peut pas être déclarée plusieurs fois dans une même portée.



```
let myVariable = 1;  
let myVariable = 2;
```

- ❑ Une variable peut être redéclarée dans une portée imbriquée. C'est alors une nouvelle variable portant localement le même nom (à éviter).

```
let myVariable = 1;  
{  
  let myVariable = 2;  
  alert(myVariable);  
}  
alert(myVariable);
```

Variables

- ❑ Les caractères valides dans les noms de variables sont :
 - les lettres,
 - les chiffres,
 - les symboles \$ et _.
- ❑ Un nom de variable ne peut pas commencer par un chiffre.
- ❑ Usuellement, on utilise la convention camelCase pour les noms de variables.

❑ Exemples :

```
let myVariable = 42;  
let _ = "Je suis un nom valide"; // plutôt pour les bibliothèques  
let θ = "θ est bien une lettre"; // à éviter
```

Constantes

- ❑ Pour introduire une constante, on utilise `const` à la place de `let`.
- ❑ Une constante doit être initialisée au moment de la déclaration. Sa valeur ne sera pas modifiable par la suite.
- ❑ Pour une vraie constante (portée globale, valeur fixe qui ne dépend pas d'un calcul), on utilise des noms en majuscules.
- ❑ Exemples :

```
const PI = 3.1425927;  
const THE_ULTIMATE_DEBUGGING_MESSAGE = "Fatal error!";  
const snapshot = application.getState();
```

Valeurs

☐ Javascript dispose de 7 types primitifs (valeur unique) :

- boolean
- number
- bigint (ES10)
- string
- null
- undefined
- symbol

☐ Et d'un seul type non-primitif (valeur composée) :

- object

☐ Est-ce que ça vous paraît cohérent ?

☐ Pour connaître le type d'une valeur, on peut utiliser l'opérateur `typeof`.

```
alert(typeof 42); // affiche number
```

```
alert(typeof undefined); // affiche undefined
```

```
alert(typeof null); // affiche object, pour des raisons historiques
```

Type boolean

- ❑ Représente une valeur logique : `true` ou `false`.
- ❑ C'est le type de valeurs retourné par les opérateurs de comparaisons (pour les types qui les supportent).
 - `==`, `!=`, `===` et `!==` pour tous les types
 - `<`, `<=`, `>`, `>=`, pour les nombres et les chaînes
- ❑ On trouve également les connecteurs logiques classiques `!`, `||` et `&&` ainsi que `??`.
- ❑ Pourquoi deux égalités (`==` et `===`)?
- ❑ Que fait `??`, l'opérateur de coalescence des nuls?
- ❑ Bientôt plus à ce sujet.

Type number

- ❑ Représente les nombres aussi bien entiers que flottants.
- ❑ En interne, flottants double précision sur 64 bits (mantisse sur 53 bits).
- ❑ Attention à la distinction entre entiers et entiers sûrs.

```
let min = Number.MIN_SAFE_INTEGER; // min = -9007199254740991 ie  $-(2^{53}) + 1$   
let max = Number.MAX_SAFE_INTEGER; // max = 9007199254740991 ie  $2^{53} - 1$   
alert(max + 1 - 1 == max); // true  
alert(max + 2 - 2 == max); // false  
alert(max + 1 == max + 2); // true
```

- ❑ Les notations exponentielles, binaires, octales et hexadécimales sont possibles. Les nombres suivants sont tous égaux :

```
31    3.1e1    0b11111    0o37    0x1F
```

- ❑ Les notations octales de la forme `037` sont interdites en mode strict.

Type number

- ❑ Opérateurs unaires: +, -
- ❑ Opérateurs binaires: +, -, *, /, %, **
- ❑ Opérateurs bits à bits (pour les entiers): ~, |, &, <<, >>, >>>
- ❑ Support des affectations combinées (+=, >>=, etc)
- ❑ Support des pré et post incrémentations et décrémentations (++ et --).
- ❑ Objet Math
- ❑ Valeurs particulières: Infinity, -Infinity, NaN

```
alert(1 / 0); // Infinity
```

```
alert(0 / 0); // NaN
```

```
alert("deux" - 1); // NaN
```

```
alert("2" - 1); // 1 ???
```

Type bigint

- ❑ Entiers en précision arbitraire

- ❑ Il suffit de suffixer un entier par un n.

```
let x = 2 ** 53 + 1 - 2 ** 53; // 0
```

```
let y = 2n ** 53n + 1n - 2n ** 53n; // 1n
```

- ❑ Réservés pour des cas particuliers (mathématique, cryptographie, ...)

- ❑ Comportements particuliers par rapport aux autres types primitifs
(introduction tardive ES10)

Type string

- ❑ Représente les chaînes de caractères, utilise les guillemets simples, doubles ou renversés.

```
'Hello'    "Hello"    `Hello`
```

- ❑ Les guillemets renversés autorisent l'échappement d'expressions.

```
let who = "World";  
let birthYear = 1985;  
alert(`Hello ${who}! I'm ${new Date().getFullYear() - birthYear} old.`);
```

- ❑ On obtient la taille d'une chaîne par `str.length` et son *i*-ème caractère par `str[i]`. On peut concaténer une chaîne avec une autre valeur à l'aide de l'opérateur `+`... et il existe beaucoup d'autres opérations.

```
alert("Hello " + "World");  
alert(1 + " + " + 2 + " = " + (1 + 2)); // pourquoi des parenthèses ?
```

- ❑ Pas de type distinct pour les caractères.

Les types `null` et `undefined`

- ❑ Chacun est restreint à une unique valeur qui porte le même nom.
- ❑ La valeur `undefined` représente une donnée non initialisée ou un champ non défini.
- ❑ Il n'est pas recommandé d'assigner manuellement `undefined`.
- ❑ La valeur `null` représente une donnée vide, absente ou inconnue.
- ❑ Les deux valeurs sont très proches (en particulier `undefined == null`) mais leur sémantique est différente : `null` est une valeur intentionnelle alors que `undefined` est la valeur par défaut d'une variable non initialisée.

Le type symbol

- ❑ Type en lien avec les objets.
- ❑ Il est mentionné ici par soucis d'exhaustivité mais nous y reviendrons plus tard.

Conversions automatiques

- ❑ Les opérateurs attendent souvent des opérandes d'un type précis.
- ❑ Il en va de même pour les fonctions et leurs arguments.
- ❑ De nombreux langages produisent des erreurs à la compilation ou à l'exécution si les types attendus ne sont pas respectés.
- ❑ À l'inverse, Javascript tente des conversions automatiques.
- ❑ Comprendre ce mécanisme est important pour éviter ou corriger des bugs parfois subtils.

Conversion vers string

- ❑ Cas d'usage : `alert(value)`, `String(value)`, ...
- ❑ Transforme la valeur en chaîne de caractères.
- ❑ Le résultat obtenu est évident dans la plupart des cas.

Conversion vers number

- ❑ Cas d'usage : `value1 - value2`, `Number(value)`, ...
- ❑ Transforme la valeur en nombre.

Valeur	Conversion
undefined	NaN
null	0
false	0
true	1
Une chaîne	On ignore les espaces de tête et de fin de la chaîne. Si la chaîne est vide, on obtient 0. Si la chaîne représente un nombre, on obtient ce nombre. Dans tous les autres cas, on obtient NaN.
Un symbole	Erreur

- ❑ Écrire `+value` est équivalent à `Number(value)`.

Conversion vers boolean

- ❑ Cas d'usage : `!value`, `Boolean(value)`, ...
- ❑ Transforme la valeur en booléen.
- ❑ Les valeurs intuitivement vides deviennent `false` (c'est-à-dire `0`, `NaN`, `""`, `undefined` et `null`).
- ❑ Les autres valeurs donnent `true`. C'est en particulier le cas des chaînes `"0"` et `" "`.

Égalités et comparaisons

- ❑ Opérateurs ==, !=, <, <=, > et >=
- ❑ Ordre naturel sur les nombres (mais attention aux flottants)
`0.1 + 0.2 != 0.3;`
- ❑ Ordre lexicographique sur les chaînes (unicode sous-jacent)
`"a" > "A"    "java" < "javascript"    "programmation" < "programs"`
- ❑ Pour tous les autres cas, conversion préalable vers number
- ❑ Cela provoque des cas étranges :

```
if ("0") {  
    alert("Display!");  
}
```

```
if ("0" == true) {  
    alert("Don't display!");  
}
```

Cas particuliers

- ❑ Le nombre NaN n'est égal à aucune valeur (pas même NaN).
- ❑ Les valeurs `null` et `undefined` sont égales mais différentes de toutes les autres valeurs.

- ❑ Expliquez les résultats suivants:

```
null >= 0; // true  
null > 0; // false  
null == 0; // false
```

```
undefined >= 0; // false  
undefined > 0; // false  
undefined == 0; // false
```

Égalité stricte

- ❑ Une forme d'égalité stricte existe: `===`.
- ❑ Ne réalise pas de conversion, deux valeurs de types différents sont nécessairement distinctes.
- ❑ En particulier, `null !== undefined`.
- ❑ Par contre, `NaN !== NaN`.

Retour sur les opérateurs logiques

Expression	Résultat
<code>!a</code>	true si a se convertit en false; false sinon
<code>a b</code>	a si a se convertit en true; b sinon
<code>a && b</code>	a si a se convertit en false; b sinon
<code>a ?? b</code>	a si <code>a !== null</code> et <code>a !== undefined</code> ; b sinon

☐ Autrement dit:

- Une séquence `a || b || c || d` renvoie la première valeur qui s'identifie à true (ou la dernière par défaut).
- Une séquence `a && b && c && d` renvoie la première valeur qui s'identifie à false (ou la dernière par défaut).
- Une séquence `a ?? b ?? c ?? d` renvoie la première valeur définie (ou la dernière par défaut).

Structures de contrôles : conditionnelles

❑ Instruction `if (condition) if-true [else if-false]`.

- L'expression *condition* est évaluée puis convertie en booléen.
- La branche adaptée est ensuite **exécutée**.

❑ Opérateur ternaire `condition ? if-true : if-false`

- L'expression *condition* est évaluée puis convertie en booléen.
- La branche adaptée est ensuite **évaluée** et sa valeur est celle de l'expression complète.

❑ Bonnes pratiques :

- Les branches d'un `if` devraient être des blocs entre accolades.
- Les branches de l'opérateur ternaire devraient éviter les effets de bord.

Structures de contrôles : les boucles

`while (condition) body`

`do body while (condition)`

`for (initialization; condition; increments) body`

- ❑ La boucle s'arrête si *condition* s'identifie à `false` et se poursuit sinon.
- ❑ Les corps des boucles devraient toujours être des blocs (même s'ils ne contiennent qu'une seule instruction).
- ❑ Pour la boucle `for`, *initialization* peut contenir des déclarations de variables locales à la boucle.

```
let i = 42;  
for (let i = 3; i >= 0; i--) {  
  alert(i); // 3 2 1 0  
}  
alert(i); // 42
```

Structures de contrôles : les boucles

`while (condition) body`

`do body while (condition)`

`for (initialization; condition; increments) body`

- ❑ On peut quitter la boucle courante à l'aide de l'instruction `break`.
- ❑ On peut passer au tour de boucle suivant à l'aide de l'instruction `continue`.
- ❑ On peut faire de même sur plusieurs niveaux dans des boucles imbriquées à l'aide d'un label.

```
loop: for (let i = 0; i <= 3; i++) {  
  for (let j = 0; j <= 3; j++) {  
    alert(`i = ${i}, j = ${j}`);  
    if (i && j) { break loop; }  
  }  
}
```



Ce n'est pas un goto, le label doit porter sur une boucle et n'être utilisé qu'à l'intérieur de celle-ci.

Structures de contrôle : switch

```
switch (expression) {  
    case expression1: instructions1; [break;]  
    case expression2: instructions2; [break;]  
    ...  
    case expressionN: instructionsN; [break;]  
    [default: instructions; [break;]]  
}
```

- ❑ Attention au passage d'un cas au suivant en l'absence de break.
- ❑ Particularité de javascript, les cas sont des expressions et non des constantes.
- ❑ Les comparaisons sont faites par égalité stricte.

Structures de contrôle : exemple de switch

```
loop: for (let a = 0; ; a++) {  
  switch (a) {  
    case a * a - 2:  
      break loop;  
    default:  
      alert(a);  
  }  
}
```

❑ N'écrivez pas ça!

❑ Uniquement à titre d'illustration.

Fonctions

- ❑ Une déclaration de fonction est de la forme :

```
function name([par1, [par2, ..., [parN]...]]) { instructions }
```

- ❑ Un appel de fonction est de la forme:

```
name([arg1, [arg2, ..., [argN]...]])
```

- ❑ Le nombre d'arguments ne doit pas nécessairement être le même que le nombre de paramètres de la fonction.
 - Les arguments surnuméraires sont ignorés.
 - Les arguments manquant prennent la valeur `undefined`.
- ❑ Les fonctions définies dans une portée sont utilisables dès le début de cette portée et peuvent capturer des variables définies plus loin. C'est le phénomène de remontée (*hoisting*).

Fonctions: exemples de *hoisting*

Exemple 1

```
alert(isEven(42));  
function isEven(n) {  
    return n ? isOdd(n - 1) : true;  
}  
function isOdd(n) {  
    return n ? isEven(n - 1) : false; // return n && isEven(n - 1)  
}
```

Exemple 2

```
show1();  
show2();  
function show1() { alert(x); }  
show1();  
show2();  
let x = 42;  
show1();  
show2();  
function show2() { alert(x); }
```

Zone morte temporaire de la variable x.
La variable est visible mais sa lecture est interdite.

Fonctions: paramètres par défaut

❑ Si un argument est manquant, on a vu que sa valeur était `undefined`.

❑ On peut donner une valeur par défaut différente.

```
function message(from, body = "nothing") { alert(`user ${from} says: ${body}`) }
```

```
message("E. Nigma"); // "user E. Nigma says: nothing"
```

```
message("E. Nigma", undefined); // "user E. Nigma says: nothing"
```

❑ La valeur par défaut n'a pas besoin d'être une constante, elle est systématiquement réévaluée.

```
let count = 0;
```

```
function defaultMessage() { return `${++count} empty message(s) detected.`; }
```

```
function message(from, body = defaultMessage()) { alert(`user ${from} says: ${body}`); }
```

```
message("A. Nonymous"); // "user A. Nonymous says: 1 empty message(s) detected."
```

```
message("A. Nonymous"); // "user A. Nonymous says: 2 empty message(s) detected."
```

Fonctions = Valeurs

- ❑ Les fonctions sont elles-mêmes des valeurs assignables.

```
function apply(f, x) { f(x); }  
apply(alert, "high-order call!"); // la fonction alert a été assignée au paramètre f
```

- ❑ Au lieu de déclarer une fonction, on peut la définir comme une expression. C'est la même syntaxe que pour une déclaration mais le nom est optionnel.

- ❑ La valeur obtenue doit être assignée ou utilisée de suite.

```
let f = function (x, y) {  
    alert(`${x} + ${y} = ${x + y}`);  
};  
f(1, 2);
```

```
(function (x, y) {  
    alert(`${x} + ${y} = ${x + y}`);  
})(1, 2);
```

Fonctions = Valeurs

❑ Y-a-t'il une différence entre

```
function f() { return; }
```

et

```
let f = function () { return; };
```

❑ Dans le second cas, pas de *hoisting*.

```
f(); // ok
```

```
function f() { return; }
```

```
g(); // erreur, dans la zone morte temporaire de g
```

```
let g = function () { return; };
```

Fonctions = Valeurs

- ❑ Pourquoi les expressions qui définissent les fonctions autorisent optionnellement un nom ?

```
let f = function g() { return; }  
g(); // erreur
```

- ❑ Important pour toutes les références internes, notamment les récursions.

```
let fac = function f(n) {  
  return n == 0 ? 1 : n * f(n - 1);  
};  
  
alert(fac(6));
```

Fonctions: callback

- ❑ Javascript est friand des fonctions de rappel.
- ❑ Ces fonctions sont passées en arguments d'une autre fonction qui les utilisera quand (et si) il y a besoin.

```
function interact(question, callbackSuccess, callbackFailure) {  
  let answer = prompt(question);  
  if (answer) { callbackSuccess(answer); }  
  else { callbackFailure(); }  
}
```

```
interact("What's your name?",  
  function (name) { alert(`Hello ${name}!`); },  
  function () { alert("Goodbye anonymous..."); });
```

Fonctions fléchées

❑ On peut définir des fonctions à l'aide de la syntaxe

```
let f = (par1, par2, ..., parN) => expression;
```

qui est équivalent à

```
let f = function (par1, par2, ..., parN) { return expression; };
```

❑ On peut adapter l'exemple précédent :

```
interact(  
  "What's your name?",  
  (name) => alert(`Hello ${name}!`),  
  () => alert("Goodbye anonymous..."));
```

❑ Un exemple où l'on renvoie vraiment une valeur

```
let compose = (f, g) => (x) => f(g(x));  
alert(compose((x) => x + 1, (x) => 2 * x)(3));
```

Fonctions fléchées

❑ Pour un corps de plusieurs lignes, il faut mettre des accolades et faire un **return** explicite.

```
let compose = (f, g) => (x) => {  
  let y = g(x);  
  let z = f(y);  
  return z;  
};
```

```
interact(  
  "What's your age?",  
  (age) => {  
    age = +age;  
    if (Number.isNaN(age) || age <= 0) { alert("Sorry, I didn't understand."); }  
    else { alert(`You are ${age} years old.`); }  
  },  
  () => alert("Don't ask a lady her age!"));
```

Interagir dans la page Web

- ❑ On a déjà beaucoup utilisé deux des fonctions prédéfinies.
- ❑ Les fonctions standards pour interagir avec l'utilisateur sont :

- `alert(message)`

Affiche message dans une fenêtre.

- `prompt(question, [default])`

Affiche question dans une fenêtre et permet d'entrer une réponse dans un champ. La fonction renvoie le champ sous forme de chaîne quand l'utilisateur appuie sur OK ou `null` s'il appuie sur Annuler.

- `confirm(message)`

Affiche message dans une fenêtre. La fonction renvoie `true` quand l'utilisateur appuie sur OK ou `false` s'il appuie sur Annuler.