

Framework JavaScript

React

1

Synthèse OCR et autres

0 - FRAMEWORKS JAVASCRIPT	5
Objectif et marché	5
Objectif	5
Le marché	5
Méthodo : construire une appli JavaScript – méta-framework	6
React	7
objectifs	7
Pré-requis	7
1 - CONCEPTS REACT	8
Bibliographie React	8
Site officiel	8
Doc officielle : à regarder !!!	8
Tuto officiel	8
Installation	8
Framework front-end : “FW front”	9
Principes	9
Bibliothèque ou Framework ?	9
React : bib. ET framework	9
Composant	10
Etat applicatif – props – état local	11
DOM virtuel	12
CRA	13
2 - INSTALLATION DE CRA	14
Présentation	14
Installation de Node	14
Gestionnaire des modules de Node : npm – Mise à jour du npm	15
Présentation de npm	15
Mise à jour du npm	15
Version du npm	15
Installation du CRA	16
Présentation	16
Installation du CRA	16
Sous Linux ou MacOS :	16
Créez le squelette d’application	17
Création du squelette de l’application : le framework	17
Création du squelette de l’application « memory »	17
Affichage à la fin de l’installation :	18
Architecture du framework : dossiers et fichiers installés	19
Organisation des dossiers et des fichiers du framework	19
Fichier README.md	19
Fichier package.json	19

Fichier yarn.lock	19
Dossier public	20
Dossier src	20
Démarrage de l'application	21
TP 22	
Exercices : installation d'un l'environnement Reac	22
3 – JAVASCRIPT ES6 / ES2015	23
Bibliographie JavaScript	23
Prérequis JS	24
Remarque	24
Tester JavaScript en ligne	24
Ré-introduction	24
Principes	25
Les types	26
Les nombres	27
Les chaines de caractères	27
Booléen	28
Symbole	28
Déclaration des variables – var et portée avant le ES6	29
Les opérateurs	30
Structures de contrôle	31
Les tableaux	32
Les objets	32
Les fonctions	33
Les closures (fermetures)	34
POO en JavaScript classique	35
Classe et POO	40
JS moderne : ES6/2015 - Standard ECMAScript	40
L'essentiel de ES6/2015	40
let 41	
const 42	
OCR - Classe	43
OCR - Fonction fléchée : fat arrow : =>	44
OCR - Fonction fléchée : fat arrow : =>	45
OCR – this et =>	46
Le mode strict	47
OCR – Destructuration	48
Exemples : un composant compteur	49
TP : mise à jour du composant compteur	49
4 – FONCTION PURE REACT	50
Rappels : création d'une application React	50
La procédure :	50
Le contenu :	51
OCR - API : module, export, import	52
Hello World : 1er composant pur fonctionnel	53
1ère fonction Component	53
Utilisation de la fonction Component	54
Tester le code : https://codepen.io/topics/	54
1ere approche de JSX	56

1ère fonction Component	56
Tester le code : https://codepen.io/topics/	57
1ère props	58
fonction Component avec props	58
Tester le code : https://codepen.io/topics/	60
Tester le code dans une application React	60
Ajouter une deuxième props	60
5 – JSX	61
Présentation	61
JSX vs React.createElement(...)	61
Notion de grappe	61
Exemple de formulaire JSX	62
Syntaxe : un peu HTML, un peu XML, un peu JSX	63
6 – COMPOSANT FONCTION, VARIABLE, CLASSE	64
TP Sublime text pour React	64
Composant dans une variable	65
Composant dans une classe	67
TP composant dans une variable ou dans une classe	69
TP memory	69
7 – LES EVENEMENTS	70
8 – TESTS EN JSX	72
Présentation - opérateur ternaire ?:	72
true, false, null et undefined ignorés par JSX	72
Tester avec l'opérateur &&	72
Application dans le memory	72
const won = new Date().getSeconds() % 3 === 0	72
9 – BOUCLE EN JSX	73
Utilisation de la fonction map : -> doc	73
Sur un tableau d'entiers :	73
Sur un tableau d'objets	74
En déstructurant :	75
La prop key	75
TP-J6-1 ->	76
10 - DIVERSES PROPRIETES DES COMPOSANTS	79
Children dans le DOM JavaScript	79
Composant parent – Composant enfant	79
Règle de passage des props	79
Les props techniques	79
Valeurs par défaut : propriété statique defaultProps	80
Définition des types : propriété statique propTypes	82
TP-J6-2 ->	83
11 - COMPOSANT CLASSE	84
Présentation	84

Une classe ressemble à :	85
Usage de cette classe comme d'une balise	85
Le problème du this (cf §3-Classe etPOO-this et => p.46)	86
L'état local : les attributs de la classe	90
TP-J7-1 ->	92
setState	94
TP-J7-2 ->	96
TP-J7-3 ->	99
12 – CYCLE DE VIE D'UN COMPOSANT	100
Principes de base	100
Cycle de vie simplifié	100
Etape de construction	101
Cycle de mise à jour	101
TP-Cycle de vie - 1 ->	102
TP-Cycle de vie - 2 ->	104
TP-Cycle de vie – Mise à jour du HandleCardClick et du getFeedbackForCard	106
13 – LES FORMULAIRES	109
Situation HTML avant React	109
Situation événementiel avant React	111
Bilan : exemple React	112
Validez et formatez avec des champs contrôlés	113
TP-J7-formulaire ->	116

Edition février 2020

0 - FRAMEWORKS JAVASCRIPT

Objectif et marché

Objectif

Développement d'applications web et mobiles « riches » sans langage serveur.

Avoir un code propre et pérenne :

- ⇒ Flash : la techno « pompée » ! Les FW JS s'appuient sur JS qui est le standard W3C.
- ⇒ Plat de spaghettis sauce JQuery ou Bootstrap : incompréhensible, difficile à faire évoluer, plein de bugs et d'effets de bord. Les FW JS mettent en œuvre un code propre, objet, orienté MVC.

Le marché

3 frameworks JS principaux en 2018

- ⇒ **React : facebook – 2013 : le plus utilisé**
- ⇒ **Angular : google – 2010-2012 : beaucoup de gens pas intéressé**
- ⇒ **VueJS : 2014 : en progrès**

React et Vue.js sont un bon choix, quoiqu'il arrive

Leur architecture et philosophie sont proches. Découvrir l'un permet de prendre la main plus rapidement avec l'autre

Les méthodes d'écriture de code JavaScript "modernes" seront nécessaires pour les apprivoiser (POO, structuré, MVC, etc.)

La syntaxe de templating en quasi-HTML de VueJS est plus "naturelle" que le JSX de React

Angular victime d'une politique d'évolution floue, de versions ayant brisé la rétro-compatibilité (la 2.0) et d'une certaine complexité pour se laisser dompter.

Source : Alsacréation - 2018

<https://www.alsacreations.com/actu/lire/1778-frameworks-javascript-angular-react-vue.html>

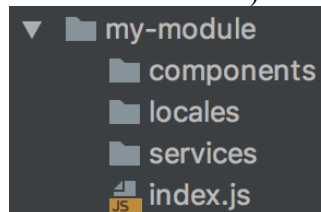
Méthodo : construire une appli JavaScript – méta-framework

Problème : rendre pérenne son application frameworkée JS

Combinaison React / Redux (stockage) / Babel / WebPack (gestion du bundling=paquetage=regroupement).

Notion de module JS : ensemble de fonctionnalités utilisables par un tier = package = API.

La structure d'un module (organisation des fichiers) va s'adapter au besoin.



L'index contient tout ce qui est publique (dans une logique de POO).

Les composants du module sont les composants du FW (React, Angular, etc.)

Les services sont les classes et les fonctions pour la logique du module.

Locales sert pour les traductions. On utilise du JSON.

Sources : Medium - OCR – 2018

<https://medium.com/openclassrooms-produit-design-et-ingénierie/construire-une-application-js-depuis-zéro-1-3-ad4047420824>

<https://medium.com/openclassrooms-produit-design-et-ingénierie/construire-une-application-js-depuis-zéro-2-3-2b0c4b227a2>

<https://medium.com/openclassrooms-produit-design-et-ingénierie/construire-une-application-js-depuis-zéro-3-3-64428629c9d9>

React

objectifs

Comprendre les concepts fondamentaux de React et ce qui les différencie d'autres FW.

Mettre en place un projet avec Create React App (CRA)

Créer des composants React complets avec la syntaxe JavaScript ES2015 et l'extension JSX

Gérer des formulaires avec ou sans contrôle de saisie

Tester ses composants React

Pré-requis

JavaScript

POO

1 - CONCEPTS REACT

Bibliographie React

Site officiel

<https://fr.reactjs.org>

Doc officielle : à regarder !!!

<https://fr.reactjs.org/docs/hello-world.html>

Tuto officiel

<https://fr.reactjs.org/tutorial/tutorial.html>

Installation

<https://fr.reactjs.org/docs/getting-started.html>

Framework front-end : “FW front”

Principes

Objectif : éviter le back-end et ses frameworks : symfony, zend, j2E, .net, etc.

Avoir une expérience utilisateur plus fluide

Comme pour tout FW : éviter de réinventer la roue à chaque fois

De nombreux FW front : React, Angular, Vue, etc.

Bibliothèque ou Framework ?

Une bibliothèque (library): offre des fonctions, des classes et/ou des packages permettant d’éviter de les réécrire et organisant même la façon de réfléchir à la solution des problèmes à résoudre.

Un framework : est une architecture semi-finie qu’on peut paramétrer et compléter en fonction des spécificités du produit à réaliser.

React : bib. ET framework

Projet open-source. Licence MIT. Piloté par Facebook. **React se concentre sur l’IHM.**

Routage, stockage sont laissés à d’autres : [React-Router](#), [Redux](#), [Redux-Offline](#)

20 000 modules sur « npm » en rapport avec React (npl : référentiel de 650 000 modules de développement orienté JavaScript).

De gros sites web ont migré leur interface web et mobile sur React.

Très populaire : <https://2017.stateofjs.com/2017/front-end/results/>

Composant

L'**approche basée composants** vient de Delphi (années 90).

React fait voler en éclat le **dogme web de la séparation stricte entre la structure (HTML), l'aspect (CSS) et le comportement (JS)**, classiquement écrits dans des fichiers distincts.

React, comme quasi tous les FW front, revient à la **notion de composant autonome**, cohérent et complet, ce qui correspond à une logique objet.

Un composant est une « **boîte noire** » classique avec une **API** clairement définie et un **cœur encapsulé** (logique objet).

Un composant contient tout le nécessaire à son bon fonctionnement : la structure HTML, le style CSS et le comportement JS.

Un **même fichier** contient les **3 volets connectés** avec une **syntaxe particulière : le JSX**. Pour comprendre un fonctionnement, tout est regroupé dans un fichier. C'est plus simple (logique objet). On peut faire des tests unitaires.

Arborescence de composants : un composant est composé d'autres composants. On dit qu'un composant est composé de « **composants fils** ». Notez que c'est trompeur ! La composition n'est pas un héritage !

<https://reactjs.org/docs/components-and-props.html>

<https://fr.reactjs.org/docs/components-and-props.html>

Etat applicatif – props – état local

L'état applicatif, c'est **l'ensemble des données utilisées pour afficher l'application**. Cet état est stable quand l'application est stable. Quand on met en œuvre une fonctionnalité de l'application, cet état peut changer (le plus souvent il change).

Il faut **éviter un couplage fort** entre les composants : autrement dit que les composants dépendent les uns des autres sans logique. Les dépendances suivent la logique arborescente des composants.

Principe React : « les données descendent, l'état remonte ».

Un composant fournit des données à ses composants fils.

Quand les composants fils modifient l'état applicatif, ils retournent une demande une mise à jour de l'état applicatif qui remonte de parent en parent.

Ce mécanisme sera détaillé dans le cours. Il utilise des « **props** » et la notion d' « **état local** ».

DOM virtuel

React permet d'utiliser un **DOM virtuel** qui gère l'interface avec le DOM réel.

C'est plus performant côté HTML.

Surtout : ça rend le code indépendant du type d'affichage : on pourra facilement passer à une application mobile ou sur du code côté serveur (node.js).

CRA

CRA : Create-React-App : « Créer une Application React ».

CRA est un outil écrit en Node pour faciliter le développement d'applications web fondées sur React.

CRA permet de générer automatiquement un squelette applicatif (des répertoires et des fichiers). En ce sens React est un framework.

CRA permet de masquer la complexité d'installation et de configuration des briques techniques associées : gestion de JavaScript moderne (ES2015+), bundling de notre application (avec Webpack), serveur de développement, génération de fichiers de production optimisés, etc.

CRA peut être mis à jour automatiquement.

On peut « ouvrir CRA » et gérer soi-même chaque aspect technique.

Au fil des versions, CRA gère de plus en plus de besoins (comme tout bon framework !).

Guide utilisateur : <https://github.com/facebook/create-react-app#user-guide>

2 - INSTALLATION DE CRA

Présentation

CRA est écrit en Node (Node 6 au minimum).

Node.js est un environnement d'exécution JavaScript.

Pour installer CRA, il faut d'abord installer Node.js

Installation de Node

Vérifiez si Node est déjà installé : en ligne de commande :

```
C :> node --version
```

Si Node n'est pas installé ou si la version est inférieure à la 6, installez Node :

- a. <https://nodejs.org/fr/download/>

Sur PC : MSI

- b. Tools for native models : on peut ne pas cocher
- c. Tapez node --version dans une console.

Tuto d'installation :

- d. <https://openclassrooms.com/fr/courses/1056721-des-applications-ultra-rapides-avec-node-js/1056956-installer-node-js>

Gestionnaire des modules de Node : npm – Mise à jour du npm

Présentation de npm

npm est le gestionnaire de modules de Node.

Il existe plus de 650 000 modules Node !

Mise à jour du npm

```
C:> npm install --global npm
```

A refaire une deuxième fois : on obtient

```
C:> npm install --global npm
C:\Users\bertrandliaudet\AppData\Roaming\npm\npx ->
C:\Users\bertrandliaudet\AppData\Roaming\npm\node_modules\npm\bin\npx-cli.js
C:\Users\bertrandliaudet\AppData\Roaming\npm\npm ->
C:\Users\bertrandliaudet\AppData\Roaming\npm\node_modules\npm\bin\npm-cli.js
+ npm@6.10.1
updated 1 package in 56.263s
```

La version 6.10.1 du npm a été installée : c'est la plus récente.

Sous Linux, MacOS, il faut faire un « sudo ».

Version du npm

```
C:> npm --version
```

Installation du CRA

Présentation

L'installation du CRA est une commande du npm.

Installation du CRA

```
C:> npm install --global create-react-app
...
+ create-react-app@3.0.1
```

La version 3.0.1 du CRA a été installée : c'est la plus récente.

Sous Linux ou MacOS :

Sous Linux, MacOS, il faut faire un « sudo ».

Créez le squelette d'application

Création du squelette de l'application : le framework

Un framework : c'est une architecture semi-finie qu'on peut paramétrer et compléter en fonction des spécificités du produit à réaliser. C'est le squelette qu'on va créer.

Création du squelette de l'application « memory »

On commence par se placer où on veut (dans le dossier react par exemple) :

```
C:> cd react
```

Ensuite on crée le squelette de l'application memory :

```
C:> create-react-app memory
```

Le squelette se situe dans le dossier react/memory

Affichage à la fin de l'installation :

Success! Created memory at C:\Users\bertrandliaudet\Desktop\react\memory\

Inside that directory, you can run several commands:

⇒ npm start // Starts the development server.

⇒ npm run build // Bundles the app into static files for production.

⇒ npm test // Starts the test runner.

⇒ npm run eject // Removes this tool and copies build dependencies, configuration files and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

⇒ cd memory

⇒ npm start

Sur Mac

Faire npm start et pas yarn start (on évite yarn).

Architecture du framework : dossiers et fichiers installés

Organisation des dossiers et des fichiers du framework

3 fichiers et 2 dossiers :

```
├─ README.md
├─ package.json
├─ public
│   ├── favicon.ico
│   ├── index.html
│   └─ manifest.json
├─ src
│   ├── App.css
│   ├── App.js
│   ├── App.test.js
│   ├── index.css
│   ├── index.js
│   ├── logo.svg
│   └─ registerServiceWorker.js
└─ yarn.lock
```

Fichier README.md

Donne des informations de tutoriel.

Fichier package.json

Décrit la configuration de notre application.

Fichier yarn.lock

Fichier technique à ne pas modifier.

Dossier public

Fichier index.html : c'est une page de support qui affiche une page vide pour le moment. C'est un fichier obligatoire pour CRA.

Dossier src

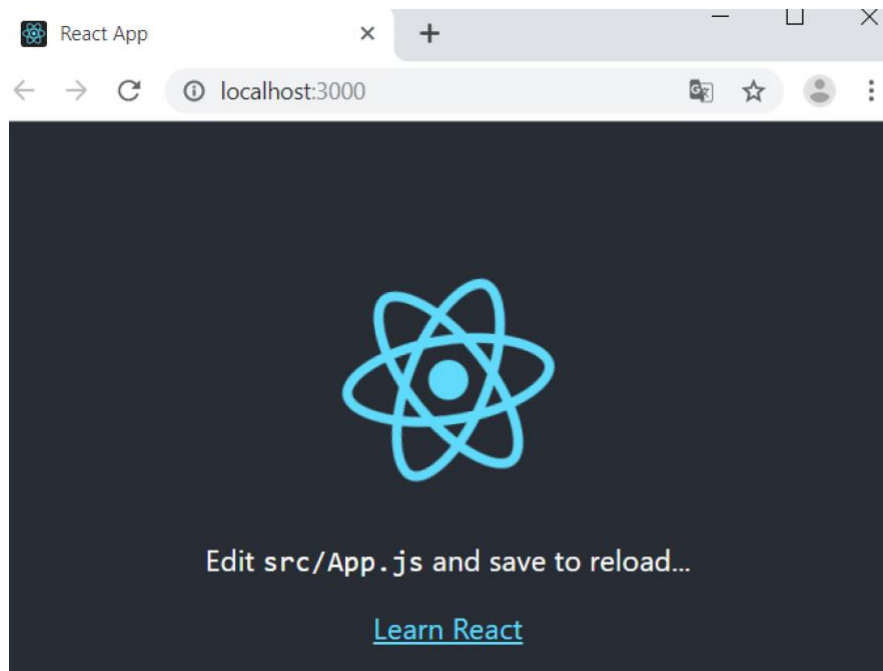
Fichier index.js : c'est **l'entrée dans l'application**. Cette entrée inclut le premier composant qui est App.js. C'est un fichier obligatoire pour CRA.

Fichiers App.js, App.css, App.test.js : ce sont les 3 fichiers du premier composant.

Démarrage de l'application

On va faire la suggestion proposée à la fin de la création du squelette de l'application memory :

```
C:> cd memory  
C:> npm start
```



Le fichier de l'application se trouve dans : src/App.js

On peut modifier un peu le texte : ici, on a ajouté ... à la fin de reload.

A noter que Firefox peut avoir des difficultés à charger l'image.

Le port utilisé est 3000

Exercices : installation d'un l'environnement Reac

Suivez le poly et installer React.

Faites tourner la première application : « memory ».

Mettez la couleur de fond en rouge.

Expliquez la rotation de l'image.

Rajoutez un « Bonjour votre nom » en <h1> sous la ligne « save and reload » et avant le <a href>

3 – JAVASCRIPT ES6 / ES2015

Bibliographie JavaScript

Réintroduction à JavaScript

https://developer.mozilla.org/fr/docs/Web/JavaScript/Une_réintroduction_à_JavaScript

ES6/2015

Tuto : <http://ccoentraets.github.io/es6-tutorial/>

let et const : <https://gist.github.com/gaearon/683e676101005de0add59e8bb345340c>

OCR : <https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664806-modernisez-votre-javascript-avec-es2015>

Prérequis JS

https://developer.mozilla.org/fr/docs/Web/JavaScript/Une_réintroduction_à_JavaScript

Remarque

Le prérequis concerne l'algorithmique et la partie objet. Il ne concerne pas la partie événementielle.

Tester JavaScript en ligne

<https://codepen.io/wlabarron/pen/yYrPRQ?editors=0011>

Ré-introduction

Pourquoi une réintroduction ? Parce que JavaScript : [le langage de programmation le plus incompris au monde](#).

Souvent raillé comme étant un simple jouet mais **langage très puissant**.

Nombreuses applications JavaScript de premier plan : <https://www.draw.io>

Une connaissance approfondie de cette technologie est une **compétence importante pour les développeurs Web**.

Créé en **1995** par Brendan Eich, un ingénieur de **Netscape**.

Rapidement soumis à l'[Ecma International](#), organisation de normalisation européenne => première édition du **standard ECMAScript en 1997** (ES1 = ES1997).

ES6=ES2015 : sixième édition qui apporte des nouveautés majeures, publié en juin 2015.

Conçu pour s'exécuter comme un langage de script dans un environnement hôte : c'est à cet environnement de fournir des mécanismes de communication avec le monde extérieur.

L'environnement hôte le plus commun est un navigateur, mais il en existe bien d'autres.

D'autres interpréteurs JS existent : dans Adobe Acrobat, Photoshop, les images SVG, le moteur de widgets de Yahoo!, des environnements côté serveur tels que [Node.js](#), les bases de données NoSQL telles que [Apache CouchDB](#), les ordinateurs embarqués ou encore des environnements de bureaux comme [GNOME](#) (interface graphique très populaire des systèmes d'exploitation GNU/Linux).

Principes

JavaScript est un **langage dynamique multi-paradigmes : procédural, objet, événementiel**.

Cf : <http://bliaudet.free.fr/IMG/pdf/Introduction-a-la-POO-Premiers-diagrammes-de-classes-UML.pdf>

Il dispose de différents types, opérateurs, objets natifs et méthodes.

Sa syntaxe s'inspire des langages Java et C.

JavaScript n'a pas de classes. La fonctionnalité des classes est reprise par les prototypes d'objet et le « sucre syntaxique pour les Classes » apparu avec ES6/ES2015.

Spécificité du JavaScript : les fonctions sont des objets. On peut donc stocker ces fonctions dans des variables et les transmettre comme n'importe quel objet.

Les types

[Number](#)

[String](#)

[Boolean](#)

[Object](#)

- [Function](#)
- [Array](#)
- [Date](#)
- [RegExp](#)

[null](#) : absence de valeur explicitement donnée

[undefined](#) : pas de valeur, ni de valeur null.

[Symbol](#)

Enfin, il y a également quelques types natifs pour gérer les exceptions : [Error](#).

Objets globaux :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux

[typeof](#) : renvoie le type de la variable.

Les nombres

Format IEEE 754 en double précision 64 bits

Attention aux calculs mathématiques !

Math.sin(), Math.PI, parseInt(), etc.

NaN, Infinity, isFinite

Les chaînes de caractères

"bonjour".length; // 7

"bonjour".charAt(0); // "b"

"coucou tout le monde".replace("coucou", "bonjour"); // "bonjour tout le monde"

"bonjour".toUpperCase(); // "BONJOUR"

Toutes les méthodes de manipulations de chaînes : [méthodes](#)

Booléen

true ou false.

Boolean(expression) convertit une expression en booléen

false, 0, chaîne vide (""), NaN, null et undefined valent false. Tout le reste vaut true.

Boolean('coucou') est true

Symbole

Apparu avec ES6/2015.

```
const ANIMAL_DOG = Symbol();
```

```
const ANIMAL_CAT = Symbol();
```

un symbole est une valeur unique et non modifiable.

Un symbole peut donc s'utiliser comme clé sans risque de collision.

Exemples : <https://putaindecode.io/articles/es6-es2015-les-symboles/>

Déclaration des variables – var et portée avant le ES6

Comportement classique en JavaScript :

On créer une variable en lui donnant une valeur : la première fois, on parle de déclaration. On peut précéder la création du mot clé « var » ou pas.

Toutes les variables créées **sans le mot clé « var »** sont des **variables globales** : elles sont utilisables **partout dans le code après leur déclaration, même dans les fonctions**.

Toutes les variables déclarées **avec un « var » dans les fonctions** sont **locales aux fonctions** : elles ne sont pas utilisables en dehors des fonctions.

```
ga='ga';
var gb='gb'
gc='gc';
f();
gd='gd';
console.log(fa)
console.log(fb)    // fb n'existe pas

function f(){
  fa='fa'
  var fb='fb'    // fb est locale à la fonction
  console.log(ga)
  console.log(gb)
  console.log(gc)
  // console.log(gd) // gd est déclaré après l'appel à la fonction
}
```

Les opérateurs

```
x += 5;
```

```
x = x + 5;
```

```
x++ ;
```

```
"coucou" + " monde" // "coucou monde"
```

```
"3" + 4 + 5; // "345"
```

```
3 + 4 + "5"; // "75"
```

```
4 + ""; // "4" : l'ajout d'une chaîne vide convertit en une chaîne.
```

Les comparaisons en JavaScript se font à l'aide des opérateurs <, >, <= et >=

```
123 == "123"; // true, comparaison non typée par défaut
```

```
1 == true; // true
```

```
123 === "123"; //false, comparaison typée à 3 =
```

```
1 === true; // false
```

&& , || et logique de court-circuit.

Structures de contrôle

if, else, switch... case... break... default,

while, do... while

for (i=0 ;i<tab.length ; i++)

for (let value of tab)

for (let propriete in objet)

Les tableaux

```
var tab = ["chien", "chat", "poule"];  
a.length; // 3
```

Les objets

```
var obj = {};  
  
var obj = {  
  nom: "Carotte",  
  prix: 10,  
  details: {  
    couleur: "orange",  
    taille: 12  
  },  
  toString :function(){  
    return this.nom+" "+this.details.couleur+" taille "  
      + this.details.taille+" : prix = "+this.prix  
  }  
};  
console.log(obj.details.couleur)  
console.log(obj["details"]["taille"])  
console.log(obj.toString())
```


Les fonctions

possibilité de fonctions récursives : RAS

possibilité de fonctions internes : RAS

[Valeur par défaut](#) des arguments.

[Tableau des arguments](#).

Nombre indéfini d'arguments : « [paramètres du reste](#) ».

Méthodes « [apply](#) » et « [call](#) » : apply permet de passer les arguments dans un tableau.

Les closures (fermetures)

Exemple

```
function creerFonction(a, b) {  
  var nom = "Mozilla";  
  function afficheNom(b) {  
    console.log(b+' '+nom);  
  }  
  return afficheNom;  
}  
  
var maFonction = creerFonction("Mozilla");  
maFonction("coucou"); // coucou Mozilla  
maFonction("super");  // super Mozilla
```

Principes

On crée un objet fonction avec une première série de paramètres.

Cette objet peut être utilisé avec une deuxième série de paramètres.

C'est très étrange !

Ces fonctions se « souviennent » de l'environnement dans lequel elles ont été créées (on dit aussi que la fonction capture son « environnement »).

Précisions [ici](#).

POO en JavaScript classique

https://developer.mozilla.org/fr/docs/Web/JavaScript/Introduction_à_JavaScript_orienté_objet

Il y a plusieurs façon de faire de la POO en JavaScript classique.

Version 1

```
function Personne(prenom, nom) {  
  this.prenom = prenom;  
  this.nom = nom;  
  this.nomComplet = function() {  
    return this.prenom + ' ' + this.nom;  
  }  
  this.nomCompletInverse = function() {  
    return this.nom + ' ' + this.prenom;  
  }  
}  
var s = new Personne("Simon", "Willison");  
console.log(s.nomComplet())  
console.log(s.nomCompletInverse())
```

Défaut : chaque objet porte les fonctions. Elles ne sont pas partagées.

Version 2 : avec prototype

https://developer.mozilla.org/fr/docs/Web/JavaScript/Introduction_à_JavaScript_orienté_objet

```
function Personne(prenom, nom) {
  this.prenom = prenom;
  this.nom = nom;
}
Personne.prototype.nomComplet = function() {
  return this.prenom + ' ' + this.nom;
}
Personne.prototype.nomCompletInverse = function nomCompletInverse()
{
  return this.nom + ' ' + this.prenom;
}
var s = new Personne("Simon", "Willison");
console.log(s.nomComplet())           // Simon Willison
console.log(s.nomCompletInverse())    // Willison Simon
```

Le prototype permet d'ajouter la fonction et qu'elle soit partagée.

ajout d'une fonction avec prototype en cours de code

```
Personne.prototype.nomEnMajuscules = function() {
  return this.nom.toUpperCase()
}
console.log(s.nomEnMajuscules()); // WILLISON
```

ajout d'une fonction avec prototype sur des classes natives

```
var s = "Simon"; // s est une String
String.prototype.inverse = function() {
  var r = "";
  for (var i = this.length - 1; i >= 0; i--) {
    r += this[i];
  }
  return r;
}
console.log(s.inverse()); // "nomiS"
```

On pourrait aussi redéfinir la méthode « toString » qui est déjà présente sur les chaînes de caractères.

Array.prototype – méthodes filter, every, some

Array utilise la technique du prototype :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/prototype

Associé au prototype on trouve de nombreuses méthodes qu'on peut redéfinir, comme par exemple filter(), every() et some()

filter() :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/filter

every() :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/every

some() :

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/some

Version 3 : avec la fonction static [Object.create](#)

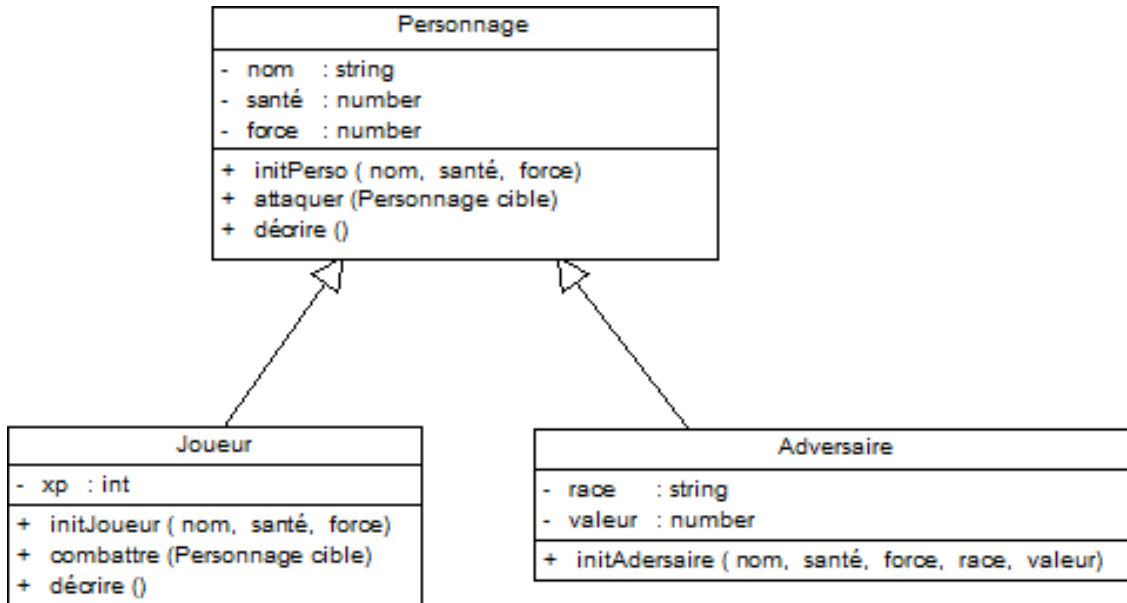
➤ **Classe : création d'une variable de type structure qui sert de prototype**

```
var Personnage = {  
  init: function (nom, sante, force) {  
    this.nom = nom;  
    this.sante = sante;  
    this.force = force;  
    this.xp = 0;  
  },  
  
  toString: function () { // retourne la description à afficher  
    return this.nom + " a " + this.sante +  
      "points de vie, " + this.force +  
      " en force et " + this.xp + " points d'expérience";  
  }  
};
```

➤ **Objet : création d'un objet et utilisation de l'objet : fonction static [Object.create](#)**

```
var perso1 = Object.create(Personnage);  
perso1.init("Aurora", 150, 25);  
console.log(perso1.toString());
```

➤ **Héritage :**



// On crée le prototype Personnage

```
var Personnage = {
  initPerso: function (nom, sante, force) {
    this.nom = nom;
    this.sante = sante;
    this.force = force;
  }
};
```

//on crée le prototype Joueur

```
var Joueur = Object.create(Personnage); // Prototype du Joueur
```

// on ajoute la fonction initJoueur au prototype

```
Joueur.initJoueur = function (nom, sante, force) {
  this.initPerso(nom, sante, force); // appelle initPerso
  this.xp = 0; // init la partie joueur
};
```

// on ajoute la fonction décrire au prototype

```
Joueur.decrire = function () {
  var description = this.nom + " a " + this.sante + " points de vie, " + this.force + "
    en force et " + this.xp + " points d'expérience";
  return description;
};
```

//idem avec Adversaire

Classe et POO

JS moderne : ES6/2015 - Standard ECMAScript

<https://apprendre-a-coder.com/es6/>

ES6 = ES2015 = ES6/2015 : une révolution pour JavaScript.

ES6/2015 : sucre syntaxique pour les Classes. JavaScript n'a pas de classes. La fonctionnalité des classes est reprise par les prototypes d'objet et le « sucre syntaxique pour les Classes » apparu avec ES6.

Pour les prototypes, voir le chapitre précédent.

Pour une introduction à l'objet, voir : http://biliaudet.free.fr/article.php3?id_article=108 : on trouve un pdf et des exemples JavaScript ES6, Python et Java.

L'essentiel de ES6/2015

<https://gist.github.com/gaearon/683e676101005de0add59e8bb345340c>

“let” et “const” : mieux contrôler ses variables. **Usage React** : on définit les variables avec des let et des const.

Classe : <https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes>

arrow function : fonction fléchée. On va utiliser ça dans les méthodes render() des composants react. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions

Principes

let : <https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/let>

Portée

let permet de déclarer une variable dont la portée est celle du bloc courant

Utilité

Avoir des variables locales explicites.

Usage

On va déclarer toutes les variables en let : une variable est locale par défaut. On évite les variables globales.

Variables globales : rappels

D'une façon générale, il vaut mieux éviter les globales.

Les variables globales de niveau fichier (ou module) peuvent être déclarées avec un var ou sans mot clé.

Les variables globales qui apparaissent dans les fonctions sont déclarées sans mot clé. Elles peuvent être utiles pour conserver un état pour un retour dans la fonction à condition de ne pas les utiliser en dehors de la fonction.

Principes

const : <https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/const>

Pour un non objet : on ne peut plus le modifier. C'est une constante classique.

Pour un objet (donc les tableaux), on ne peut pas le redéfinir, mais on peut modifier son contenu ;

```
const obj={ ... }
```

```
const tab=[]
```

Portée

const permet de déclarer une variable dont la portée est celle du bloc courant : une constante déclarée dans une fonction n'est visible que dans la fonction.

Utilité

Quand un objet est déclaré en constante, on ne peut plus le modifier => on ne peut plus le transformer en entier, le passer à null, etc => c'est une protection syntaxique

Usage

On va déclarer tous les objets en constante !

OCR - Classe

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664806-modernisez-votre-javascript-avec-es2015#/id/r-4718441>

```
class Screen extends Component {  
  constructor (props) {  
    super(props)  
    this.state = { loginState: 'logged-out' }  
  }  
  
  render () { // render : means getting or fetching data.  
    // ...  
  }  
}
```

L'appel au constructeur parent « super(props) » est obligatoire quand il y a héritage « extends ».

Cet appel doit être fait avant tout usage du « this » qui est obligatoire.

Sinon, on aura des `SyntaxError` ou des `ReferenceError`.

Les méthodes s'écrivent directement « nom(paramètres) { ... } » et non plus « nom : fonction(paramètres) { ... } »

OCR - Fonction fléchée : fat arrow : =>

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664806-modernisez-votre-javascript-avec-es2015#/id/r-4718486>

De :

```
const people=[], adults = [], minors = []
people[0]={name:"adulte", age:30}; people[1]={name:"enfant",
age:10}

people.forEach(function (person) {
  if (person.age >= 18) {
    adults.push(person)
  } else {
    minors.push(person)
  }
})
console.log(adults)
console.log(minors)
```

À :

```
people.forEach((person) => {
  if (person.age >= 18) {
    adults.push(person)
  } else {
    minors.push(person)
  }
})
```

OCR - Fonction fléchée : fat arrow : =>

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664806-modernisez-votre-javascript-avec-es2015#/id/r-4718486>

De :

```
tab = people.map(function (person) {  
  return person.name  
})
```

À :

```
tab = people.map(person => person.name)
```

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664806-modernisez-votre-javascript-avec-es2015#/id/r-4718486>

map : la fonction map() retourne un tableau avec les valeurs passées en paramètres.

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Objets_globaux/Array/map

OCR – this et =>

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664806-modernisez-votre-javascript-avec-es2015#/id/r-4718486>

En **JS classique**, le **this** dans une fonction fait référence à **l'objet qui appelle la fonction**.

Dans une **fonction anonyme classique**, le **this** fait référence à **l'objet global (la fenêtre)**.

Dans les **fonctions fléchées**, le **this** fait référence à **l'objet le plus proche**.

```
const name = 'Extérieur'

const test = {
  name: 'Intérieur',
  testerThis () {
    let tab=[1]

    console.log("IN : testerThis")
    console.log(this)           // this c'est l'objet test
    console.log(this.name)

    tab.forEach(function(elt) { // 1 seul élément dans tab
      console.log("TAB : forEach")
      console.log(this)         // this c'est la fenêtre
      console.log(elt+':'+this.name) // name n'existe pas
    })

    tab.forEach((elt) => {       // 1 seul élément dans tab
      console.log("TAB => : forEach")
      console.log(this)         // this c'est l'objet test
      console.log(elt+':'+this.name)
    }, 0)
  }
}

test.testerThis ()
```

Le mode strict

mode laxiste JS, mode strict JS

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Strict_mode

Le mode strict rend explicite certaines erreurs silencieuses. Il améliore ainsi la qualité et la maintenabilité du code.

Dans l'exemple précédent, le cas du milieu :

```
tab.forEach(function(elt) {           // 1 seul élément dans tab
  console.log("TAB : forEach")
  console.log(this)                   // this c'est la fenêtre
  console.log(elt+':'+this.name)      // name n'existe pas
})
```

affiche

```
TAB : forEach
Window
1:
```

il n'y a pas de this.name : c'est une erreur silencieuse.

Si on ajoute au début du code :

```
'use strict';
```

On obtiendra une erreur :

```
TypeError: this is undefined
```

OCR – Destructuration

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664806-modernisez-votre-javascript-avec-es2015#/id/r-4718497>

La déstructuration nous permet de récupérer plus facilement plusieurs informations au sein d'un objet ou d'un tableau.

Sur un objet

A l'ancienne :

```
const firstName = this.props.firstName
const lastName = this.props.lastName
const onClick = this.props.onClick
```

Syntaxe ES6

```
const { firstName, lastName, onClick } = this.props
```

Exemple :

```
obj={props:{firstName:'bertrand', lastName:'liaudet'}}
const { firstName, lastName } = obj.props
console.log(firstName, lastName)
```

Sur un tableau

A l'ancienne sur un tableau

```
const names = fullName.split(' ')
const firstName = names[0]
const lastName = names[1]
```

Syntaxe ES6

```
const [firstName, lastName] = fullName.split(' ')
```

Exemple :

```
let fullName='Bertrand Liaudet'
const [firstName, lastName] = fullName.split(' ')
console.log(firstName, lastName)
```


Exemples : un composant compteur

On présente 4 versions d'un petit compteur :

- Version sans variable globale
- Version avec variable globale
- Version avec Classe-fonction et prototype
- Version avec Classe

Les versions avec classes permettent de construire un composant HTML avec son HTML, son CSS et son JavaScript.

Les versions composant distinguent 2 fichier JavaScript : le « main » et la classe « Compteur ».

TP : mise à jour du composant compteur

Dans la version avec classe, ajoutez des fonctionnalités pour arriver aux résultats suivants :

Gestion d'un composant compteur

Incrementer

Decrementer

Compteur = 5 - Increment = 1

set Cpt

set Inc

raz Cpt

raz tout

Incrementer

Decrementer

Compteur = 2 - Increment = 1

set Cpt

set Inc

raz Cpt

raz tout

4 – FONCTION PURE REACT

Rappels : création d'une application React

La procédure :

1) **Installation de node, du gestionnaire de module, du CRA**

```
C:> node --version           // version de node
C:> npm --version            // version npm
C:> npm install --global npm  // mise à jour du npm
C:> npm install --global create-react-app // installation du CRA
```

2) **Creation d'une application monApp**

```
C:> cd react
C:> create-react-app mon-app
```

3) **Contenu de l'application monApp**

```
|— README.md ..... // info tuto
|— package.json      // configuration de monApp
|— public
|   |— favicon.ico
|   |— index.html .....// fichier obligatoire,
|                               // page de support, vide pour le moment
|   |— manifest.json
|— src
|   |— App.css ..... // css du composant
|   |— App.js         // js du composant
|   |— App.test.js    // outil de test du composant
|   |— index.css
|   |— index.js ..... // entrée dans l'application
|   |— logo.svg
|   |— registerServiceWorker.js
|— yarn.lock .....// fichier technique à ne pas modifier
```

4) **Démarrer l'application**

```
C:> cd mon-app
C:> npm start
```

Le contenu :

src/index.js : ReactDOM.render()

- Le fichier src/index.js est le fichier d'entrée.
- Sa seule instruction est d'appel à **ReactDOM.render()** :

```
ReactDOM.render(<App />, document.getElementById('root'));
```

- Il fait référence à « **App** ».

src/app.js : la fonction App() { return du HTML } est une fonction Component

- App correspond au fichier src/app.js qui contient une fonction App() qui retourne du HTML.
- On a une balise de className= «App » qui contient tout le HTML.
- La fonction ne retourne qu'une balise.

```
function App() {  
  return (  
    <div className="App">  
      <header className="App-header">  
        ...  
      </header>  
    </div>  
  );  
}
```

- Dans l'exemple, il y a un <header>. On peut mettre ce qu'on veut, en plus ou en moins.
- La fonction importe des éléments et s'exporte elle-même. Le chapitre suivant aborde cette question.

OCR - API : module, export, import

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664806-modernisez-votre-javascript-avec-es2015#/id/r-4718505>

- Une application est découpée en modules. Un module est un fichier.
- Les déclarations faites dans un fichier sont locales au fichier.
- Pour rendre certaines parties accessibles, on les **exportent**. On exporte des variables, des fonctions et des classes.
- Pour utiliser ces éléments exportés, on fera un **import** from en précisant les fonctions et classes importées et le fichier concerné dans le from.

Exemple :

```
// Dans le module Compteur.js :  
export default class Compteur {  
  // ...  
}  
  
// Dans le module main, dans le même répertoire :  
import Compteur from './SuperComponent'
```

Attention, **ça ne fonctionne que dans un environnement client-serveur** (sous node.js).

Pour gérer plusieurs fichiers côté client, on fera plusieurs inclusion dans le fichier HTML.

Hello World : 1^{er} composant pur fonctionnel

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664821-ecrivez-des-fonctions-pures>

<https://fr.reactjs.org/docs/hello-world.html>

1ère fonction Component

Principe

React permet de créer des composants sous forme de fonction ou de classe.

La forme fonction est la plus simple et la plus répandue. On parle de « **composant pur fonctionnel** », CPF (pur = sans état) ou « **stateless fonctionnel composant** », SFC.

Pour les mêmes « **props** » en entrée, un CPF « **render** » un même élément du DOM.

function MyComponent

On crée une fonction MyComponent qui retourne un élément du DOM virtuel (l'équivalent d'un nœud du DOM réel).

Pour créer cet élément on utilise la méthode **React.createElement()**.

```
function MyComponent() {  
  return React.createElement('p', {}, 'Hello World !')  
}
```

React.createElement()

La méthode **React.createElement()** permet de créer des éléments du DOM virtuel. Elle retourne un composant.

La méthode a 3 paramètres :

1. le nom d'une balise ou d'un composant
2. les props (ici rien{ } : ce sont les attributs de la balise ou du composant)
3. le contenu

Les paramètres 2 et 3 sont facultatifs.

Utilisation de la fonction Component

ReactDOM.render()

Pour injecter le DOM virtuel dans le DOM effectif du navigateur, l'application (le programme) utilise la méthode **ReactDOM.render()** qui se trouve dans le fichier src/index.js

La méthode a 2 paramètres :

1. **un composant** : on le crée avec `React.createElement()`. On lui fournit la fonction composant créé précédemment.
2. **`document.getElementById('root')`** : c'est une constante dans le FM React. C'est la position du composant dans le fichier HTML.

```
ReactDOM.render(  
  React.createElement(MyComponent),  
  document.getElementById('root')  
)
```

MyComponent est une fonction qui retourne un composant, c'est-à-dire un élément du DOM virtuel (l'équivalent d'un nœud pour le DOM réel).

Tester le code : <https://codepen.io/topics/>

Pour tester le code : <https://codepen.io/topics/>

Choisir React, puis React Function Component Examples

On garde le HTML. On vide le CSS et le JS.

Dans le JS, copier la fonction `MyComponent()` et le `ReactDOM.render`

Rappels JS : passer une fonction sans paramètre en paramètre

On peut passer une fonction sans paramètres en paramètre d'une autre fonction :

```
function ff(a) { return a() + 100 }
```

« a » est une fonction : on la retrouve avec des parenthèses dans le code de ff

Pour utiliser ff, on écrit :

```
console.log( ff(test) ) // 110
```

« test » est une fonction : on la passe en paramètre sans parenthèses.

« test » c'est par exemple :

```
function test(){return 10}
```

On ne peut pas avoir de paramètres dans « test »

On peut mettre « test » dans une variable et passer cette variable en paramètre.

```
vf=test
console.log( vf() ) // 10
console.log( ff(vf) ) // 110
```

On ne peut pas mettre des parenthèses au passage de la fonction :

```
console.log( ff(test() ) ) // BUG !!!
```

Ca bugue car on appelle la fonction test() qui retourne 10.

C'est donc 10 qui est passé en paramètre : ça ne « matche » pas avec a().

Remarque : on ne peut pas passer une fonction avec des paramètres en paramètre d'une autre fonction. Si on veut faire ça, il faut passer par un objet qui contient la fonction à passer en paramètre : c'est alors de la classique programmation objet.

1^{ère} approche de JSX

1^{ère} fonction Component

Principe

- L'utilisation de **React.createElement()** est lourde puisqu'il faut faire un createElement par composant constituant la page, le constituant élémentaire étant la balise HTML.
- Pour compacter le code, on utilise le JSX.
- La syntaxe JSX est proche du HTML et du XML.

Codage du composant précédent en JSX :

- On retourne le code de la balise HTML

```
function MyComponent() {  
  return <p>Hello World !</p>  
}
```

Utilisation du composant en JSX :

- L'utilisation est aussi simplifiée : on peut passer le nom du composant en format XML en premier paramètre plutôt que l'appel à la méthode React.createElement()
- A noter que la balise XML est une balise orpheline avec un /> en fermeture.

```
ReactDOM.render(  
  <MyComponent />,  
  document.getElementById('root')  
)
```

- On retrouve la syntaxe de notre premier projet créé avec le CRA.

Tester le code : <https://codepen.io/topics/>

Pour tester le code : <https://codepen.io/topics/>

Choisir React, puis React Function Component Examples

On garde le HTML. On vide le CSS et le JS.

Dans le JS, copier la fonction MyComponent() et le ReactDOM.render

1^{ere} props

fonction Component avec props

Principe

- On veut passer des paramètres à une fonction Component.
- On va utiliser la syntaxe JSX.
- En paramètre de la fonction, on met la props : {firstName = 'Bertrand'}
- La valeur de la props, 'Bertrand', est une valeur par défaut.
- Dans le code XML, on accède à la props en écrivant : {firstName}

Codage avec JSX :

```
function MyComponent({firstName = 'Bertrand'}) {  
  return <p>Hello {firstName} !!!</p>  
}
```

Principe de l'utilisation du composant avec props en JSX :

- L'utilisation d'un composant consiste à le passer en balise.
- On peut ajouter des attributs à la balise : ces attributs correspondent aux paramètres de la fonction.

```
ReactDOM.render(  
  <MyComponent firstName = 'Rachid' />,  
  document.getElementById('root')  
)
```

Utilisation du composant avec props en JSX :

➤ *Version 1 : simple appel à MyComponent sans paramètre*

```
ReactDOM.render(  
  <MyComponent />,  
  document.getElementById('root')  
)
```

➤ *Version 2 : appel à MyComponent avec paramètre*

```
ReactDOM.render(  
  <MyComponent firstName = 'Rachid' />,  
  document.getElementById('root')  
)
```

➤ *Version 3 : appel à MyComponent avec un code plus complexe*

```
ReactDOM.render(  
  <div>  
    <h1>Test avec props</h1>  
    <MyComponent firstName = 'Yuwei' />  
    <h1>Test avec props par défaut</h1>  
    <MyComponent />  
  </div>,  
  document.getElementById('root')  
)
```

On a une seule balise en premier paramètre : une div. Ensuite on peut mettre du HTML et/ou des composants React.

On appelle 2 fois le composant : avec ses paramètres par défaut, ou avec des paramètres en entrée à l'appel du composant.

Tester le code : <https://codepen.io/topics/>

Pour tester le code : <https://codepen.io/topics/>

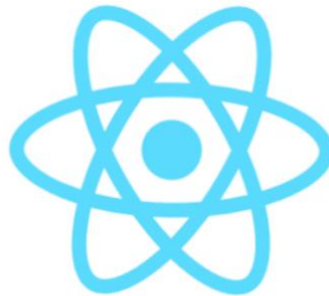
Choisir React, puis React Function Component Examples

On garde le HTML. On vide le CSS et le JS.

Dans le JS, copier la fonction MyComponent() et le ReactDOM.render

Tester le code dans une application React

- Créez une application React : mon-app
- Dans cette application, créez un composant MyComponent sur le même modèle que App.js et qui reprenne le code de MyComponent : Hello Bertrand (Bertrand par défaut dans les props).
- Dans MyComponent, mettez l'image de l'application créée par React.
- Gérez le css au minimum : selecteur .MyComponent, .MyComponent-logo, @media et @keyframes.
- L'objectif est d'avoir le résultat suivant : avec l'image qui tourne.
- A noter que le prénom est passer en paramètre dans l'index.js.



Hello Nabila !!!

Ajouter une deuxième props

- Ajouter une props « lastName » au composant, sans valeur par défaut. Les props sont séparées par des virgules.
- Donner un lastName à l'utilisation du composant.

5 – JSX

Présentation

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664826-decrivez-un-composant-avec-jsx>

JSX vs React.createElement(...)

- On a vu qu'on peut utiliser les 2 syntaxes.
- La syntaxe JSX est plus pratique.
- Pour plus d'exemples de syntaxe React.createElement(...) :
<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664826-decrivez-un-composant-avec-jsx#/id/r-4871160>

Notion de grappe

- Une grappe est un DOM virtuel React créé avec la méthode React.createElement(...) ou directement par la fonction render en JSX. On peut parler de grappe JSX.
- Une grappe c'est un nœud parent qui contient des enfants sur plusieurs générations et qui correspond à une balise HTML ou à un composant contenant d'autres balises ou composants plus ou moins imbriqués
- Une grappe correspond à un composant.
- La grappe c'est la partie HTML-JSX qui se trouve dans le return du render().

Exemple de formulaire JSX

```
<form method="post" action="/sessions" onSubmit={this.handleSubmit}
>
  <p className="field">
    <label>E-mail</label>
    <input type="email" name="email" required autoFocus
      value={this.state.email}
      onChange={this.handleChange}
    />
  </p>
  <p className="field">
    <label> Mot de passe </label>
    <input type="password" name="password" required
      Maxlength={12}
      value={this.state.password}
      onChange={this.handleChange}
    />
  </p>
  <p>
    <input type="submit" value="Connexion" />
  </p>
</form>
```

- Le code n'est pas très différent du HTML.
- La connexion aux traitements se repère facilement avec les { }
- Le this.state correspondra à l'état de l'objet en cours.
- Exemple détaillé : <https://react-reform.codecks.io/docs/form/>

Syntaxe : un peu HTML, un peu XML, un peu JSX

- Sensible à la casse : MyComponent est différent de mycomponent
- Balise orpheline : `<MyComponent />`
- Les attributs s'appellent des « props »
- Deux types possibles pour les props : chaîne de caractère et tout le reste.
- Les chaînes de caractères sont entre " ", le reste entre { }
- Comme en HTML, `required={true}` peut s'écrire simplement : `required`
- On ne peut pas utiliser les mots-clés du JavaScript comme nom de props. On écrit donc `className` au lieu de `class` et `htmlFor` au lieu de `for`.
- Les commentaires s'écrivent : `{/* */}`

6 – COMPOSANT FONCTION, VARIABLE, CLASSE

TP Sublime text pour React

Option 1 : Babel

- On a vu que la coloration syntaxique ne suit plus.
- Dans sublime text : ctrl-shif-p : tapez install : on arrive sur install package control. On tape entrée. Un message nous dit que ça a été installé.
- Arrêter et relancer sublime text.
- Dans sublime text : ctrl-shif-p : taper install : choisir install Package et taper entrée.
- Il y a beaucoup de choix.
- Taper Babel : il s'installe très vite (on a un petit message en bas de fenêtre).
- En bas à droite, on clique sur « javascript » : dans le menu on choisit Babel /JavaScript Babel
- C'est fini ! On a une coloration syntaxique adaptée. Mais pas plus.

Option 2 : TypeScript

- On va désinstaller ce qu'on a fait.
- Dans sublime text : ctrl-shif-p : tapez remove : choisir Package. Taper entrée. Le texte perd toute coloration.
- En bas à droite, on est en « plain-text ». On peut choisir JavaScript. Babel a disparu.
- On est revenu à la case départ.
- On refait l'installation mais pas pour Babel, mais pour TypeScript
- Dans sublime text : ctrl-shif-p : taper install : choisir install Package et taper entrée.
- Il y a beaucoup de choix.
- Taper TypeScript : il s'installe pas très vite (on a un petit message en bas de fenêtre).
- En bas à droite, on clique sur « javascript » : dans le menu on choisit TypeScript /TypeScriptReact
- C'est fini ! On a une coloration syntaxique adaptée.
- Mais en plus, il propose de la complétion automatique.
- clas + tab : structure complète de la classe.
- Il propose aussi du formatage automatique : ctrl+T ctrl F : c'est plus ou moins efficace

Remarque sur TypeScript

TypeScript est un langage open source développé par Microsoft pour sécuriser le JavaScript. Donc tout code JavaScript est un code TypeScript. Tout code ES6 est aussi un code TypeScript. Le TypeScript est concurrencé par l'ES6.

Composant dans une variable

- A la place d'une fonction :

```
function MonComposant({firstName = 'Bertrand'}) {  
  return (  
    <div className="MonComposant">  
      <img src={logo} className="MonComposant-logo" alt="logo" />  
      <p> Hello {firstName} !!!</p>  
    </div>  
  );  
}
```

- On peut avoir une variable qui contient une fonction :

```
const MonComposant = function ({firstName }) {  
  return (  
    <div className="MonComposant">  
      <img src={logo} className="MonComposant-logo" alt="logo" />  
      <p> Hello {firstName} !!!</p>  
    </div>  
  );  
}
```

- On peut écrire la fonction avec des => :

```
const MonComposant = ({firstName }) => (  
  <div className="MonComposant">  
    <img src={logo} className="MonComposant-logo" alt="logo" />  
    <p>coucou</p>  
    <p> Hello {firstName} !!!</p>  
  </div>  
)
```

- Ici, on a supprimé les mots clés « function » et « return » ainsi que les { }. On ajoute la =>

- L'utilisation de la variable se fait de la même manière qu'avec une fonction :

```
function App() {  
  return (  
    <div className="App">  
      <MonComposant firstName='Nabila' />  
    </div>  
  );  
}
```

Composant dans une classe

- A la place d'une fonction :

```
function MonComposant ({firstName = 'Bertrand'}) {  
  return (  
    <div className="MonComposant">  
      <img src={logo} className="MonComposant-logo" alt="logo" />  
      <p> Hello {firstName} !!!</p>  
    </div>  
  );  
}
```

- On peut avoir une classe qui « render » notre grappe :

```
export default class MonComposant {  
  constructor(firstName) {  
    this.firstName = firstName  
  }  
  render() {  
    return (  
      <div className="MonComposant">  
        <img src={logo} className="MonComposant-logo" alt="logo" />  
        <p>coucou classe</p>  
        <p> Hello {this.firstName} !!!</p>  
      </div>  
    );  
  }  
}
```

- Ici, on passe les props dans le constructeur.
- La méthode render() sert pour le retour de la grappe.
- A l'instanciation d'un objet, le constructeur est appelé et aussi le render() qui retourne sa grappe : la classe se comporte comme une fonction.
- On exporte la classe à sa définition pour pouvoir l'importer.

- L'utilisation de la classe se fait par un new dans une const. Ensuite, on accède à la méthode directement dans le JSX.

```
function App() {  
  const monComposant = new MonComposant('Nabila')  
  return (  
    <div className="App">  
      {monComposant.render()}  
    </div>  
  );  
}
```

TP composant dans une variable ou dans une classe

- Tester les codes proposées ci-dessus.

TP memory

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664826-decrivez-un-composant-avec-jsx#/id/r-4871320>

- Dans le répertoire de travail « react », créer une application react appelée « memory » :

```
C:\chezMoi\react>create-react-app memory
```

- Dans le répertoire « **src** » de l'application memory, supprimez : **App.js**, **App.css**, **App.test.js**, **index.css**, et **logo.svg**.
- Télécharger « memory-elt-1.zip » sur le site du cours.
- Mettez tous les fichiers du zip dans le répertoire « **src** » : **App.js**, **App.css**, **Card.js** , **Card.css**, **GuessCount.js**. **GuessCount.css**.
- Démarrer l'application.
- Regarder chaque fichier 1 par 1, dans l'ordre pour comprendre le fonctionnement.
 - index.js
 - App.js et App.css
 - Card.js et Card.css
 - GuessCount.js et GuessCount.css
- Lisez le code et les commentaires pour bien tout comprendre.
- Le principe est qu'on veut fabriquer la page suivante :



- Pour ça, on crée un composant App qui va contenir les éléments à afficher. Les textes sont de simples balises <p>. Le 0 correspond au nombre de coup joués. Ce sera un nouveau composant. Chaque image correspond à une carte. Elle correspond à un nouveau composant.
- Il y a plusieurs type d'affichage pour une carte : le ? veut dire qu'elle est face cachée. Le contour gris veut dire qu'elle n'est plus jouable. Le contour vert voudra dire qu'on a gagné. Le contour rouge voudra dire qu'on perdu.
- On fait un premier affichage avec tous les cas possibles, pour voir.

7 – LES EVENEMENTS

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664831-reagissez-aux-evenements>

React permet de déclarer nos gestionnaires d'événements dans le code JSX de notre composant.

Exemple : <http://bliaudet.free.fr/IMG/zip/memory-base-evt.zip>

8 – TESTS EN JSX

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664836-contextualisez-le-contenu-de-vos-composants>

Présentation - opérateur ternaire ?:

- **JSX** produit une **expression JavaScript** et non une instruction.
- On ne peut pas utiliser le « if ».
- On peut utiliser l'opérateur ternaire : `?:`
- Exemple ?:

```
<p>{1>0 ? <a href="/admin">ok</a>:pas ok}</p>
```

true, false, null et undefined ignorés par JSX

- JSX ignore les valeurs true, false, null et undefined. Autrement dit, il fait comme si elle n'était pas présentes.
- Donc :
 - `{true && <p>coucou true</p>}` vaut `{<p>coucou true</p>}`
 - `{false && <p>coucou false</p>}` vaut `{}` car c'est faux donc ignoré.

Tester avec l'opérateur &&

- Du fait de la propriété précédente, une expression de la forme :

```
expression_1 && expression_2
```

- est équivalente à

```
expression_1 ? expression_2 : null
```

- Exemple de && avec une variable booléenne : myBool

```
<p>{myBool && <a href="/admin">ok</a>}</p>
```

- L'expression "`myBool && ok`" est évaluée.
- Si myBool est faux, ça vaut faux et donc c'est ignoré.
- Si myBool est vrai, myBool est ignoré et le résultat est `<a ...>ok`

Application dans le memory

- On ajoute dans le composant App, en bas, l'affichage « gagné » si on a gagné :

```
{won && <p>GAGNÉ !</p>}
```

- On simule au début du composant, dans le render(), avant le return, le calcul de won (vrai si les secondes sont divisibles par 3)

```
const won = new Date().getSeconds() % 3 === 0
```


9 – BOUCLE EN JSX

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664841-manipulez-des-listes-de-composants>

Utilisation de la fonction map : -> [doc](#)

- **JSX** produit une **expression JavaScript** et non une instruction.
- On ne peut pas utiliser le « for ».
- Le but est par exemple de transformer une liste de données en une liste de composants.
- La **méthode « map »** du JavaScript permet de faire ça : -> [doc](#)
- On peut associer d'autres méthode pratiques pour faire des calculs : **filter** (pour filtrer les éléments), **every** (pour tester tous les éléments), **some** (pour tester si au moins un élément répond au critère proposé), etc. -> [doc](#)

Sur un tableau d'entiers :

```
const numbers = [1, 2, 3, 4]
const doubles = numbers.map(x => x * 2) // [2, 4, 6, 8]
console.log(doubles)
```

Tester le code : <https://codepen.io/topics/>

Pour tester le code : <https://codepen.io/topics/>

Choisir React, puis React Function Component Examples

On garde le HTML. On vide le CSS et le JS.

Dans le JS, copier le code

Sur un tableau d'objets

- Soit un tableau d'objets :

```
const users = [  
  { id: 1, name: 'Alice' },  
  { id: 2, name: 'Bob' },  
  { id: 3, name: 'Claire' },  
  { id: 4, name: 'David' },  
]
```

- Pour produire une liste de liens avec les noms des utilisateurs, on écrirait ceci, dans une classe, avec `this.props` :

```
render () {  
  return (  
    <div className="userList">  
      {this.props.users.map((user) => (  
        <a href={` /users/${user.id}`} > {user.name} </a>  
      ))}  
    </div>  
  )  
}
```

Tester le code : <https://codepen.io/topics/>

- Pour tester le code : <https://codepen.io/topics/>
- Choisir React, puis React Function Component Examples
- On garde le HTML. On vide le CSS et le JS.
- On teste avec une simple fonction. Dans le JS, copier le code :

```
function MyComponent() {  
  const users = [  
    { id: 1, name: 'Alice' },  
    { id: 2, name: 'Bob' },  
    { id: 3, name: 'Claire' },  
    { id: 4, name: 'David' },  
  ]  
  return (  
    <div className="userList">  
      {users.map((user) => (  
        <li><a href={` /users/${user.id}`} >{user.name}</a></li>  
      ))}  
    </div>  
  );  
}  
ReactDOM.render(  
  <MyComponent />,  
  document.getElementById('root')  
)
```

En déstructurant :

```
return (
  <div className="userList">
    {users.map(({id, name}) => (
      <li><a href={` /users/${id}`}>{name}</a></li>
    ))}
  </div>
)
```

Tester le code : <https://codepen.io/topics/>

La prop key

- React introduit une prop « technique » appelé « key »
- Cette prop sert à identifier les éléments d'un tableau au sein d'une grappe.
- Ici, c'est l'id de nos users qui servira d'identifiant.
- Il faut ajouter cette key pour que le tableau soit bien géré par React.

```
return (
  <div className="userList">
    {users.map(({id, name}) => (
      <li><a href={` /users/${id}`} key={id}>{name}</a></li>
    ))}
  </div>
)
```

Tester le code : <https://codepen.io/topics/>

Ajouter un filter : `users.filter(user=>user.id>1).map(...)`

Téléchargements

- Téléchargez J6-React : il contient les J5-memory-*, HallOfFame.*, symbols.js

Installation de l'environnement de travail

- Créez un dossier J6
- Dans J6 créez un dossier AppJ6
- Dans J6 mettez le dossier node_module le plus récent
- Dans J6 AppJ6, mettez les J5-memory-*
- Dans J6, copiez-collez le contenu de J5-memory-base-evt-plus
- Dans J6/src, mettez les fichiers HallOfFame.*

Installation

- Ouvrez une console. Mettez-vous dans J6.
- Faites : `npm install --save lodash.shuffle`
- Éventuellement, selon les messages reçus, faites : `npm audit fix`
- Refaites : `npm install --save lodash.shuffle` : ça réduit les vulnérabilités.
- Démarrez l'application : `npm start`

Démarrage de l'application

- Une console est ouverte. Vous êtes dans J6 : `npm start`.

Mise à jour de l'application : App.js

- juste après import React from 'react';

```
import shuffle from 'lodash.shuffle'
```

- en dernier import

```
import HallOfFame, { HALL_OF_FAME } from './HallOfFame'
```

• TESTER

- après les import (récupérer ces lignes dans symbols.js)

```
const SIDE = 6
const SYMBOLS = '😬🍷❤️📺🐼🐱🐰🌐🌙🌟🌀🍎🍌🍓🍏🍷🍷'
// 18 SYMBOLS * 2 = 36 : jeu de 6 x 6
```

- Au début de la classe App :

```
cards = this.generateCards()
generateCards() {
  const result = []
  const size = SIDE * SIDE
  const candidates = shuffle(SYMBOLS) // on mélange tout
  while (result.length < size) {
    const card = candidates.pop()
    result.push(card, card) // on met 2 fois la carte
  }
  return shuffle(result) // on mélange tout
```

```
}
```

- **TESTER**

- A la place des 6 cards, on met (entre GuessCount et « jouons au memory ») :

```
{this.cards.map((card, index) => (  
  <Card  
    card={card}  
    feedback="visible" // visible pour le moment, ou hidden  
    key={index}        // la key est l'indice dans le tableau  
    onClick={this.handleCardClick}  
  />  
))}
```

- **TESTER**

- La mise en page est moche ! On veut un tableau de 6 * 6 : il suffit de passer la valeur à 350 dans App.css

```
width: 350px;                /* largeur de l'affichage */
```

- **TESTER**

- On passe le « GAGNE » à la ligne en le centrant :

```
{won && <p className ="won">GAGNÉ !</p> */}
```

- On ajoute dans App.css :

```
.memory > .won {                /* > : enfant direct */  
  color : red;  
  width: 100%;                  /* pour s'étaler sur 100% et passer à  
  la ligne */  
  text-align: center;  
}
```

- **TESTER**

- Dans App.js : passer le feedback du composant Card à « hidden »

- **TESTER**

- On constate que la mise en page ne fonctionne plus.
- Il faut s'assurer que les cards prennent bien la bonne place sur une ligne.
- On met à jour la propriété flex de .card dans Card.css. Dans .memory > .card, remplacez le flex :

```
.memory > .card {                /* > : enfant direct */  
  flex: 1 1 calc(100% / 6 - 0.4em);  
      /* flex détaillé :grow shrink basis */  
      /* grow=1 : prend toute la place */  
      /* shrink=1 : peut se rétrécir */  
      /* 1 1 basis=100%/6-marge : s'adapte à cette taille */
```

- **TESTER**

- Reste à gérer le tableau d'honneur : Hall Of Fame. Ajoutez, dans App.js, juste avant la fermeture de la <div> du render :

```
{won && <HallOfFame entries={HALL_OF_FAME} />}
```

- **TESTER** plusieurs fois pour avoir l’affichage
- Regardez le code de gestion du HallOfFrame

10 - DIVERSES PROPRIETES DES COMPOSANTS

Children dans le DOM JavaScript

- En JavaScript, tout nœud du DOM a un parent et un seul. Un nœud peut avoir plusieurs enfants qui l'ont tous comme parent direct.

Composant parent – Composant enfant

- En React, un composant enfant est un composant « renderé », quel que soit le niveau (enfant, petit-enfant, arrière petit-enfant, etc.).
- Le composant qui fournit le render() est le parent.
- Tous les composants figurants dans le render() sont des enfants.

Règle de passage des props

- Une prop est passée du parent à l'enfant
- Une prop est uniquement passée en lecture (elle n'est pas modifiée par l'enfant).
- Rappel du principe React : « les données descendent, l'état remonte ». On reviendra sur la notion d'état.

Les props techniques

key

- La key permet d'identifier chaque élément d'un tableau. Ça permet à React de gérer proprement les modifications d'un tableau d'un render à l'autre.

children

- c'est un tableau rempli automatiquement par les enfants du composant parent.

dangerouslySetInnerHTML

- permet de fournir du HTML « à la papa » !
- Attention, il faut fournir un objet ayant un attribut __html (deux tirets bas)

```
balise.dangerouslySetInnerHTML={ __html: 'First &middot; Second' }
```

Valeurs par défaut : propriété statique defaultProps

Propriété statique

Comme en POO classique, une propriété statique vaut pour la classe et est donc partagée pour tous les objets.

defaultProps

- C'est une propriété statique : quand on la définit, la valeur sera la même chaque instance.
 - Comme en POO classique, une propriété classique vaut pour la classe et est donc partagée pour tous les objets.
 - On rajoute des defaultProps après la définition de la fonction.
 - Les defaultProps sont utilisables dans la fonction
-
- A noter : les paramètres de la fonction sont en JSX : avec des { } : {myTest="coucou"}
 - Dans la fonction, la partie JavaScript est en JavaScript : myTest sans accolades
 - Dans la fonction, la partie HTML écrit son JavaScript en JSX : avec des accolade { }

```
function MyComponent({myTest}) {
  console.log("myTest :", myTest)
  const nameId1 = 'Alice'+myTest // pour montrer l'usage
  const users = [
    { id: 1, name: nameId1},
    { id: 2, name: 'Bob'},
    { id: 3, name: 'Claire'},
  ]

  return (
    <div className="userList">
      {users.filter(user=>user.id>1).map(({ id, name}) => (
        <li><a href={`/users/${id}`} key={id}>
          {name}-{myTest}
        </a></li>
      ))}
      <p>{users[0].id} - {users[0].name} - {myTest}</p>
    </div>
  );
}

MyComponent.defaultProps = {
  myTest: "valDef",
}

ReactDOM.render(
  <MyComponent/>,
  document.getElementById('root')
)
```


Valeur par défaut sans defaultProps : à éviter

```
function MyComponent({myTest="valDef"}) {
  const users = [
    { id: 1, name: 'Alice'},
    { id: 2, name: 'Bob'},
    { id: 3, name: 'Claire'},
  ]

  return (
    <div className="userList">
      {
        users.map(({ id, name, test}) => (
          <li><a href={` /users/${id}`} key={id}>
            {name}-{myTest}
          </a></li>
        ))
      }
    </div>
  );
}

ReactDOM.render(
  <MyComponent />,
  document.getElementById('root')
)
```

Définition des types : propriété statique propTypes

Présentation

- propTypes une propriété statique qui permet de définir des **contraintes de type** sur les props et aussi de préciser si la **valeur est obligatoire ou pas**.
- Ainsi React permet de rendre le **JavaScript typé** ! Comme du Java ! L'intérêt est que la compilation vérifie la cohérence de ce que l'on écrit.
- Les props sont les API de nos composants : quand on utilise un composant, on passe des props en paramètre. Si le code peut vérifier que l'usage qu'on fait des composants est bon, c'est mieux. C'est une **logique de programmation objet**.
- Les vérifications ne seront toutefois mises en œuvre que pendant le développement. En phase de production, elles sont « élaguées » automatiquement.

Exemple de syntaxe

```
MonComposant.propTypes = {
  max: PropTypes.number,
  guesses: PropTypes.number.isRequired,
  card: PropTypes.string.isRequired,
  feedback: PropTypes.oneOf([
    'hidden',
    'justMatched',
    'justMismatched',
    'visible',
  ]).isRequired,
  onClick: PropTypes.func.isRequired,
  entries: PropTypes.arrayOf(
    PropTypes.shape({
      date: PropTypes.string.isRequired,
      guesses: PropTypes.number.isRequired,
      id: PropTypes.number.isRequired,
      player: PropTypes.string.isRequired,
    })
  ).isRequired,
}
```

Installation

```
npm install --save prop-types
npm audit fix
npm install --save prop-types
```

Ajoutez des contraintes de type

A partir de l'exemple donnée, ajoutez les contraintes de type au TP-J6-1

- Dans Card.js

```
import PropTypes from 'prop-types'
// définition de la fonction
Card.propTypes = {
  // mettre les attributs : cf exemples du cours
}
```

- Mettre « invisible » comme valeur pour feedback dans App.js et constatez le warning.

- Dans GuessCount.js

```
import PropTypes from 'prop-types'
// définition de la fonction
GuessCount.propTypes = {
  // mettre les attributs : cf exemples du cours
}
```

- Mettre « ok » comme valeur pour guesses dans App.js et constatez le warning.

- Dans HallOfFame.js

```
import PropTypes from 'prop-types'
// définition de la fonction
HallOfFame.propTypes = {
  // mettre les attributs : cf exemples du cours : entries
}
```

- Où faut-il faire une modification pour constater la prise en compte du type ?

Ajoutez des valeurs par défaut

- Mettez la valeur de la prop guesses passée en paramètre au composant GuessCount en valeur par défaut.
- Dans GuessCount.js, après le propTypes :

```
GuessCount.defaultProps={
  guesses:1,
}
```

- Dans App.js, après le propTypes, on supprime la valeur par défaut à la création de GuessCount :

```
<GuessCount/>
```

11 - COMPOSANT CLASSE

Présentation

- React encourage à définir le plus de composants possibles sous forme de fonction, les Stateless Function Component, SFC, composant sans état (sans état local) : ils facilitent l'optimisation automatique.
- Toutefois 3 usages justifient l'utilisation de classe :
 - pour ajouter des **méthodes métier**
 - pour ajouter des **méthodes de cycle de vie** de React,
 - pour conserver, faire circuler et modifier de l'information d'un d'un render à l'autre au sein du composant, il faut définir un **état local au composant**.
- **3 sortes de méthodes :**
 - Gestionnaires d'événement : `handleCardClick`. On les appelle « `handleEvenement` », ici « `handleClick` » en précisant éventuellement le nom du composant = « `handleCardClick` ».
 - Méthodes de calcul de données ou de récupération des données : `generateCard`
 - Méthodes du cycle de vie, par exemple `ComponentWillReceiveProps()` (cf. §12).

Une classe ressemble à :

```
class CoolComponent extends Component {
  static defaultProps = {
    initialCollapsed: false,
  }
  static propTypes = {
    initialCollapsed: PropTypes.bool.isRequired,
    items: PropTypes.arrayOf(CoolItemPropType).isRequired,
  }
  constructor(props) {
    super(props)
    this.state = {
      collapsed: props.initialCollapsed
    }
  }
  render() {
    // ...
  }
}
```

- Le `super(props)` est obligatoire s'il y a `extends`.
- Le `this.state` correspond à l'état local. On y reviendra.
- La question du `this` est problématique. On y reviendra.
- Le `render`, c'est la fonction qui est appelée automatiquement à l'utilisation de la classe comme d'un composant. Dans ce cas, il y a instanciation d'un objet avec appel au constructeur qui utilise les `defaultProps`, et `render` qui retourne une grappe HTML pour le DOM virtuel.

Usage de cette classe comme d'une balise

```
<CoolComponent />
```

```
<CoolComponent initialCollapsed />
```

Ici le `initialCollapsed` est équivalent à `initialCollapsed=« true »`

Présentation

- En **JS classique**, le **this** dans une fonction fait référence à **l'objet qui appelle la fonction**.
- Dans une **fonction anonyme classique**, le **this** fait référence à **l'objet global (la fenêtre)**.
- Dans les **fonctions fléchées**, le **this** fait référence à **l'objet le plus proche**.
- Donc :
 - **les méthodes de calcul gèrent correctement le this.**
 - **Les fonctions d'événement n'ont pas de this en standard.**

Exemples

➤ *Exemple sur une méthode de calcul*

```
generateCards() {  
  console.log('this : '+this+ ' - classe: '+this.constructor.name)  
  ...  
}
```

- this : [object Object] - classe: App
- this.constructor.name permet de récupérer le nom de la classe

➤ *Exemple sur une fonction d'événement :*

```
handleCardClick(card, feedback) {  
  console.log(card+ ' '+feedback+ ' : clicked', ' this: ',this)  
}
```

- Et son utilisation dans le render() :

```
render() {  
  return (  
    <div className="memory">  
      ...  
      onClick={this.handleClick}  
      ...  
    </div>  
  )  
}
```

- Le this est undefined :
- 🌐 visible : clicked this: undefined
- Le problème vient du fait que la méthode est un « callback » appliqué à onClick

Solution 0 : rustine à éviter, fonction fléchée sur l'événement

```
onClick={this.handleClick}
```

- devient :

```
onClick={ (card, feedback) => this.handleClick(card, feedback) }
```

- On a une syntaxe de fonction fléchée pour le onClick. Ainsi le this se maintient
- La fonction fléchée **délègue** au handleClick
 - C'est une syntaxe sujet-verbe-complément : this-méthode-paramètre
 - Les paramètres de la fonction fléchée se retrouvent dans la fonction déléguée.

➤ *C'est à éviter pour des questions d'efficacité*

- Chaque render refabrique une méthode

Solution 1 : le binding

- Dans le constructeur, on « bind » (lier, attacher). Le bind permet que chaque instance du composant se dote de ses variantes. Ici la variante, c'est le this.

```
constructor(props) {  
  super(props)  
  this.handleClick = this.handleClick.bind(this)  
}
```

- Le défaut c'est que la définition de la méthode ne nous dit rien sur le fait que le this sera conservée.

Solution 2 : décorateur

- Il suffit d'ajouter un décorateur

```
@autobind  
handleClick(card, feedback) {  
  console.log(card+' '+feedback+' : clicked', ' this: ',this)  
}
```

- C'est le plus élégant, mais ce n'est pas encore standard dans Create React App.

Solution 3 : la fonction devient un champ : à choisir

- On transforme handleClick en champ qu'on initialise.

```
// arrow fx for binding
handleCardClick = (card, feedback) => {
  console.log(card+' '+feedback+' : clicked', ' this: ',this)
}
```

- C'est une fonction fléchée : le this est le bon. Mais rien ne nous dit que c'est une syntaxe pour accéder au this.
 - Il faut le mettre en commentaire : // arrow fx for binding
 - On va privilégier cette écriture. On positionne ces fonctions après le constructeur et après l'état local.
- C'est équivalent à :

```
constructor(props) {
  super(props)

  this.cards=this.generateCards()

  // arrow fx for binding
  this.handleClick = (card, feedback) => {
    console.log(card+' '+feedback+' : clicked', ' this: ',this)
  }
}
```

Doc React

- Il existe deux types de données qui contrôlent un composant : **les props** (les accessoires) et l'état (**state**).
- **Les props** (accessoires) sont **fixés par le parent** et ils sont **fixes** pendant toute la durée de vie d'un composant.
- **L'état** concerne les **données qui vont changer**.

Présentation

- C'est l'approche objet classique : une classe a des attributs.
- Ils sont locaux : ils ne seront visibles que dans le composant : ni par son parent, ni par ses enfants.
- C'est ce qu'on appelle l'état local.
- L'attribut s'appelle « state ». C'est un objet qui contiendra les attributs spécifiques qu'on veut gérer dans notre classe-composant.
- Comme pour l'attribut « cards » déjà abordé, le state peut être géré comme un attribut directement dans la classe ou alors dans le constructeur.

Exemple

```
state = {  
  cards: this.generateCards(), // les cartes  
  currentPair: [],             // les cartes en train d'être retournées  
  guesses: 0,                  // le nombre de coups joués  
  matchedCardIndices: [],      // toutes les cartes retournées ok  
}
```

Destructuration dans le render :

- On peut récupérer l'état avec une destructuration (cf. p.48) :

```
const {cards, guesses, matchedCardIndices} = this.state
```

- On a récupéré 3 variables, cards, guesses et matchedCardIndices à partir de this.state, en une seule ligne : c'est pratique !

TP-J7-1 ->

- Dans le composant App, **structurer la classe avec un constructeur** et ajouter l'état (le state) au TP-J6-2 (cf cours)

```
constructor(props) {  
  super(props)  
  this.state = {  
    cards: this.generateCards(), // les cartes  
    currentPair: [],             // les cartes en train d'être retournées  
    guesses: 0,                  // le nombre de coups joués  
    matchedCardIndices: [],      // toutes les cartes retournées ok  
  }  
}
```

- Récupérer les variables par destructuration au début du render.

```
const {cards, guesses, matchedCardIndices} = this.state
```

- Mettre à jour le this.cards dans le render() : ce n'est plus this.cards mais le cards sorti du state par la destructuration.

```
{cards.map((card, index) => (
```

- Mettre à jour le guesses du GuessCount : il ne vaut plus 0 mais l'état local guesses, qu'on incrémentera ailleurs : il faut revenir à la valeur fournie à l'appel.

```
<GuessCount guesses={guesses}/>
```

- Dans le composant Card, ajouter une prop : index.** Ce sera l'index de la carte dans jeu (dans le tableau).

```
const Card = ({ card, feedback, index, onClick }) => (
```

- Mettre à jour l'appel à onclick

```
onClick={() => onClick(index)}
```

- Déclarer la propTypes de index dans le composant Card

```
index : PropTypes.number.isRequired,
```

- Dans le render de App : mettre à jour la création des card en conséquence. Dans la création des card qui est fait en boucle avec le map, il faut ajouter la props index en lui donnant la valeur de l'index correspondant :

```
index={index}
```

- Dans le composant App, gérer le this proprement dans le **handleCardClick** (cf cours)

```
// arrow fx for binding  
handleCardClick = (card, feedback) => {
```

- On change de handelCardClick :

- C'est une méthode avec 1 seul paramètre : index (et non plus card + feedback). L'index, c'est le numéro de la carte dans le tableau. C'est l'index qu'on passe en paramètre quand on appelle onclick.
- Etant donné qu'on appelle une méthode qui n'est pas écrite quand on a une paire avec 1 card, au deuxième clic sur le jeu, ça plante !

```
// Arrow fx for binding
```

```

handleCardClick = index => {
  const { currentPair, cards } = this.state
  console.log('handleCardClick -> La carte: '+cards[index]
    +' - La paire: '+currentPair
    +' - Les cartes: '+cards)
  console.log('carte n°'+index
    +' - this: ' + this
    +' - classe: '+this.constructor.name)
  // si la paire est pleine : on ne fait rien
  if (currentPair.length === 2) {
    return
  }
  // si la paire est vide : on met l'index dans la paire
  if (currentPair.length === 0) {
    this.setState({ currentPair: [index] })
    return
  }
  // sinon : la paire contenait une valeur : là il y a du travail !
  // il faudra écrire cette méthode
  this.handleNewPairClosedBy(index)
}

```

- Gérer le feedback des card : **méthode getFeedbackForCard**. A la création de la carte, on va appeler une fonction qui calculera l'état de la carte à partir de son index. Pour bien comprendre la dynamique du programme, on regardera plus tard le cycle de vie.

```
feedback={this.getFeedbackForCard(index) }
```

- Il faut écrire la méthode getFeedbackForCard :
 - Pour le moment on prend cette méthode sans la comprendre.
 - Elle retourne l'état de la carte en fonction du contexte.
 - On a le hidden de départ

```

getFeedbackForCard(index) {
  const { currentPair, matchedCardIndices } = this.state
  const indexMatched = matchedCardIndices.includes(index)
  if (currentPair.length < 2) {
    return indexMatched || index === currentPair[0] ?
      'visible' : 'hidden' // c'est le hidden de départ
  }
  if (currentPair.includes(index)) {
    return indexMatched ? 'justMatched' : 'justMismatched'
  }
  return indexMatched ? 'visible' : 'hidden'
}

```

- **TESTER** : à ce stade, l'affichage est propre. On peut cliquer une fois, mais pas 2 : la fonction « handleNewPairClosedBy » n'existe pas. On a un message cliquable dans le navigateur qui ramène directement dans l'éditeur ! A noter qu'on peut tester les étapes une par une.

setState

doc

- <https://fr.reactjs.org/docs/react-component.html#setstate>
- https://www.w3schools.com/react/react_state.asp

usage basique

```
this.setState({ name: 'toto', age: 20 })
```

- Attention : la fonction est asynchrone et l'exécution regroupera des actions identiques.

- Si on écrit, avec state.open à false

```
this.setState({ open: true })  
console.log(this.state.open === true) // `false`
```

- Dans console.log, state.open vaut toujours false

- Si on écrit, avec state.count à 0 au début

```
this.setState({ count: this.state.count + 1 })  
this.setState({ count: this.state.count + 1 })  
console.log(this.state.count === 2) // `false`
```

- Dans console.log, state.count vaut toujours 0 et il vaudra 1 quand tout sera fait car les deux instructions sont réduites à une seule.

usage sécurisé de setState

- Le paramètre de setState est une fonction fléchée avec state et props en paramètres :

```
this.setState((state, props) => {  
  return {counter: state.counter + props.step};  
});
```

- Avec cet usage, si on fait plusieurs appels successifs, chaque appel est pris en compte

TP-J7-2 ->

- On va pouvoir écrire la fonction `handleNewPairClosedBy(index)`
- Cette fonction se trouve dans la casse `App`.
- Dans cette classe on définit la constante pour le temps avant de retourner une paire non conforme.

```
const PAUSE_MSECS = 750
```

- Ensuite, la fonction :

```
handleNewPairClosedBy(index) {  
  // la fonction est appelée sur la deuxième carte soulevée  
  
  // PARTIE MODELE  
  // destructuration : on récupère l'état local  
  const {cards,currentPair,guesses,matchedCardIndices}=this.state  
  
  // initialisation de variables locales : le modèle local  
  // pour simplifier le traitement du contrôleur  
  const newPair = [currentPair[0], index] // on remplit la pair  
  const newGuesses = guesses + 1          // on incrémente newGuesses  
  const matched = cards[newPair[0]] === cards[newPair[1]]  
  
  // PARTIE CONTROLEUR et VUE  
  // On fait les setState nécessaires qui engendrent  
  // les mises à jour de la page  
  this.setState({ currentPair: newPair, guesses: newGuesses })  
  
  if (matched) {  
    this.setState(  
      {matchedCardIndices: [...matchedCardIndices, ...newPair] }  
    )  
  }  
  
  setTimeout(  
    () => this.setState( { currentPair: [] } ),  
    PAUSE_MSECS  
  )  
}
```

- Mettre à jour le won :

```
const won = matchedCardIndices.length === cards.length
```

- On pourra mettre `=== 4` pour tester rapidement une victoire !

```
const won = matchedCardIndices.length === 4
```

- On n'affiche gagné en console que dans on a gagné :

```
if (won) {console.log('gagné : ',won)};
```


- **TESTER** : à ce stade, ça fonctionne. Mais si on reclique sur une carte qu'on vient de découvrir, ça plante ! On va réfléchir à ça avec le cycle de vie.

Trois petits points ...

<https://blog.nathanaelcherrier.com/fr/rest-parameter-et-spread-operator-en-javascript/>

➤ ***rest parameter :***

...var : stocke une liste d'éléments séparés par des virgules dans un tableau

➤ ***spread parameter : fait l'opération inverse ! :***

...var : transforme un tableau en liste de valeurs séparées par des virgules

Que fait :

```
this.setState({matchedCardIndices:[...matchedCardIndices, ...newPair]})
```

- Dans le tableau des cartes retournées, on met : le tableau des cartes retournées et le tableau des nouvelles cartes.
- Les ... sont des « spread parameter » : on transforme un tableau en une liste de valeurs séparées par des virgules.
- On récupère la liste des valeurs des cartes déjà retournés, séparées par des virgules, puis les deux nouvelles cartes, elles aussi séparées par des virgules. Le tout est mis entre crochets. C'est un tableau de valeurs qu'on affecte à matchedCardIndices.

setTimeout (fonction, delai)

- setTimeout (fonction, delai) exécute la fonction après un délai en millisecondes.
- On peut passer une fonction anonyme à setTimeout et même une fonction fléchées.

TP-J7-3 ->

- On a toujours un bouton « Cliquez » qui ne sert à rien.
- Ce bouton est géré par le composant TestEvt (c'était un composant de test au départ).
- On va lui changer de nom : il s'appellera : recommencer la partie.

```
Recommencer la partie
</button>
```

- Il faut créer une méthode qui videra les tableaux de notre jeu : ça se fera sur le click sur recommencer la partie.
- On va donc avoir un handleBtnRazClick

```
handleBtnRazClick = () => {
  const {cards, currentPair, guesses, matchedCardIndices} = this.state
  console.log('handleBtnRazClick ->'
    +' - guesses: '+guesses
    +' - paire: '+currentPair
    +' - matched: '+matchedCardIndices
    +' - Les cartes: '+cards
  )
  this.setState({ currentPair: [] })
  this.setState({ guesses: 0 })
  this.setState({ matchedCardIndices: [] })
}
```

- Du coup on supprime le console.log détaillé du handleClick. On a que la currentPair dans la déstructuration.
- Quand on crée le composant, on passe en paramètre la méthode :

```
TestEvt
  cestQui="SuperJoueur"
  onClicke={this.handleBtnRazClick}
/>
```

- On met à jour le composant TestEvt :

```
const TestEvt = ({ cestQui, onClicke }) => (
  <button
    onClick={() =>{
      // console.log(`Bonjour ${cestQui} !`) ou alors :
      console.log('cestQui: ' + cestQui + '!' + ' - onClicke: ' + onClicke)
      onClicke(cestQui)}
  >
    Recommencer la partie
  </button>
)
```

- Pour finir, on change le nom du composant : il s'appelle désormais BtnRaz. Il faut aussi changer le nom du fichier et des includes. On va supprimer l'attribut cestQui qui ne sert à rien.

12 – CYCLE DE VIE D'UN COMPOSANT

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664881-apprivoisez-le-cycle-de-vie-des-composants>

<https://fr.reactjs.org/docs/state-and-lifecycle.html>

Principes de base

- Les composants React passent par **plusieurs étapes durant leur cycle de vie**.
- **On peut contrôler chacune de ces étapes** en implémentant des méthodes exécutées automatiquement.
- Un composant à au moins un `render()` : une étape de son cycle de vie est l'appel au `render()`.
- Certains composants ont un constructeur.
- Une troisième méthode sera utile : `ComponentWillReceiveProps`.
- D'autres méthodes peuvent aussi intervenir.

Cycle de vie simplifié

- 1 : Etape de construction
 - 1 : `Constructeur()` : props et state par défaut
 - 2 : `Render()` : c'est la première mise en place dans le DOM virtuel.
- 2 : Cycle de mise à jour
 - 3 : `ComponentWillReceiveProps()` : c'est appelé quand le parent refait un render. Ses enfants sont conservés et seulement mis à jour.
 - 4 : `Render()` : après la mise à jour des props, le composant rend, ce qui met à jour le DOM virtuel et permet un cycle de mise à jour de ses enfants
- 3 : Etape de suppression du DOM
 - 5 : `ComponentWillUnmount()` : le composant annule toutes les initialisations qui avaient été faites et supprime tous ses enfants.
- Les étapes de construction et de suppression n'ont lieu qu'une fois.
- Le cycle de mise à jour est répété à chaque render du parent. Il peut ne pas y en avoir.

Etape de construction

1 : La construction du composant : rôle du constructeur

- Le composant va apparaître dans le DOM (virtuel). Un composant, c'est un nœud dans l'arbre du DOM (virtuel).
- Il reçoit les props initiales en argument.
- C'est l'occasion d'initialiser l'état local et de garantir le this de certaines méthodes (avec un bind() manuel ou en faisant de la méthode un attribut).

2 : le render()

<https://fr.reactjs.org/docs/render-props.html>

- Le render est une prop qui contient une fonction qui retourne du HTML : une grappe HTML, c'est à dire une arborescence de balises contenues dans une balise racine.
- Le render peut construire d'autres composant, ou les supprimer, ou les mettre à jour s'ils étaient déjà présents.

Cycle de mise à jour

- Quand un composant parent fait nouveau render(), ses composants enfants font un cycle de mise à jour avec les nouvelles props.
- Le cycle de mise à jour fait appel à la méthode componentWillReceiveProps(). On peut l'implémenter dans notre composant enfant pour, par exemple, mettre à jour l'état local avec un setState(). Ensuite, le cycle de mise à jour se termine par un render().

TP-Cycle de vie - 1 ->

- ```
console.log('-> render App')
```

```
console.log('-> render App')
```

- ```
export default class Card extends React.Component {
```

```
export default class Card extends React.Component {
  render() {
    console.log('-> render Card');
    return(
      <div
        className={`card ${this.props.feedback}`}
        onClick={() => this.props.onClick(this.props.index)}
      >
        <span className="symbol">
          {this.props.feedback === 'hidden' ? HIDDEN_SYMBOL :
this.props.card}
        </span>
      </div>
    )
  }
}
```

- > render App App.js:140

```
-> render Card 36 Card.js:32
```

- On a 1 render App et 36 appels à render Card

- **TESTER** : On clique sur Recommencer la partie :

- ```
handleBtnRazClick -> - guesses: 0 - paire: - matched: - App.js:48
```

Les cartes: 

```
-> render Card 36 Card.js:32
```

TESTER : On clique sur une carte :

- ```
carte n°0 - this: [object Object] - classe: App App.js:63
```

```
-> render App App.js:140
```

-> render Card 36 Card.js:32

- On a 1 render App et 36 appels à render Card

- **TESTER** : On clique sur une 2^{ème} carte :

- QUESTION 1** On the first day, the bank balance was £1000. On the second day, the bank balance was £1000 + £100 = £1100. On the third day, the bank balance was £1100 + £100 = £1200. On the fourth day, the bank balance was £1200 + £100 = £1300. On the fifth day, the bank balance was £1300 + £100 = £1400. On the sixth day, the bank balance was £1400 + £100 = £1500. On the seventh day, the bank balance was £1500 + £100 = £1600. On the eighth day, the bank balance was £1600 + £100 = £1700. On the ninth day, the bank balance was £1700 + £100 = £1800. On the tenth day, the bank balance was £1800 + £100 = £1900. On the eleventh day, the bank balance was £1900 + £100 = £2000. On the twelfth day, the bank balance was £2000 + £100 = £2100. On the thirteenth day, the bank balance was £2100 + £100 = £2200. On the fourteenth day, the bank balance was £2200 + £100 = £2300. On the fifteenth day, the bank balance was £2300 + £100 = £2400. On the sixteenth day, the bank balance was £2400 + £100 = £2500. On the seventeenth day, the bank balance was £2500 + £100 = £2600. On the eighteenth day, the bank balance was £2600 + £100 = £2700. On the nineteenth day, the bank balance was £2700 + £100 = £2800. On the twentieth day, the bank balance was £2800 + £100 = £2900. On the twenty-first day, the bank balance was £2900 + £100 = £3000. On the twenty-second day, the bank balance was £3000 + £100 = £3100. On the twenty-third day, the bank balance was £3100 + £100 = £3200. On the twenty-fourth day, the bank balance was £3200 + £100 = £3300. On the twenty-fifth day, the bank balance was £3300 + £100 = £3400. On the twenty-sixth day, the bank balance was £3400 + £100 = £3500. On the twenty-seventh day, the bank balance was £3500 + £100 = £3600. On the twenty-eighth day, the bank balance was £3600 + £100 = £3700. On the twenty-ninth day, the bank balance was £3700 + £100 = £3800. On the thirtieth day, the bank balance was £3800 + £100 = £3900. On the thirty-first day, the bank balance was £3900 + £100 = £4000. On the thirty-second day, the bank balance was £4000 + £100 = £4100. On the thirty-third day, the bank balance was £4100 + £100 = £4200. On the thirty-fourth day, the bank balance was £4200 + £100 = £4300. On the thirty-fifth day, the bank balance was £4300 + £100 = £4400. On the thirty-sixth day, the bank balance was £4400 + £100 = £4500. On the thirty-seventh day, the bank balance was £4500 + £100 = £4600. On the thirty-eighth day, the bank balance was £4600 + £100 = £4700. On the thirty-ninth day, the bank balance was £4700 + £100 = £4800. On the fortieth day, the bank balance was £4800 + £100 = £4900. On the forty-first day, the bank balance was £4900 + £100 = £5000. On the forty-second day, the bank balance was £5000 + £100 = £5100. On the forty-third day, the bank balance was £5100 + £100 = £5200. On the forty-fourth day, the bank balance was £5200 + £100 = £5300. On the forty-fifth day, the bank balance was £5300 + £100 = £5400. On the forty-sixth day, the bank balance was £5400 + £100 = £5500. On the forty-seventh day, the bank balance was £5500 + £100 = £5600. On the forty-eighth day, the bank balance was £5600 + £100 = £5700. On the forty-ninth day, the bank balance was £5700 + £100 = £5800. On the fiftieth day, the bank balance was £5800 + £100 = £5900. On the fifty-first day, the bank balance was £5900 + £100 = £6000. On the fifty-second day, the bank balance was £6000 + £100 = £6100. On the fifty-third day, the bank balance was £6100 + £100 = £6200. On the fifty-fourth day, the bank balance was £6200 + £100 = £6300. On the fifty-fifth day, the bank balance was £6300 + £100 = £6400. On the fifty-sixth day, the bank balance was £6400 + £100 = £6500. On the fifty-seventh day, the bank balance was £6500 + £100 = £6600. On the fifty-eighth day, the bank balance was £6600 + £100 = £6700. On the fifty-ninth day, the bank balance was £6700 + £100 = £6800. On the sixtieth day, the bank balance was £6800 + £100 = £6900. On the sixty-first day, the bank balance was £6900 + £100 = £7000. On the sixty-second day, the bank balance was £7000 + £100 = £7100. On the sixty-third day, the bank balance was £7100 + £100 = £7200. On the sixty-fourth day, the bank balance was £7200 + £100 = £7300. On the sixty-fifth day, the bank balance was £7300 + £100 = £7400. On the sixty-sixth day, the bank balance was £7400 + £100 = £7500. On the sixty-seventh day, the bank balance was £7500 + £100 = £7600. On the sixty-eighth day, the bank balance was £7600 + £100 = £7700. On the sixty-ninth day, the bank balance was £7700 + £100 = £7800. On the seventieth day, the bank balance was £7800 + £100 = £7900. On the seventy-first day, the bank balance was £7900 + £100 = £8000. On the seventy-second day, the bank balance was £8000 + £100 = £8100. On the seventy-third day, the bank balance was £8100 + £100 = £8200. On the seventy-fourth day, the bank balance was £8200 + £100 = £8300. On the seventy-fifth day, the bank balance was £8300 + £100 = £8400. On the seventy-sixth day, the bank balance was £8400 + £100 = £8500. On the seventy-seventh day, the bank balance was £8500 + £100 = £8600. On the seventy-eighth day, the bank balance was £8600 + £100 = £8700. On the seventy-ninth day, the bank balance was £8700 + £100 = £8800. On the eightieth day, the bank balance was £8800 + £100 = £8900. On the eighty-first day, the bank balance was £8900 + £100 = £9000. On the eighty-second day, the bank balance was £9000 + £100 = £9100. On the eighty-third day, the bank balance was £9100 + £100 = £9200. On the eighty-fourth day, the bank balance was £9200 + £100 = £9300. On the eighty-fifth day, the bank balance was £9300 + £100 = £9400. On the eighty-sixth day, the bank balance was £9400 + £100 = £9500. On the eighty-seventh day, the bank balance was £9500 + £100 = £9600. On the eighty-eighth day, the bank balance was £9600 + £100 = £9700. On the eighty-ninth day, the bank balance was £9700 + £100 = £9800. On the ninetieth day, the bank balance was £9800 + £100 = £9900. On the hundredth day, the bank balance was £9900 + £100 = £10000.

carte n°8 - this: [object Object] - classe: App	App.js:63
-> render App	App.js:140
-> render Card	36 Card.js:32
-> render App	App.js:140
-> render Card	36 Card.js:32

- On a 2 séries de render App et render Card.
- On ajoute un console.log à l'entrée de handleNewPairClosedBy et dans le setTimeout :

```
console.log('handleNewPairClosedBy')
```

```
setTimeout(
  () => {
    console.log('handleNewPairClosedBy : setTimeout')
    this.setState( { currentPair: [] } )
  },
  PAUSE_MSECS
)
```

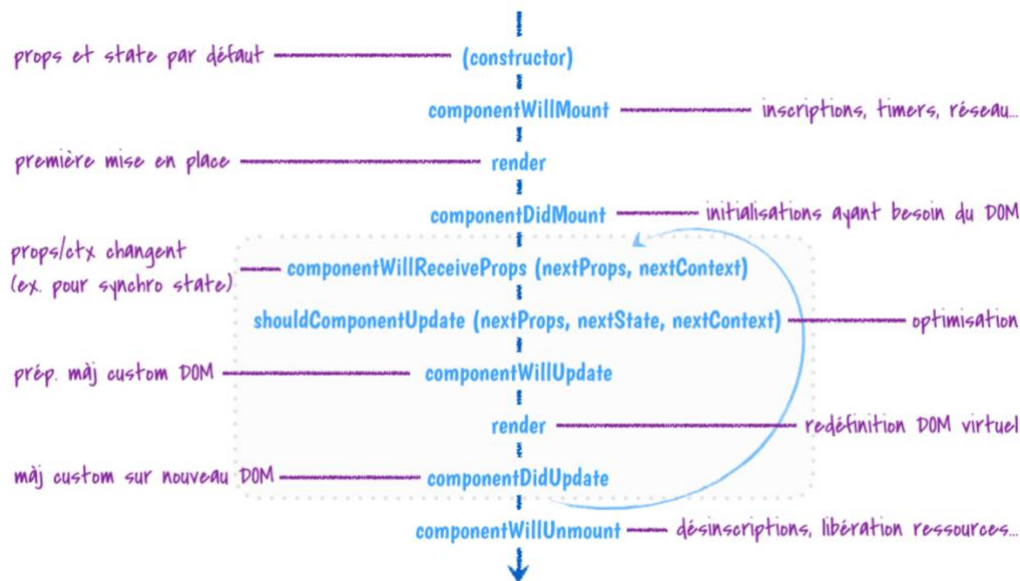
- **TESTER :** On clique sur une 2ème carte :

handleCardClick -> carte n°1 - this: [object Object] - classe: App	App.js:62
handleNewPairClosedBy	App.js:79
-> render App	App.js:153
-> render Card	36 Card.js:42
handleNewPairClosedBy : setTimeout	App.js:103
-> render App	App.js:153
-> render Card	36 Card.js:42

- Explications : quand clique sur une 2ème carte, on appelle la méthode handleClickCard qui appelle handleNewPariCloseBy.
- Plusieurs setState sont faits : ils seront traités ensemble. Quand ils sont traités, tous ensemble, il y a un render de l'App et les 36 render des Card.
- Ensuite, on entre dans le setTimeout. De ce fait, les setState sont traités après et il y a un nouveau render de l'App et les 36 render des Card. C'est le setTimeout sert de « commit » des setState.

TP-Cycle de vie - 2 ->

- Le cycle de vie est décrit dans ce schéma :



Les étapes du cycle de vie d'un composant défini par classe

- On ajoute des méthodes de cycle de vie, avant le render de App :

```
componentWillMount() {  
  console.log('-> componentWillMount App');  
}  
componentWillReceiveProps() {  
  console.log('-> componentWillReceiveProps App');  
}  
componentWillUnmount() {  
  console.log('-> componentWillUnmount Card');  
}
```

- Et avant le render de Card :

```
componentWillMount() {  
  // console.log('-> componentWillMount Card :  
  '+this.props.index);  
}  
componentWillReceiveProps() {  
  console.log('-> componentWillReceiveProps Card :  
  '+this.props.index);  
}  
componentWillUnmount() {  
  // console.log('-> componentWillUnmount Card :  
  '+this.props.index);  
}
```

- On regarde ce qui se passe quand on utilise le programme :
- On voit que l'App est montée et renderée. Puis chaque carte est montée et renderée.

- Sur un click sur carte, chaque, chaque carte est renderée et les props mises à jour : il n'y a plus de montage, mais il y a un appel à `componentWillReceiveProps`.
- Sur un click sur une deuxième carte, on a en plus le passage par le `setTimeout`.

TP-Cycle de vie – Mise à jour du HandleCardClick et du getFeedbackForCard

- On va fusionner handleNewPairClosedBy dans HandleCardClick pour mieux suivre le cycle de vie.
- Nouveau code de HandleCardClick :

```
// Arrow fx for binding
handleCardClick = index => {
  const {cards, currentPair, guesses,
matchedCardIndices}=this.state
  console.log('handleCardClick -> la carte
'+index+':'+cards[index]+' - La paire: ['+currentPair+']')
  // si la paire est vide :
  if (currentPair.length === 0) {
    console.log('currentPair.length === 0')
    // on met l'index dans la currentPair
    this.setState({ currentPair: [index] })
    return
  }
  // si on reclique sur la carte qu'on vient de jouer : on ne
fait rien
  else if (currentPair.length === 1 && currentPair[0] === index ){
    return
  }
  // sinon, on a une carte à ajouter dans la paire
  else{
    console.log('currentPair.length === 1')
    const newPair = [currentPair[0], index] // on remplit la
paire
    const newGuesses = guesses + 1           // on incrémente
newGuesses
    // on met à jour currentPair et guesses
    this.setState({ currentPair: newPair, guesses: newGuesses })
    // si les 2 cartes correspondent
    if (cards[newPair[0]] === cards[newPair[1]]) {
      console.log('handleCardClick : matched !!!')
      // on met à jour matchedCardIndices
      this.setState(
        {matchedCardIndices: [...matchedCardIndices, ...newPair]
}
      )
    }
  }
  setTimeout(
    // on vide currentPair après un délai
    () => {
      console.log('handleCardClick : setTimeout')
      this.setState( { currentPair: [] } )
    },
```

```
        PAUSE_MSECS  
    )  
}  
}
```

- Ici, on voit bien qu'on a plusieurs `setState`. Les 3 premiers seront synchronisés ensemble. Le dernier est dans un `setTimeout` qui fait office de « commit ». Le `setTimeout` engendrera un nouveau `render()`.

- On fait une version développée et plus explicite du `getFeedbackForCard()` avec des `console.log`

```
getFeedbackForCard(index) {
  const {cards, currentPair, matchedCardIndices} = this.state
  const indexDansCouple = matchedCardIndices.includes(index)
  const indexDansLaPaire = currentPair.includes(index)
  const laPaireEstVide = currentPair.length==0
  const unDanslaPaire = currentPair.length==1
  const laPaireEstPleine = currentPair.length==2
  let retourne
  // console.log('getFeedbackForCard-> '+currentPair.length +
  index: '+index+'-currentPair: '+currentPair+'-
  matchedCardIndices: '+matchedCardIndices, cards[currentPair[0]],
  cards[currentPair[1]])
  // si déjà affiché et la paire est vide : il faut les 2
  if (indexDansCouple && laPaireEstVide) retourne='visible'
  // si pas affiché et la paire est vide : initialisation
  else if (!indexDansCouple && laPaireEstVide )
  retourne='hidden'
  // si pas affiché et pas dans la pair : hidden
  else if (!indexDansCouple && !laPaireEstVide &&
  !indexDansLaPaire) retourne='hidden'
  // si seul dans la pair : hidden
  else if (unDanslaPaire && indexDansLaPaire) retourne='visible'
  // si dans la paire pleine :
  else if (laPaireEstPleine && indexDansLaPaire &&
  cards[currentPair[0]]===cards[currentPair[1]])
  retourne='justMatched'
  else if (laPaireEstPleine && indexDansLaPaire &&
  cards[currentPair[0]]!==cards[currentPair[1]])
  retourne='justMismatched'
  console.log('getFeedbackForCard: '+index+':'+retourne)
  return retourne
}
```

- Ici, on peut constater avec les `console.log` que quand on clique sur un bon couple, le `matchedCardIndices` est rempli et donc `indexDansCouple` est vrai alors qu'on pourrait penser qu'il est faux. Du coup, on teste directement `laPaireEstPleine`.

13 – LES FORMULAIRES

<https://openclassrooms.com/fr/courses/4664381-realisez-une-application-web-avec-react-js/4664881-apprivoisez-le-cycle-de-vie-des-composants>

Situation HTML avant React

HTML

En HTML, la valeur réelle d'un champ dépend largement du type de balise utilisée. Elle peut être définie par :

- **l'attribut value=...**
 - Exemple : `<input type="text" name="nom" value="toto" ... />`
- **les attributs checked=...**
 - Exemple : `<input type="checkbox" ... checked />`
- **multiple= et selected=...**
 - Exemple : `<select... multiple><option selected> ... <option/></select>`
- **le contenu de l'élément :**
 - Exemple : `<textarea> contenu (</textarea>)`

Dom

On retrouve cette variété dans le DOM avec les propriétés

- **value**
- **selectedIndex**
- **options[...].selected**
- **checked**
- ...

Solution JQuery

- méthode `.val()`

Solution React : prop value ou defaultValue

- `prop value = ...` pour tous les composants.
- S'il s'agit de valeurs multiples, on passe un tableau de valeurs.
- On pourra aussi utiliser la `prop defaultValue`.
- A noter que pour les cases à cocher, on garde l'attribut `checked`.
- Le composant va apparaître dans le DOM (virtuel). Un composant, c'est un nœud dans l'arbre du DOM (virtuel).

Situation événementiel avant React

Dom

Pour détecter un changement de valeur de champs, de nombreux événements peuvent intervenir :

- change
- click
- keydown
- focus
- blur
- ...

Solution React : `prop onChange`

- `prop onChange = ...` pour tous les changement de valeur.

```
<input
  onChange={this.handleLiveChange}
  type="text"
  value={this.state.nom}
/>
<input
  onChange={this.handleLiveChange}
  type="checkbox"
  checked={this.state.choisi}
/>
<select
  onChange={this.handleLiveChange}
  multiple
  value={this.state.lesPropositions}
/>
<textarea
  onChange={this.handleLiveChange}
  value={this.state.commentaire}
/>
```


Objectifs

- Les champs contrôlés permettent d'intervenir au fil de la saisie pour valider et retravailler la saisie.

Technique

- Les valeurs des champs sont saisies dans l'état local du composant : on peut les gérer comme n'importe quel attribut d'une classe.
- Un champ contrôlé a une prop `value={this.state.value}` (par exemple, ou `checked=` pour les cases à cocher et les radio boutons).
- Un champ contrôlé a une prop `onChange= {this.handleChange}` (par exemple) qui conduit à un gestionnaire de contrôle qui assure la vérification, le formatage et le stockage dans l'état local. Sans le stockage dans l'état local (un `setState`), le champ n'est pas saisissable.
- Le gestionnaire de contrôle ressemble à :

```
handleChange = ({ target: { value } }) => {  
  ...  
  this.setState({ value })  
}
```

Remarque

- prop `onChange= {this.handleChange}` : on donne toujours le nom `handleChange` pour une méthode attachée à un `onChange`.

Exemple

➤ *Composant Classe pour la saisie d'un champ téléphonique :*

La structure sera à peu près la même pour chaque composant à saisir :

```
import React from 'react';
import PropTypes from 'prop-types';

export default class FrenchPhoneField extends React.Component {
  static defaultProps = {
    name: 'tel',
    placeholder: '0x xx xx xx xx',
    required: false,
  }
  static propTypes = {
    name: PropTypes.string.isRequired,
    placeholder: PropTypes.string.isRequired,
    required: PropTypes.bool.isRequired,
  }
  constructor(props) {
    super(props)
    this.state = { value: '' }
  }
  // arrow fx for binding
  handleChange = ({ target: { value } }) => {
    value = ...
    this.setState({ value })
  }
  render() {
    const { name, placeholder, required } = this.props
    return (
      <input
        name={name}
        value={this.state.value}
        type="text"
        placeholder={placeholder}
        required={required}
        onChange={this.handleChange}
      />
    )
  }
}
```

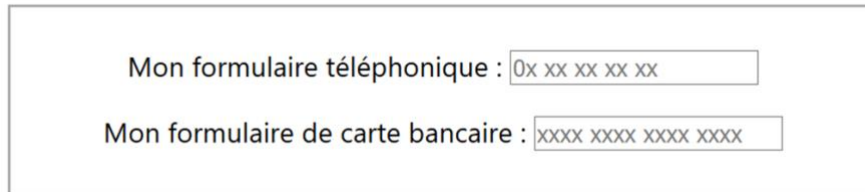
➤ **Composant fonctionnel App construisant un formulaire :**

```
function App() {
  return (
    <div className="App">
      <h1>Exemple de formulaire</h1>
      <form>
        <fieldset>
          <p>
            <label>Mon formulaire téléphonique : </label>
            <FrenchPhoneField />
          </p>
          <p>
            <label>Mon formulaire de carte bancaire : </label>
            <CreditCardField />
          </p>
        </fieldset>
      </form>
    </div>
  );
}
```

TP-J7-formulaire ->

- Chargez le dossier « livraison-formulaire.zip ». Il contient la classe téléphone et les spécificité pour de la classe carte de crédit.
- Créez une application qui affiche :

Exemple de formulaire



Mon formulaire téléphonique :

Mon formulaire de carte bancaire :

- Pour ça, commencer par créer une application « formulaire » avec create-react-app formulaire.
- Mettez à jour l’application pour afficher l’exemple proposé.
- C’est le composant App qui se charge de l’affichage souhaité. On utilisera une balise `<form>` et une balise `<fieldset>`