Cours JS - 1

Les types

Les types primitifs

Qu'est-ce qu'un type primitif ?

Un type primitif est une donnée qui ne peut pas être modifiée.
Les types primitifs sont:

- 📌 **les nombres**
- 📌 **les chaînes de caractères**
- 📌 **les booléens**
- 📌 **les valeurs nulles et les valeurs indéfinies.**

Déclaration d'un type primitif

```
const nombre = 42;
const chaine = "Bonjour";
const booleen = true;
const nul = null;
const indefini = undefined;
```

Les types objets

Qu'est-ce qu'un type objet ?

Un type objet est une donnée qui peut être modifiée. Les types objets sont:

- 📌 **les tableaux**
- 📌 **les fonctions**
- 📌 **les objets**
- 📌 **les dates**

Déclaration d'un type objet

```
const tableau = [1, 2, 3];
const fonction = function() {};
const objet = { nom: "Aurora" };
const date = new Date();
```

Les types objet predefinis

Qu'est-ce qu'un type objet prédéfini ?

Un type objet prédéfini est un type objet qui est défini par le langage JavaScript. Les types objets prédéfinis sont:

- 📌 **les dates**
- 📌 **les erreurs**
- 📌 **les regexp**
- 📌 **les math**
- 📌 **les JSON**

Déclaration d'un type objet prédéfini

```
const date = new Date();
const erreur = new Error("Oups!");
const regexp = /ab+c/;
const math = Math.random();
const json = JSON.stringify({ nom: "Aurora" });
```

Les fonctions

Qu'est-ce qu'une fonction ?

Une fonction est un bloc de code qui peut être exécuté plusieurs fois. Une fonction peut prendre des paramètres et retourner une valeur.

Déclaration d'une fonction

```
function direBonjour() {
  console.log("Bonjour !");
}

direBonjour();
```

Déclaration d'une fonction avec des paramètres

```
function direBonjour(nom) {
  console.log(`Bonjour ${nom} !`);
}

direBonjour("Aurora");
```

Déclaration d'une fonction avec un paramètre par défaut

```
function direBonjour(nom = "Aurora") {
  console.log(`Bonjour ${nom} !`);
}

direBonjour();
```

Déclaration d'une fonction avec un paramètre optionnel

```
function direBonjour(nom) {
  if (nom) {
    console.log(`Bonjour ${nom} !`);
  } else {
    console.log("Bonjour !");
  }
}

direBonjour();
```

Déclaration d'une fonction qui retourne une valeur

```
function additionner(a, b) {
  return a + b;
}

console.log(additionner(1, 2));
```

Déclaration d'une fonction fléchée

```
const direBonjour = nom => console.log(`Bonjour ${nom} !`);

direBonjour("Aurora");
```

Déclaration d'une fonction fléchée avec un paramètre par défaut

```
const direBonjour = (nom = "Aurora") => console.log(`Bonjour ${nom} !`);

direBonjour();
```

Déclaration d'une fonction avec paramètres restants

```
function direBonjour(nom, ...autresNoms) {
  console.log(`Bonjour ${nom} !`);
  console.log(autresNoms);
}

direBonjour("Aurora", "Maxime", "Ludovic");
```

Les callbacks

Qu'est-ce qu'un callback ?

Un callback est une fonction qui est passée en paramètre d'une autre fonction. Par exemple ces fonctions necessite un callback:

- 📌 `setTimeout()`
- 📌 `setInterval()`
- 📌 `addEventListener()`

Déclaration d'un callback

```
setTimeout(function() {
  console.log("1 seconde!");
}, 1000);
```

Déclaration d'un callback avec une fonction fléchée

```
setTimeout(() => {
  console.log("1 seconde!");
}, 1000);
```

Les fonctions de closure

Qu'est-ce qu'une fonction de fermeture ?

- 📌 **Une fonction de fermeture est une fonction qui est définie à l'intérieur d'une autre fonction.**
- 📌 **Une fonction de fermeture a accès aux variables de la fonction parente.**

Exemple de fonction de fermeture

```
function direBonjour(prenom) {
  const message = "Bonjour, " + prenom + " !";
  return function() {
    console.log(message);
  };
}

const direBonjourJohn = direBonjour("John");
direBonjourJohn(); // Bonjour, John !

const direBonjourPaul = direBonjour("Paul");
direBonjourPaul(); // Bonjour, Paul !
```

Exercice

Exercice 1: Creer une fonction qui permet de dire bonjour au bout de 1,5 seconde

Exercice 2: Creer une fonction qui a pour prototype `function aumoins3(array, verifcallback) {}` qui permet de verifier si un tableau contient au moins 3 elements qui verifient la condition de la fonction de callback
elle retourne true ou false

Exercice 3: Creer une fonction qui a pour prototype `function filter(array, verifcallback) {}` qui permet de renvoyer un tableau contenant tous les elements qui verifient la condition de la fonction de callback

Le hoisting

Qu'est-ce que le hoisting ?

Le hoisting est un mécanisme par lequel les déclarations de variables et de fonctions sont déplacées en haut de leur portée (scope) avant l'exécution du code.

Le hoisting n'affecte que les déclarations et pas les initialisations.

Exemple

```
function test() {
  console.log(a); // undefined
  console.log(foo()); // 2

  var a = 1;
  function foo() {
    return 2;
  }
}

test();
```

Quel est l'avantage du hoisting ?

📌 **Le hoisting permet de déclarer les variables et les fonctions où elles sont utilisées.**

Quel est le problème du hoisting ?

📌 **Le hoisting peut être source d'erreurs.**

Quelle est la solution ?

Utiliser des variables déclarées avec le mot-clé `let` ou `const`.

De plus, il est recommandé de déclarer les variables en haut de leur portée.

Puisque de toute maniere c'est la maniere dont le code est executé par le moteur JS.

C'est egalement la maniere de faire dans les autres langages de programmation.

Exemple

Cette exemple mets en avant plusieurs effets du hoisting

```
let x = 3;
let y = 4;
if (x === 3) {
  // y n'est pas défini ici
  console.log(y);
  let y = 5;
  // y est défini ici
  console.log(y);
}
// y est défini ici
console.log(y);
```

Les classes

Qu'est-ce qu'une classe ?

Elles sont utilisées pour créer des objets.

Ce qui permet de representer des entités du monde réel.

Lorsque l'on crée une classe, on crée un nouveau type d'objet.

Chaque objet de ce type est appelé instance de cette classe.

Les attributs

Une classe peut avoir des attributs.

- 📌 **Ces attributs sont des variables qui sont propres à l'objet.**
- 📌 **Elles permettent de stocker des informations sur l'objet.**
- 📌 **Elles sont déclarées dans le constructeur de la classe.**
- 📌 **Elles sont déclarées avec le mot clé `this`.**
- 📌 **Elles sont accessibles avec l'operateur `.`**
- 📌 **Elles sont modifiables avec l'operateur `=`**

Les attributs prive

Une classe peut avoir des attributs prive.

Ce sont des attributs normaux a ceci près qu'ils ne sont accessibles qu'a la classe elle meme.

Les attributs prive peuvent paraître étrange à première vue, mais si nous représentons un humain cela fait sens.

Personne n'est censé pouvoir manipuler les organes d'une autre personne.

Ainsi, si nous représentions un humain l'attribut coeur serai prive.

Les constructeurs

Un constructeur est une methode qui est appelée lors de la creation d'un objet.

- 📌 **Il permet d'initialiser les attributs de l'objet.**
- 📌 **Pour déclarer un constructeur, on utilise le mot clé `constructor`.**

Declaration d'une classe

📌 **Pour déclarer une classe, on utilise le mot clé `class`.**

On peut les declarer de deux manieres differentes.

Maniere 1:

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
}
```

Maniere 2 :

```
var Rectangle = class {
  constructor(height, width) {
    this.height = height;
    this.width = width
          }
        }
```

Les methodes

Qu'est-ce qu'une méthode ?

Une méthode est une fonction qui est déclarée dans une classe.

- 📌 **Elle permet d'effectuer des actions sur l'objet.**
- 📌 **Elle peut accéder aux attributs de l'objet.**

Déclaration d'une méthode

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  getArea() {
    return this.height * this.width;
  }
}
```

Appel d'une méthode

```
let rectangle = new Rectangle(2, 3);
let area = rectangle.getArea();
```

Les méthodes statiques

Une méthode statique est une méthode qui peut être appelée sans instancier la classe.

📌 **Pour déclarer une méthode statique, on utilise le mot clé `static` devant le nom de la méthode.**

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  static getArea(height, width) {
    return height * width;
  }
}

let area = Rectangle.getArea(2, 3);
```

Les méthodes privées

Une méthode privée est une méthode qui ne peut pas être appelée depuis l'extérieur de la classe.

- 📌 **Pour déclarer une méthode privée, on utilise le mot clé `#` devant le nom de la méthode.**
- 📌 **Pour appeler une méthode privée, on utilise le mot clé `this` devant le nom de la méthode.**

```
class Rectangle {
        #getArea() {
                return this.height * this.width;
        }

        printArea() {
        console.log(this.#getArea());
  }
}


let rectangle = new Rectangle(2, 3);
rectangle.printArea();
// rectangle.#getArea(); // Erreur
```

Les getter et setter

Qu'est-ce qu'un getter ?

Un getter est une méthode qui permet de récupérer la valeur d'un attribut.

📌 **Il est déclaré avec le mot clé `get` suivi du nom de l'attribut**

Exemple

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  get area() {
    return this.height * this.width;
  }
}

let rectangle = new Rectangle(2, 3);
let area = rectangle.area;
```

Qu'est-ce qu'un setter ?

Un setter est une méthode qui permet de modifier la valeur d'un attribut.

- 📌 **Il est déclaré avec le mot clé `set` suivi du nom de l'attribut**
- 📌 **Il prend en paramètre la nouvelle valeur de l'attribut.**

```
class Personnage {
  constructor(nom, sante, force) {
    this.nom = nom;
    this._sante = sante;
    this.force = force;
  }

  decrire() {
    return `${this.nom} a ${this.sante} points de vie et ${this.force} en force`;
  }

  set sante(valeur) {
    if (valeur < 0) {
      throw new Error("la santé d'un personnage ne peut pas être négative");
    }
    this._sante = valeur;
  }

  get sante() {
    return this._sante;
  }
}

const aventurier = new Personnage("Aurora", 150, 25);
console.log(aventurier.decrire());

aventurier.sante = 1000;
console.log(aventurier.decrire());

aventurier.sante = -20;
```

Exercice 1:

Creer un village qui aura les attributs :

- 📌 **nbvillageois | 1**
- 📌 **nbbois | 100**
- 📌 **nbpierre | 100**
- 📌 **nbor | 0**
- 📌 **Batiments | [HDV]**

Exercice 2:

Ajouter les methodes :

- 📌 **construiremine() une mine coutera 100 pierre a construire et ajoutera un batiment (MINEP)**
- 📌 **construirescierie() une scierie coutera 100 bois a construire et ajoutera un batiment (SCIE)**
- 📌 **construiremineor() une scierie coutera 100 bois et 100 pierre a construire et ajoutera un batiment (MINEP)**

Exercice 3:

Creer une classe batiment qui contiendra:

Attribut:

- 📌 **type**
- 📌 **hp**
- 📌 **production**
- 📌 **cout_reparation**

Methode:

- 📌 **restaurerHP(ordisponible) qui restaurera 1 hp si le parametre est superieur a cout_reparation et renvoie un objet {restaure : true|false , nouveausoldeor: }**
- 📌 **produire() qui renverra l'objet {type: "bois"|"pierre"|"or", uniteproduite: }**

Exercice 3 suite:

Classe village:

Ajouter un attribut :

tour | 0

et une methode:

createbatiment(type) qui renverra un booleen en fonction de la possibiliter de construire un batiment.

jouertour()

Elle permettra de jouer un tour de jeu a chaque appel

Pour chaque tour de jeu:

- 📌 **chaque MINEP gagne 15 pierre**
- 📌 **chaque SCIE gagne 15 bois**
- 📌 **chaque MINEO gagne 2 OR**
- 📌 **L'Hotel de ville perd 1 hp**

A chaque tour la console affichera les ressources disponibles et demandera a l'utilisateur son prochain coup :

- 📌 Construire mine de pierre
- 📌 Construire scierie
- 📌 Construire mine d'or
- 📌 ne rien faire.

Si l'utilisateur ne peut pas construire un batiment, il ne pourra pas le faire.
Le jeu s'arrete quand l'hotel de ville n'a plus de hp.
En fin de partie, le jeu affichera le nombre de tour joue.

Heritage

Qu'est-ce que l'héritage ?

L'héritage est un mécanisme qui permet à une classe de bénéficier des propriétés et méthodes d'une autre classe.

- 📌 **La classe qui hérite d'une autre classe est appelée classe fille.**
- 📌 **La classe dont une autre classe hérite est appelée classe mère.**
- 📌 **On utilise le mot clé `extends` pour déclarer une classe fille.**
- 📌 **On utilise le mot clé `super` pour appeler le constructeur de la classe mère.**

Déclaration d'un héritage

```
class Personnage {
  constructor(nom, sante, force) {
    this.nom = nom;
    this.sante = sante;
    this.force = force;
  }

  decrire() {
    return `${this.nom} a ${this.sante} points de vie et ${this.force} en force`;
  }
}

class Aventurier extends Personnage {
  constructor(nom, sante, force, xp) {
    super(nom, sante, force);
    this.xp = xp;
  }

  decrire() {
    return `${super.decrire()} et ${this.xp} points d'expérience`;
  }
}

const aventurier = new Aventurier("Aurora", 150, 25, 1000);
console.log(aventurier.decrire());
```

Exercice 1:

Créer une classe Personnage qui aura les attributs:

- 📌 **nom**
- 📌 **sante**
- 📌 **force**

Exercice 2:

Créer une classe Aventurier qui hérite de Personnage et qui aura les attributs:

- 📌 **xp**

Exercice 3:

Créer une classe Guerrier qui hérite de Personnage et qui aura les attributs:

- 📌 **rage**

Exercice 4:

Créer une classe Paladin qui hérite de Aventurier et qui aura les attributs:

- 📌 **vertu**

Exercice 5:

Créer une classe Mage qui hérite de Personnage et qui aura les attributs:

- 📌 **mana**

Les modules

Qu'est-ce qu'un module ?

Un module est un fichier qui contient du code JavaScript.

📌 Un fichier JavaScript peut contenir plusieurs modules.

Déclaration d'un module

Pour déclarer un module, on utilise le mot clé `export` devant le nom de la classe.

```
export class Personnage {
  constructor(nom, sante, force) {
    this.nom = nom;
    this.sante = sante;
    this.force = force;
  }

  decrire() {
    return `${this.nom} a ${this.sante} points de vie et ${this.force} en force`;
  }
}
```

Import d'un module

Pour importer un module, on utilise le mot clé `import` suivi du nom de la classe.

Import simple:

```
import { Personnage } from "./personnage.js";

const aventurier = new Personnage("Aurora", 150, 25);
console.log(aventurier.decrire());
```

Import multiple:

```
import { Personnage, Aventurier } from "./personnage.js";

const aventurier = new Aventurier("Aurora", 150, 25, 1000);
console.log(aventurier.decrire());
```

Import avec alias:

```
import { Personnage as Perso } from "./personnage.js";

const aventurier = new Perso("Aurora", 150, 25);
console.log(aventurier.decrire());
```

Import de tout:

```
import * as Personnage from "./personnage.js";

const aventurier = new Personnage.Aventurier("Aurora", 150, 25, 1000);
console.log(aventurier.decrire());
```

Exercice

Exercice 1:

module Personnage

Créer un module Personnage qui contiendra les classes:

- 📌 **Personnage**
- 📌 **Aventurier**
- 📌 **Guerrier**
- 📌 **Paladin**
- 📌 **Mage**

Exercice 2:

module Combat

Créer un module Combat qui contiendra les classes:

- 📌 **Arme**
- 📌 **Sort**
- 📌 **Combat**

Exercice 3:

utilser les modules Personnage et Combat afin de creer un jeu de combat au tour par tour en se faisant attaquer par un monstre.

DOM

Qu'est-ce que le DOM ?

Le DOM est une interface de programmation qui représente les documents HTML et XML sous forme d'arbres. Il fournit une représentation structurée du document, permettant ainsi l'accès et la modification du contenu du document.

Comment accéder aux éléments du DOM ?

Pour accéder aux éléments du DOM, on utilise les méthodes suivantes:

- `getElementById()`
- `getElementsByClassName()`
- `getElementsByTagName()`
- `querySelector()` **et** `querySelectorAll()`
- `document`

GetElementById()

```
const titre = document.getElementById("titre");
console.log(titre);
```

GetElementsByClassName()

```
const paragraphes = document.getElementsByClassName("paragraphe");
console.log(paragraphes);
```

GetElementsByTagName()

```
const paragraphes = document.getElementsByTagName("p");
console.log(paragraphes);
```

QuerySelector()

```
const titre = document.querySelector("#titre");
console.log(titre);
```

QuerySelectorAll()

```
const paragraphes = document.querySelectorAll(".paragraphe");
console.log(paragraphes);
```

Comment modifier les éléments du DOM ?

Pour modifier les éléments du DOM, on utilise les méthodes suivantes:

- 📌 `innerHTML`
- 📌 `style`
- 📌 `setAttribute()`
- 📌 `classList`

InnerHTML

```
const titre = document.getElementById("titre");
titre.innerHTML = "Nouveau titre";
```

Style

```
const titre = document.getElementById("titre");
titre.style.color = "red";
```

SetAttribute()

```
const titre = document.getElementById("titre");
titre.setAttribute("style", "color: red;");
```

ClassList

```
const titre = document.getElementById("titre");
titre.classList.add("rouge");
titre.classList.remove("rouge");
titre.classList.toggle("rouge");
```

Comment créer des éléments du DOM ?

Pour créer des éléments du DOM, on utilise les méthodes suivantes:

- 📌 `createElement()`
- 📌 `createTextNode()`
- 📌 `appendChild()`
- 📌 `insertBefore()`

CreateElement()

```
const titre = document.createElement("h1");
titre.id = "titre";
titre.innerHTML = "Titre";
document.body.appendChild(titre);
```

CreateTextNode()

```
const titre = document.createElement("h1");
titre.id = "titre";
const texte = document.createTextNode("Titre");
titre.appendChild(texte);
document.body.appendChild(titre);
```

AppendChild()

```
const titre = document.createElement("h1");
titre.id = "titre";
const texte = document.createTextNode("Titre");
titre.appendChild(texte);
document.body.appendChild(titre);
```

InsertBefore()

```
const titre = document.createElement("h1");
titre.id = "titre";
const texte = document.createTextNode("Titre");
titre.appendChild(texte);
document.body.insertBefore(titre, document.body.firstChild);
```

Comment supprimer des éléments du DOM ?

Pour supprimer des éléments du DOM, on utilise les méthodes suivantes:

- 📌 `removeChild()`
- 📌 `remove()`

RemoveChild()

```
const titre = document.getElementById("titre");
document.body.removeChild(titre);
```

Remove()

```
const titre = document.getElementById("titre");
titre.remove();
```

Comment éviter les erreurs ?

Pour éviter les erreurs, on utilise les méthodes suivantes:

📌 `querySelector()` **et** `querySelectorAll()`

Les evenements

Qu'est-ce qu'un événement ?

Un événement est une action effectuée par l'utilisateur. Les événements sont:

- `click`
- `mouseover`
- `mouseout`
- `keydown`
- `keyup`
- `keypress`
- `load`
- `scroll`
- `submit`

Déclaration d'un événement

```
document.getElementById("bouton").addEventListener("click", function() {
  console.log("Bouton cliqué!");
});
```

Les evenements de souris

Qu'est-ce qu'un événement de souris ?

Un événement de souris est un événement qui se déclenche lorsqu'une action de la souris est effectuée. Les événements de souris sont:

- 📌 `click`
- 📌 `mouseover`
- 📌 `mouseout`

Déclaration d'un événement de souris

```
document.getElementById("bouton").addEventListener("mouseover", function() {
  console.log("Bouton survolé!");
});
```

Les evenements du clavier

Qu'est-ce qu'un événement du clavier ?

Un événement du clavier est un événement qui se déclenche lorsqu'une touche du clavier est pressée. Les événements du clavier sont:

- 📌 `keydown`
- 📌 `keyup`
- 📌 `keypress`

Déclaration d'un événement du clavier

```
document.addEventListener("keydown", function(e) {
  console.log(e.which);
});
```

Les evenements de formulaire

Qu'est-ce qu'un événement de formulaire ?

Un événement de formulaire est un événement qui se déclenche lorsqu'une action est effectuée sur un formulaire. Les événements de formulaire sont:

- 📌 `submit`

Déclaration d'un événement de formulaire

```
document.querySelector("form").addEventListener("submit", function(e) {
  e.preventDefault();
  console.log("Formulaire soumis!");
});
```

Les evenements formulaires

Qu'est-ce qu'un événement de formulaire ?

Un événement de formulaire est un événement qui se déclenche lorsqu'une action est effectuée sur un formulaire. Les événements de formulaire sont:

- 📌 `submit`

Déclaration d'un événement de formulaire

```
document.querySelector("form").addEventListener("submit", function(e) {
  e.preventDefault();
  console.log("Formulaire soumis!");
});
```

Les evenements du navigateur

Qu'est-ce qu'un événement du navigateur ?

Un événement du navigateur est un événement qui se déclenche lorsqu'une action est effectuée sur le navigateur. Les événements du navigateur sont:

- 📌 `load`
- 📌 `resize`
- 📌 `scroll`

Déclaration d'un événement du navigateur

```
window.addEventListener("load", function() {
  console.log("Page chargée!");
});
```

Exercice

Exercice 1:

Integrer une interface dans le navigateur pour le jeu de gestion de village

Exercice 2:

Integrer une interface dans le navigateur pour le jeu de combat avec les personnages

Assynchronisme

Qu'est-ce que l'assynchronisme ?

L'assynchronisme est une technique permettant d'exécuter plusieurs tâches en même temps.
Il permet de réduire le temps d'exécution d'un code.

Quels sont les avantages de l'assynchronisme ?

- 📌 **Il permet d'exécuter plusieurs tâches en même temps.**
- 📌 **Il permet d'exécuter une tâche longue sans bloquer le programme.**

Quels sont les inconvénients de l'assynchronisme ?

- 📌 **Il est plus complexe à mettre en place.**
- 📌 **Il est plus difficile à débugger.**

Quelles sont les méthodes permettant de gérer l'assynchronisme ?

- 📌 **Promesses**
- 📌 **Async / await**

L'assynchronisme avec promesse

Qu'est-ce qu'une promesse ?

Une promesse est un objet qui représente la réussite ou l'échec d'une opération asynchrone. Les promesses sont:

- 📌 `then()`
- 📌 `catch()`
- 📌 `finally()`

Then()

Qu'est-ce que then() ?

then() est une méthode qui permet de gérer la réussite d'une promesse.
Elle prend en paramètre une fonction de rappel qui sera exécutée lorsque la promesse sera résolue.
C'est la fonction resolve() qui est appelée dans le code asynchrone.

Quand then est executee ?

then() est exécutée lorsque la promesse est résolue.
c'est a dire lorsque la fonction resolve() est appelée dans le code asynchrone.

Exemple

```
const promesse = new Promise(function(resolve, reject) {
  setTimeout(function() {
          resolve("Promesse tenue!");
  }, 3000);
});

console.log("before")

promesse.then(function(message) {
        console.log(message);
});

console.log("after")

setInterval(function(){
  console.log("log")
},500)
```

Catch()

Qu'est-ce que catch() ?

catch() est une méthode qui permet de gérer l'échec d'une promesse.
Elle prend en paramètre une fonction de rappel qui sera exécutée lorsque la promesse sera rejetée.
C'est la fonction reject() qui est appelée dans le code asynchrone.

Quand catch est executee ?

catch() est exécutée lorsque la promesse est rejetée.
C'est a dire lorsque la fonction reject() est appelée dans le code asynchrone.

Exemple

```
const promesse = new Promise(function(resolve, reject) {
  setTimeout(function() {
          reject("Ho mince!");
  }, 3000);
});

console.log("before")

promesse.then(function(message) {
        console.log(message);
}).catch(function(error) {
        console.log(error);
});

console.log("after")

setInterval(function(){
  console.log("log")
},500)
```

Finally()

Qu'est-ce que finally() ?

finally() est une méthode qui permet de gérer la fin d'une promesse.
Cela permet d'exécuter du code une fois que la promesse a été traitée, quel que soit le résultat.
On évite ainsi de dupliquer du code entre les gestionnaires then() et catch()

Exemple

```
const promise = new Promise((resolve, reject) => {
  const random = Math.random();
  if (random < 0.5) {
        resolve();
  } else {
        reject();
  }
});

promise
  .then(data => console.log(data))
  .catch(err => console.log(err))
  .finally(() => console.log('Done'));
```

Async / await

Qu'est-ce qu'async / await ?

async / await est une syntaxe permettant de simplifier l'utilisation des promesses.

Déclaration d'async / await

```
async function direBonjour(prenom) {
  return new Promise(function(resolve, reject) {
    const message = "Bonjour, " + prenom + " !";
    resolve(message);
  });
}

async function direBonjourAvecAsync(prenom) {
  const message = await direBonjour(prenom);
  console.log(message); // Bonjour, John !
}

direBonjourAvecAsync("John");
```

AJAX avec axios

Qu'est-ce que AJAX ?

AJAX est une technique permettant de faire des requêtes asynchrones. C'est à dire que le navigateur peut continuer à exécuter du code pendant que la requête est en cours.
AJAX permet de récupérer des données sans recharger la page.

Qu'est-ce que axios ?

Axios est une bibliothèque JavaScript permettant de faire des requêtes HTTP.

Exemple d'AJAX avec axios

```
axios.get("https://api.github.com/users")
  .then(function(response) {
    console.log(response.data);
  })
  .catch(function(error) {
    console.log(error);
  });
```

Les websocket avec socket.io

Qu'est-ce que socket.io ?

Socket.io est une bibliothèque JavaScript permettant de créer des applications temps réel.
C'est à dire que les données sont envoyées en temps réel sans recharger la page.
Un socket est une connexion entre un client et un serveur.
Par exemple, un chat est une application temps réel car les messages sont envoyés en temps réel sans recharger la page.

Les événements socket.io et leurs fonctionnement

Les événements socket.io sont des événements qui permettent de communiquer entre le client et le serveur.
Ils permettent de gérer les connexions, les déconnexions, les messages, etc.
Vous pouvez definir vos propres événements.

Fonctionnement des événements socket.io

- 📌 Le serveur émet un événement
- 📌 Le client écoute un événement
- 📌 Le client émet un événement
- 📌 Le serveur écoute un événement

On()

```
io.on("connection", function(socket) {
  socket.on("message", function(message) {
    console.log(message);
  });
});
```

Emit()

```
io.on("connection", function(socket) {
  socket.emit("message", "Bienvenue sur le chat");
});
```

TP Pong avec socket.io

Les tips and tricks

Les Proxy

Un proxy est un objet qui permet de contrôler l'accès à un autre objet.
C'est à dire que vous pouvez intercepter les appels à un objet et les modifier avant de les transmettre à l'objet cible.

Avantages des proxy

- 📌 Le proxy permet de contrôler l'accès aux propriétés d'un objet
- 📌 Le proxy permet de contrôler l'accès aux méthodes d'un objet
- 📌 Le proxy permet de contrôler l'accès aux fonctions d'un objet

Inconvénients des proxy

Exemple de proxy

```
const objet = {
  prenom: "John",
  nom: "Doe",
  age: 30
};

const proxy = new Proxy(objet, {
  get: function(target, property) {
    return target[property];
  },
  set: function(target, property, value) {
    target[property] = value;
  }
});

proxy.prenom = "Jane";
console.log(proxy.prenom); // Jane
```

Les jump table

Qu'est-ce qu'une jump table ?

Une jump table est un tableau qui permet de stocker des fonctions.

Elle permet de stocker des fonctions dans un tableau et de les appeler par leur index.

Ce mécanisme permet de gagner en performance.

Elle est utilisée dans les langages compilés comme C, C++, Java, etc.

Avantages des jump table

- 📌 Les jump table permettent de gagner en performance
- 📌 Les jump table permettent de gagner en lisibilité
- 📌 Les jump table permettent de gagner en maintenabilité
- 📌 Les jump table permettent de simplifier l'écriture de code

Inconvénients des jump table

📌 **Les jump table sont difficiles à comprendre**

Exemple de jump table

```
const direBonjour = function() {
  console.log("Bonjour");
};

const direAuRevoir = function() {
  console.log("Au revoir");
};

const jumpTable = [direBonjour, direAuRevoir];

jumpTable[0](); // Bonjour
jumpTable[1](); // Au revoir
```

Les decorators

Qu'est-ce qu'un decorator ?

Un decorator est une fonction qui permet d'ajouter des fonctionnalités à une autre fonction.

Avantages des decorators

- 📌 Les decorators permettent d'ajouter des fonctionnalités à une fonction
- 📌 Les decorators permettent de gagner en lisibilité
- 📌 Les decorators permettent de gagner en maintenabilité
- 📌 Les decorators permettent de simplifier l'écriture de code

Inconvénients des decorators

📌 **Les decorateurs necessitent une mise en place.**

Exemple de decorator

```
function direBonjour(target) {
  target.prototype.direBonjour = function() {
    console.log("Bonjour");
  };
}

@direBonjour
class Personne {
  constructor(prenom) {
    this.prenom = prenom;
  }
}

const john = new Personne("John");
john.direBonjour(); // Bonjour
```