JavaScript

Table des matières

Algorithmie

Les variables

En JavaScript, il est possible de créer des conteneurs à valeur, afin de stocker jusqu'à la fin de l'exécution du code une valeur, dans le but de la réutiliser.

On appelle ces conteneurs « **variables** », et elles doivent être déclarées comme ceci :

```
let exemple = "toto"
```

Ici, nous avons créé la variable « **exemple** », et stockés la valeur « **toto** » (chaîne de caractères) à l'intérieur : le placement de la valeur se fait à l'aide de l'opération d'affectation « **=** ».

Lors de la première utilisation, il faut toujours déclarer leur portée :

- « **let** » permet d'utiliser la variable dans son bloc uniquement
- « **var** » permet d'utiliser la variable partout (variable globale)
- « **const** » permet de n'affecter une valeur à la variable une seule et unique fois

Types de valeur

Les différents types de valeurs en JavaScript sont les suivants :

- **Number** (nombres entiers, nombres réels) `1`
- **Boolean** (VRAI | FAUX) `true | false`
- **String** (chaîne de caractères) `"Hello world!"`
- **Array** (tableaux) `[1, 2, 3]`
- **Object** (objets) `{nom: "John", prenom: "Doe"}`
- **Function** (fonctions |procédures) `() => {console.log('Hello world!')}`

Opérateur

Affectation

L'affectation est le fait d'associer ou de modifier une valeur à une variable.
On retrouve différents opérateurs d'affectation :

- « **=** » Affectation de la valeur à la variable
- « **+=** » Ajout de la valeur à la variable (addition pour Number | concaténation pour String)
- « **-=** » Soustraction de la valeur à la variable
- « ***=** » Multiplication de la variable
- « **/=** » Division de la variable
- « **%=** » Affectation du reste de l'opération de division
- « ****=** » Affectation du résultat de la variable à la puissance du nombre spécifié

Arithmétique

Différentes opérations peuvent être réalisées en JavaScript, les opérateurs mis à disposition sont les suivants :

- « **%** » A % B renvoie le reste de la division de A par B
- « **++** » A++ renvoie A et ajoute 1 ensuite | ++A ajoute 1 à A puis le renvoie
- « **--** » A-- renvoie A et soustrait 1 ensuite | --A soustrait 1 à A puis le renvoie
- « **+** » A + B renvoie le résultat de l'addition de A et B
- « **-** » A – B renvoie le résultat de la soustraction de A et de B
- « **/** » A / B renvoie le résultat de la division de A par B
- « ***** » A * B renvoie le résultat de la division de A par B
- « ****** » A ** B renvoie le résultat de A puissance B

Comparaison

Les opérateurs de comparaison servent à comparer deux valeurs, ils renvoient « **true** » si la condition est vraie, sinon ils renvoient « **false** ».

En JavaScript on retrouve les opérateurs de comparaison suivants :

- « **==** » Teste l'égalité simple de deux valeurs (valeurs égales)
- « **!=** » Teste l'inégalité simple de deux valeurs (valeurs inégales)
- « **===** » Teste l'égalité stricte de deux valeurs (valeurs et types égaux)
- « **!==** » Teste l'inégalité stricte de deux valeurs (valeurs et types inégaux)
- « **<** » Teste l'infériorité stricte d'une valeur par rapport à une autre
- « **>** » Teste la supériorité stricte d'une valeur par rapport à une autre
- « **<=** » Teste l'infériorité ou l'égalité d'une valeur par rapport à une autre
- « **>=** » Teste la supériorité ou l'égalité d'une valeur par rapport à une autre

Logique

Les opérateurs logiques servent à composer des conditions complexes (plusieurs conditions fusionnées). Ils renvoient TRUE si leur logique est respectée, sinon FALSE.

En JavaScript on retrouve :

- « **&&** » Les deux conditions l'entourant doivent être TRUE (A && B)
- « **||** » A moins l'une des deux conditions doit être TRUE (A || B)
- « **!** » Renvoie l'opposé du résultat de la condition | du booléen (!A)

instanceOf

L'opérateur **instanceOf** permet de tester si la valeur a été générée à partir d'un constructeur de classe. Renvoie « **true** » si la valeur appartient à la classe spécifiée, sinon « **false** ».

*Ex : « now instanceof Date » renverra « **true** » si now = new Date()*

typeof

A l'instar de **instanceof** vu précédemment, **typeof** ne teste pas la valeur, mais renvoie simplement le type à laquelle elle appartient.

*Ex : **typeof** 1000 renvoie « **number** »*

Structures de contrôle

Divers outils sont mis à disposition pour que vous puissiez gérer au maximum le comportement de votre code. Parmi ces outils, on trouve les structures de contrôle : des éléments de syntaxe permettant de modifier l'exécution du code « **classique** » du haut vers le bas.

Conditionnement

SI, SINON

La structure **IF** permet de conditionner l'exécution d'un bloc de code.

On vient tester si une condition est vraie, et si c'est le cas on vient exécuter le code spécifié. Il est possible de spécifier le code à exécuter dans le cas où la condition est fausse.

```
if(CONDITION) {
  console.log("Condition vraie")
}
else {
  console.log("Condition fausse")
}
```

Il est possible d'utiliser des « **else if** » : combinaison d'un if et d'une **else**, qui permet de tester une nouvelle condition si la première a été fausse.

```
if (CONDITION === 1) {
  console.log('CONDITION est égal à 1');
} else if (CONDITION === 2) {
  console.log('CONDITION est égal à 2');
} else if (CONDITION === 3) {
  console.log('CONDITION est égal à 3');
} else {
  console.log('CONDITION est supérieur à 3');
}
```

Ici, toutes les conditions vont être testées jusqu'à trouver celle qui sera vraie (si aucune n'est vraie alors le « **else** » s'exécutera).

SWITCH

Un **switch** permet de programmer l'exécution de code en fonction de l'état d'une variable ou d'une expression.

```
let variable = 3;
switch (variable) {
  case 1:
    console.log('La variable vaut 1');
    break;
  case 2:
    console.log('La variable vaut 2');
    break;
  case 3:
    console.log('La variable vaut 3');
    break;
  default:
    console.log('La variable vaut ...?');
    break;
}
```

Dans l'exemple ci-dessus, en fonction de la valeur de la variable « **variable** », un cas différent du switch sera déclenché (si un cas se déclenche, les autres cas passent à la trappe).

Si vous utilisez un switch, pensez bien à toujours mettre un cas « **default** » au cas où aucun des cas ne soit vrai (même principe que le « **else** »).

Ternaire

Un ternaire est une version minifiée d'un « **if** ».

Son rôle est de retourner telle ou telle valeur en fonction d'une condition, le tout de manière ultra compacte.

```
CONDITION ? 'La condition est vraie' : 'La condition est fausse';
```

Ici, notre ternaire retourne une chaine de caractère qui va changer en fonction du retour de la condition.

```
const ternaireFonction = () => {
  if (CONDITION) return 'La condition est vraie';
  else return 'La condition est fausse';
};
```

Si on devait reproduire le comportement d'un ternaire avec une fonction, on l'écrirait comme la fonction ci-dessus.

Boucles

Parfois, il est utile de répéter des portions de code, en algorithmie en général il est possible de le faire avec les structures de contrôle **WHILE** et **FOR**, qui vont boucler chacune à leur manière, leur point d'arrêt étant paramétré au travers d'une condition à respecter.

WHILE

Les boucles « **while** » (tant que... en français) sont des structures permettant de répéter en boucle une portion de code, tant que sa condition est respectée (vraie).

```
let variable = 0;
while (variable !== 3) {
  console.log('Valeur de variable: ' + variable);
  ++variable;
}
```

Ici, variable prends +1 à chaque tour, ainsi, au troisième tour, la condition ne sera plus respectée, et la boucle s'arrêtera.

FOR

Une boucle « **for** » est une boucle qui va boucler un nombre de fois prédéfini.

A chaque tout son « **indice** » (i,j,k en général) va être changé, la plupart du temps en y rajoutant 1, mais on peut très bien le multiplier par 2, soustraire 1… etc.

```
for (let i = 0; i < 10; ++i) {
  console.log('Tour n°' + i);
}
```

Fonctions

En algorithmie en général, il est possible d'enregistrer certaines portions de code dans le but de les utiliser à plusieurs endroits dans le code plutôt que de copier-coller le même code plusieurs fois.

On distingue deux types de fonctions :

- Les **procédures**, des fonctions qui exécutent la portion de code associé sans avoir besoin de demander une quelconque donnée supplémentaire.
- Les fonctions « **classiques** », qui vont exécuter la portion de code associée en leur fournissant une valeur récupérée lors de son appel, on appelle cette valeur (il peut y en avoir plusieurs) « **paramètre** ».

```
function travailler() {
  console.log('Je travaille');
}

function travailler(travail) {
  console.log('Je travaille dans le domaine: ' + travail);
}
```

Ici, la fonction supérieure est une procédure car elle ne dépend d'aucune donnée passée lors de son utilisation.

La fonction inférieure quant à elle est une fonction « **classique** », elle affiche le paramètre « **travail** » qui lui sera passé lors de l'appel.

```
travaillerProcedure();
travaillerFonction('Ingénieur logiciel');
```

On utilise les fonctions (qui sont des modèles) en les « **appelant** » : on marque le nom de la fonction et on y colle des parenthèses, qui, contiendront les potentiels paramètres, sinon vide.

ATTENTION : si vous appelez la fonction sans les parenthèses, vous obtiendrez la valeur de celle-ci : un objet contenant tout le bloc de code ; cela peut être utile lorsque vous utiliserez les fonctions en tant que call-back (fonction appelée par une autre fonction).

Type String

Le type **String** est caractérisé par les guillemets qui l'entourent.

On y retrouve plusieurs fonctions utilisables afin de manipuler ses données :

- **String.trimStart()** & **String.trimEnd()** & **String.trim()**
  Enlève de la chaine de caractère les caractères « **espace** » du début & de la fin.
- **String.toUpperCase()**
  Transforme la chaine de caractère en majuscule.
- **String.toLowerCase()**
  Transforme la chaine de caractères en minuscule.
- **String.substring(start, end)**
  Retourne une chaine de caractère correspondant aux caractères entre l'indice « **start** » et l'indice « **end** ».
- **String.startsWith(chaine)** & **String.endsWith(chaine)**
  Retourne un booléen qui dit si la chaine de caractère commence ou termine par la chaine passée en paramètre.
- **String.split(separateur)**
  Retourne un tableau contenant la découpe de la chaine de caractères en fonction du séparateur passé en paramètres.
- **String.slice(start, end)**
  Retourne une nouvelle string contenant les caractères a partir de l'indice start jusqu'à l'indice end.
- **String.replace(car_a_remplacer, car_remplaçant)**
  Remplace le premier caractère trouvé correspondant au 1er paramètre par le 2è paramètre.
- **String.replaceAll(car_a_remplacer, car_remplaçant)**
  Remplace tous les caractères trouvés correspondants au 1er paramètre par le 2è paramètre.
- **String.repeat(count)**
  Retourne une chaine de caractères contenant la répétition de la chaine en question « **count** » (paramètre) fois.
- **String.indexOf(modèle)** & **String.lastIndexOf(modèle)**
  Retourne respectivement l'index de la première, et dernière occurrence correspondant au modèle(paramètre).
- **String.includes(modèle)**
  Retourne un booléen indiquant si la chaine de caractères contient le modèle (paramètre).
- **String.At(index)**
  Retourne le caractère se trouve à l'indice « index » (paramètre).

Type Array

Le type **Array** permet de gérer des données mixtes, en les stockant de manière ordonnée dans un tableau, où chaque élément possède un indice (sa position) permettant de récupérer son élément. On récupère une valeur en faisant :
tableau[indice] => retourne l'élément à l'indice donné.

De nombreuses fonctions existent pour manipuler ses données :

- **Array.concat(autre_tableau)**
  Retourne la fusion des deux tableaux l'un après l'autre dans un nouveau tableau.
- **Array.every(fonction_de_test)**
  Retourne un booléen indiquant si tous les éléments du tableau retournent « **true** » à la fonction de test.
- **Array.some(fonction_de_test)**
  Retourne un booléen indiquant si au moins un élément du tableau retourne « **true** » à la fonction de test.
- **Array.fill()**
  Rempli le tableau de différentes manières en fonction des paramètres.
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/fill
- **Array.filter(fonction_de_test)**
  Retourne un tableau contenant les éléments ayant retournés « **true** » à la fonction de test.
- **Array.find(fonction_de_test)**
  Retourne le premier élément du tableau ayant retourné **true** à la fonction de test.
- **Array.findIndex(fonction_de_test)**
  Retourne l'index du premier élément ayant retourné « **true** » à la fonction de test.
- **Array.flat(profondeur)**
  Si le tableau contient des tableaux, alors la fonction va retourner un nouveau tableau avec les sous tableaux ayant été « **dissous** » et leurs éléments fusionnés avec le tableau d'origine.
  Possibilité de spécifier une profondeur.
- **Array.forEach(fonction_de_traitement)**
  Applique la fonction de traitement à tous les éléments du tableau.
- **Array.includes(valeur)**
  Retourne un booléen indiquant si le tableau contient la valeur passée en paramètre.
- **Array.indexOf(valeur)**
  Si « **valeur** » existe dans le tableau, alors retourne l'index du premier élément correspondant, sinon retourne -1.
- **Array.join()**
  Retourne une string contenant tous les éléments du tableau à la suite.
- **Array.lastIndexOf(valeur)**
  Si « **valeur** » existe dans le tableau, alors retourne l'index du dernier élément correspondant, sinon retourne -1.

- **Array.map(fonction_de_traitement)**
  Retourne un tableau contenant tous les retours de la fonction de traitement pour chaque élément.

- **Array.pop()**
  Retire le dernier élément du tableau, et retourne sa valeur.

- **Array.shift()**
  Retire le premier élément du tableau, et retourne sa valeur.

- **Array.push(élément)**
  Ajoute « **élément** » à la fin du tableau, et retourne la nouvelle longueur du tableau.

- **Array.unshift(élément)**
  Ajoute « **élément** » au début du tableau, et retourne la nouvelle longueur du tableau.

- **Array.reverse()**
  Inverse l'ordre des éléments du tableau (premier => dernier | dernier => premier)

- **Array.slice()**
  Retourne une portion du tableau, en fonction des paramètres passés.
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice

- **Array.sort()**
  Retourne un nouveau tableau correspondant au tableau originel trié.
  Par défaut, trie en fonction de l'ordre alphanumérique des éléments, mais il est possible de passer en paramètres une fonction de comparaison pour permettre un nouveau tri. Exemple :

```
const croissant = (a, b) => {
  if (a < b) return 1;
  if (a > b) return -1;
  return 0;
};
```

- **Array.splice()**
  Supprime des éléments d'un tableau, la quantité, la position, varient en fonction des paramètres fournis à la fonction.
  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/splice

Type Object

Le type **Object** permet d'organiser des données en spécifiant un « **nom** » à cette dernière. Ainsi les données sont gérées sous forme CLE : **VALEUR** où **CLE** est le nom de la donnée, et VALEUR sa valeur.

On récupère la donnée en faisant :
objet.CLE => retourne la VALEUR

Il existe plusieurs fonctions pour manipuler le type Object :

- Object.values(objet)
  Retourne un tableau contenant toutes les valeurs de l'objet.
- Object.keys(objet)
  Retourne un tableau contenant toutes les clés de l'objet.
- Object.entries(objet)
  Retourne un tableau, contenant pour chaque entrée de l'objet, un tableau avec pour 1er élément le nom de la cle, et en 2e élément la valeur.
- Object.fromEntries(entrées)
  Retourne un objet depuis un tableau correspondant au format de retour de Object.entries().
- Object.defineProperty(objet, nomCle, infosObjet)
  Ajoute une entrée à l'objet, en y ajoutant les informations récupérées dans les paramètres ; pour plus d'informations sur le format des données à fournir : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperty
- Object.defineProperties(objet, propriétés)
  Ajoute plusieurs étrées à l'objet, en y ajoutant les informations récupérées dans les paramètres ; plus d'infos : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/defineProperties
- Object.freeze(objet)
  Gèle l'objet en empêchant toute modification ultérieure sur ce dernier.
- Object.isFrozen(objet)
  Retourne un booléen indiquant si l'objet est actuellement gelé.
- Object.seal(objet)
  Scelle l'objet, empêchant ainsi toute suppression de données de ce dernier.
- Object.isSealed(objet)
  Retourne un booléen indiquant si l'objet est actuellement scellé.
- Object.preventExtensions(objet)
  Empêche toute extension de l'objet en enlevant la possibilité de rajouter des entrées dans ce dernier.
- Object.IsExtensible(objet)
  Retourne un booléen indiquant si l'objet est actuellement extensible.

Type Number

Le type **Number** regroupe tous toutes les valeurs numériques (nombres entiers, nombres réels).
Il est possible d'effectuer des opérations mathématiques basiques sur ces derniers (addition, soustraction…), mais aussi des opérations plus complexes au travers de fonctions mises à disposition :

- **isFinite(number)**
  Retourne un booléen indiquant si le nombre est un nombre fini.
- **Number.isInteger(number)**
  retourne un booléen indiquant si le nombre est un nombre entier (pas de virgule)
- **Number.isSafeInterger(number)**
  Retourne un booléen indiquant si le nombre est « sûr » (compris entre -2^53 -1 et 2^53 + 1).
- **Number.isNaN(number)**
  Retourne un booléen indiquant si la valeur du nombre est NaN.
- **parseFloat(stringFloatNumber)**
  Retourne stringFloatNumber (réel sous forme de chaine de caractères) sous forme de Number.
- **parseInt(stringIntNumber)**
  Retourne stringIntNumber (entier sous forme de chaine de caractères) sous forme de Number.
- **nombre.toExponential(exponentielle)**
  retournes une chaine de caractères représentant le nombre élevé à l'exponentielle passée en paramètres.
- **nombre.toFixed(nombre_de_chiffres_apres_virgule)**
  Retournes une chaine de caractères représentant le nombre avec le nombre de chiffres après la virgule spécifié (arrondi).
- **nombre.toString()**
  Retourne le nombre en question sous forme de chaine de caractères.

API Fetch

JavaScript étant un langage assez complet, il est possible de l'utiliser pour faire des interactions réseaux.

Pendant longtemps, des librairies tierces étaient utilisées car plus pratiques et faciles à utiliser (axios, jquery…), mais aujourd'hui il est possible d'utiliser « **fetch()** », une api du langage servant à faire des requêtes de tous types, et avec la possibilité de modifier pas mal de choses grâce aux objets (corps de la requêtes, options… etc.).

Fetch s'utilise comme ceci :

```
1   fetch('https://mon-api.com')
2   .then(request => request.json())
3   .then(data => {
4       //traitement de la donnée reçue ici
5   })
```

1. Exécution de la requête (ici requête simple de type **GET**, pas de paramètres)
2. Fonction asynchrone chainée, se déclenchant lors de la réception de la requête, et retournant le résultat de la requête filtré de toutes informations inutiles (il ne reste que la donnée demandée)
3. Fonction asynchrone chainée, se déclenchant lorsque la requête a été filtrée, et récupère le résultat de l'opération dans son paramètre « **data** » (le nom importe peu, c'est un paramètre), permettant ainsi d'effectuer un traitement sur la donnée reçue.

Il est possible de gérer une éventuelle erreur en rajoutant un

```
.catch((err) => {
  // traitement de l'erreur
});
```

Tout ceci est évidemment asynchrone, étant donné le caractère imprévisible d'une requête réseau (temps de réception, erreur sur le réseau…).

Rappel : un code asynchrone est un code qui sort du cadre d'exécution « **classique** » (de haut en bas) en s'exécutant en parallèle du code (d'où la nécessité de passer des procédures qui s'exécuteront à certains moments) et ainsi permettant au reste du code de s'exécuter paisiblement.

Async | Await

Les dernières versions de Javascript ont introduites des fonctionnalités permettant de gérer l'asynchrone de manière bien plus simple qu'auparavant : « **async** » et « **await** ».

Il est possible de spécifier qu'une fonction est asynchrone (elle sera alors exécutée en parallèle du reste du code et non plus en impératif) avec le préfixe « **async** ».

Ainsi, dans une fonction dite asynchrone, il devient possible de spécifier avec le préfixe « **await** », si un code initialement asynchrone (ex : Fetch) doit être exécuté en tant que tel ou en synchrone (on attend la fin de son exécution pour exécuter la suite).

```
const asyncFunction = async () => {
  await fetch('https://mon-api.com')
    .then((request) => request.json())
    .then((data) => {
      //traitement de la donnée reçue ici
    })
    .catch((err) => {
      // traitement de l'erreur
    });

    //Suite du programme (exécuté apres la fin de la requête)
};
```

Programmation Orientée Objet

Introduction

JavaScript, par défaut, est un langage orienté objet.
En effet, beaucoup de fonctions que vous utilisez surement déjà sont issues d'un système de classes et d'héritage.

Ex : « **[1, 2, 3].sort()** » on exécute la fonction sort() qui trie le tableau depuis lequel il est appelé.

Dans l'exemple ci-dessus, [1, 2, 3] est ce que l'on appelle un prototype, une « **instance** » de la classe « **Array** », utilisée pour manipuler les tableaux.

Cette classe « **Array** » agit comme un modèle, lorsque elle a été créée, diverses propriétés lui ont été données, ainsi que des fonctions, comme par exemple la fonction sort().

Lorsque vous créez un tableau, que ce soit via le constructeur de la classe (peu de chances), ou directement en définissant un tableau à la volée (comme l'exemple), vous aurez la possibilité d'accéder à toutes les fonctionnalités qui sont disponibles pour les tableaux, car ils appartiennent tous à cette même classe : « **Array** ».

Ces fonctionnalités ont toutes un comportement qui leur est propre : certaines vont transformer le tableau sur lequel elles sont appelées, d'autres vont simplement retourner quelque chose à partir de ce dernier.

Au-delà d'utiliser sans vous en rendre compte ce système de classes, vous pouvez en créer vous-même.

Cette nouvelle approche de la programmation ne révolutionne rien, n'apporte rien, si ce n'est une meilleure compréhension du code potentielle : c'est ce que l'on appelle du sucre syntaxique.

Classes personnalisées

Il est possible de définir une nouvelle classe grâce au mot-clé « **class** ».

```
class personne {

}
```

A partir d'ici, vous devrez créer ce que l'on appelle un « **constructeur** » avec le mot-clé « **constructor** ».

Ce dernier agira comme la fonction qui se lancera lorsqu'une instance de votre classe est créée : il aura pour devoir d'initialiser les données qui caractérisent votre classe.

```
class personne {
  constructor(nom, prenom, taille) {
    this.nom = nom;
    this.prenom = prenom;
    this.taille = taille;
    this.dateNaissance = new Date().toDateString();
  }
}
```

Comme illustré ci-dessus, la fonction de construction prend tout comme une fonction classique des paramètres : ce seront les données que l'utilisateur devra renseigner afin de créer l'instance.

Vous remarquerez aussi l'utilisation du mot-clé « **this** » : il représente la ou les futures instances créées à partir de ce modèle, ainsi il permet de différencier les données utilisées dans le modèle (comme les paramètres du constructeur) des « **attributs** » qui seront gardés par notre instance.

Ex : « **this.nom** » représente l'attribut « **nom** » de l'instance alors que « **nom** » représente le paramètre de la fonction de construction.

On peut très bien créer des attributs à partir d'informations n'étant pas renseignées par l'utilisateur, ici on stocke dans l'attribut « **dateNaissance** » la date au moment du lancement de la fonction de construction, et donc au moment de sa création.

Méthodes

Il est désormais possible de créer des méthodes (fonctions) qui seront utilisables par nos futures instances basées sur la classe « **personne** ».

```
sePresenter = () => {
  console.log(`Bonjour, je m'appelle ${this.prenom} ${this.nom} et je mesure ${this.taille}.`)
}
```

La méthode doit être déclarée dans le corps de la classe en question.
Attention de bien utiliser « **this** » pour récupérer les attributs, car il permet d'indiquer qu'il s'agira bien de l'attribut de la future instance (la classe n'étant qu'un modèle).

Cette méthode sera donc renseignée par les infos de l'instance qui l'appelle.

Maintenant, imaginons que je veuille comparer la taille de deux instances de la classe personne, on pourrait faire appel à la méthode de l'un des deux instances, mais ce ne serait pas très logique, laquelle des deux serait la plus légitime à être appelée ?

C'est pour ce genre de cas qu'existe les méthodes dites « **statiques** », des méthodes qui n'appartiennent non pas aux instances comme l'est la fonction « **sePresenter()** » mais à la classe en elle-même.

```
15    static comparerTaille = (personne1, personne2) => {
16      if (personne1.taille < personne2.taille)
17        return console.log(
18          personne2.prenom + ' est plus grand que ' + personne1.prenom
19        );
20      if (personne1.taille > personne2.taille)
21        return console.log(
22          personne1.prenom + ' est plus grand que ' + personne2.prenom
23        );
24      console.log(
25        personne1.prenom + ' et ' + personne2.prenom + ' font la même taille.'
26      );
27    };
```

Il suffit donc d'ajouter le préfixe « **static** » lors de la déclaration de la fonction pour faire de cette dernière une fonction statique.

Il est désormais possible d'appeler cette fonction en lui passant deux instances de la classe personne en paramètres, et elle affichera dans la console lequel des deux a la plus grande taille.

Création d'une instance

Une fois la classe définie, il devient alors possible de « **l'instancier** », c'est-à-dire de créer un prototype à partir de ce dernier, et donc un objet « **concret** ».

```
10    let Paul = new personne('Dupont', 'Paul', 160);
```

Ici, une instance de la classe personne a été créée, et les paramètres passés dans les parenthèses seront, dans le même ordre, passés au constructeur de la classe.

L'instance est désormais stockée dans la variable Paul, qui va servir pour appeler ses fonctions (dites « **méthodes** » dans le cadre d'un objet) ou ses attributs.

```
38    console.log(Paul.dateNaissance);
```

```
31    Paul.sePresenter();
```

Pour appeler une fonction statique, il faut tout simplement partir de la classe en elle-même au lieu d'une instance.

```
30    let Paul = new personne('Dupont', 'Paul', 160);
31    Paul.sePresenter();
32
33    let Jean = new personne('Dupont', 'Jean', 180);
34    Jean.sePresenter()
35
36    personne.comparerTaille(Paul, Jean);
```

Héritage

L'héritage, c'est le fait de créer une classe basée sur un classe déjà existante.

Ainsi on va indiquer en créant notre nouvelle classe de quelle classe elle va hériter, cela aura pour effet de transmettre tous les attributs et méthodes du parent à notre classe.

Pour créer notre classe héritée, il faut utiliser la même syntaxe qu'une classe normale, en rajoutant à la fin de la ligne « **extends** » suivi du nom du parent.

```
38  class sportif extends personne {
39    constructor(nom, prenom, taille, sport) {
40      super(nom, prenom, taille);
41      this.sport = sport;
42      this.medailles = 0;
43    }
44  }
```

Dans l'exemple, j'ai créé la classe « **sportif** », qui n'est en fait qu'une « **personne** » mais avec deux attributs en plus « **sport** » et « **medailles** » ; « **sport** » contient le nom du sport que la personne pratique, et « **medailles** » contient le nombre de médailles qu'elle a amassée.

On peut voir une nouveauté : l'utilisation de la fonction « **super()** » : elle ne s'utilise que dans le constructeur d'une classe héritée et permet d'appeler le constructeur du parent, il lui faut donc les paramètres du parents entre les parenthèses (il faut quand même passer par le constructeur de la classe héritée pour demander les paramètres en question : voir ligne 39 du code ci-dessus).

Ainsi on peut rajouter les attributs souhaités, ainsi que des nouvelles fonctions accessibles uniquement depuis cette classe héritée.

```
38  class sportif extends personne {
39    constructor(nom, prenom, taille, sport) {
40      super(nom, prenom, taille);
41      this.sport = sport;
42      this.medailles = 0;
43    }
44
45    gagnerMedaille = () => {
46      if (Math.ceil(Math.random() * 2) > 1) this.medailles++;
47    };
48
49    afficherMedailles = () => {
50      console.log("j'ai actuellement " + this.medailles + ' médailles !');
51    };
52  }
```

Si je créé une instance de « **sportif** » je constaterai qu'elle a accès à ses attributs ainsi que ceux de son parent, de plus que ses méthodes et celles de ses parents.

```
54  let rafael = new sportif('Nadal', 'Rafaël', 180, 'Tennis');
55  rafael.afficherMedailles();
56  rafael.sePresenter();
```