POO en JavaScript

[1. POO et POOP 2](#_Toc106229323)

[1.1. POO 2](#_Toc106229324)

[1.1.1. Concepts fondamentaux de la programmation orientée objet 3](#_Toc106229325)

[1.1.1.1. Classe 3](#_Toc106229326)

[1.1.1.1.1. Membres de classe 3](#_Toc106229327)

[1.1.1.1.2. Instanciation d’une classe 4](#_Toc106229328)

[1.1.1.1.3. Représentation UML d’une classe 4](#_Toc106229329)

[1.1.1.2. Objet 4](#_Toc106229330)

[1.1.1.2.1. Le concept d’objet 5](#_Toc106229331)

[1.1.1.2.2. Représentation UML d’un objet 5](#_Toc106229332)

[1.1.1.3. Encapsulation 6](#_Toc106229333)

[1.1.1.3.1. Définition de l’encapsulation 6](#_Toc106229334)

[1.1.1.3.2. Schéma de l’encapsulation 6](#_Toc106229335)

[1.1.1.3.3. Niveaux de visibilité des éléments de la classe 7](#_Toc106229336)

[1.1.1.4. Héritage 8](#_Toc106229337)

[1.1.1.4.1. Définition de l’héritage 8](#_Toc106229338)

[1.1.1.4.2. Représentation UML de l’héritage 8](#_Toc106229339)

[1.1.1.5. Polymorphisme 9](#_Toc106229340)

[1.1.1.5.1. Définition du polymorphisme 9](#_Toc106229341)

[1.1.1.5.2. Caractéristiques du polymorphisme 10](#_Toc106229342)

[1.1.1.5.3. Polymorphisme versus héritage 11](#_Toc106229343)

[1.1.1.5.4. Représentation UML du polymorphisme 11](#_Toc106229344)

[1.1.1.6. Classe abstraite 11](#_Toc106229345)

[1.1.1.6.1. Définition d’une classe abstraite 11](#_Toc106229346)

[1.1.1.6.2. Intérêts de la classe abstraite 12](#_Toc106229347)

[1.1.1.6.3. Représentation UML d’une classe abstraite 12](#_Toc106229348)

[1.1.1.7. Interface 13](#_Toc106229349)

[1.1.1.7.1. Définition d’une interface 13](#_Toc106229350)

[1.1.1.7.2. Le principe d’obligation contractuelle 14](#_Toc106229351)

[1.1.1.7.3. Interface versus classe abstraite 14](#_Toc106229352)

[1.1.1.7.4. Représentation UML d’une interface 14](#_Toc106229353)

[1.1.1.8. Méthodes statiques 15](#_Toc106229354)

[1.1.1.8.1. Définition d’une méthode statique 15](#_Toc106229355)

[1.1.1.8.2. Représentation UML d’une méthode statique 15](#_Toc106229356)

[1.2. POOP 15](#_Toc106229357)

[1.2.1. Définition de la Programmation orientée prototype 15](#_Toc106229358)

[1.2.2. Comparaison des modèles à classes et à prototypes 15](#_Toc106229359)

[1.2.3. Intérêt de la programmation orientée prototype 16](#_Toc106229360)

[2. Les classes en JavaScript 16](#_Toc106229361)

[2.1. Sucre syntaxique 16](#_Toc106229362)

[2.2. Présentation des classes 16](#_Toc106229363)

[2.3. Définissez des objets et leurs attributs avec des classes 17](#_Toc106229364)

[2.3.1. Découvrez les objets 17](#_Toc106229365)

[2.3.2. La notion de classe 18](#_Toc106229366)

[2.3.2.1. Exemple de classe 18](#_Toc106229367)

[2.3.2.2. Instance d’une classe 19](#_Toc106229368)

[2.3.2.3. Pratiquez : créez un objet 19](#_Toc106229369)

[2.3.2.3.1. Solution 19](#_Toc106229370)

[2.3.2.4. Accédez aux données d'un objet avec la notation pointée 21](#_Toc106229371)

[2.3.2.4.1. Pratiquez : récupérez des valeurs depuis un Object 21](#_Toc106229372)

[2.3.2.4.1.1. Solution 21](#_Toc106229373)

[2.3.3. Manipulez des classes 22](#_Toc106229374)

[2.3.3.1. Définition de la classe & constructeur 23](#_Toc106229375)

[2.3.3.2. Construction d’une classe 23](#_Toc106229376)

[2.3.3.2.1. Pratiquez : les classes 24](#_Toc106229377)

[2.3.3.2.1.1. Solution 24](#_Toc106229378)

[2.3.4. Résumé 26](#_Toc106229379)

[2.3.5. To do : UML 26](#_Toc106229380)

[2.4. Définir des méthodes d’instance et des propriétés 27](#_Toc106229381)

[2.4.1.1. Rappel sur la notion de propriété de classe 27](#_Toc106229382)

[2.4.2. Tirez parti des classes avec des méthodes d'instance 28](#_Toc106229383)

[2.4.2.1. Instance de classe newAccount() 28](#_Toc106229384)

[2.4.2.2. Méthode d'instance showBalance() 29](#_Toc106229385)

[2.4.2.3. Ajouter des méthodes capables de modifier les propriétés de l'instance 30](#_Toc106229386)

[2.4.3. Pratiquez les méthodes instance 30](#_Toc106229387)

[2.4.4. Solution 31](#_Toc106229388)

[2.4.5. Découvrez les méthodes statiques 34](#_Toc106229389)

[2.4.6. En résumé 36](#_Toc106229390)

[3. Mixins 36](#_Toc106229391)

[4. Links 36](#_Toc106229392)

# POO et POOP

En JavaScript tout est objet. Tous les types héritent du type Object. Ainsi, que ce soient les string, les array, les bool, presque tout est objet.

JavaScript est certes un langage orienté objet, mais avec une subtilité. En effet, il est orienté objet à prototype.

## POO

La POO (Programmation Orientée Objet) est un paradigme de programmation qui consiste à encapsuler les données et les traitements en relation avec ces données dans des objets. Les algorithmes consistent alors à orchestrer les opérations sur ces objets et non plus sur ce qui les compose.[[1]](#footnote-1)

### Concepts fondamentaux de la programmation orientée objet

Les huit concepts fondamentaux de la programmation orientée objet sont :

* Classe
* Objet
* Encapsulation
* Héritage
* Polymorphisme
* Abstraction
* Interface
* Méthodes statiques

#### Classe

**Une classe est une** **structure abstraite** qui décrit des objets du monde réel sous deux angles : ses **propriétés** (ses caractéristiques) et ses **méthodes** (les actions qu’elle peut effectuer ou son comportement).

* La classe est une sorte de moule, de **modèle**.
* La classe est un concept **abstrait** qui permet de représenter toutes les **entités** d'un **système** ou d’un **domaine** donné.
* Une classe est une description d’un ensemble d’objets ayant une **sémantique**, des **attributs**, des **méthodes** et des **relations** en commun.
* Une classe est, donc, un ensemble d’objets similaires, c’est-à-dire possédant la **même structure** et le **même comportement** et constitués des **mêmes** **attributs** et **méthodes**.
* Ses attributs et ses méthodes, sont appelés **membres** de la classe.
* Toutes les **instances** de classe s’appellent des **objets**.
* Les objets sont construits à partir de la classe, par le processus d’**instanciation**.
* De ce fait, **tout objet est une instance de classe.**

##### Membres de classe

* **Chaque attribut** est décrit par :
* un **nom unique** dans une même classe,
* **par un type d**e **données[[2]](#footnote-2)**.
* **Une opération, ou méthode**, est décrite par :
* un nom,
* un **ensemble d’arguments** ,
* un **type de retour(**s’il y a lieu**)[[3]](#footnote-3)**.

##### Instanciation d’une classe

L’instanciation d’une classe fait appel à 3 méthodes :

1. Le constructeur
2. Les accesseurs (***get***) et les mutateurs (***set***)
3. Le destructeur

##### Représentation UML d’une classe

Une image contenant texte, clipart

Description générée automatiquement

* Graphiquement, la classe est représentée par un rectangle, divisé en 3 parties :
* son nom
* ses attributs
* ses méthodes
* Le nom d’une classe doit :
* apparaître au singulier
* débuter par une majuscule (écrit en snake case ou en Pascal case[[4]](#footnote-4))
* être significatif de l’ensemble des objets constituant la classe

Une image contenant texte

Description générée automatiquement

Figure Représentation UML d’une classe

#### Objet

**Un objet est une** **structure concrète.**

Donc un objet (concret) est une instance d’une classe (abstraite).

##### Le concept d’objet

Un objet est caractérisé par 3 éléments :

1. **une identité** : l’identité doit permettre d’identifier sans ambiguïté l’objet (**adresse**/ **référence** ou **nom**)
2. **des états** : chaque objet a une valeur par défaut (lorsqu’elle est indiquée à l’instanciation) pour chacune de ses propriétés. On appelle ces valeurs, des états de l’objet.
3. **des méthodes** : chaque objet est capable d’exécuter les actions ou le comportement défini dans la classe. Ces actions sont traduites en POO concrètement sous forme de méthodes. Les actions possibles sur un objet sont déclenchées par des appels de ces méthodes ou par des messages envoyées par d’autres objets.

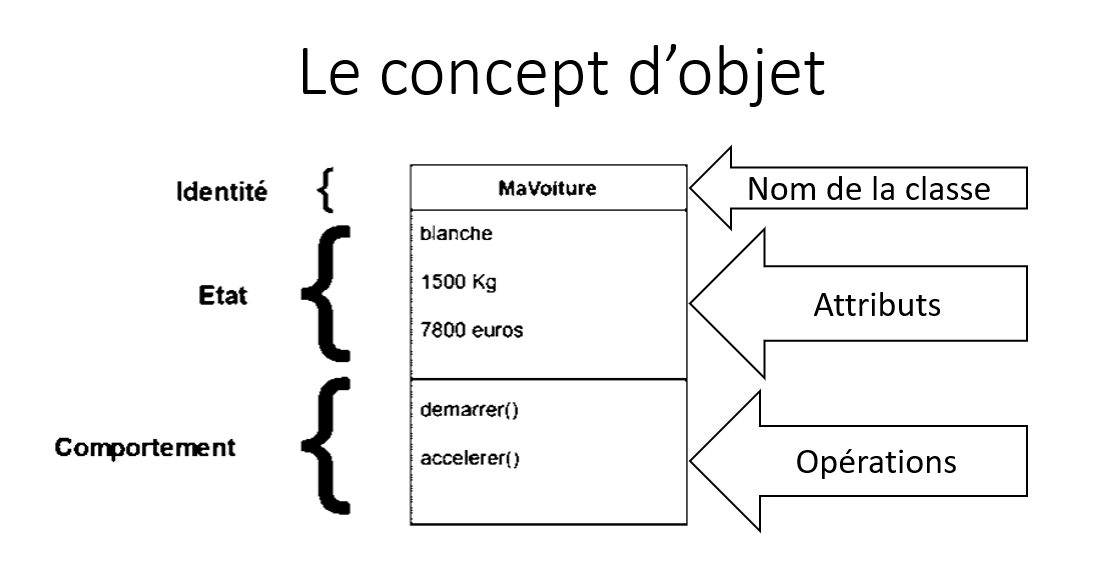


Figure Le concept d’objet

##### Représentation UML d’un objet

* Le nom de l’objet est toujours souligné indiquant le nom de la classe précédé de ":"
* Un objet peut être nommé ou anonyme

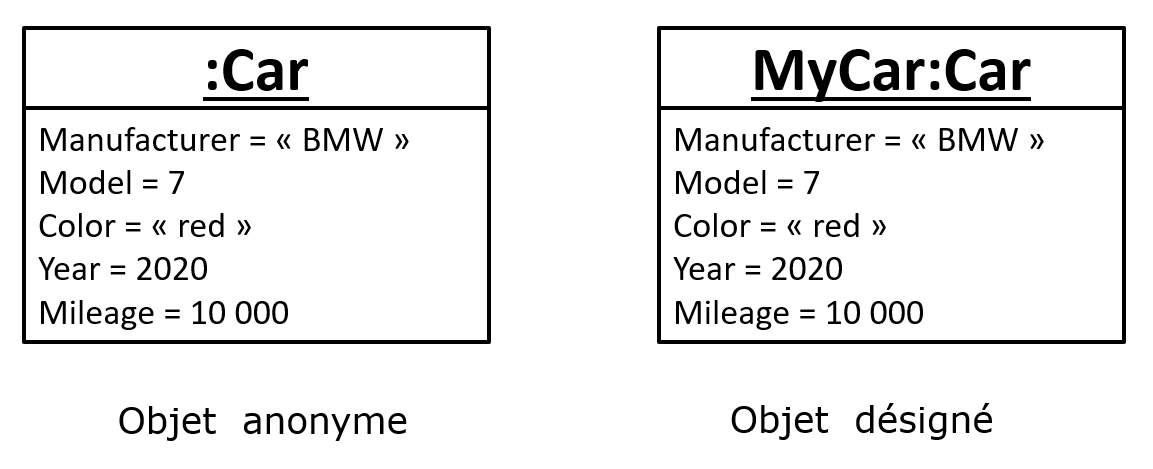


Figure . Représentation UML d’un objet

#### Encapsulation

##### Définition de l’encapsulation

* L’encapsulation consiste à **masquer** des attributs et des méthodes de l’objet vis-à-vis des autres objets.
* C’est donc un mécanisme consistant **à rassembler les données et les méthodes au sein d’une structure en cachant l’implémentation de l’objet**
* C’est-à-dire **en empêchant l’accès aux données par un autre moyen que les services proposés**
* L’encapsulation est une **abstraction** puisque l’on simplifie la représentation de l’objet vis-à-vis des objets extérieurs.
* L’encapsulation permet donc de **garantir l’intégrité des données** contenues dans l’objet.
* L'encapsulation **empêche qu'une application accède directement** aux propriétés de l'objet.
* L'encapsulation permet donc d'exposer aux applications clientes uniquement **l'interface**.
* **Conséquence** , si l’on modifie le code mais pas l'interface, l'application cliente n'a pas à être modifiée.

##### Schéma de l’encapsulation

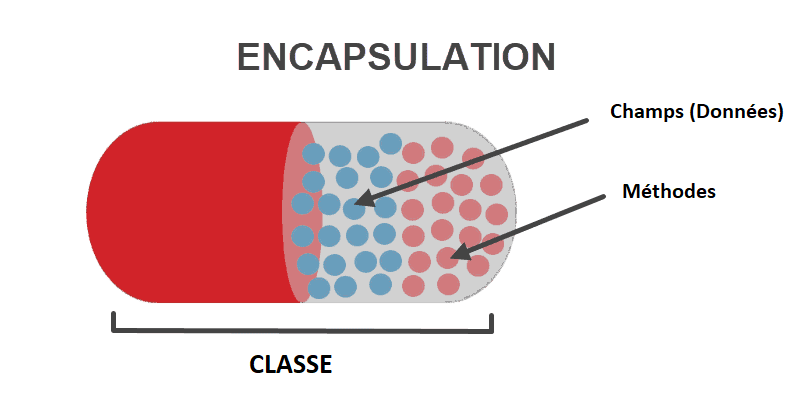


Figure Schéma de l’encapsulation

##### Niveaux de visibilité des éléments de la classe

L’encapsulation permet de définir des niveaux de visibilité des éléments de la classe. Ces niveaux de visibilité définissent ce qu’on appelle la **portée** (scope) de **l’attribut** ou de la **méthode**. La portée indique les **droits à leur accès**.

L’utilisation des **modificateurs d'accès** permet d'appliquer les principes d'encapsulation, en définissant principalement[[5]](#footnote-5), trois niveaux de visibilité :

1. **Privée** (-): les attributs privés sont accessibles seulement par la classe elle-même.
2. **Protégée** (#): les attributs protégés sont accessibles seulement dans la classe elle-même et aux classes dérivées.
3. **Publique** (+): les attributs publics sont accessibles à tous

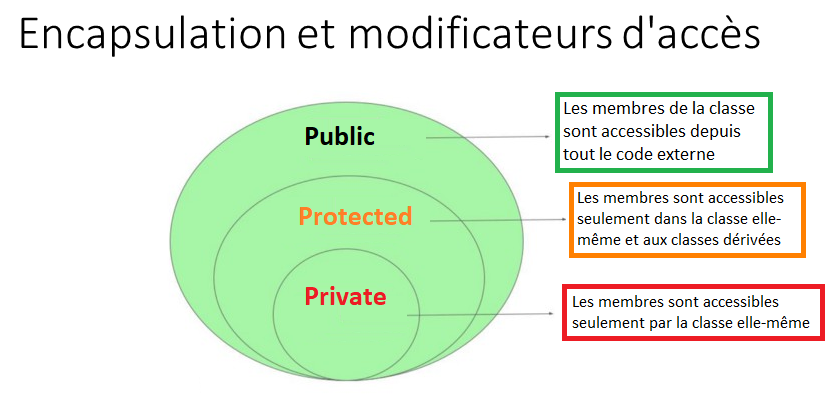


Figure Modificateurs d'accès

#### Héritage

##### Définition de l’héritage

* L'héritage est un mécanisme par lequel une **classe** **dérivée** (ou classe fille) hérite de toutes les caractéristiques de sa **classe de base** (ou classe mère).
* La classe fille (sous-classe) **récupère**, ainsi, les champs et méthodes de la classe mère (surclasse) ; et par conséquent, de toutes les fonctionnalités de cette dernière.
* La Classe fille peut utiliser **les membres de la classe mère** (super-classe), mais aussi **ajouter ses propres membres** ou **redéfinir certains membres de la classe mère**.

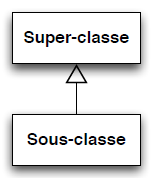


Figure Super classe et sous classe

* Pour qu’un attribut ou une méthode appartenant à une surclasse puisse être utilisé dans une sous-classe, **il faut que son type d’accès soit** **public** ou **protected.**
* **En revanche les attributs et méthodes de type private ne sont pas hérités par la classe fille.**

##### Représentation UML de l’héritage

Exemple : Carre hérite de Rectangle :

Une image contenant texte, clipart

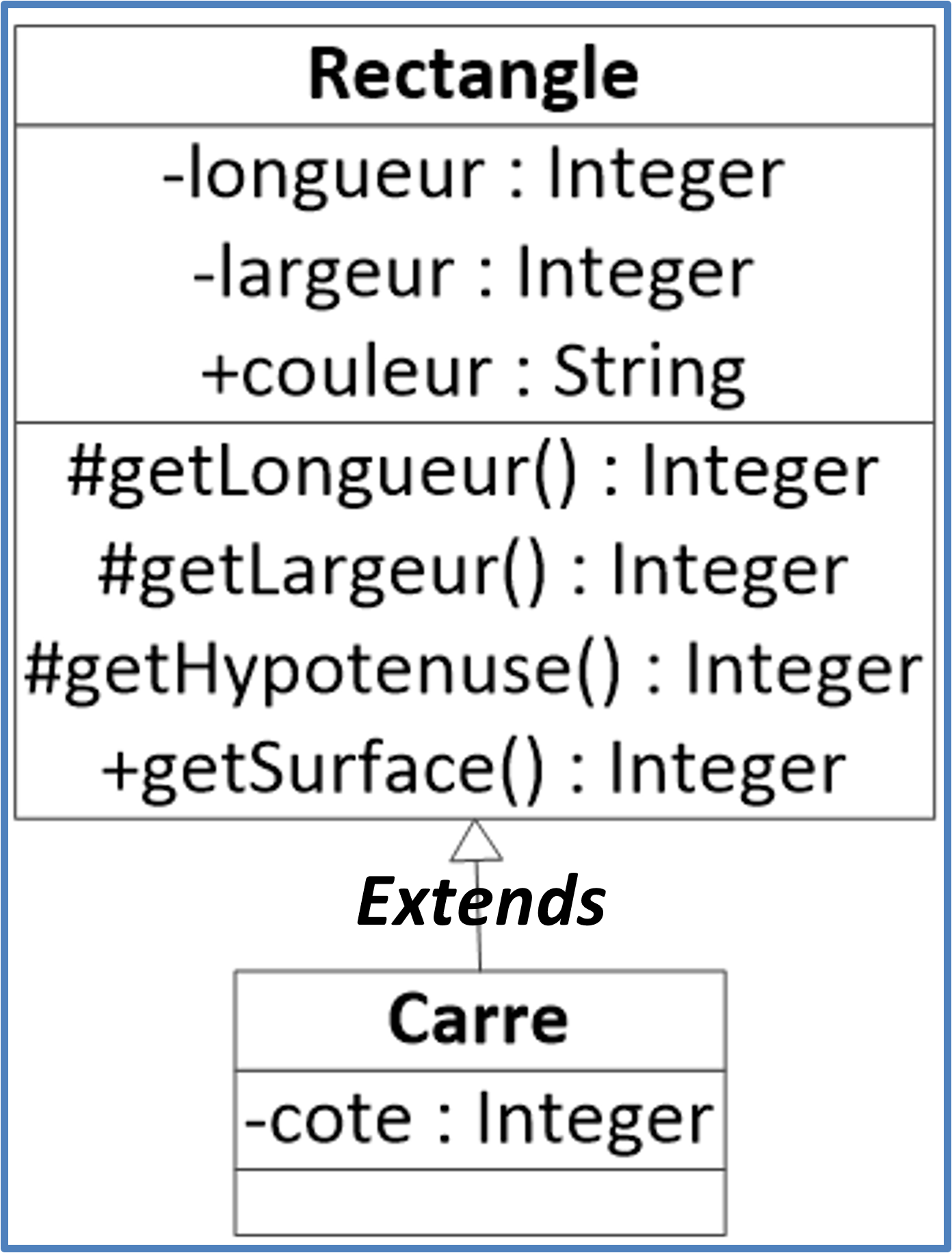
Description générée automatiquement

Figure Représentation UML de l’héritage

**La classe Carre hérite :**

* Des attributs de niveau public ou protégé:
  + **couleur**
* Des méthodes de niveau public ou protégé:
  + **getLongueur**()
  + **getLageur**()
  + **getHypotenuse**()
  + **getSurface**()

**La classe Carre ne peut accéder :**

* Aux attributs de niveau privé:
  + **longueur**
  + **largeur**

#### Polymorphisme

##### Définition du polymorphisme

* Le polymorphisme est le concept consistant à fournir une interface unique à des entités pouvant avoir **différents types**. Par exemple, des opérations telles que la multiplication peuvent ainsi être étendues à des scalaires aux vecteurs ou aux matrices.
* **Selon le langage informatique** employé, le polymorphisme peut être réalisé par différents moyens, inhérents au langage[[6]](#footnote-6)
* Le polymorphisme est la faculté attribuée à un objet d’être une **instance de plusieurs classes.**
* **Alors que l'héritage concerne les classes (et leur hiérarchie),** le **polymorphisme est relatif aux méthodes des objets**.
* On distingue généralement trois types de polymorphisme :

1. Le polymorphisme **ad hoc** , également **surcharge** ou **overloading**.
2. Le polymorphisme **d'héritage**, également **redéfinition**, spécialisation ou **overriding**.
3. Le polymorphisme **paramétrique**, également **généricité** ou **template**.

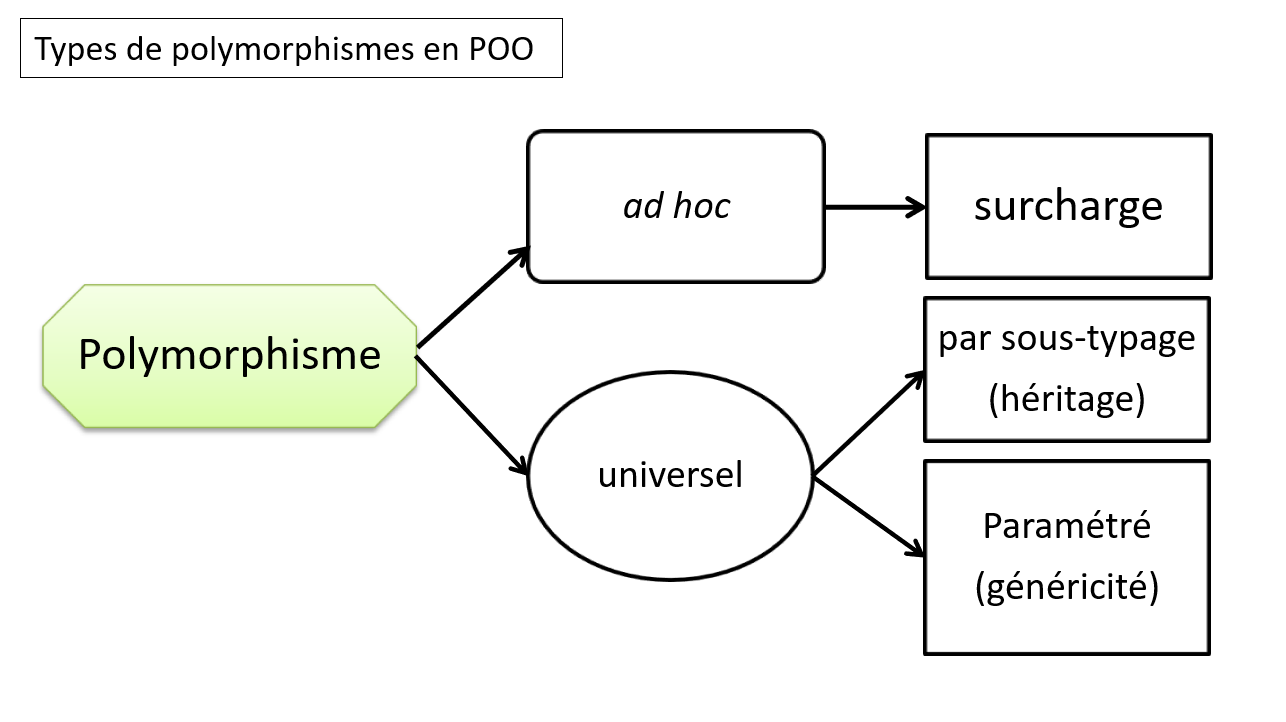


Figure Types de polymorphismes en POO

##### Caractéristiques du polymorphisme

Le polymorphisme permet de donner le même nom à des traitements différents ou encore, permet le **choix dynamique d’un** traitement en fonction de l’objet auquel il est appliqué ; **En effet le choix de la méthode polymorphe à exécuter est déterminé à l’exécution du programme.**

C’est pour cela qu’on dit qu’elle est dynamique, par opposé à statique qui est déterminé lors de la compilation.

Le polymorphisme, va de pair avec l’héritage.

Car En réalité, le but ultime de l’héritage est de faire du polymorphisme : un procédé qui permet d’offrir une même interface de programmation pour des types différents.

##### Polymorphisme versus héritage

* L’héritage désigne **l’utilisation du structure et du comportement d’une** **super classe dans une sous-classe**. Bien que le polymorphisme se réfère à la **modification du comportement d’une super-classe dans une sous-classe**.
* L’héritage permet, la **réutilisabilité du code** et le polymorphisme permet à une fonction d’avoir **différentes formes**.
* La différence fondamentale entre l’héritage et le polymorphisme est que l’héritage permet de **réutiliser** le code existant dans la classe mère, et le polymorphisme fournit un mécanisme permettant de **décider dynamiquement[[7]](#footnote-7)** quelle forme d’une fonction à invoquer[[8]](#footnote-8).

##### Représentation UML du polymorphisme

L'opération Dessiner est **polymorphe** car sa réalisation peut prendre **plusieurs formes :**

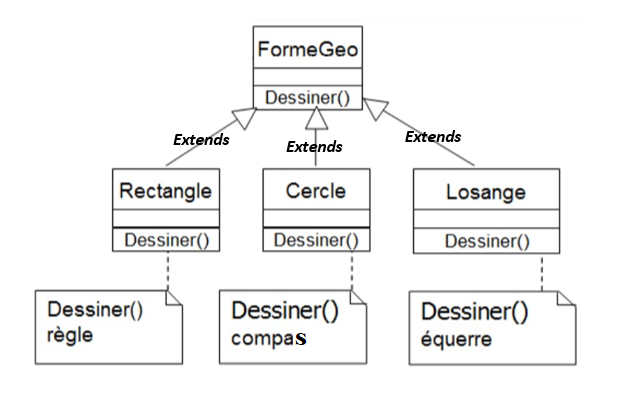


Figure Représentation UML du polymorphisme

#### Classe abstraite

##### Définition d’une classe abstraite

* Le concept de classe abstraite se situe entre celui de classe et celui d’interface.
* **Une classe est dite abstraite** lorsqu’elle définit au **moins une méthode abstraite** ou lorsqu’une classe dite parent possède une méthode abstraite non encore réalisée.
* Le concept de **classe abstraite** se situe entre celui de classe et celui **d’interface**. Une classe abstraite **est une classe ne pouvant pas être instanciée**.
* Une classe abstraite encapsule des attributs et méthodes **abstraites** qui sont utilisés par les instances des classes qui en **héritent**.
* Une classe abstraite **peut très bien** **contenir des méthodes concrètes**.
* Les classes héritant d’une classe abstraite doivent **obligatoirement redéfinir** ses méthodes abstraites ou alors être elles-mêmes abstraites.
* Elle va permettre de créer des **classes dérivées**, en clair, cette classe **va seulement servir de classe de base** (super-classe) pour des classes qui vont lui dériver.
* Une classe abstraite **peut hériter** **d’une classe concrète ou d’une classe abstraite**.
* L’indication d’une classe abstraite se fait par l’ajout du mot-clef ***abstract*** derrière son nom

##### Intérêts de la classe abstraite

Une classe abstraite permet de:

* regrouper plusieurs classes sous un même nom de classe, par **polymorphisme**,
* **factoriser du code**
* ainsi que de **décrire partiellement des attributs et méthodes communs** à plusieurs classes.
* De **clarifier** la conception.
* D’imposer à des classes les mêmes attributs la **même signature pour certaines méthodes**
* D’écrire des **algorithmes plus génériques grâce au polymorphisme.**

##### Représentation UML d’une classe abstraite

* En **UML**, le mot-clé **{abstrait}** (ou **{abstract})** est accolé aux classes et méthodes abstraites.
* Une autre manière souvent usitée de représenter ces méthodes ou classes est d’écrire leur nom en ***italique.***

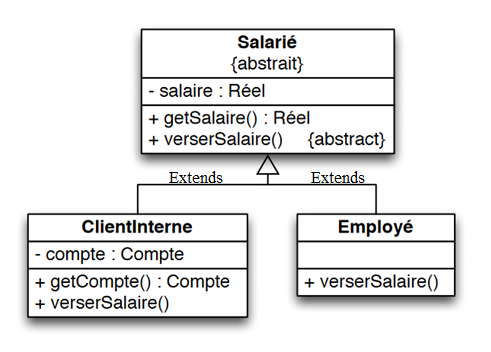


Figure Représentation UML d’une classe abstraite

#### Interface

##### Définition d’une interface

* Une interface est un **type**, au même titre qu’une classe.
* Elle est déclarée avec le mot clé ***interface***.
* Selon les principes de l’encapsulation, l’interface est la **vue externe d’un objet** : elle définit les services accessibles (offerts) aux utilisateurs de l’objet, qui ne savent pas comment ces services sont effectués.
* Une interface est un ensemble de signatures de méthodes, **qui sont par essence abstraites et publiques.**
* **Une interface ne peut contenir ni attribut, ni méthode implémentée.**
* **Une interface ne peut être instanciée.**
* Le terme d’héritage n’est pas utilisé entre classes et interfaces : on dit qu’une classe **implémente** (ou « réalise ») une interface , à l’aide du mot clé ***implements***
* Lorsqu’une classe implémente une interface, **elle doit** **redéfinir** toutes ses méthodes abstraites ou bien être abstraite..
* Une interface peu**t hériter** d’une autre interface.
* Une classe ,abstraite ou non, peut **implémenter plusieurs interfaces**, ce qui permet de pallier le manque de l’héritage multiple dans certains langages[[9]](#footnote-9).
* Les **constantes** d’une interface sont accessibles à toutes les classes implémentant l’interface.
* L’intérêt d’une interface est de regrouper plusieurs classes par **polymorphisme**.

##### Le principe d’obligation contractuelle

Les interfaces servent à définir des **contrats de service**, définis (implémentés) dans différentes

**classes concrètes**.

Une classe qui déclare implémenter une interface **s'engage à fournir le service** (le « contrat ») spécifié par l'interface.

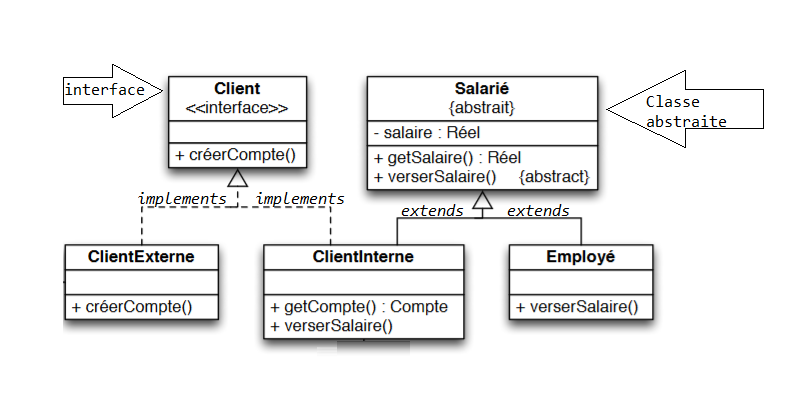
Elle doit donc fournir une **implémentation pour chacune des méthodes abstraites** listées dans l'interface.

En fait, ce n'est le cas que si la classe est concrète. Une classe abstraite qui implémente une interface n'est pas tenue de fournir une implémentation pour toutes les méthodes abstraites déclarées dans l'interface. Mais ses sous-classes concrètes devront le faire, bien sûr (sans quoi elles ne seraient pas concrètes !).

##### Interface versus classe abstraite

* l'interface est une classe abstraite **100% abstraite.**
* La notion d’interface est proche de la notion de classe abstraite, avec une capacité de **découplage** plus grand ; en effet, le comportement d'une interface est purement abstrait, en revanche une classe abstraite peut implémenter tout ou partie du comportement.
* Une interface est une coquille vide. Il y a **seulement les signatures des méthodes.**
* L'interface sert de **modèle**.
* Implémenter une interface consomme très peu de **ressources** CPU.
* Les classes abstraites, contrairement aux interfaces, sont des classes, donc elles sont plus couteuses en ressources.

##### Représentation UML d’une interface



#### Méthodes statiques

##### Définition d’une méthode statique

##### Représentation UML d’une méthode statique

## POOP

Le JavaScript est un langage qui utilise la Programmation Orienté Objet à Prototype : POOP.

### Définition de la Programmation orientée prototype

La programmation orientée prototype est une forme de programmation orientée objet sans classe, fondée sur la notion de prototype.

Un prototype est un objet à partir duquel on crée de nouveaux objets.

Self[[10]](#footnote-10) fut le premier langage à prototypes. Il a été conçu dans les laboratoires de Sun dans les années 1990. Le plus connu actuellement est JavaScript[[11]](#footnote-11).

### Comparaison des modèles à classes et à prototypes

* Objets à classes :
  + Une classe définie par son code source est statique ;
  + Elle représente une définition abstraite de l'objet ;
  + Tout objet est instance d'une classe ;
  + L'héritage se situe au niveau des classes.
* Objets à prototypes :
  + Un prototype défini par son code source est mutable ;
  + Il est lui-même un objet au même titre que les autres ;
  + Il a donc une existence physique en mémoire ;
  + Il peut être modifié, appelé ;
  + Il est obligatoirement nommé ;
  + Un prototype peut être vu comme un exemplaire modèle d'une famille d'objet ;
  + Un objet hérite des propriétés (valeurs et méthodes) de son prototype.

### Intérêt de la programmation orientée prototype

# Les classes en JavaScript

## Sucre syntaxique

<https://www.freecodecamp.org/news/javascript-classes-how-they-work-with-use-case/#Prerequisites>

<https://buzut.net/programmation-orientee-objet-javascript/>

## Présentation des classes

Les classes ont été introduites dans **EcmaScript 2015**[[12]](#footnote-12) (ES6) pour fournir un moyen plus propre de suivre les modèles de programmation orientés objet.

JavaScript suit toujours un modèle **d'héritage basé sur un prototype.** Les classes en JavaScript sont du sucre syntaxique par rapport au modèle d'héritage basé sur un prototype que nous utilisons pour implémenter les concepts POO.

Ainsi, l'introduction de classes dans JS a permis aux développeurs de créer plus facilement des logiciels autour des concepts OOP. Il a également apporté des similitudes avec différents langages de programmation basés sur la POO tels que C++ et Java.

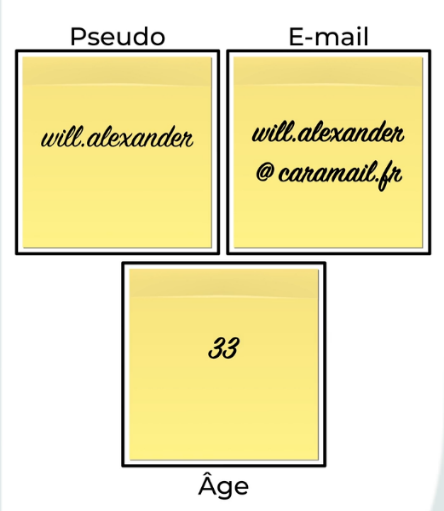
## Définissez des objets et leurs attributs avec des classes

Vous remarquez des **points communs** entre différents objets et vous notez ces informations pour construire une **représentation mentale d'une catégorie d'objets**.

Cette **liste mentale d'attributs** sert de **modèle** **pour cet objet**. En programmation, on l'appelle une **classe.**

Pour construire une classe, vous pouvez choisir le nom de votre choix. C'est pour cela qu'on l'appelle un **type nommé**. Vous le verrez, les classes permettent aussi de regrouper beaucoup de détails ; c'est pourquoi elles s'appellent aussi des **types complexes**.

Une image contenant texte

Description générée automatiquement  Une image contenant texte

Description générée automatiquement

### Découvrez les objets

Les objets JavaScript sont écrits en **JSON** (JavaScript Object Notation).

* Ce sont des séries de **paires clés/valeurs**
* séparées par des virgules,
* entre des accolades.

Les objets peuvent être enregistrés dans une variable :

Une image contenant texte

Description générée automatiquement

Chaque **clé** (« nom » « alias ») **est une chaîne** : un **string**.

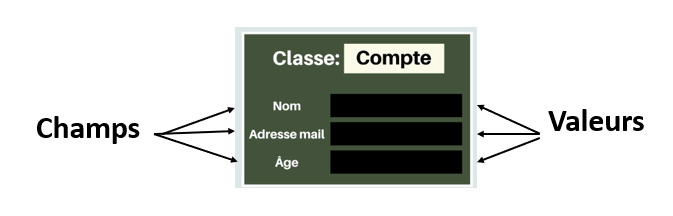
Et les **valeurs** associées peuvent avoir **tout type de données.**

Construire des objets présente un avantage essentiel : cela permet de **regrouper les attributs** d'une chose unique à un même emplacement.

### La notion de classe

Une image contenant texte

Description générée automatiquement



**►La** [**classe**](https://fr.wikipedia.org/wiki/Classe_(informatique)) **(ou** [**class**](https://en.wikipedia.org/wiki/Class_(computer_programming))**)**  contient des ***champs*** qui correspondent aux ***clés*** de l’objet final

Le nom d’une **C**lasse doit commencer par une majuscule.

À partir de l’édition de 2015 (ECMAScript 6), une [syntaxe de définition de classes](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Classes) a été ajoutée, simplifiant l’utilisation de son mécanisme d’héritage prototypal pour le développement orienté objet.

#### Exemple de classe

[class Point](https://fr.wikipedia.org/wiki/Classe_(informatique)#Javascript):

**class Point {**

**constructor(x, y) {**

**this.\_x = x;**

**this.\_y = y;**

**}**

**getX() {**

**return this.\_x;**

**}**

**getY() {**

**return this.\_y;**

**}**

**isOrigin() {**

**return this.\_x === 0 && this.\_y === 0;**

**}**

**translate(pt) {**

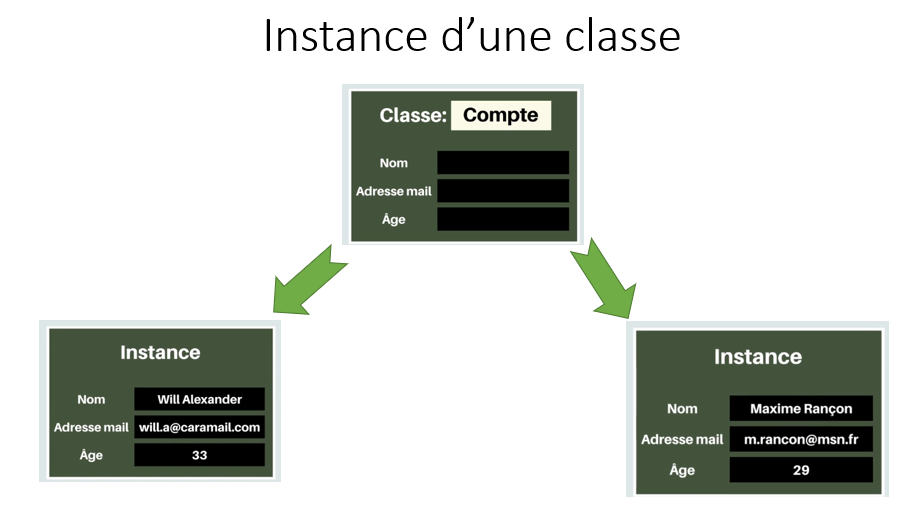
**return new Point(this.\_x + pt.\_x, this.\_y + pt.\_y);**

**}**

**}**

#### Instance d’une classe

Chaque objet construit à partir de cette **classe est appelé instance de cette classe.**

****

#### Pratiquez : créez un objet

Créez un objet et stockez-le dans une variable objet appelée **episode** . Utilisez bien des accolades et mettez les trois attributs suivants :

**title** : le titre de l'épisode

**duration** : la durée de l'épisode

**hasBeenWatched** : si l'épisode a été visionné ou non

Associez des valeurs appropriées à chaque attribut.

##### Solution

**HTML :**

<h3>Les objets</h3>

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="episode-info-o"></p>

</div>

**CSS :**

body{

background-color: #b7ffb7;

color: #000;

font-family: 'Roboto Slab', serif;

}

.series-frame{

max-width: 600px;

font-size: 0.9em;

display: flex;

justify-content: space-between;

align-items: center;

box-sizing: border-box;

border: 2px solid #804040;

margin: 30px;

padding: 10px;

}

.my-form {

padding: 20px;

}

**JS :**

// =========================================

let oEpisode = {

title : 'Gone with the Wind',

duration : 243,

hasBeenWatched : true

};

// =========================================

document.querySelector('#episode-info').innerText = `Episode: ${oEpisode.title}

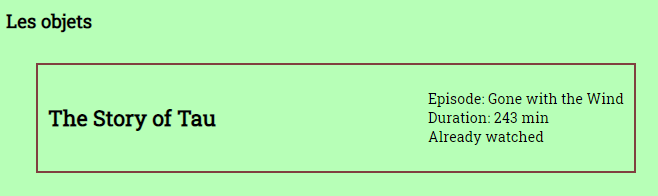
Duration: ${oEpisode.duration} min

${oEpisode.hasBeenWatched ? 'Already watched' : 'Not yet watched'}`

Une image contenant texte

Description générée automatiquement

**Ecran :**



#### Accédez aux données d'un objet avec la notation pointée

Une fois qu'un **objet est enregistré dans une variable**, vous pouvez accéder à ses données.

Pour cela, utilisez **le nom de la variable qui contient l'objet**, **un point (.),** **puis le nom de la clé** dont vous souhaitez récupérer la valeur.

Une image contenant texte

Description générée automatiquement

##### Pratiquez : récupérez des valeurs depuis un Object

**Reprendre l’objet vu précédemment et y ajouter :**

1. Créez les trois variables suivantes :
   1. episodeTitle : le titre de l'épisode (string)
   2. episodeDuration : la durée de l'épisode (number)
   3. episodeHasBeenWatched : si l'épisode a été regardé ou non (boolean)
2. Assignez-y les valeurs correspondantes de l'objet episode en utilisant la notation "dot".

###### Solution

**HTML :**

<h3>Accéder aux données d'un objet</h3>

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="episode-info-aO"></p>

</div>

CSS : inchangé

**JS :**

let oEpisod = {

title : 'Gone with the Wind',

duration : 243,

hasBeenWatched : true

};

// Create variables here

// =====================================

let episodeTitle2 = oEpisod.title ;

let episodeDuration2 = oEpisod.duration ;

let episodeHasBeenWatched2 = oEpisod.hasBeenWatched ;

// =====================================

document.querySelector('#episode-info-aO').innerText = `Episode: ${episodeTitle2}

Duration: ${episodeDuration2} min

${episodeHasBeenWatched2 ? 'Already watched' : 'Not yet watched'}`

Une image contenant texte

Description générée automatiquement

**Ecran :**

Une image contenant texte

Description générée automatiquement

### Manipulez des classes

La construction d'un objet à la main, par la notation à accolades vue précédemment, convient bien à des objets simples et uniques.

Mais vous aurez souvent besoin de **beaucoup d'objets du même type**. C'est là que les **classes** sont utiles.

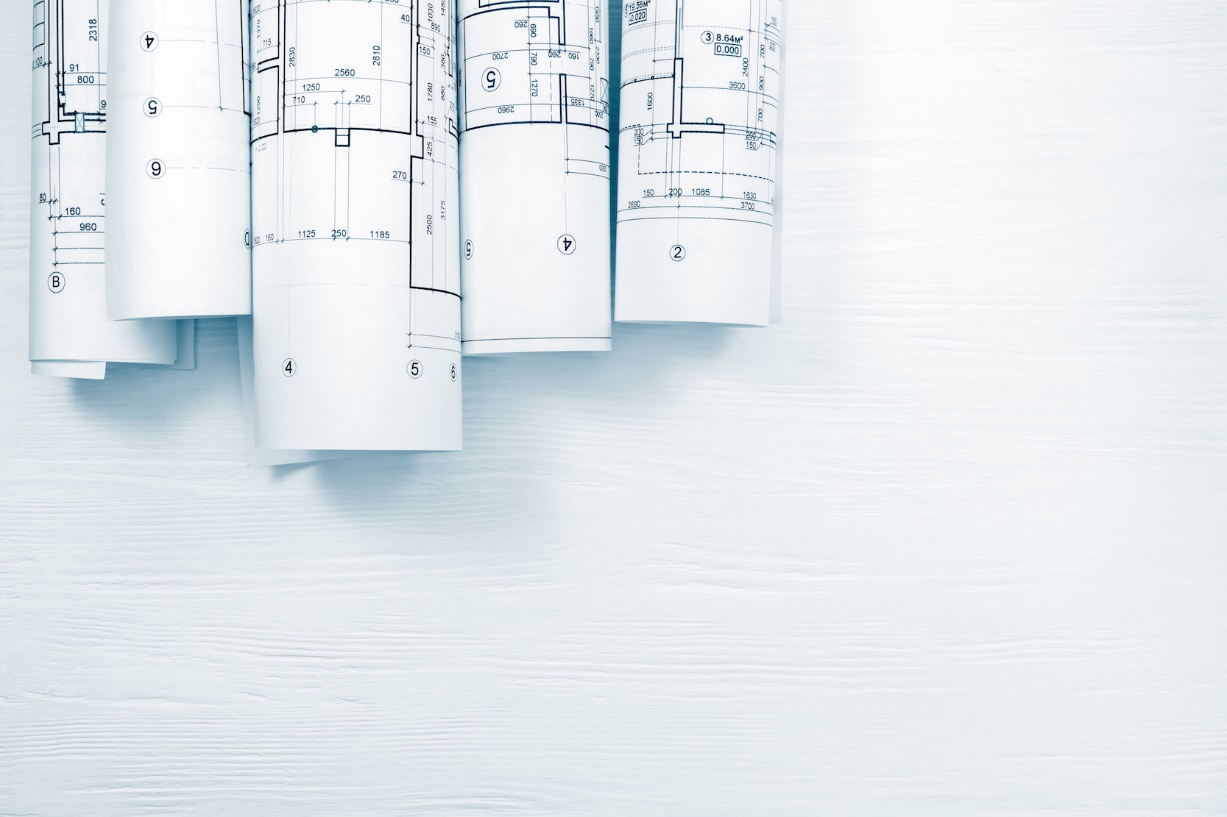


Figure : La classe est comme un plan

#### Définition de la classe & constructeur

Une classe est un **modèle**[[13]](#footnote-13) pour un objet dans le code. Elle permet de **construire plusieurs** **objets du même type,** appelés ***instances*** de la même classe ; plus facilement, rapidement et en toute fiabilité.

#### Construction d’une classe

1. Pour construire une classe dans JavaScript, utilisez le mot clé **class**, suivi par un nom
2. Encadrez ensuite le code de la classe **entre accolades**
3. Utiliser un ***constructor***
   * Utiliser lemotclé **this** avecla notation dot, pour attribuer les valeurs reçues **aux clés** correspondantes.

Le ***constructor*** d'une classe est la fonction qui est appelée quand on crée une nouvelle instance de cette classe avec le mot clé **new**.

Ce qui donne :

Une image contenant texte

Description générée automatiquement

Maintenant que la classe est terminée, on peut construire des instances de la classe ***Book*** par le mot clé **new** :



Gardez cette notion de classe en tête, on y reviendra un peu plus loin en abordant la notion de **propriété de classe** et les **méthodes d’instance**.

##### Pratiquez : les classes

Cette fois-ci, il y a **trois composants épisodes** 🡺

1. **Donc** la manière logique de procéder serait de construire **une classe Episode**
2. Et puis de construire **trois instances de cette classe** — une pour chaque épisode.

Créez une **classe Episode** avec le mot-clé **class**.

Créez un ***constructor*** pour votre classe **Episode** qui devra accepter trois arguments[[14]](#footnote-14) :

**title** : le titre de l'épisode (string)

**duration** : la durée de l'épisode (number)

**hasBeenWatched** : si l'épisode a été regardé ou non (boolean)

Avec le mot-clé **this** , assurez-vous que le ***constructor*** assigne correctement les arguments reçus aux champs correspondants de chaque nouvelle instance.

Avec le mot-clé **new** , créez trois instances de la classe **Episode** et stockez-les dans les trois variables :

**firstEpisode**

**secondEpisode**

**thirdEpisode**

N'oubliez pas de passer des arguments appropriés à chaque instance.

###### Solution

**HTML :**

<h3>Les classes</h3>

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="first-episode-info"></p>

</div>

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="second-episode-info"></p>

</div>

<div class="series-frame">

<h2>The Story of Tau</h2>

<p id="third-episode-info"></p>

</div>

**CSS :**

.series-frame{

    max-width: 600px;

    /\*font-size: 0.9em;\*/

    display: flex;

    /\*justify-content: space-between;\*/

    align-items: center;

    box-sizing: border-box;

    border: 2px solid #804040;

    margin: 30px;

    padding: 10px;

}

.my-form {

    padding: 20px;

  }

.series-frame h2 {

  margin-right: 20px;

}

**JS :**

class cEpisode{

constructor(title, duration, hasBeenWatched){

this.title = title;

this.duration = duration;

this.hasBeenWatched = hasBeenWatched;

}

}

let oFirstEpisode = new cEpisode('Gone with the Wind', 243, true)

let oSecondEpisode = new cEpisode("2001: A Space Odyssey", 156, true)

let oThirdEpisode = new cEpisode('A Beautiful Mind', 135, true)

// =====================================

document.querySelector('#first-episode-info').innerText = `Episode: ${oFirstEpisode.title}

Duration: ${oFirstEpisode.duration} min

${oFirstEpisode.hasBeenWatched ? 'Already watched' : 'Not yet watched'}`;

document.querySelector('#second-episode-info').innerText = `Episode: ${oSecondEpisode.title}

Duration: ${oSecondEpisode.duration} min

${oSecondEpisode.hasBeenWatched ? 'Already watched' : 'Not yet watched'}`;

document.querySelector('#third-episode-info').innerText = `Episode: ${oThirdEpisode.title}

Duration: ${oThirdEpisode.duration} min

${oThirdEpisode.hasBeenWatched ? 'Already watched' : 'Not yet watched'}`;

**Une image contenant texte

Description générée automatiquement**

**Ecran :**

Une image contenant texte

Description générée automatiquement****

### Résumé

1. Les **objets** avec les paires clés/valeurs en notation **JSON**. Ils permettent d'enregistrer plusieurs éléments de données associés dans une même variable ;
2. La **notation** **pointée** (**dot**) qui donne accès aux valeurs d'un objet et la possibilité de les modifier ;
3. **Les** **classes**, et comment l'utilisation de classes peut vous permettre de **construire** **des** **objets** plus facilement et de façon plus lisible.

### To do : UML

<https://openclassrooms.com/fr/search?page=1&query=UML>

[Une image contenant texte

Description générée automatiquement](https://openclassrooms.com/fr/courses/4055451-modelisez-et-implementez-une-base-de-donnees-relationnelle-avec-uml)

## Définir des méthodes d’instance et des propriétés

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

#### Rappel sur la notion de propriété de classe

Il n’est pas rare d’utiliser le terme “**attribut**” au lieu de "**propriété**", mais cela représente bien la même chose.

Prenons l’exemple d’une classe qui représenterait un compte bancaire, et que l’on appellerait du coup BankAccount .

On pourrait trouver comme propriétés owner (permettant d’identifier le propriétaire du compte) et balance (permettant de connaître le montant disponible sur le compte).

Une image contenant table

Description générée automatiquement

Ainsi, lorsque que l’on crée une instance de BankAccount avec un propriétaire et un montant initial en argument, on pourra exploiter par la suite ces **propriétés** pour afficher leurs valeurs, les utiliser pour des calculs, les modifier, etc.

### Tirez parti des classes avec des méthodes d'instance

Maintenant que vous avez commencé à découvrir les **fonctions** vous pouvez ajouter des **méthodes d'instance** à des classes, pour augmenter leur puissance et leur utilité.

**Une méthode d'instance est une fonction faisant partie d'une classe, et qui agit sur une instance de cette classe**

Reprenons notre exemple de classe BankAccount :

**class BankAccount {**

**constructor(owner, balance) {**

**this.owner = owner;**

**this.balance = balance;**

**}**

**}**

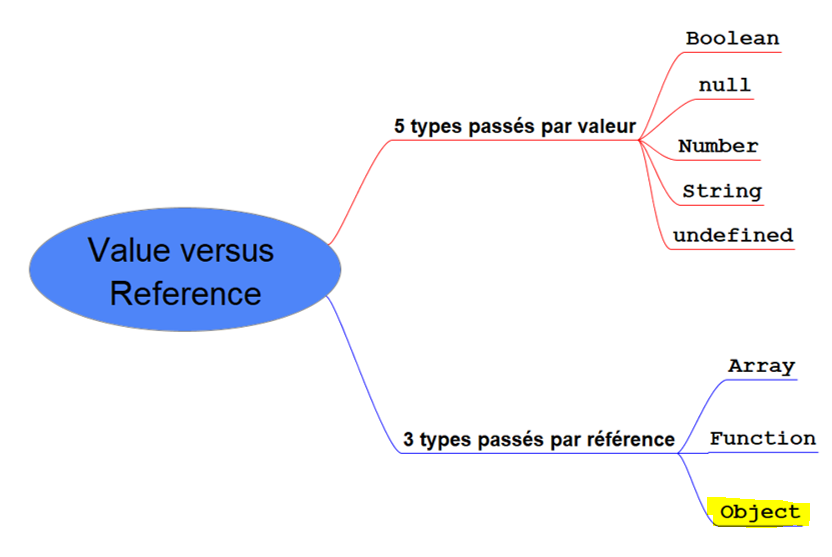
#### Instance de classe newAccount()

►Vous pouvez ensuite créer une **instance de cette classe** appelée newAccount :

**const newAccount = new BankAccount("Bill Gates", 98000000000);//2020-07-31**

**►Pour rappel : une instance d'une classe est un objet.**

…►Et un objet est un type par **référence** :



..►donc vous pouvez toujours apporter des modifications à l'instance de newAccount

…►la partie constante **désigne une référence[[15]](#footnote-15) à cette instance**

Telle quelle, l'instance n'est pas très utile. Vous pourriez afficher son solde à la console par newAccount.balance

#### Méthode d'instance showBalance()

**class BankAccount {**

**constructor(owner, balance) {**

**this.owner = owner;**

**this.balance = balance;**

**}**

**showBalance() {**

**console.log("Account balance: " + this.balance + " $");**

**}**

**}**

La nouvelle **méthode** ci-dessus, déclarée par son nom suivi par (), utilise le mot clé this pour accéder à la **propriété** balance de l'instance, et l'afficher sur la console.

Ceci signifie que vous pouvez utiliser la notation dot sur l'instance newAccount pour appeler sa méthode showBalance() :

**let newAccount = new BankAccount("Bill Gates", 98000000000); //2020-07-31**

**newAccount.showBalance();**

Une image contenant texte

Description générée automatiquement

#### Ajouter des méthodes capables de modifier les propriétés de l'instance

Vous pouvez aussi ajouter des méthodes capables **de modifier les propriétés** de l'instance. Dans ce cas, ajoutez les méthodes deposit() (dépôt) et withdraw() (retrait) à la classe, en n'oubliant pas que les deux ont besoin d'un paramètre amount (montant).

**class BankAccount {**

**constructor(owner, balance) {**

**this.owner = owner;**

**this.balance = balance;**

**}**

**showBalance() {**

**console.log("Solde: " + this.balance + " EUR");**

**}**

**deposit(amount) {**

**console.log("Dépôt de " + amount + " EUR");**

**this.balance += amount;**

**this.showBalance();**

**}**

**withdraw(amount) {**

**if (amount > this.balance) {**

**console.log("Retrait refusé !");**

**} else {**

**console.log("Retrait de " + amount + " $");**

**this.balance -= amount;**

**this.showBalance();**

**}**

**}**

**}**

Dans le corps d'une classe, le mot clé this fait référence à **l'instance créée de la classe**. Dans cet exemple, il fait référence à newAccount .

### Pratiquez les méthodes instance

Plutôt que d'utiliser une fonction pour calculer la note moyenne pour une série, on a décidé que la classe Show (ou "Série") **devrait gérer les nouvelles notes automatiquement**

Comme vous pouvez le constater, la classe Show contient un Array ratings , pour les notes, qui est initialisé vide, et une propriété averageRating (note moyenne) de zéro.

1. Déclarez une méthode d’instance dans la classe Show appelée addRating() . Elle prendra une rating (la note), qui sera un nombre entre 1 et 5, comme paramètre.
2. La méthode addRating() doit :

* ajouter la note reçue au tableau ratings de l'instance ;
* recalculer la valeur averageRating de l'instance.

### Solution

**HTML**

<link href="https://fonts.googleapis.com/css?family=Roboto+Slab" rel="stylesheet">

<button id="refresh">Check ratings</button>

**CSS**

body, button {

background-color: #111;

color: #EEE;

font-family: 'Roboto Slab', serif;

}

.series-frame {

max-width: 600px;

display: flex;

justify-content: space-between;

align-items: center;

box-sizing: border-box;

border: 2px solid #EEE;

margin: 30px;

padding: 10px;

}

#refresh {

float: right;

margin-right: 20px;

font-size: 1.2em;

font-weight: 200;

padding: 10px;

}

#refresh:hover {

background-color: #000090;

cursor: pointer;

}

**JS**

**class Show {**

**constructor(title, numberOfSeasons) {**

**this.title = title;**

**this.numberOfSeasons = numberOfSeasons;**

**this.ratings = [];**

**this.averageRating = 0;**

**}**

**addRating(rating) {**

**this.ratings.push(rating);**

**let sum = 0;**

**for (let rating of this.ratings) {**

**sum += rating;**

**}**

**this.averageRating = sum / this.ratings.length;**

**}**

**}**

// ======================

const tau = new Show('The Story of Tau', 5);

const colin = new Show('The Hero of Old Meldrum', 3);

const clara = new Show('The Bugs of Isla Clara', 6);

const shows = [tau, colin, clara];

const body = document.querySelector('body');

const refresh = document.querySelector('#refresh');

refresh.addEventListener('click', () => {

removeShows();

addRandomRatings();

updateShows();

})

const updateShows = () => {

for (let show of shows) {

const showPane = document.createElement('div');

showPane.classList.add('series-frame');

const showHeading = document.createElement('h2');

showHeading.innerText = show.title;

const showDetails = document.createElement('p');

const seasons = document.createElement('p');

seasons.innerText = show.numberOfSeasons + ' seasons';

const ratings = document.createElement('p');

ratings.innerText = show.averageRating > 0 ? show.ratings.length + ' ratings\n' + show.averageRating.toFixed(1) + ' stars' : 'No ratings yet';

showDetails.append(seasons);

showDetails.append(ratings);

showPane.append(showHeading);

showPane.append(showDetails);

body.append(showPane);

}

};

const removeShows = () => {

const children = [];

for (let childNode of body.childNodes) {

children.push(childNode);

}

for (let child of children) {

if (child.tagName == 'DIV') {

body.removeChild(child);

}

}

}

const addRandomRatings = () => {

for (let show of shows) {

if (Math.random() >= 0.2) {

const numberOfRatings = Math.floor(Math.random() \* 4 + 1);

for (let i = 0; i < numberOfRatings; i++) {

const rating = Math.floor(Math.random() \* 5 + 1);

show.addRating(rating);

}

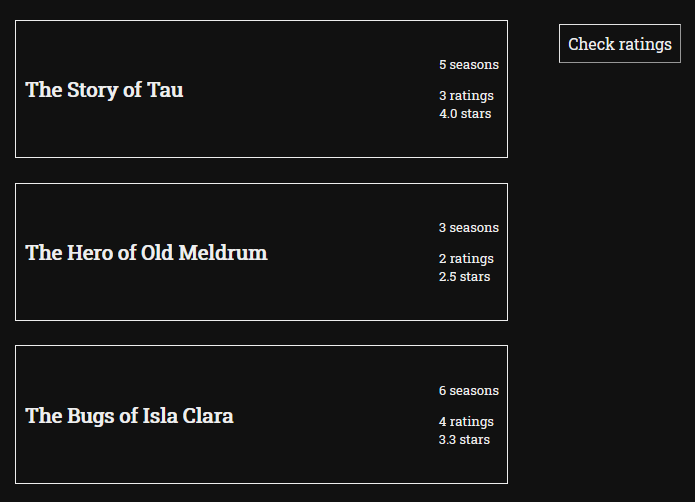
}

}

}

updateShows();

**Affichage**

****

### Découvrez les méthodes statiques

Il existe un type spécial de méthode pouvant être ajoutée à une classe : la méthode statique. Elle est différente des méthodes d'instance parce qu'elle n'est pas liée à une instance particulière d'une classe, mais à la classe elle-même.

Utilisez-la pour créer des **méthodes utilitaires** (**helper en anglais**) où vous n'aurez pas besoin d'une instance d'une classe pour les utiliser. Vous pourrez vous en servir comme **boîte à outils de fonctions** que vous utiliserez souvent.

Par exemple, en JavaScript, l'objet Math contient beaucoup de méthodes utiles :

**let x = 5.95;**

**const randomNumber = Math.random();**

**const roundDown = Math.floor(x);**

**const roundTop = Math.ceil(x);**

**const maxData = Math.max(23,17,83,2,13);**

**const minData = Math.min(23,17,83,2,13);**

Une image contenant texte

Description générée automatiquement

**console.log(randomNumber);**

**console.log(roundDown);**

**console.log(roundTop);**

**console.log(maxData);**

**console.log(minData);**

Vous n'avez pas besoin de créer par new une instance de l'objet Math pour utiliser ces méthodes ; il suffit de les appeler sur l'objet Math global.

►Vous pouvez créer vos propres méthodes statiques par le mot clé **static** !

**class BePolite {**

**static sayHello(){**

**console.log("Hello ☺");**

**}**

**static sayHelloTo(anyName){**

**console.log("Hello" + String.fromCharCode(32) + anyName + " ☺");**

**}**

**static remainder(dividend, divider){**

**return dividend % divider;**

**}**

**}**

**let modulo = BePolite.remainder(17,5);**

**BePolite.sayHello();**

**BePolite.sayHelloTo("Person");**

**console.log(modulo);**

Une image contenant texte

Description générée automatiquement

Vous n'avez pas besoin d'ajouter un **constructor** à votre classe, car vous n'allez pas l'instancier.

Vous avez une méthode qui :

* imprime un message **générique** ;
* accepte un argument et l'utilise pour créer un message ;
* renvoie une valeur à partir des arguments que vous lui envoyez.

Toutes **ces fonctionnalités pourraient être des fonctions**,

mais l'avantage d'utiliser des méthodes de classe statiques est par exemple de **pouvoir les regrouper par catégorie ou par type.**

### En résumé

nous nous sommes intéressés à **deux types de méthodes de classe** :

1. les **méthodes d'instance**, **qui agissent** sur les instances individuelles d'une classe ;
2. les **méthodes statiques**, qui ne **s'appuient pas** sur une instance d'une classe.

*Dans le chapitre qui suit, nous allons voir comment écrire des fonctions, et du code propre qui soit facile à lire, à maintenir et à adapter !*

# Mixins

# Links

<https://www.freecodecamp.org/news/javascript-classes-how-they-work-with-use-case/>

<https://www.freecodecamp.org/news/javascript-classes-how-they-work-with-use-case/>

1. <https://developer.mozilla.org/fr/docs/Glossary/OOP> [↑](#footnote-ref-1)
2. Fondamental ! [↑](#footnote-ref-2)
3. Ces 3 caractéristiques représentent la signature de la méthde. [↑](#footnote-ref-3)
4. Ou Upper Camel Case [↑](#footnote-ref-4)
5. En fait, le nombre de niveaux de visibilité dépend de chaque langage de POO. [↑](#footnote-ref-5)
6. <https://fr.wikipedia.org/wiki/Polymorphisme_(informatique)> [↑](#footnote-ref-6)
7. Par le compilateur [↑](#footnote-ref-7)
8. <https://waytolearnx.com/2018/09/difference-entre-heritage-et-polymorphisme.html> [↑](#footnote-ref-8)
9. Comme Java. [↑](#footnote-ref-9)
10. <https://en.wikipedia.org/wiki/Self_(programming_language)> [↑](#footnote-ref-10)
11. <https://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_prototype> [↑](#footnote-ref-11)
12. <https://262.ecma-international.org/6.0/> [↑](#footnote-ref-12)
13. Quid d’UML ? [↑](#footnote-ref-13)
14. Ou attributs [↑](#footnote-ref-14)
15. Équivalent du pointeur en C. [↑](#footnote-ref-15)