Les Classes

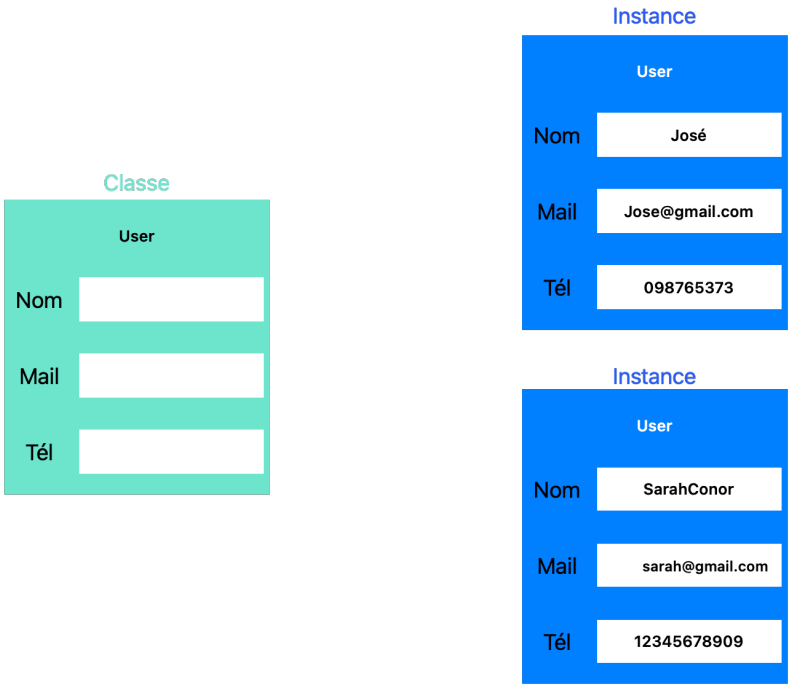
Pré requis : Les objets

Les classes sont un élément essentiel de le P.O.O (Programmation orientée objet), pour résumer le concept les classes vont nous permettre de créer plus facilement et rapidement plusieurs objets avec des caractéristiques, d'architectures similaires.

Par exemple si une application gère des utilisateurs on peut définir une classe « user » et pour chaque user on gère les données suivantes : un nom, un mail, un num de téléphone.

Pour créer / **construire** des nouveaux user, le système de class utilise une fonction qui s'appelle **constructor** (**construct** dans d'autres langages)

On va pouvoir plus facilement créer de nouveaux user (des nouvelles instances de la classe user)



Syntaxe

Donc côté code, on crée notre class UserProfile, pour pouvoir créer des nouvelles instance on renseigne les données dont on a besoin, que l'on recevra en paramètre (nameUser, mialUser, phoneUser)

Le mot clé this va représenter l'objet courant, celui que l'on est entrain de créer, c'est le contexte. Ici pour résumer on assigne aux propriété de notre classe les valeur qu'on va recevoir en paramètre

```
class UserProfile {
    constructor(nameUser, mailUser, phoneUser) {
        this.nameUser = nameUser;
        this.mailUser = mailUser;
        this.phoneUser = phoneUser;
    }
    getProfileInfo() {
        return `infos de l'utilisateur :
            son nom : ${this.nameUser}
            son mail : ${this.mailUser}
            son Tél : ${this.phoneUser}`;
    }
}
```

Bonus : on a écrit une fonction au sein de la classe, c'est une méthode de classe et elle ne pourra s'utiliser QUE sur des objets (des nouvelles instances) de cette classe

Une fois qu'on a définit la structure de notre classe on va pouvoir utiliser le constructeur pour créer un nouvel utilisateur en faisant new UserProfile( )

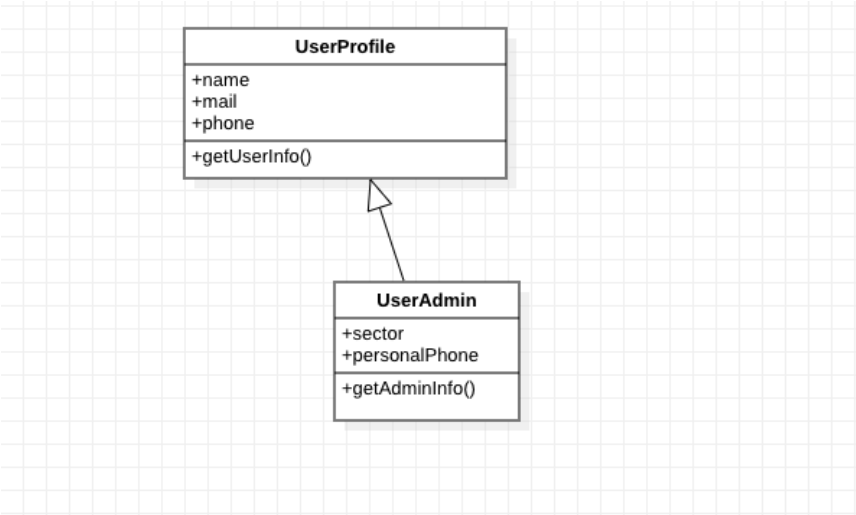
```
const exampleUser1 = new UserProfile("José", "jose@gmail.com",
« 09876543");
```

```
const exampleUser2 = new UserProfile("Sarah", "sarah@gmail.com",
"063736252");
exampleUser2.getProfileInfo();
```

Dans notre cas Il n'y aura que sur des new UserProfile que l'on pourra utiliser la méthode getProfileInfo( ).

L'héritage

Avec les classes on peut également profiter d'un système d'héritage, cela signifie que nous pouvons étendre (**extends**) les propriétés, les méthodes d'une classe vers une autre,
Par exemple dans notre application on gère déjà des utilisateurs, mais on veut aussi gérer des utilisateurs un peu plus spécifiques : des Admin, les admins ils auraient les mêmes propriétés que les utilisateurs (un nom, un mail, un téléphone) mais avec des informations en plus (le **secteur** dans lequel l'admin travaille, et son **Téléphone personnel**)
on crée une nouvelle classe « enfant » qui hérite des propriétés et des méthodes d'une classe parent.



⚠ une classe enfant peut hériter d'une classe parent mais l'inverse n'est pas possible.
Sur une instance de **UserAdmin** on pourra utiliser **getProfileInfo**( ) mais sur une instance de **UserProfile** on ne peut pas utiliser **getAdminInfo**( ).

Dans le code : on va utiliser **extends** et **super( )**

```
class UserProfile {
    //! Pas besoin de déclarer function devant le constructor et méthodes
    constructor(nameUser, mailUser, phoneUser) {
        this.nameUser = nameUser;
        this.mailUser = mailUser;
        this.phoneUser = phoneUser;
    }
    getProfileInfo() {
        return `infos de l'utilisateur :
            son nom : ${this.nameUser}
            son mail : ${this.mailUser}
            son Tél : ${this.phoneUser}`;
    }
}

const exampleUser1 = new UserProfile("José", "jose@gmail.com", "09876543");
const exampleUser2 = new UserProfile("Sarah", "sarah@gmail.com", "063736252");
exampleUser2.getProfileInfo();

class UserAdmin extends UserProfile{
    constructor(unNom,unMail,unPhone,sector,personnalPhone){
        super(unNom,unMail,unPhone); //! Appel au constructor du parent
        this.sector = sector;
        this.personnalPhone = personnalPhone;
    }
    getAdminInfo(){
        return `infos de l'utilisateur :
        son nom : ${this.nameUser}
        son secteur d'intervention : ${this.sector}
        son Tél Personnel : ${this.personnalPhone}`;
    }
}

const exampleAdmin1 = new UserAdmin('Jacky','jack@gmail.com','012345678','administration','0987654323');

console.log(exampleAdmin1.getAdminInfo());
```

La guerre des langages

⚔️ En programmation on peut distinguer à peu près tous les langage en 2 familles, les langages basés sur les **classes** et ceux basés sur les **prototypes**.
Js est un langage orienté objet basé sur les prototypes. (C'est pour cela que l'on dit qu'en Javascript TOUT est Objet)

Le JavaScript est un langage objet basé sur les prototypes. Cela signifie que le JavaScript ne possède qu'un type d'élément : **les objets** et que tout objet va pouvoir partager ses propriétés avec un autre, c'est-à-dire servir de prototype pour de nouveaux objets.
L'héritage en JavaScript se fait en remontant la chaine de prototypage.

En plus de la manière déclarative (créer un variable et lui assigner directement une valeur) dans Javascript on va retrouver également un système de constructor pour créer tout type d'objet

Exemple de fiche récapitulative des types d'objets en JS (non exhaustif) :

On peut déclarer soit en utilisant le constructor new Array( ), mais il faut penser à stocker dans une variable, nativement dans JS pour chaque objet on aura des propriétés de base, ici length commun à plusieurs type et JS propose aussi des fonctions de base, utiles pour manipuler chaque type d'objets. (Toujours avoir le réflexe d'aller consulter la documentation, NE PAS ré inventer la ROUE)

Violet : créer une variable (via constructor ou en mode déclaratif)
Jaune : exemple d'une propriété
Blanc : des fonctions de bases