Cégep de Saint-Hyacinthe
Département d'informatique

Programmation client-serveur

420-2RP-HY

Notes de cours

JavaScript

Programmation orientée objet

Enseignante :
Giovana Velarde

Hiver 2022

1 Table des matières

JavaScript orienté objet (JSOO)

Le JavaScript est un langage qui possède un fort potentiel objet. En effet, ce langage utilise les objets dans sa syntaxe même et la grande partie des éléments que nous manipulons en JavaScript sont en fait des objets ou vont pouvoir être convertis en objets et traités en tant que tel.

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 (Giraud, 2020).

1. La programmation orienté objet

L'idée de base de la programmation orientée objet (POO) consiste à apporter une structure au code (en utilisant des objets) pour modéliser les objets du monde réel que l'on souhaite représenter dans nos programmes et de fournir un moyen simple d'accéder à une fonctionnalité qu'il serait difficile d'utiliser autrement.

La POO encode à la fois les données et les manipulations qu'on veut faire. Elle nous permet de décomposer notre projet en objets. Chaque objet est indépendant des autres objets, mais ensemble ils forment le programme complet. Les objets peuvent aussi servir pour stocker des données et les transférer facilement sur un réseau.

Imaginons que nous voulons utiliser une stéréo. La stéréo a plusieurs propriétés.

- On n'a pas besoin de comprendre ce qui se passe en dessous pour l'utiliser (encapsulation).
- On peut modifier son état avec des boutons
- On peut ajouter des choses dessus, comme de meilleurs haut-parleurs.
- On n'a pas besoin d'acheter autres choses pour le faire fonctionner.
- La stéréo ne plantera pas !

1.1 Classe

On va utiliser un type qu'on appelle `Classe` pour représenter des structures complexes.

- Une classe doit avoir une **interface** bien définie pour que l'utilisateur puisse l'utiliser facilement.
- Une classe doit représenter un concept clair.
- Une classe doit être complète et documentée.
- Le code ne doit pas planter.

Une classe aura deux types de composantes :

- Les **propriétés** : L'état dans lequel on est.
- Les **méthodes** : des fonctions qui modifient notre état.

Imaginons qu'on veut programmer notre stéréo. Les propriétés seraient :

- `volumeCourant : int` Pour garder le niveau du volume
- `posteCourant : int` Pour garder le poste d'émission courant
- `allume : boolean` Pour garder l'état (allumé / non allumé)

Donc, à tous moments, notre stéréo à un volume, un poste et on sait si elle est éteinte ou non. Maintenant, on veut pouvoir modifier ces états. On aura donc différentes fonctions pour modifier ces états :

- `changerVolume(int nouveauVolume) : void` Pour modifier le volume
- `changerPoste(int nouveauPoste) : void` Pour modifier le poste d'émission
- `allumer() : void` Pour l'allumer (`allume = true`)
- `eteindre() : void` Pour l'éteindre (`allume = false`)

1.2 Objet

Un objet est une **instance** d'une classe. Imaginons notre stéréo. On a un seul plan pour toutes nos stéréos, mais on en a créé plusieurs. Une **classe** c'est le plan et un objet c'est un stéréo créé avec ce plan. Ce qu'on fait sur le plan répercute dans tous les objets créés avec ce plan. Mais, ce qu'on fait sur un objet ne se fait pas sur tous les autres objets. Par exemple, si je monte le volume sur mon stéréo, cela n'affectera pas le volume des autres stéréos créés avec le même plan.

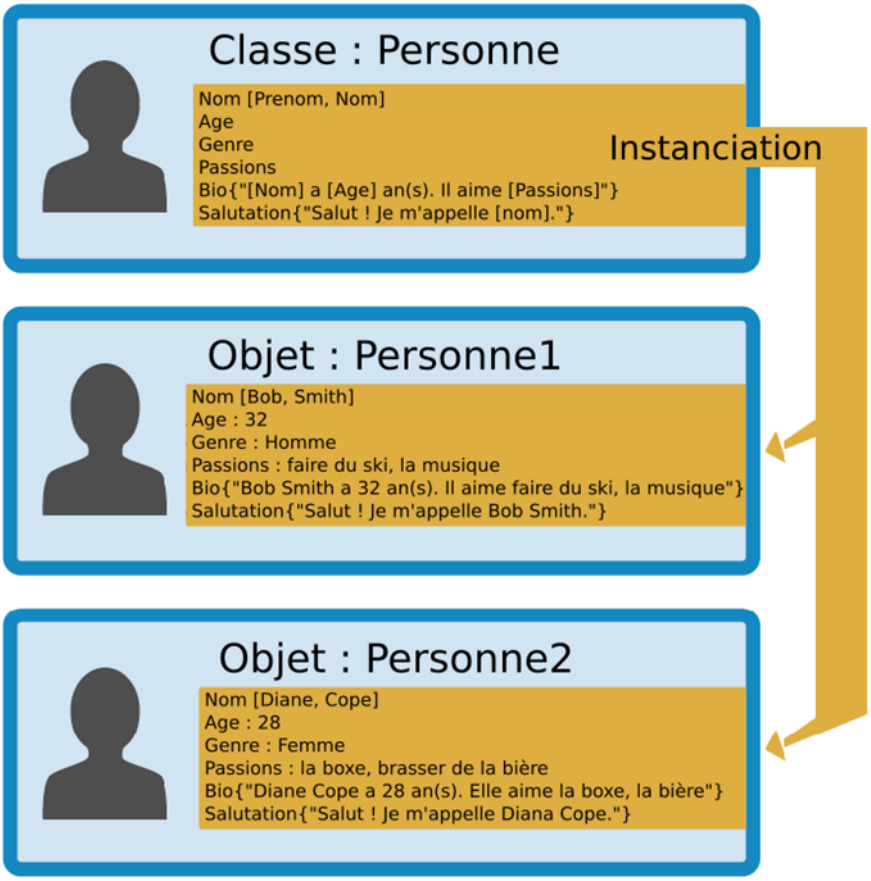

Figure 1 (MDN, 2021)

1.3 Une classe en JavaScript.

Pour faire une **classe**, donc un « plan » on utilise le mot clef `classe`. Ensuite, on doit faire une fonction appelée « **constructeur »** dans la classe qui permettra de créer l'**objet** lors de l'instanciation et définir son **état initial** avec des **propriétés**. On utilise le mot clef `constructor`. Le constructeur prend en argument les valeurs des propriétés à initialiser et met l'état initial de notre objet. Si les objets ont toujours le même état initial, on n'a pas besoin de passer des arguments, on peut simplement les mettre à la bonne valeur.

Exemple :

```
class Stereo {
   constructor(volumeCourant, posteCourant, allume) {
      this.volumeCourant = volumeCourant;
      this.posteCourant = posteCourant ;
      this.allume = allume;
   }
}

monStereo = new Stereo(3, 98.5, 0);
monDeuxiemeStereo = new Stereo(0, 93.3, 1);

console.log(monStereo.allume);
console.log(monStereo.posteCourant);
console.log(monStereo.volumeCourant);

console.log(monDeuxiemeStereo.allume);
console.log(monDeuxiemeStereo.posteCourant);
console.log(monDeuxiemeStereo.volumeCourant);
```

```
0
98.5
3
1
93.3
0
```

Dans l'exemple précédent, on suppose que les stéréos n'ont pas tous le même état initial. On crée donc une classe `Stereo`. Son constructeur prend donc l'état initial en entrée : le volumen courant (`volumeCourant`), le poste d'émission courant (`posteCourant`), et l'état allumé ou éteint (`allume`). Ensuite, il met à jour les propriétés respectifs (`volumeCourant`, `posteCourant`, `allume`). Le mot clef `this` permet d'accéder à la propriété en dehors de la classe. Si on ne le met pas, on ne pourrait pas y accéder en dehors de la définition de la classe.

```
class Stereo {
   constructor(volumeCourant, posteCourant, allume) {
      this.volumeCourant = volumeCourant;
      this.posteCourant = posteCourant ;
      allume = allume;
   }
}
```

```
monStereo = new Stereo(3, 98.5, 0);
monDeuxiemeStereo = new Stereo(0, 93.3, 1);

console.log(monStereo.allume);
console.log(monStereo.posteCourant);
console.log(monStereo.volumeCourant);

console.log(monDeuxiemeStereo.allume);
console.log(monDeuxiemeStereo.posteCourant);
console.log(monDeuxiemeStereo.volumeCourant);
```

```
undefined
98.5
3
undefined
93.3
0
```

Dans l'exemple précédent, il n'y a pas le mot clef `this` devant allume. Cela signifie que, en dehors du constructeur, on ne peut pas accéder à la valeur `allume`.

1.4 Méthodes

Maintenant que nous avons notre état initial et nos propriétés, nous voulons pouvoir modifier notre objet. Nous allons donc définir des **méthodes** dans notre « plan ». Les méthodes sont énumérées après le constructeur.

- On doit avoir une méthode `changerVolume(nouveauVolume)`. Lorsqu'on appelle cette méthode, on doit lui passer en argument le nouveau volume de notre stéréo.
- On doit avoir une méthode `changerPoste(nouveauPoste)`. Lorsqu'on appelle cette méthode, on doit lui passer en argument le nouveau poste de notre stéréo.
- On peut aussi avoir une méthode `allumer()` qui allume la radio et une méthode `eteindre()` qui éteint la radio.

```
class Stereo {
   //constructor(volumeCourant, posteCourant, allume) ...

   changerVolume(nouveauVolume){
      this.volumeCourant = nouveauVolume;
   }

   changerPoste(nouveauPoste){
      this.posteCourant = nouveauPoste ;
   }

   allumer(){
      this.allume = 1;
   }

   eteindre(){
      this.allume = 0;
```

```
    }
}

monStereo = new Stereo(3, 98.5, 0);
monDeuxiemeStereo = new Stereo(0, 93.3, 1);

monStereo.allumer();
monStereo.changerVolume(5);
monStereo.changerPoste(107.9);

monDeuxiemeStereo.eteindre();
monDeuxiemeStereo.changerVolume(30);
monDeuxiemeStereo.changerPoste(89.5);

console.log(monStereo.allume);
console.log(monStereo.posteCourant);
console.log(monStereo.volumeCourant);

console.log(monDeuxiemeStereo.allume);
console.log(monDeuxiemeStereo.posteCourant);
console.log(monDeuxiemeStereo.volumeCourant);
```

```
0
98.5
3
1
93.3
0
30
```

DEUXIÈME EXEMPLE. Imaginons que nous voulons créer un objet « chien ». Quelles propriétés peut avoir un chien? Il peut avoir un nom, un âge et une race. Est-ce que ces propriétés ont une valeur par défaut? Non puisqu'ils sont différents pour tous les chiens que nous allons créer. Nous allons donc devoir les passer en argument dans le constructeur comme dans l'exemple précédent.

```
class Chien{
   constructor(nom, age, race) {
      this.nom = nom;
      this.race = race;
      this.age = age;
   }
}
```

Quelle méthode pourrait avoir notre classe `Chien`? Elle pourrait avoir une méthode `aboyer` qui imprime « bark ! ». Elle pourrait avoir une méthode `vieillir` qui ajoute un à l'âge du chien. Elle pourrait avoir une méthode `obtenirInformation` qui retourne le nom et la race du chien. Ces méthodes sont faites dans l'exemple suivant.

```
class Chien{
   //constructor
```

```
   aboyer(){
      console.log("bark!");
   }

   vieillir(){
      this.age += 1;
   }

   obtenirInformation(){
      return this.nom + ", " + this.race;
   }
}

monChien = new Chien("Firoulaïs", 4, "Golden");

monChien.aboyer();
monChien.vieillir();

console.log(monChien.age);
console.log(monChien.obtenirInformation());
```

```
bark!
5
Firoulaïs, Golden
```

IL EST POSSIBLE D'INCLURE UNE INSTANCE D'UNE CLASSE COMME ENTRÉE D'UNE MÉTHODE D'UNE CLASSE. Imaginons une classe `Compte`. Cette classe représente l'état de compte d'une personne. Elle a comme propriétés `numeroCompte` et `balance`.

```
class Compte{
   constructor(numeroCompte, balance){
      this.numeroCompte = numeroCompte;
      this.balance = balance;
   }
}
```

Les méthodes possibles pour la classe `Compte` sont `ajouterArgent(montant)` ou `retirerArgent(montant)`.

```
class Compte{
   //constructor(numeroCompte, balance)

   ajouterArgent(montant){ //créditer
      this.balance += montant;
   }

   retirerArgent(montant){ //débiter
      this.balance -= montant;
   }
}
```

On pourrait aussi avoir une méthode `transfert(compte, montant)` qui prendre un autre **compte** (instance de la classe `Compte`) et qui transfert un montant de ce compte vers un autre.

```
class Compte{
   //constructor

   ajouterArgent(montant){
      this.balance += montant;
   }

   //retirerArgent

   transfert(compte, montant){
      this.balance -= montant;
      compte.ajouterArgent(montant);
   }
}
```

IL EST POSSIBLE D'UTILISER UNE INSTANCE D'UNE CLASSE COMME PROPRIÉTÉ D'UNE AUTRE CLASSE. Par exemple, on pourrait avoir une classe `Client` qui a comme propriétés: `nom`, `prenom`, `compteEpargne`, `compteCheque`. Cette classe pourrait avoir des méthodes telles que : `ajouterArgentEpargne(montant)`, `ajouterArgentCheque(montant)`, `retirerArgentEpargne(montant)`, `retirerArgentCheque(montant)`, `transfertVersEpargne(montant)`, `tranfertVersCheque(montant)`, `obtenirInformation`.

```
class Client{
   constructor(nom, prenom, compteEpargne, compteCheque){
      this.nom = nom;
      this.prenom = prenom;
      this.compteEpargne = compteEpargne;
      this.compteCheque = compteCheque;
   }

   obtenirInformation(){
      return this.nom + ", " + this.prenom
                + "  compte chèques: " + this.compteCheque.balance
                + "  compte d'épargne: " + this.compteEpargne.balance;
   }

   ajouterArgentEpargne(montant){
      this.compteEpargne.ajouterArgent(montant);
   }

   ajouterArgentCheque(montant){
      this.compteCheque.ajouterArgent(montant);
   }

   retirerArgentEpargne(montant){
      this.compteEpargne.retirerArgent(montant);
```

```
    }

    retirerArgentCheque(montant){
        this.compteCheque.retirerArgent(montant);
    }

    transfertVersEpargne(montant){
        this.compteCheque.transfert(this.compteEpargne, montant);
    }

    tranfertVersCheque(montant){
        this.compteEpargne.transfert(this.compteCheque, montant);
    }
}
compteEpargne = new Compte('CE10255', 0);
compteCheque = new Compte('CC10255', 1000);

compteCheque.ajouterArgent(500);
compteCheque.retirerArgent(200);
compteCheque.transfert(compteEpargne, 200);

client = new Client('Vigoureux','Christian', compteEpargne, compteCheque);
console.log(client.obtenirInformation());

client.ajouterArgentCheque(1000);
console.log(client.obtenirInformation());

client.transfertVersEpargne(500);
console.log(client.obtenirInformation());
```

```
Vigoureux, Christian  compte chèques: 1100  compte d'épargne: 200
Vigoureux, Christian  compte chèques: 2100  compte d'épargne: 200
Vigoureux, Christian  compte chèques: 1600  compte d'épargne: 700
```

1.5 Héritage

En 2015, le JavaScript (ES6) a introduit une syntaxe utilisant les classes pour son modèle objet. Cette syntaxe est copiée sur les langages orientés objets basés sur les classes et nous permet concrètement de mettre en place l'héritage en JavaScript plus simplement.

Attention cependant : cette syntaxe n'introduit pas un nouveau modèle d'héritage dans JavaScript ! En arrière-plan, le JavaScript va convertir les classes selon le modèle prototypé (Giraud, 2020).

Un héritage est une liste de bien qu'une personne qui décède remet à quelqu'un. C'est ce concept qu'on veut représenter dans la POO. Une classe `B` hérite d'une classe parent `A` si elle a les mêmes propriétés et les mêmes méthodes que la classe `A`. L'implémentation de la classe `B` est donc la même que la classe `A`, mais il se peut que la classe `B` ait plus de méthodes et des propriétés que la classe `A`.

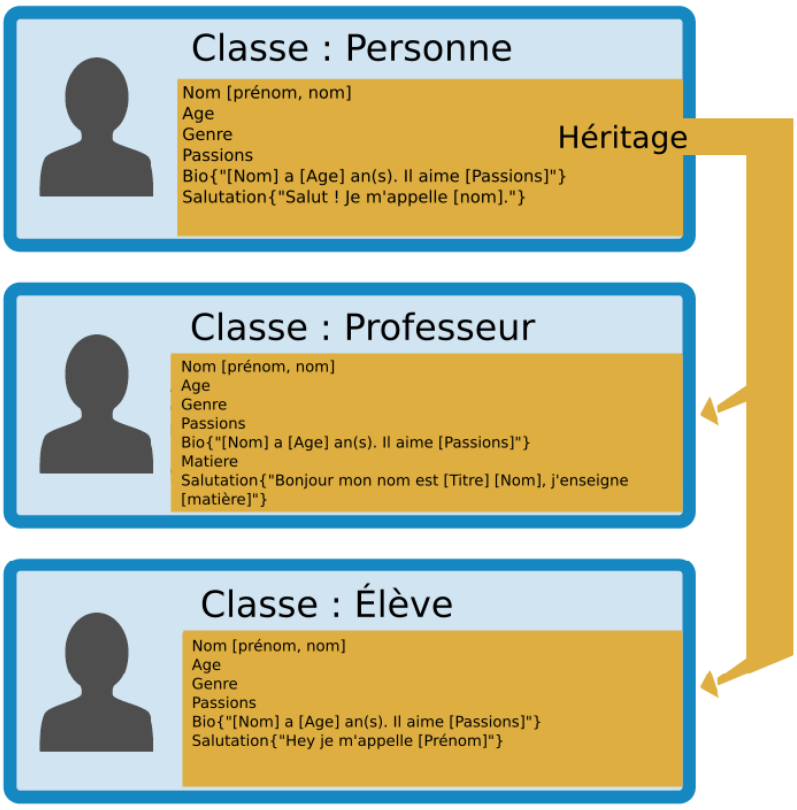

Figure 2 (MDN, 2021)

Une fois la classe fille créée il est alors possible de l'instancier et de créer des objets.

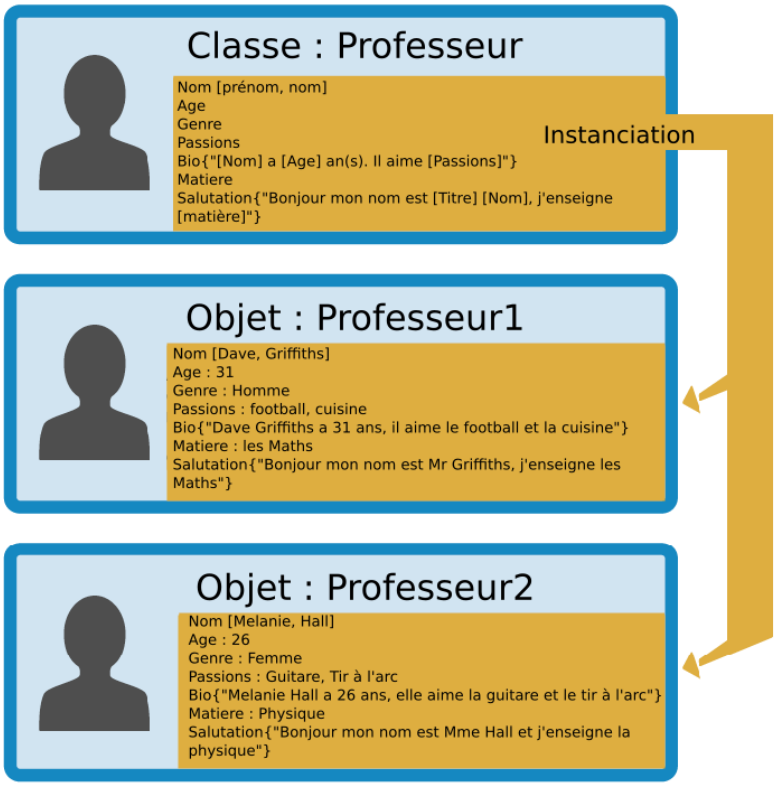

Figure 3 (MDN, 2021)

Prenons une classe `Docteur` telle que présentée dans l'exemple suivant. Les propriétés de cette classe sont `nom` et `prenom` et sa seule méthode est `obtenirInformation` qui permet de retourner une information.

```
class Docteur{
   constructor(nom, prenom, endroitInternat, endroitExternat){
      this.nom = nom;
      this.prenom = prenom;
      this.endroitExternat = endroitExternat;
      this.endroitInternat = endroitInternat;
   }

   obtenirInformation(){
      return this.nom + ", " + this.prenom;
   }
}
```

Imaginons maintenant une classe `Chirurgien`. Un chirurgien est un docteur, avec une spécialisation. Il a donc les mêmes compétences qu'un docteur, mais il a dû se spécialiser en chirurgie. On ne veut donc pas réécrire tout ce que nous avons écrit dans la classe `Docteur`. Ce serait redondant. On veut pouvoir partir de la classe `Docteur` et étendre cette classe pour avoir un classe `Chirurgien`. Pour ce faire, il faut déclarer la classe comme on le fait habituellement et ajouter le mot clef `extends` avec le nom de la classe parent `Docteur`. Si aucun constructeur n'est déclaré, celui de la classe `Chirurgien` prendra automatiquement le constructeur de la classe parent.

```
class Chirurgien extends Docteur{

}
unDocteur = new Docteur("Charles", "Carmichael", "Burbank", "Burbank");
console.log(unDocteur.obtenirInformation());
unChirurgien = new Chirurgien("Maxime", "Tremblay", "Burbank", "Burbank");
console.log(unChirurgien.obtenirInformation());
```

```
Charles, Carmichael
Maxime, Tremblay
```

Il est maintenant possible d'ajouter des méthodes à notre classe `Chirurgien`. Ces méthodes ne seront pas disponibles pour la classe `Docteur`. Dans l'exemple suivant, nous ajoutons la méthode `obtenirProfession` dans la classe `Chirurgien`.

```
class Chirurgien extends Docteur{

   obtenirProfession(){
      console.log("Je suis chirurgien");
   }
}

unChirurgien = new Chirurgien("Maxime", "Tremblay", "Burbank", "Burbank");
unChirurgien.obtenirProfession()
unDocteur = new Docteur("Charles", "Carmichael", "Burbank", "Burbank");
```

```
unDocteur.obtenirProfession()
```

```
Je suis chirurgien
TypeError : unDocteur.obtenirProfession is not a function
```

On peut vouloir ajouter des entrées au constructeur initial. Par exemple, on pourrait vouloir ajouter l'endroit où le chirurgien a fait sa spécialisation. On va donc réécrire un constructeur comme dans l'exemple suivant. Ce constructeur va remplacer le constructeur hérité du parent. Pour conserver les propriétés provenant de la classe parent `Docteur` (`nom`, `prenom`, `endroitInternat`, `endroitExternat`), on va utiliser le mot clef `super` sur les arguments qui sont les mêmes que ceux de la classe parent. Ce mot clef appellera le constructeur de la classe parent avec les propriétés passés en entrée. On assigne ensuite les autres entrées à leurs propriétés propres à la classe enfant. Dans l'exemple suivant on ajoute la propriété `endroitSpecialisation` a la classe `Chirurgien`.

```
class Chirurgien extends Docteur{
   constructor(nom, prenom, endroitInternat, endroitExternat, endroitSpecialisation){
      super(nom, prenom, endroitInternat, endroitExternat);
      this.endroitSpecialisation = endroitSpecialisation;
   }

   obtenirProfession(){
      console.log("Je suis chirurgien");
   }
}
unChirurgien = new Chirurgien("Maxime","Tremblay","Burbank","Burbank","San Francisco");
console.log(unChirurgien.endroitSpecialisation);

unDocteur = new Docteur("Charles", "Carmichael", "Burbank", "Burbank");
console.log(unDocteur.endroitSpecialisation);
```

```
San Francisco
undefined
```

On peut également vouloir réécrire une fonction héritée. Il suffit de la redéfinir dans la classe enfant.

```
class Chirurgien extends Docteur{
   //constructor

   obtenirProfession (){
      console.log("Je suis chirurgien");
   }

   obtenirInformation(){
      return this.nom + ", " + this. prenom + ". Chirurgien";
   }
}

unChirurgien = new Chirurgien("Maxime","Tremblay","Burbank","Burbank","San Francisco");
console.log(unChirurgien.obtenirInformation());

unDocteur = new Docteur("Charles", "Carmichael", "Burbank", "Burbank");
console.log(unDocteur.obtenirInformation());
```

```
Maxime, Tremblay Chirurgien
Charles, Carmichael
```

1.6 Diagramme de classe UML

UML (*Unified Modeling Language*) est un langage de modélisation orienté objet, c'est-à-dire que toutes les entités modélisées sont des objets ou se rapportent à des objets. C'est la notation de modélisation la plus répandue dans le monde qui permet de représenter le monde réel dans un modèle abstrait et simplifié. Étant graphique, il permet de visualiser le système réalisé. Il y a plusieurs types de diagramme UML.

Le **diagramme de classe** est un type de diagramme UML de structure qui représente la conception logique et physique d'un système en décrivant ses **classes** et les **liens** entre celles-ci.

On représente une classe par un rectangle divisé en trois parties. La première partie est le nom de la classe, la deuxième partie est utilisée pour décrire ses propriétés et la dernière partie est utilisée pour décrire ses méthodes.

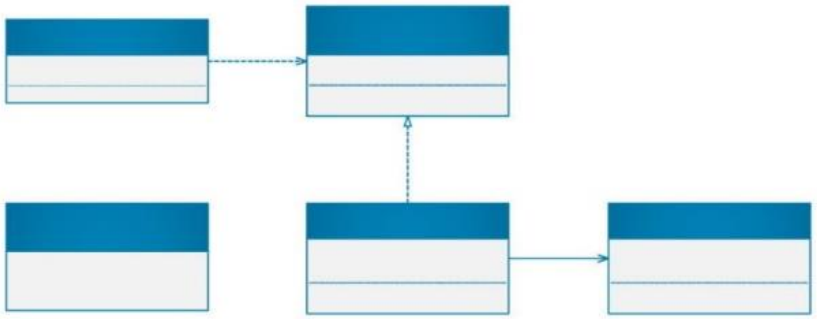

Figure 4 (Team, 2019)

1.6.1 Représentation d'une classe

Une classe est composée d'un nom, d'attributs et d'opérations.

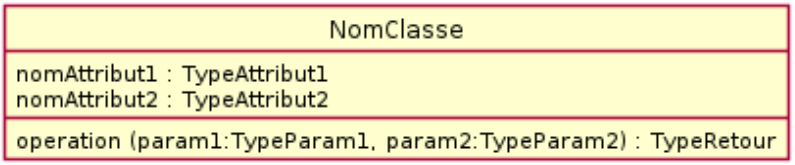

Figure 5 (Lipn, s.d.)

- Le nom d'une classe commence par une majuscule
- Le nom d'une propriété commence par une minuscule
- Les types de base (string, number, boolean, …) sont en minuscules
- Il n'a pas d'espace dans les noms de classes ou de propriétés
- Pour les noms composés, on fait commencer chaque mot par une majuscule (Camel case)

DESCRIPTION DES PROPRIÉTÉS : On va mettre le nom de la propriété et sa multiplicité entre crochets. Par exemple, si quelqu'un a deux adresses courriel, on mettra `courriel[2]` au lieu de deux propriétés `courriel1` et `courriel2`.

DESCRIPTION DES MÉTHODES : On va mettre le nom de la méthode, les arguments qu'elle prend en entrée (entre parenthèses) et deux points avec le type que la méthode retourne en sortie. Par exemple, une méthode retournant un entier et prenant un format date en entrée sera décrite comme ceci : `calculerAge(dateNaissance : Date) : int`.

1.6.2 Relations entre classes

- La relation **d'héritage** est une relation de généralisation/spécialisation permettant l'abstraction de concepts.
- Une relation de **composition** décrit une relation de contenance et d'appartenance.
- Une **association** représente une relation possible entre les objets d'une classe.
- Une **dépendance** est une relation unidirectionnelle exprimant une dépendance sémantique entre les éléments du modèle.

REPRÉSENTATION DE L'HÉRITAGE : On représente une classe `B` (classe enfant) héritant d'une classe `A` (parent) avec un **triangle** suivi d'une flèche comme dans la figure 6. Ici, `B` a deux propriétés supplémentaires, `d` et `unObject`.

REPRÉSENTATION DE LA COMPOSITION : On représente une classe `B` se composant d'instances d'une classe `C` avec un **losange** (suivi d'une flèche) comme dans la figure 6. Ici, `B` a donc une propriété d'une classe `C`. (Sur la flèche), on peut écrire `1` si `B` a une seule propriété de la classe `C` ou `N` (ou `1..*`) si il peut en avoir plusieurs.

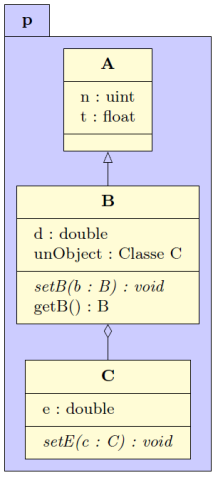

Figure 6

Exemple montrant les liens entre classes :

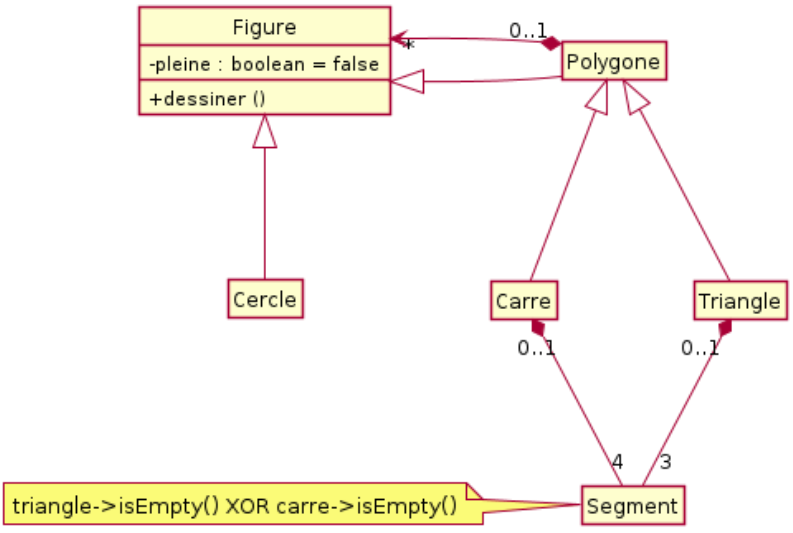

Figure 7 (Lipn, s.d.)

2 Références

Giraud, P. (2020). *Apprendre à coder en JavaScript*. Récupéré sur https://www.pierre-giraud.com/javascript-apprendre-coder-cours/

Lipn. (s.d.). *UML Cours 1 : Diagrammes de classes : associations*. Récupéré sur Lipn: https://lipn.univ-paris13.fr/~gerard/uml-s2/uml-cours01.html

MDN. (2021). *Le JavaScript orienté objet pour débutants*. Récupéré sur MDN Web Docs Mozilla: https://developer.mozilla.org/fr/docs/Learn/JavaScript/Objects/JS_orient%C3%A9-objet

Team, M. 3. (2019, septembre 24). *Guide simple des diagrammes UML et de la modélisation de base de données*. Récupéré sur Microsoft: https://www.microsoft.com/fr-ca/microsoft-365/business-insights-ideas/resources/guide-to-uml-diagramming-and-database-modeling