



**KEEP
CALM
AND
<CODE/>**

Programmation orientée objet

Objet et classe : getter et setter

Sylvie TROUILHET - www.irit.fr/~Sylvie.Trouilhet

La classe Voiture

```
class Voiture {  
    constructor(marque, an) {  
        this.marque=marque  
        this.an=an  
    }  
}
```

```
const v1=new Voiture("Dacia", 2013)  
console.log(v1.marque)
```

```
3\Voiture.js  
Dacia
```

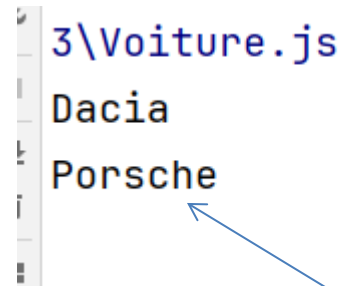
Peut-on transformer la Dacia en Porsche ?

```
const v1=new Voiture("Dacia", 2013)
```

```
console.log(v1.marque)
```

```
v1._marque="Porsche"
```

```
console.log(v1.marque)
```



```
3\Voiture.js  
Dacia  
Porsche
```

Contrôler la modification ?

Attribut privé

Un attribut privé ne peut pas être utilisé en dehors de la classe
Il doit être déclaré en premier dans la classe :

```
class Voiture {  
    #marque  
    constructor(marque, an) {  
        this.#marque=marque  
        this.an=an  
    }  
}
```

→ L'utilisation à l'extérieur de la classe provoque une erreur.

```
C:\Users\trouilhe\hubic\Enseign  
console.log(v1.#marque)  
                ^
```

Opérateur get

L'opérateur **get** permet de lier un attribut avec une fonction.

```
class Voiture {  
    #marque  
    constructor(marque, an) {  
        this.#marque=marque  
        this.an=an  
    }  
    get marque() { return this.#marque }  
}
```

→ Un pseudo-attribut `marque` est créé

Opérateur get

C'est la fonction qui est appelée lorsqu'on accède au pseudo-attribut.

```
class Voiture {  
    #marque  
    ...  
    get marque( ) { return this.#marque }  
}
```

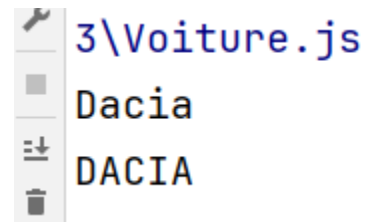
```
const v1=new Voiture("Dacia", 2013)  
console.log(v1.marque)
```

Opérateur set

L'opérateur set permet de lier un attribut à une fonction qui sera appelée lorsqu'on tente de modifier cet attribut.

```
set marque (val) {  
    if (val.toLowerCase() ==  
        this.#marque.toLowerCase())  
        this.#marque = val  
}
```

```
const v1 = new Voiture("Dacia", 2013)  
v1.marque = "Porsche"  
console.log(v1.marque)  
v1.marque = "DACIA"  
console.log(v1.marque)
```



```
3\Voiture.js  
Dacia  
DACIA
```



UNIVERSITÉ
TOULOUSE III
PAUL SABATIER

**KEEP
CALM
AND
<CODE/>**

Programmation orientée objet

Relation entre objets

Sylvie TROUILHET - www.irit.fr/~Sylvie.Trouilhet

Relations entre classes

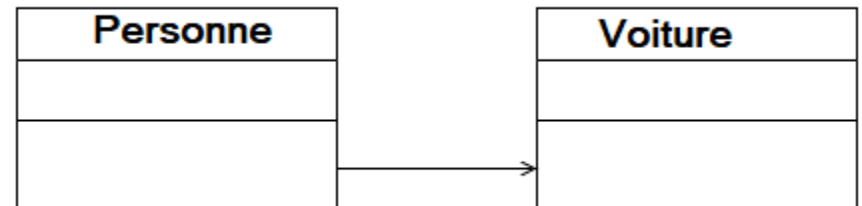
Deux types de relation sont possibles entre 2 classes :

- la délégation : relations “est relié à”, “a un / a des”
 - association
 - agrégation
 - composition
- l’héritage : relation “est un “

Association

Un objet peut faire appel à un autre objet pour rendre un service (relation de collaboration).

```
class Voiture {  
    accélérer (vitesse) {  
    }  
}  
  
class Personne {  
    déplacer () {  
        this.maVoiture.accélérer(10)  
    }  
}
```

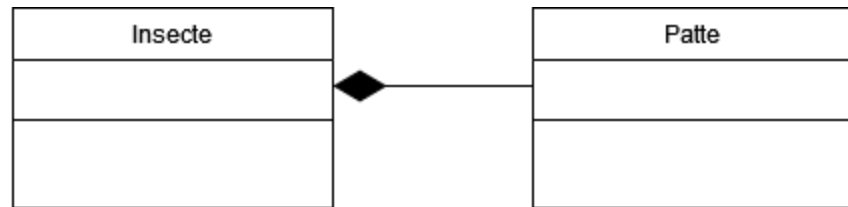


Composition

Une classe est composée d'instances d'une autre classe.

Couplage fort entre les deux classes : la suppression d'une instance de la classe composée entraîne la suppression des objets qui la compose

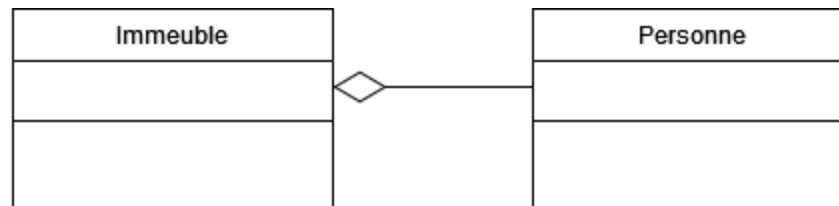
Par exemple : **class** Insecte {
 #pattes
 ...
}



Agrégation

Couplage moins fort que la composition : la disparition d'une instance de la classe agrégée n'entraîne pas la suppression des objets qui la compose

Par exemple : **class** Immeuble {
 occupants
 ...
}



Héritage

L'héritage permet de réutiliser une classe, en la spécialisant.

Une classe fille hérite de tous les attributs et méthodes non privés de la classe mère.

Il est possible d'ajouter des attributs et/ou des méthodes supplémentaires dans la classe fille.

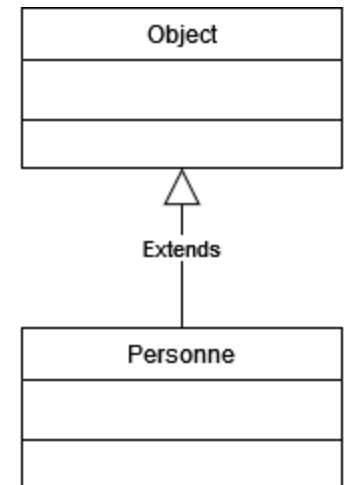
Héritage

Toute classe hérite d'une et d'une seule classe (à l'exception de la classe Object) : par défaut, c'est la classe Object

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Global_Objects/Object

class Personne

⇔ **class** Personne **extends** Object



Exemple d'héritage

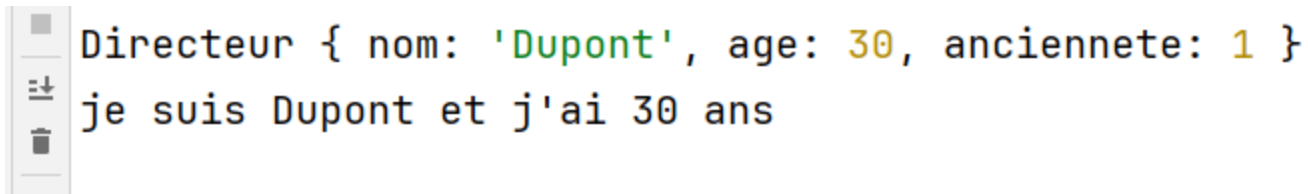
```
class Personne {  
    constructor (nom, age) {  
        this.nom = nom  
        this.age = age  
    }  
    sePresenter () {  
        return `je suis ${this.nom} et j'ai ${this.age} ans`  
    }  
}
```

```
class Directeur extends Personne {  
    constructor(nom, age, anciennete){  
        super(nom, age)  
        this.anciennete=anciennete  
    }  
}
```

Toujours sur la
première ligne

Exemple d'héritage

```
const d1=new Directeur("Dupont", 30, 1)
console.log(d1)
console.log(d1.sePresenter())
```



```
Directeur { nom: 'Dupont', age: 30, anciennete: 1 }
je suis Dupont et j'ai 30 ans
```

C'est la méthode de *Personne* qui est appelée.


Comment faire pour qu'un directeur se présente avec
« je suis le directeur » ?

Redéfinir une méthode héritée

Redéfinition : **changer** le corps de la méthode héritée.

```
class Directeur extends Personne {  
    ...  
    sePresenter () {  
        return `je suis le directeur`  
    }  
}
```

```
const d1=new Directeur("Dupont", 30, 1)  
console.log(d1.sePresenter())
```


 je suis le directeur

Redéfinir une méthode héritée

Redéfinition : **spécialiser** la méthode héritée.

```
class Directeur extends Personne {  
  ...  
  sePresenter () {  
    return `${super.sePresenter()}, je suis le directeur`  
  }  
}
```

```
const d1 = new Directeur('Dupont', 30, 1)  
console.log(d1.sePresenter())
```

 je suis Dupont et j'ai 30 ans, je suis le directeur