# CHAPITRE IV: La Programmation Orientée Objet jusqu'à ES5

Ce chapitre a pour but de mieux faire comprendre quelques concepts clés qui permettront de faire de la programmation orientée objet en JavaScript. Si vous êtes un programmeur chevronné, versé dans la programmation orientée objet, votre première approche du JavaScript risque d'être fortement influencée par votre expérience de langages tels que le Java et le C++, et il se peut que vous luttiez péniblement avec ce nouveau langage que vous essayez de maîtriser.

**JavaScript est il un langage de programmation orienté objet ?**

En fait le JavaScript est un langage **de programmation orienté objet par prototype.**

Le prototype est un objet à part entière et qui sert "de prototype de définition" de la structure interne et des méthodes. De ce prototype, par le mécanisme de clonage, sont créés les autres objets de mêmes types. Dans ce modèle, il n'y a plus de distinction entre attributs et méthodes : ce sont tous des **slots**. Un slot est un label de l'objet, privé ou public, auquel est attachée une définition : ce peut être une valeur ou des opérations. Cet attachement peut être modifié à l'exécution. Chaque modification ou ajout d'un slot impacte l'objet et l'ensemble de ses clones. Parmi les autres langages à prototype on trouve Self, Io, Slater, Lisaac, etc.

Quelles sont les principales spécificités des langages de POO ? L'encapsulation, l'héritage et le polymorphisme...

JavaScript, *à* sa manière, offre toutes ces possibilités

**Concepts des " langages objet" supportés par JavaScript**

| | |
|---|---|
| **Classe** | JavaScript supporte partiellement les classes. Il permet de définir des classes en se fondant sur les mécanismes des fonctions et des closures ainsi que sur leur attribut **prototype** mais ne fournit pas d'élément de langage dédié |
| **Classes et méthodes abstraites** | JavaScript ne supporte pas explicitement les classes et méthodes abstraites. Il peut néanmoins les **mettre en œuvre**, puisqu'une méthode d'une classe peut en appeler une autre qui n'existe pas dans la classe. La classe fonctionne correctement si une classe fille la définit. Si tel n'est pas le cas, une erreur se produit. JavaScript ne permet pas de verrouiller ce mécanisme. |
| **Composition et agrégation** | JavaScript supporte ces deux mécanismes, puisqu'il permet d'utiliser des variables par référence. |
| **Contrôle de l'héritage et visibilité** | JavaScript ne supportant pas le modificateur **final,** il ne permet pas de contrôler l'héritage dans le but d'empêcher de créer des sous-classes ou de surcharger des méthodes. Le langage ne supporte pas non plus le type de visibilité **protégé** |
| **Encapsulation et visibilité** | JavaScript ne fournit pas de mécanisme permettant de mettre en œuvre l'encapsulation et de supporter convenablement tous les niveaux de visibilité. Comme il ne permet pas de gérer dans tous les cas le niveau privé, seul le niveau de visibilité public peut être utilisé tout le temps. Dans ce cas, les différents attributs et méthodes d'un objet peuvent être accèdes depuis des entités extérieures. L'encapsulation **n'est alors pas garantie** par le langage.<br><br>Il est néanmoins possible de définir des attributs et des méthodes **privés** |
| **Héritage** | JavaScript permet de mettre en œuvre l'héritage mais ne fournit pas de mot-clé **extends** à cet effet. Comme nous le verrons, différentes techniques, fondées sur les constructeurs des classes mères et l'attribut prototype des fonctions, permettent cependant de mettre en œuvre l'héritage multiple. |
| **Mot-clé t h i s** | JavaScript supporte le mot-clé **this** dans une fonction en faisant référence à l'objet sur lequel la méthode est appelée. |
| **Package** | JavaScript ne fournit pas à proprement parler la notion de **package**. Il est cependant possible d'en simuler un en l'incorporant dans le nom d'une classe. |
| **Polymorphisme**<br><br>**Surcharge** | JavaScript ne supporte pas ce concept puisque la mise en œuvre de l'héritage avec ce langage consiste en une recopie des méthodes et attributs. De plus le langage n'est pas typé.<br><br>Il est possible par contre de simuler une **surcharge** de méthodes.( utilisation de arguments et typeof ) |
| **Typage** | JavaScript possède un typage dynamique. Le type d'un objet n'est donc connu qu'au moment de l'exécution du code et peut varier au cours de l'exécution. Le langage fournit toutefois les mots-clé **typeof** et **instanceof** afin de déterminer le type d'un objet. |

# I. LES OBJETS EN JAVASCRIPT

Le JavaScript ne requiert pas que l'on utilise des objets ou même des fonctions. Bien souvent, Il est possible d'écrire un programme JavaScript sous la forme d'un simple flux de texte exécuté directement à sa lecture par l'interpréteur. *A* mesure qu'un programme prend de l'ampleur, les fonctions et les objets deviennent cependant de plus en plus utiles pour l'organisation du code.
Le moyen le plus simple de créer un nouvel objet JavaScript consiste à exécuter le constructeur prédéfini de la classe Object ( classe de base de toutes les classes )

```
var monObjet = new Object();
```

Nous considérerons d'autres approches et verrons la fonction du mot clé **new** plus loin dans ce cours. L'objet **monObjet** est initialement "vide", ce qui signifie qu'il n'a ni propriété ni méthode. Voyons comment les ajouter. La classe Object() possède un attribut nommé **prototype** de type objet . Toutes les classes héritent de cet attribut.

# II. CONSTRUIRE DES OBJETS IMPROVISÉS

L'objet JavaScript n'est au fond rien de plus qu'un tableau associatif, avec des champs et des méthodes reliés par leur nom. Une syntaxe de type C est flanquée par-dessus afin de rendre le langage plus familier aux programmeurs C, mais l'implémentation sous-jacente peut être exploitée d'autres manières également. Il est possible de construire des objets complexes ligne par ligne, en ajoutant de nouvelles variables et méthodes à mesure que l'on y pense.

Il existe deux moyens de créer des objets de cette manière **improvisée**. Le premier consiste à se servir du JavaScript pour créer l'objet. Le second consiste à employer une notation spéciale connue sous le nom de **JSON**. Commençons par la technique JavaScript standard.

## II.1 Utiliser des instructions JavaScript

Au milieu d'un bloc de code compliqué, il peut être souhaitable d'attribuer une valeur à la propriété d'un objet. Les propriétés d'objets JavaScript sont en lecture/écriture et peuvent se voir attribuer des valeurs à l'aide de l'opérateur =. Ajoutons une propriété à un objet simple :

```
var voiture= new Object() ;       // voiture est un nouvel objet
voiture.couleur="rouge" ;         // couleur est une propriété de voiture
Object.prototype.longueur = 4  // Toute modification de l'attribut prototype de la classe
// Object (ajout d'attributs ou de méthodes) se répercute sur  les instances créées ou à venir
```

**Exercice:** Ajouter à la classe Object la méthode each() permettant de faire afficher chaque objet créé , tous les attributs et les méthodes le composant.

**Correction:**

```
Object.prototype.each=function(){for( i in this){alert(i + " : " + this[i])}}

var t = {"nom":"Dupont","prenom":"Pierre"}
t.each()  // affiche les propriétés et les valeurs du tableau t   plus la méthode each()

navigator.each()  // affiche les propriétés et les valeurs de l'objet navigator plus la méthode
each()
```

Dans un langage de programmation orientée objet structuré, nous serions contraints de définir une classe déclarant la propriété couleur et la méthode each(), sans quoi le compilateur générerait une erreur. Rien de tel en JavaScript. En fait, nous pouvons même référencer des propriétés à l'aide d'une syntaxe de tableau (associatif) :

```
voiture[ 'couleur' ]="rouge" ;
alert( voiture.couleur)
```

Cette notation est assez maladroite pour un usage ordinaire mais sera utile si le nom de la propriété se trouve dans une variable

```
var option="couleur"
voiture[ option ]="rouge" ;
alert( voiture.couleur)
```

Nous pouvons également ajouter une nouvelle fonction (anonyme) à notre objet de manière dynamique :

```
voiture.afficheCouleur=function ( ) {
                        alert( "Couleur   :   "+this. couleur) ;
}
```

Dans cette notation  le mot-clé **this** fait référence à l'objet voiture.

Ou emprunter une fonction prédéfinie :

```
function afficheCouleur(){
        alert('Couleur : '+this.couleur) ;
}
voiture.afficheCouleur=afficheCouleur;
```

Notez que lors de l'attribution de la fonction prédéfinie, nous omettons les parenthèses. Si nous écrivions :

```
voiture.afficheCouleur=afficheCouleur () ;
```

Nous exécuterions la fonction **afficheCouleur**() et attribuerions la valeur de retour (soit null dans ce cas) à la propriété **afficheCouleur** de **voiture**.

Des objets peuvent être attachés à d'autres objets de manière à construire des modèles de données complexes :

```
var maBiblio=new Object();
maBiblio .nombreDeLivres =345;
maBiblio.themes=new Array();
maBiblio.themes[0]="MERISE";
maBiblio.themes[1]="Javascript";
maBiblio.themes[2]="PHP";
maBiblio.themes[3]="UML";
maBiblio.themes[4]="PostgreSQL";
maBiblio.livres = new Array();
maBiblio.livres[0] = new Object();
maBiblio.livres[0].titre="Découvrir MERISE";
maBiblio.livres[0].auteurs = new Array();
```

```
maBiblio.livres[0].auteurs[0]= new Object();
maBiblio.livres[0].auteurs[0].nom="GRANET";
maBiblio.livres[0].auteurs[0].prenom="Jean";
maBiblio.livres[0].auteurs[1]= new Object();
maBiblio.livres[0].auteurs[1].nom="GRAS";
maBiblio.livres[0].auteurs[1].prenom="Paul";
maBiblio.livres[0].dateParution="12/11/2005"
maBiblio.livres[1] = new Object();
maBiblio.livres[1].titre="Tout sur UML";
maBiblio.livres[1].auteurs = new Array();
maBiblio.livres[1].auteurs[0]= new Object();
maBiblio.livres[1].auteurs[0].nom="WILD";
maBiblio.livres[1].auteurs[0].prenom="JOHN";
maBiblio.livres[1].dateParution="12/08/2004"
alert(maBiblio.livres[0].auteurs[1].prenom)
```

**Exercice:** Ajouter la méthode afficheAuteur(x,y) qui affiche le nom et le prénom du xième auteur du livre y

**Corrigé:**

```
maBiblio.afficheAuteur = function(x,y){
                                alert(this.livres[y].auteurs[x].nom+ this.livres[y].auteurs[x]. nom }
```

**Ou**

```
function afficheAuteur(x,y){
        alert(this.livres[y].auteurs[x].nom+ this.livres[y].auteurs[x]. nom
}

maBiblio.afficheAuteur = afficheAuteur
```

Cette syntaxe peut rapidement devenir fastidieuse et le JavaScript propose une notation compacte qui sert à assembler plus rapidement les graphes d'objets, connue sous le nom de **JSON**. Etudions-la à présent.

## II.2 Utiliser JSON

La notation **JSON** (**J**ava**S**cript **O**bject **N**otation) est une fonction essentielle du langage. Elle fournit un mécanisme concis pour la création de tableaux et de graphes d'objets. Pour comprendre **JSON**, vous devez connaître le fonctionnement des tableaux JavaScript. Commençons donc par là.

Le JavaScript possède une classe prédéfinie Array qui peut être instanciée à l'aide du mot clé **new** :

```
mois=new Array();
```

Les tableaux peuvent se voir attribuer des valeurs avec un index numérique, comme un tableau C ou Java classique :

```
mois[0]=31;
mois[1]=29;
```

Ils peuvent aussi être associés à une valeur de clé (propriété), comme un tableau associatif Java.

```
mois["janvier"]=31;
mois["fevrier"] =29;
```

Cette syntaxe est intéressante pour les ajustements précis, mais la construction d'un tableau ou d'un objet de grande taille peut être fastidieuse. Le raccourci pour la création des tableaux indexés numériquement consiste à employer des crochets, les entrées étant écrites sous la forme d'une liste de valeurs séparées par des virgules, comme ceci :

```
mois=[31,29,31,30,31,30,31,31,30,31,30,31];
```

Pour construire un objet JavaScript, on **utilise des accolades**, chaque valeur étant écrite sous la forme d'une paire **clé** (**propriété): valeur**,

```
mois = {
        janvier :    31,
        fevrier     : 29,
        mars        : 31
}
Utilisation: alert(mois.janvier) ou alert(mois['janvier'])
```

Les notations **JSON** peuvent être imbriquées afin de créer des définitions sur une (longue) ligne de hiérarchies d'objets complexes :

```
var maBiblio={
  nombreDeLivres : 345,
  themes: ["merise","javascript","php","uml","postgreSql"],
  livres:[
    { titre: "découvrir MERISE",
       Auteurs:[{nom:"GRANET",prenom:"Jean"},{nom:"GRAS",prenom:"Paul"}],
      dateParution : "12/11/2005"
     },
    { titre: "Tout sur UML",
      Auteurs : [{nom:"Wild",prenom:"John"}],
      dateParution : "12/08/2004"
     }
   ],
   afficheTitre : function(x){                  // déclaration d'une méthode
      alert(this.livres[x].titre)
  }

}
Exemples d'utilisations:

alert(maBiblio.nombreDeLivres)
alert(maBiblio.themes[3])
alert(maBiblio.livres[1].Auteurs[0].nom)
maBiblio.afficheTitre(1);
```

On peut aussi déclarer la méthode afficheTitre de la manière suivante:

```
maBiblio.afficheTitre=function(x){
        alert(this.livres[x].titre)
}
```

**Exercice**: Ajouter la méthode nombreAuteur(x) qui retourne le nombre d'auteurs du livre x.

**JSON** a été reconnu par JavaScript, après que l'ECMA ait définit la fonction **eval()** qui parse le format, dans le standard ECMAScript, en 1999.
Il a été popularisé par le développement **d'Ajax**. Le terme JSON revient souvent lorsque l'on parle d'Ajax. On sait qu'il s'agit d'un format de fichier alternatif à XML, et ce format à ses adeptes. L'intérêt est que l'on n'a plus besoin de parser un fichier XML pour extraire des informations à travers le net, car JSON est reconnu nativement par JavaScript.

**Les avantages de JSON:**

- La facilité de lecture.
- La simplicité de mise en œuvre

**menu.json**

```
var doc = {
  "menu": "Fichier",
  "commandes": [
      {
          "value": "Nouveau",
          "action":"CreateDoc"
      },
      {
          "value": "Ouvrir",
          "action": "OpenDoc"
      },
      {
          "value": "Fermer",
          "action": "CloseDoc"
      }
   ]
}
```

**Menu.xml**

```
<?xml version="1.0" ?>
<doc>
  <menu>Fichier</menu>
  <commands>
     <item>
         <value>Nouveau</value>
         <action>CreateDoc</action>
     </item>
     <item>
         <value>Ouvrir</value>
         <action>OpenDoc</action>
     </item>
     <item>
         <value>Fermer</value>
         <action>CloseDoc</action>
     </item>
  </commands>
</doc>
```

**Comparaison JSON et XML**

**Comment utiliser le format JSON**

**Coté client**

C'est particulièrement simple. JSON faisant partie de la norme JavaScript le contenu d'un fichier JSON, ou la définition de données dans ce format sont assignés à une variable, laquelle devient un objet du programme.

**Coté serveur**

Les fichiers au format JSON s'utilisent dans différents langages de programmation, notamment PHP et Java grâce à des parseurs qui permettent d'accéder au contenu, et éventuellement de le convertir en classes et attributs, dans ce langage.
Le site json.org fournit un parseur en C et donne une liste de parseurs pour d'autres langages.

**L'échange de données :**

La récupération d'un fichier peut se faire à partir de JavaScript de deux façons:
- inclusion directe du fichier dans la page HTML au même titre qu'un fichier .js de JavaScript.
- emploi de XMLHttpRequest (sera vu dans les prochains chapitres)

**1ère façon** :

```
<script language="JavaScript" src="menu.json"></script>
<script language="JavaScript" > alert(doc.menu)<script>
 Le fichier menu.json est identique à celui présenté ci-dessus
```

## II.3 Fonction constructeur, classes et prototypes

En programmation orientée objet, on crée généralement des objets en déclarant la classe à partir de laquelle on souhaite que ces objets soient instanciés. Le javaScript supporte le mot clé **new** qui permet de créer des instances d'un objet prédéfini.

## II.4 Création de la classe.

Il faut tout d'abord donner un **nom** à la classe et **nommer** ses propriétés.
Choisissons la classe **Produit,** ses propriétés seront:

- Reference
- Designation
- Prix
- Tva
- Photo

Pour créer la classe on utilise une fonction appelée: **constructeur d'objet** (le nom des classes commence en général par une majuscule).

```
function Produit (ref,des,pri,tv,pht){
        this.reference=ref;
        this.designation=des;
        this.prix=pri;
        this.tva=tv;
        this.photo=pht;
}
```

## II.5 Création d'une méthode pour la classe.

Chaque instance de la classe (objet) aura les mêmes propriétés que la classe.

Pour créer une instance il suffira d'écrire:

```
x= new Produit() ; x.reference="hp500" etc…

ou

x= new Produit("hp500","imprimante hp","100","19.6","hp500.gif")
```

Il est donc intéressant de disposer d'une méthode permettant d'afficher clairement toutes les propriétés
La fonction afficheProduit() sera une méthode pour la classe Produit.

```
function afficheProduit(){
        var r="";
        r +="<b>Reference: </b>"+this.reference+"<br/>";
        r +="<b>Designation: </b>"+this.designation+"<br/>";
        r +="<b>Prix: </b>"+this.prix+"<br/>";
        r +="<b>Tva: </b>"+this.tva+"<br/>";
        r +="<img src= "+this.photo+">";
        document.write(r);
}
```

**Remarque** : les notations utilisées pour la fonction **afficheProduit()** sont remplacées aujourd'hui par des méthodes de l'API **DOM** ( détaillée dans un futur chapitre ).

Cette méthode permet un affichage formaté d'un produit, il ne reste plus qu'à l'intégrer comme méthode de la classe Produit de la manière suivante

```
function Produit (reference,designation,prix,tva,photo){
        this.reference=reference;
        this.designation=designation;
        this.prix=prix;
        this.tva=tva;
        this.photo=photo;
        this.afficheProduit=afficheProduit;  // Attention sans ()
}
```

**Remarque** : on aurait pu ne pas déclarer la fonction afficheProduit() et l'intégrer directement dans la classe de la façon suivante:

```
function Produit (reference,designation,prix,tva,photo){
        this.reference=reference;
        this.designation=designation;
        this.prix=prix;
        this.tva=tva;
        this.photo=photo;
        this.afficheProduit=function(){     // utilisation d'une fonction anonyme
                var r="";
                r +="<b>Reference: </b>"+this.reference+"<br/>";
                r +="<b>Designation: </b>"+this.designation+"<br/>";
                r +="<b>Prix: </b>"+this.prix+"<br/>";
                r +="<b>Tva: </b>"+this.tva+"<br/>";
                r +="<img src= "+this.photo+">";
                document.write(r);
    }

}
```

```
Exemple d'utilisation:

t= new Array(19);

for ( i=0 ; i<=19 ; i++) {

        t[i]= new produit();

}

t[0].reference="hp500";t[0].designation="Imprimante HP";…….

for ( i=0 ; i<=19 ; i++) {
        t[i].afficheProduit();

}
```

**Remarque**: Cette approche fonctionne, mais elle est loin d'être idéale car pour chaque instance de Produit il y a création d'une nouvelle fonction afficheProduit. Si vous souhaitez faire de **l'AJAX** et que vous prévoyez de créer un grand nombre de ces objets, c'est une solution à éviter pour ne pas avoir de pertes mémoire. La meilleure solution consiste à utiliser la propriété **prototype** de tous les objets javaScript (il n'existe pas d'équivalent dans les langages de programmation purement orientée objet).

# III. PERSONNALISATION DES CLASSES EXISTANTES.

## III.1 Ajout de méthodes a des classes prédéfinies par javaScript.

S'il est possible de créer de nouvelles classes, il est possible d'étendre les fonctions des classes existantes et des classes prédéfinies par javaScript et ce de manière dynamique.

**Exemple:** on peut ajouter ou modifier des méthodes à la classe **String** . Il faudra pour cela utiliser sa propriété **prototype** (commune à toute les classes et héritée d'**Object** ). Toutes les modifications de la classe seront alors accessibles par les futurs objets et par les objets déjà créés. **Remarque** les modifications de l'attribut prototype de la classe **Object** se répercutent sur toutes les classes

```
x = new String("bonjour")
// ajout de la méthode long() identique à la méthode length
String.prototype.long = function(){return this.length}
alert(x.long())   // affiche 6

// affichage du contenu du prototype de la classe String
for (i in String.prototype) {alert(i+">>>"+String.prototype[i])}

// ou avec la méthode each vue précédemment
   String.prototype.each() ;
```

**Exercices:** Ajouter la méthode droite(n) à l'objet String, qui extrait les n caractères de droite d'une chaîne de caractères.

```
function droite(n){
   return this.substring(this.length-n)
}
String.prototype.droite=droite;      //     droite(n) est disponible
```

## III.2 Modification d'une méthode d'une classe prédéfinie

La méthode substring(x,y) extrait les caractères de la position x à la position y-1. On désire modifier la méthode pour que l'extraction se fasse de x à y.

```
function extract(d,f){
        var r="";

        for (var i=d;i<=f ; i++){
                r+=this.charAt(i)
        }
        return r;
}
String.prototype.substring=extract;   // la méthode substring a été modifiée
```

## III.3 Ajout de méthodes aux classes définies par le programmeur

Pour éviter les fuites mémoire, il est conseillé aux programmeurs de définir les méthodes de ses nouvelles classes par l'utilisation de la propriété **prototype**.
Exemple:

```
function Produit (reference,designation,prix,tva,photo){
        this.reference=reference;
        this.designation=designation;
        this.prix=prix;
        this.tva=tva;
        this.photo=photo;
}
Produit.prototype.afficheProduit=function(){
                    var r="";
                    r +="<b>Reference: </b>"+this.reference+"<br/>";
                    r +="<b>Designation: </b>"+this.designation+"<br/>";
                    r +="<b>Prix: </b>"+this.prix+"<br/>";
                    r +="<b>Tva: </b>"+this.tva+"<br/>";
                    r +="<img src= "+this.photo+">";
                    document.write(r);
    }
}
```

Beaucoup de bibliothèques ou frameWork utilisent la notation **JSON** pour définir l'attribut **prototype** ( *Attention*: cette écriture plus concise ne permet pas d'étendre une classe car le prototype courant est écrasé ):

```
function Produit (reference,designation,prix,photo){
        var z =  0                          // propriété privée          avec var
        this.reference=reference;           // propriétés publiques      avec this
        this.designation=designation;
        this.prix=prix;
        this.photo=photo;

}

Produit.prototype={
        // Dans le prototype les méthodes et les attributs sont toujours publiques
        afficheProduit  : function() {……},
        getDesignation : function() {.....…},          // exemple d'accesseur
        setDesignation : function(x)  {.....} ,        // exemple de mutateur
        TVA            :  20                           // initialisation d'un attribut public
}

// L'écriture précédente  n'est pas possible pour redéfinir les classes prédéfinies ( String etc.)
// Pour étendre une  classe prédéfinie, il faudra utiliser la notation suivante:

                    String.prototype.nomFonction= function() {……}
```

## III.4 Réutilisation d'une classe définie par le programmeur

Il est possible de stocker une classe dans un fichier externe (.js) pour s'en servir à de multiples reprises.

**Exemple**: soit le fichier "Biblio.js", contenant un constructeur pour la classe Personne. Cette classe est caractérisée par trois propriétés (nom, prénom, date de naissance) et par une méthode presenter() qui permet un affichage des propriétés d'une personne

```
function Personne(n, p, d) {
   this.nom =n;
   this.prenom =p;
   this.date =d;
}
Personne.prototype={
    presenter : function(){   alert(this.nom+"   "+this.prenom+"    "+this.date)}
}
```

Si dans un programme on souhaite gérer des étudiants pour lesquels on désire connaître le nom, le prénom, la date de naissance et le sexe on peut utiliser la classe **Personne** en lui ajoutant une propriété et une nouvelle méthode d'affichage ( la spécialisation peut-être faite de manière dynamique , sans avoir recours à un héritage ).

Exemple:

```
<html>
<head>
<script type="text/JavaScript" src="biblio.js"></script>
<script type="text/JavaScript"
          <!--
          // Ajout de la méthode getSexe à la classe Personne
          Personne.prototype.getSexe = function() { return(this.sexe);}

          // Ajout de la méthode setSexe à la classe Personne
          Personne.prototype.setSexe = function(x) { this.sexe=x ;}

          // Ajout de la méthode presenter2 à la classe Personne
          Personne.prototype.presenter2= function(){
                                          alert(this.nom+" "+this.prenom +" "+this.sexe)
                                        }
          // -->
</script>
</head>
<body>
<script language="JavaScript">
<!—
// on ne peut pas donner une valeur directement à la nouvelle propriété,  si tel est le souhait il faudra
// mettre  en place un héritage

   r=new Personne("BENOIST","Paul","27/08/78")

// cette écriture est obligatoire pour donner des valeurs aux nouvelles propriétés
   r.sexe="Masculin"
```

```
   alert(r.getSexe())    // affichage de la propriété sexe
   r.presenter()         // affichage de l'identité sans le sexe
   r.presenter2()        //  affichage de l'identité avec le sexe

// si nous appelons la nouvelle méthode d'affichage presenter() au lieu de présenter2()
// la première méthode n'est plus accessible. ( polymorphisme impossible )
// -->
</script>
</body>
</html>
```

## III.5 Définir une classe à partir d'une autre classe.

Il est possible qu'une propriété d'un objet soit une instance d'une autre classe ( cette solution permet de traduire une association ou une agrégation au sens UML ).

**Exemple**: soit les classes Voiture et Personne.

```
function Personne(n, p, d) {
   this.nom=n;
   this.prenom=p;
   this.date=d;
}
Personne.prototype={
       presenterPersonne : function () {alert(this.nom+"   "+this.prenom+"    "+this.date)}
}

function Voiture(ma, mo, a ) {
   this.marque=ma;
   this.modele=mo;
   this.annee=a;

   // le proprietaire est un "objet" de type Personne
   this.proprietaire = new Personne();   // instance de Personne
}

Voiture.prototype={
       presenterVoiture : function() {alert(this.marque+" "+this.modele+"   "+this.annee+" "
                                  +this.proprietaire.nom+"  "+this.proprietaire.prenom) ;}
}

vr= new Voiture("Ford","Fiesta",1993);
vr.proprietaire.nom="Grant";
vr.proprietaire.prenom="Marckus";
vr.proprietaire.date="12/08/1989";

vr.proprietaire.presenterPersonne();    // affiche l'identité du propriétaire
vr.presenterVoiture()    ;              // affichage des caractéristiques de la voiture avec
                                        // l'identité du propriétaire

alert(vr.proprietaire.nom) ;              // nom du propriétaire de la voiture
```

## III.6 Utiliser un destructeur d'objet

En JavaScript, il existe un opérateur **delete** qui permet de supprimer des propriétés, des variables, des éléments de tableau, des objets  etc..

.

Exemple:

```
r=new Personne("Dupont","Robert","27/02/68")

alert( r. nom)    //      affiche Dupont

delete r.nom ;
delete r.prenom ;
delete r.date ;

alert( r. nom)    //      affiche Undefined
```

# IV. COMPLEMENT SUR LA POO AVEC ES6

Le terme de **class** en JavaScript a été introduit avec ECMAScript 6. Cette syntaxe n'introduit pas un nouveau modèle d'objet dans JavaScript ! Elle fournit uniquement une écriture plus simple et plus claire pour créer des objets et manipuler l'héritage.

## IV.1 Définir des classes

En réalité, les classes sont juste des fonctions spéciales. Ainsi, les classes sont définies de la même façon que les fonctions : par déclaration, ou par expression.

### *IV.1.1 Les déclarations de classes*

Pour utiliser une déclaration de classe simple, on utilisera le mot-clé **class**, suivi par le nom de la classe que l'on déclare (ici « Personne »).

```
class Personne {
  constructor(nom, prenom,dateN) {
    this.nom = nom;
    this.prenom = prenom;
    this.dateN = dateN ;
  }
}
```

## IV.2 Corps d'une classe et définition des méthodes

Le corps d'une classe est la partie contenue entre les accolades. C'est dans cette partie que l'on définit les propriétés d'une classe comme ses méthodes et son constructeur.

### *IV.2.1 Mode strict*

Le corps des classes, pour les expressions et pour les déclarations de classes, est exécuté en mode strict.

### *IV.2.2 Constructeur*

La méthode constructor est une méthode spéciale qui permet de créer et d'initialiser les objets de la classe. Il ne peut y avoir qu'une seule méthode avec le nom "constructor" pour une classe donnée. Si la classe contient plusieurs occurences d'une méthode constructor, cela provoquera une exception SyntaxError.
Le constructeur ainsi déclaré peut utiliser le mot-clé super afin d'appeler le constructeur de la classe parente.

### *IV.2.3 Déclaration de Méthodes*

```
class Personne {
  constructor(nom, prenom,dateN) {
    this.nom = nom;
    this.prenom = prenom;
    this.date= dateN;
  }

  affiche() {
    alert(this.nom+"  "+this.prenom+"   "+ this.date);
  }

}

const n1 = new Personne("Dupont","Robert","27/02/68");

n1.affiche();
```

### *IV.2.4 Méthodes et propriétés statiques*

Le mot-clé static permet de définir une méthode statique pour une classe. Les méthodes statiques sont appelées par rapport à la classe entière et non par rapport à une instance donnée (ces méthodes ne peuvent pas être appelées « depuis » une instance). Ces méthodes sont généralement utilisées sous formes d'utilitaires au sein d'applications.

```
class Personne {
  constructor(nom, prenom,dateN) {
    this.nom = nom;
    this.prenom = prenom;
    this.date= dateN;
    if (typeof Personne.compt =="undefined") Personne.compt = 1
                else Personne.compt++;
  }

  static compteur(){
    alert( Personne.compt );
  }

  affiche() {
    alert(this.nom+"  "+this.prenom+"   "+ this.date);
  }

}

const n1 = new Personne("Dupont","Robert","27/02/68");
const n2 = new Personne("Dupont","Jean","27/02/68");
const n3 = new Personne("Dupont","Jeanne","27/02/68");


Personne.compteur()   //    affiche 3 car trois instances de Personne ont été créées
```

**Si votre navigateur n'est pas compatible ES6.**

Vous pouvez créer une page web html vide et à y coller le contenu du code ci-dessous. Votre navigateur n'est pas compatible ES6 ? Ce n'est pas un problème. Effectivement nous allons utiliser un script qui va nous transformer à la volée le code ES6 en code compatible avec votre navigateur.

```
<!doctype html>
<html lang="fr">
<head>
   <meta charset="utf-8" />
   <title>ECMAScript 6 - Demos</title>
</head>
<body>
   <!-- Le compilateur « Traceur » de Google -->
   <script src="https://google.github.io/traceur-compiler/bin/traceur.js"></script>
   <!-- Optionnel : Utilisé pour les tests et traces -->
   <script src="https://google.github.io/traceur-compiler/bin/BrowserSystem.js"></script>
   <!-- Optionnel : Cadriciel pour les tests -->
   <script src="https://google.github.io/traceur-compiler/src/bootstrap.js"></script>

   <script type="module">
      //  Mettre votre code ES6 ici
   </script>
</body>
</html>
```

Cette opération est réalisée par le compilateur Traceur, un outil mis au point par Google dont le but est de convertir le JavaScript de demain en JavaScript d'aujourd'hui. Il est disponible sous plusieurs formes (Serveur , client ). Ici nous utilisons la version web. Ainsi, tout le code JavaScript placé dans la balise script type="module" sera automatiquement converti à la volée.

# V. EXERCICES

**EXERCICE 1:**

Que signifient ces écritures :

a) miw=_=$=function(){return document.getElementById(arguments[0])}

b)
```
(function(){                    //  nous appellerons ceci une fei ou ( ief en anglais )
    var reg1=/-/g;
    var reg2=/^[a-z]{3}$/g
    test1 =  function(){ …. utilise reg1 }
    test2 =  function(){ …. utilize reg2 }
 })() ;
```

c)
```
$={
    merge  : function(t1,t2){…… },
    grep    : function(t,f){….},
    each    : function(t,f){….},
    trim     : function( c ){…}
   }
```

**EXERCICE 2:**

Etendre la classe **String** avec les méthodes suivantes:

- left(n) qui extrait les n caractères à gauche d'une chaîne.
- right(n) qui extrait les n caractères à droite d'une chaîne.
- capitalize() qui met la première lettre d'une chaîne en majuscule et les autres en minuscule.
- convertCss() qui remplace dans une chaine toutes les occurrences "-lettre minuscule" par "lettre majuscule".
  Exemples: "background-color" >> "backgroundColor" ; "border-color" >> "borderColor"
- trim() pour éliminer les espaces en début et fin de chaine

Etendre la classe **Array** avec la méthode suivante :

- merge(t) qui permet de concaténer deux tableaux
  Exemple : t1=["Thomas","Eric"] t2=["Léo"]

  t = t1.merge(t2) t et t1 contiendront ["Thomas","Eric","Léo"] et t2 condiendra ["Léo"]

Etendre la classe **Number** avec la méthode suivante :

- p(n) qui calcule un nombre à la puissance n.

  Exemple: var x=5;
  alert(x.p(3)) ; // affichera 125

Etendre la classe **Node** avec les méthodes suivantes:

- changeId(val) qui permet d'initialiser l'attribut id du nœud concerné avec la valeur val, cette méthode retournera le nœud modifié

  Exemple : $("val1").changeId("val2").changeId("val3")

  Cette écriture se positionne sur le nœud ayant l'attribut id = "val1" puis change sa valeur en val2 puis en val3.

- css(obj) qui permet d'ajouter au nœud concerné les propriétés CSS contenues dans l'objet obj.

  Exemple: $("val1").css({"height":"60px","color":"green"})
  Remarque: noeud.style.height="60px" ⇔ nœud.style["height"]="60px"

Etendre les classes **String, Array, Number, Node** avec la méthode suivante :

- extend(obj) qui permet d'étendre les slots de l'objet concerné avec les slots de l'objet obj

Exemples:
```
String.prototype.extend({
      left : function(n)  {…},
      right : function(n) {…},
      convertCss() :function(){},
      capitalize : function() {….},
      trim : function(){….}
}) ;

Array.prototype.extend({
      merge: function(t)  {…},
}) ;

Number.prototype.extend({
      p : function(n)  {…},
}) ;

Node.prototype.extend({
            changeId : function(val){
            },
            css : function(obj) {
            },
            attrib :function(obj){
            }
})
```

Toutes ces méthodes seront ajoutées à notre première version de la bibliothèque **miw**

**EXERCICE 3:**

Faire afficher deux listes déroulantes, une image représentant le verso d'une carte et un bouton .La première liste contient des chiffres de 1 jusqu'à 10 puis les valeurs: Valet, Dame Roi. La deuxième liste contient quatre valeurs: cœur, carreau, pique et trèfle. En cliquant sur le bouton on fait afficher la carte correspondante à la place de la carte retournée.

Pour cette réalisation, il faudra créer une Classe Carte contenant 3 propriétés (numéro, couleur, image) et la méthode afficheCarte(n) qui remplace la nième image de la page.