IFT1015 Programmation 1

tableaux associatifs et programmation orientée objets

Pascal Vincent et Aaron Courville

Université de Montréal

Première partie

Tableaux associatifs

Tableau v.s. tableau associatif

- **Tableau:** liste de *n* éléments (*valeurs*) indexés par leur ***position*** (entier de *0* à *n-1*) aussi appelé ***index***
  Associe ***position -> valeur (élément du tableau)***

```
          position:    0       1       2       3
var t1 = [ 3.5, -3*5, 1>2, "allo"];
var t2 = Array(1000); // initialisé à undefined
t2[0] = t1[3];
t1[1] = t2;
print(t1.length);
```

- **Tableau associatif:** éléments (*valeurs*) indexés par une ***clé***, souvent non numérique.
  Associe ***clé -> valeur***

```
                        clé: valeur                    clé: valeur
var ta = { marque: "honda", annee: 2000+3 };
```

Peut être vu comme un ensemble de paires *(clé,valeur)*

Syntaxe des tableaux associatifs

- Déclaration:

  ```
  var ta = { marque: "honda", annee: 2000+3 };
  ```

  (dans la syntaxe {}, les clés ne sont *pas* évaluées mais converties en string)

- Accès aux éléments:

  `ta["marque"]` ou `ta.marque`

- Permet l'ajout dynamique

  ```
  ta["modele"] = "civic";
  ```

  ou bien `ta.modele = "civic";`

- En JavaScript les **clés** sont toujours d'abord converties en **string**:

  ```
  ta[-3.50] = "bonjour";
  print( ta["-3.5"] );  // affiche bonjour
  print( ta[-3.50] );  // affiche bonjour
  print( ta["-3.50"] );  // n'affiche rien!
  ```

  Pourquoi? Car `-3.50` a été convertie en clé string `"-3.5"`

Terminologie

- **Tableau associatif** = **«map»** = **«dictionnaire»** ***clé -> valeur***
  (souvent implémenté par une *table de hachage* ou un *arbre de recherche*)

- Sont (quasi-)synonymes, en JavaScript seulement:
  tableau associatif = **enregistrement** = **structure** = **objet**
  (*clé* d'un tableau associatif = *attribut* ou *propriété* ou *champ* d'un objet)

- Beaucoup d'autres langages (ex: Java) font une distinction nette
  - **enregistrement** ou **structure** ou **objet**
    ont une liste de **propriétés (attributs) fixés à l'avance**
    syntaxe d'accès: *obj.propriete*
  - **tableaux associatifs** (**map**)
    permettent ajout/supression de *clés->valeur* **dynamiquement**
    syntaxe d'accès (très) différente.

Deux syntaxe d'accès aux éléments

- En JavaScript, *clé* d'un tableau associatif =

  attribut ou *propriété* ou *champ* d'un objet

  Possibilité d'accéder à un élément avec

  syntaxe tableau associatif ou syntaxe objet:

  `ta["nom_de_cle"]` équivalent à **`ta.nom_de_cle`**

- Attention: **`ta[`***string***`]`** mais **`ta.`***identificateur*

  Donc **`var s ="age"; print(ta[s]);`**
  équivaut à **`print(ta.age);`** pas à **`print(ta.s);`**
  et **`ta.s`** équivaut à **`ta["s"]`** pas à **`ta[s]`**

- Seule la syntaxe d'accès avec [] permet d'accéder à des clés qui ne correspondent pas à un identificateur valide:
  `ta["grand mère"]` ou d'un autre type **`ta[-3.1], ta[false]`**

Tableaux, ex de codes équivalents:

```
var t1 = [ 3.5, -3*5, "allo"];
print(t1.length);
```

```
var t1 = Array(3);
t1[0] = 3.5;
t1[1] = -3*5;
t1[2] = "allo";
print(t1.length);
```

```
var t1 = [];
t1[2] = "allo";
t1[0] = 3.5;
t1[1] = -3*5;
print(t1.length);
```

D'autres langages donneraient une erreur d'accès hors du tableau. JavaScript l'aggrandit automatiquement!

Tableaux associatifs/objets, codes équivalents:

```
var ta = { marque: "honda", annee: 2000+3, 0.50: "un demi"};
```

```
var ta = { "marque": "honda", "annee": 2000+3, "0.5": "un demi"};
```

```
var ta ={};
ta["marque"] = "honda";
ta["annee"] = 2000+3;
ta[0.500] = "un demi";
```

```
var ta ={};
ta.marque = "honda";
ta.annee = 2000+3;
ta["0.5"] = "un demi";
```

Parcourir les éléments d'un tableau ordinaire

Boucle for classique (syntaxe C; while déguisé)

```
var tab = [ 3.5, -3*5, "allo"];
for (var i = 0; i<tab.length; i++)
{
   var valeur = tab[i];
   print( i + " -> " + valeur );
}
```

Parcourir les clés d'un tableau associatif

Boucle for ... in (parfois appelée foreach)

parcourt toutes les ***clés*** d'un tableau associatif

```
var ta = { "marque": "honda",
           "annee": 2000+3,
           "0.5": "un demi"};

for (var cle in ta)
{
   print( cle + " -> " + ta[cle] );
}
```

Attention: les tableaux associatifs n'ont pas de .length

Attention, fortement déconseillé: n'utilisez pas de for ... in pour parcourir des tableaux ordinaires en JavaScript.

(pourquoi? un raison -- l'order d'itération n'est pas garantie)

Vérifier si une clé est présente dans un tableau associatif

Utiliser l'opérateur **in**

Syntaxe: ***clé*** in tableau associatif (où clé est de type *string*)
vaudra *true* si cette clé est présente dans le tableau

```
var ta = { "marque": "honda",
           "annee": 2000+3,
           "0.5": "un demi"};

print ( "marque" in ta);    // affiche true
print ( "honda" in ta);    // affiche false

if ( "annee" in ta )
  ta.annee = ta.annee+1;

print ( ta["annee"]);  // affchera 2004
```

Deuxième partie

Programmation **Orientée Objet** en JavaScript

Paradigme de programmation orienté objet

- Programmation impérative (ou procédurale): programme vu comme un ensemble de fonctions qui s'appellent mutuellement

- Programmation orientée objet: programme vu comme un ensemble d'objets qui communiquent (s'envoient des messages)

- Un objet peut communiquer avec un autre en appelant une méthode (fonction) associée à l'autre objet.

Objets

- Objet composite = structure = enregistrement
- (en JavaScript) = tableau associatif
- Un objet possède des propriétés nommées (aussi appelés attributs ou champs)
- La valeur d'une propriété (attribut) d'un objet peut être de n'importe quel type.
- L'ensemble des valeurs des attributs d'un objet constitue son état (qui peut changer)

```
var veh = { marque: "honda",age: 5 };
var coord = { x: 3, y: 5, z: 2*10 };
var naiss = { annee: 1995, mois: 12, quant: 9 };

print( veh.age ); // affiche 5
veh.age = veh.age+1;
print( veh.age ); // affiche 6
```

Objets

Exemple:

```
var mon_perroquet = { nom: "Coco",
                      age: 3,
                      altitude: 0 };
```

On aurait aussi pu écrire:

```
var mon_perroquet = {};

mon_perroquet.nom = "Coco";
mon_perroquet.age = 3;
mon_perroquet.altitude = 0;
```

Ou bien:

```
var mon_perroquet = new Object();

mon_perroquet.nom = "Coco";
mon_perroquet.age = 3;
mon_perroquet.altitude = 0;
```

Méthodes des objets

- En programmation objet, les objets regroupent des attributs/champs/propriétés, et des fonctions (appelées méthodes) pour manipuler ce genre d'objet.
- Les méthodes accèdent ou modifient souvent les attributs de l'objet.

Définissons un «objet»:

```
var mon_perroquet = { nom: "Coco",
                      age: 3,
                      altitude: 0 };
```

Approche procédurale classique: fonctions globales

```
var repete = function(perroquet, phrase) {
  print( perroquet.nom + " dit " + phrase );   };

var envoleToi = function(perroquet, altitude){
  perroquet.altitude = altitude;  };

repete(mon_perroquet, "Bonjour les pirates!");
envoleToi(mon_perroquet, 20);
```

Déplaçons une fonction ***dans*** l'objet...

```
mon_perroquet.envoleToi = function(perroquet, altitude){
  perroquet.altitude = altitude;  };

// la syntaxe d'appel est différente, et logique...
mon_perroquet.envoleToi(mon_perroquet, 40); // on semble se répéter!
```

Syntaxe d'appel des méthodes

- Les fonctions classiques s'invoquent avec la syntaxe:

  nom_de_fonction(paramètres)

- Les méthodes d'un objet s'invoquent avec la syntaxe:

  objet.nom_de_fonction(paramètres)

- On dit qu'on appelle la méthode **«sur»** l'objet.

Appel de méthode et this

- **objet.**nom_de_fonction(**paramètres**)
- L'objet sur lequel on appelle une méthode (à gauche du **.**) est semblable à un *paramètre implicite* additionnel (en plus des paramètres entre parenthèses).
- **Dans** la **définition** de la méthode on peut y accéder par l'identifiant ***this***.
- ***this*** réfère à l'objet sur lequel la méthode a été apelée.

```
mon_perroquet.envoleToi = function(altitude){
  this.altitude = altitude;  };

mon_perroquet.envoleToi(60);     // on ne se répète plus!
```

Et si on voulait créer plusieurs objets du même type ? (ex: Perroquet)

- Si on voulait créer 10 ou 1000 perroquets différents...
- Avec l'approche précédente, il faudrait pour chacun **répéter:**
  - La liste des propriétés
  - L'ajout des méthodes
- Meilleure approche en JS (pour éviter de répéter 10 fois): ***fonction Constructeur*** et ***prototype***

Constructeur

- On peut définir une fonction spéciale (un *constructeur*) pour construire et initialiser des objets d'un certain type.

```
var Perroquet = function(nom, age_en_annees) {
  this.nom = nom;
  this.age = age_en_annees;
  this.altitude = 0; };
```

- On peut ensuite facilement créer plusieurs objets de ce type, à l'aide du mot-clé **new**. Syntaxe: new Constructeur(paramètres)

```
// Créer des perroquets:

var mon_perroquet = new Perroquet("Coco",3);
var jaco = new Perroquet("Jaco",0);
```

Constructeurs et classes

- L'orienté objet de nombreux langages (Java, C++, Python...) est basé sur des «Classes».
- Pas JavaScript: c'est un langage objet basé sur des «prototypes». Il n'y a pas de vraies «classes».
- Mais on peut voir un constructeur comme correspondant à une «Classe»: il permet de créer (construire) autant d'objets que l'on veut du même genre (~classe) ex: plusieurs Perroquets.
- Convention recommandée: nommez vos fonctions constructeurs avec une Majuscule initiale.

Ajouter des méthodes

- L'exemple de constructeur qu'on a vue initialise les **propriétés** de tout objet qu'il construit.

- On pourrait aussi ajouter les **méthodes** à l'objet depuis le code du constructeur avec ex: this.nom_de_methode = function(...) { ... };

- Mais il y a une meilleure manière...

Ajouter des méthodes

- **Manière recommandée:** ajouter les méthodes au *prototype* de la fonction constructeur:

Soit le constructeur

```
var Perroquet = function(nom, age_en_annees) {
  this.nom = nom;
  this.age = age_en_annees;
  this.altitude = 0; };
```

On peut rajouter des méthodes à son «prototype»

```
Perroquet.prototype.envoleToi = function(altitude){
  this.altitude = altitude;  };
```

Les objets céés avec **`new NomDuConstructeur(...)`** reçoivent en partage toutes les méthodes (et propriétés) présentes dans **`NomDuConstructeur.prototype`**.

Ajouter des méthodes

C'est quoi la différence entre ceci:

(A.)

```
var Perroquet = function(nom, age_en_annees) {
  this.nom = nom;
  this.age = age_en_annees;
  this.altitude = 0; };

Perroquet.prototype.envoleToi = function(altitude){
  this.altitude = altitude;
};
```

Et ceci:

(B.)

```
var Perroquet = function(nom, age_en_annees) {
  this.nom = nom;
  this.age = age_en_annees;
  this.altitude = 0;
  this.envoleToi = function(altitude) {
    this.altitude = altitude;
  };
};
```

Ajouter des méthodes

(A.)

```
var Perroquet = function(nom, age_en_annees) {
   this.nom = nom;
   this.age = age_en_annees;
   this.altitude = 0; };

Perroquet.prototype.envoleToi = function(altitude){
   this.altitude = altitude;
};
```

- Avantage d'utiliser `fooObjet.prototype.méthode`
  - Utilise moins de mémoire: la méthode est instancié une seule fois pour tout objet de type `fooObjet`. Le contexte de chaque objet unique est transmis à la méthode comme paramètres implicites.

Ajouter des méthodes

(B.)

```
var Perroquet = function(nom, age_en_annees) {
  this.nom = nom;
  this.age = age_en_annees;
  this.altitude = 0;
  this.envoleToi = function(altitude) {
    this.altitude = altitude;
  };
};
```

- Avantage d'utiliser `this.méthode`
  - Cela crée une fonction privilégiée qui a accès à des *variables et fonctions privées* définies dans le constructeur (ex. Perroquet).

Exemple: comparé ceci:

(A.)

```
var Perroquet = function(nom, age_en_annees) {
  var niveauTerre = 10; // variable privée
  this.nom = nom;
  this.age = age_en_annees;
  this.altitude = 0; };

Perroquet.prototype.envoleToi = function(altitude){
  this.altitude = altitude+niveauTerre;
};
```

Avec ceci:

(B.)

```
var Perroquet = function(nom, age_en_annees) {
  var niveauTerre = 10; // variable privée
  this.nom = nom;
  this.age = age_en_annees;
  this.altitude = 0;
  this.envoleToi = function(altitude) {
    this.altitude = altitude+niveauTerre;
  };
};
```

(A.) donne une erreur, mais (B.) fonctionne.

Résumé: exemple complet

```
// Constructeur
var Perroquet = function(nom, age_en_annees) {
  this.nom = nom;
  this.age = age_en_annees;
  this.altitude = 0; };

// Méthodes (ajoutees au prototype du constructeur)
Perroquet.prototype.envoleToi = function(altitude){
  this.altitude = altitude;  };

Perroquet.prototype.repete = function(phrase) {
  print( this.nom + " dit " + phrase );   };

// Ex d'utilisation: on crée deux objets de type Perroquet
var mon_perroquet = new Perroquet("Coco",3);
var jaco = new Perroquet("Jaco",0);
// On appelle des méthodes sur ces objets
mon_perroquet.envoleToi(20);
jaco.repete("Coco s'envole!");
```

Précision: syntaxe de définition de fonction

- Deux syntaxes équivalentes pour définir des fonctions:
  `var nomDeFonction = function(paramètres) { ... };`
  Ou
  `function nomDeFonction(paramètres) { ... }`
- La 2ème est plus proche de la syntaxe Java/C/...
- Mais pour ajouter *directement* une méthode à un prototype on peut seulement utiliser la première:
  `NomConstructeur.prototype.nomDeMethode = function(paramètres) { ... };`

Précision: paramètres d'une fonction

- En JavaScript vous pouvez appeler une fonction avec un nombre d'arguments (paramètres) différent de ceux que vous avez déclaré
- La liste des arguments est accessible par la variable ***arguments*** (un tableau ordinaire)

```
var f = function(x, y) { print(arguments); };

f(3, 5); // affiche 3, 5
f(3); // affiche 3
f(3, "Bonjour", 5, true, -0.25);
 // affiche 3,Bonjour,5,true,-0.25
```

- Cela permet d'écrire des fonctions qui peuvent gérer un nombre variable d'arguments.

Méthodes prédéfinies utiles

- `valueOf`: JS appelle la méthode `valueOf()` pour convertir un objet à une valeur primitive. JS invoque automatiquement lorsqu'il rencontre un objet où une valeur primitive est attendue.

```
// Exemple:
var Perroquet = function(nom, age_en_annees) {
  this.nom = nom;
  this.age = age_en_annees;
  this.altitude = 0; };

var jaco = new Perroquet("Jaco",3);
print(jaco.valueOf()); // affiche: [object Object]
```

- Mais ça peut être surcharger:

```
Perroquet.prototype.valueOf = function() {
  return this.altitude;
};
```

Méthodes prédéfinies utiles (cont.)

- `toString`: Chaque objet a une méthode `toString()` qui est appelée automatiquement lorsque l'objet devrait représentée comme une valeur de texte ou quand un objet est appelé d'une manière dans laquelle une chaîne est attendue (ex. print(objet)).

```
// Exemple:
var Perroquet = function(nom, age_en_annees) {
  this.nom = nom;
  this.age = age_en_annees;
  this.altitude = 0; };

var jaco = new Perroquet("Jaco",3);
print(jaco); // affiche: [object Object]
```

- Mais ça aussi peut être surcharger:

```
Perroquet.prototype.toString = function() {
  return this.nom;
};
```

Héritage, polymorphisme

- Des fonctionnalités orienté objet plus avancées (telles que l'*héritage*) sont aussi possibles en JavaScript mais dépassent le cadre de ce cours.

  (Si ça vous intéressés, voir ex.

  https://developer.mozilla.org/en-US/docs/JavaScript/Introduction_to_Object-Oriented_JavaScript)

- Vous les verrez en Java dans le cours Programmation 2