Les objets prédéfinis

- Tour d'horizon de quelques objets prédéfinis :
  - le constructeur `Object` : ancêtre de tous
  - le constructeur `Function` : les fonctions
  - les constructeurs `Boolean`, `Number` et `String`
  - le constructeur `Array` : les tableaux
  - le constructeur `Date` : les dates
  - le constructeur `RegExp` : les expressions régulières
  - l'objet `Math` : compilation de fonctions utilitaires

Les objets prédéfinis : Object

- Quelques méthodes de `Object.prototype`
  - `obj.hasOwnProperty("ident")` : teste l'existence d'une propriété propre (non héritée)
    - Différent de `"ident" in obj` (propriété héritée ou non)
  - `obj.propertyIsEnumerable("ident")` : indique si la propriété est énumérable (citée dans un for-in)
  - `obj.toString()` : conversion en chaîne de caractères
    - par défaut : `[object type]` ; souvent : `[object Object]`
  - `obj.valueOf()` : conversion en valeur de base
    - par défaut : renvoie l'objet lui-même, écrit `[object Object]`

Les objets prédéfinis : Object

- Quelques méthodes proposées par `Object`
  - `Object.create(o)` : pour créer un héritier
  - `Object.getOwnPropertyNames(o)` : liste de toutes les propriétés propres (non héritées) à un objet
  - `Object.getPrototypeOf(o)` : renvoie le prototype
    - équivalent à : `o.__proto__`
  - `Object.setPrototypeOf(o, proto)` : modifie le prototype de o
    - équivalent à : `o.__proto__ = proto`
    - `get/setPrototype` sont plus propres que la manipulation de `__proto__` mais pas couverts par tous les navigateurs.

Les objets prédéfinis : Object

- Récupérer les propriétés d'un objet
  - Distinguer propriétés propres vs héritées
  - Distinguer propriétés énumérables vs non énum

- Détection (test d'existence)
  - `id in obj` : détecte n'importe quelle propriété
  - `obj.hasOwnProperty("id")` : propre
  - `obj.propertyIsEnumerable("id")` : propre et énum

- Récupération des noms des propriétés
  - `for id in obj …` : énumérables (propres ou non)
  - `Object.keys(obj)` : propres et énumérables
  - `Object.getOwnPropertyNames(obj)` : propres

Les objets prédéfinis : String

- Informations
  - `s.length` : longueur de la chaîne
- Transformations (renvoient une copie de s)
  - `s.toUpperCase()` et `s.toLowerCase()` : tout en majuscules / minuscules
  - `s.trimLeft()` et `s.trimRight()` : supprime les blancs en début/fin de chaîne
  - `s.trim()` : supprime les blancs en début et en fin
- Opération
  - `s.concat(s1, s2…)` : renvoie une nouvelle chaîne correspondant à s + s1 + s2 + …

Les objets prédéfinis : String

- Chaînes et caractères
  - `s.charAt(i)` : caractère situé à la position i
  - `s.charCodeAt(i)` : code (Unicode) du caractère situé à la position i
  - `String.fromCharCode(c1,c2…)` : chaîne constituée des caractères dont les codes (Unicode) sont donnés

- Extraction de sous-chaînes
  - `s.slice(i1)` : sous-chaîne commençant à la position i1 (jusqu'à la fin)
  - `s.slice(i1,i2)` : sous-chaîne de la position i1 à la position i2 – 1
  - `s.substr(i1,n)` : sous-chaîne de n caractères à partir de la position i1

  Note : un indice (i1 ou i2) négatif repère une position à partir de la fin de la chaîne (-1 = dernier caractère, -2 = avant-dernier, etc.)

Les objets prédéfinis : String

- Recherche d'une sous-chaîne (renvoie -1 si elle n'est pas présente)
  - `s.indexOf(s1)` : position de la chaîne s1 dans s
  - `s.indexOf(s1,i)` : idem en commençant la recherche à la position i1
  - `s.lastIndexOf(s1)` : idem en commençant la recherche à partir de la fin de la chaîne s

- Conversion en tableau
  - `s.split(sep)` : renvoie un tableau constitué des éléments obtenus en découpant s selon le séparateur sep

- Voir aussi les fonctions relatives aux expressions régulières !

Les objets prédéfinis : Array

- Créer un tableau (pas forcément homogène !)
  - Syntaxe "orienté objet"

```
var jour = new Array("lun", "mar", "mer", "jeu", "ven");
```

(le "new" n'est pas obligatoire ici.)

  - Syntaxe littérale

```
var jour = ["lun", "mar", "mer", "jeu", "ven"];
var tab = [1, "2", true, function () { return 42; }];
var tab2D = [[1, 2], [3, 4, 5]];
```

- Un tableau est un objet, et donc un tableau associatif !
  - jour :
    0 → "lun", 1 → "mar", 2 → "mer", 3 → "jeu", 4 → "ven", length → 5
  - tab :
    0 → 1, 1 → "2", 2 → true, 3 → function…, length → 4

Les objets prédéfinis : Array

- Créer un tableau vide

  `var tabVide = new Array ();` ou
  `var tabVide = [];`
  tabVide : *length* → *0*

- Créer un tableau sans valeurs initiales

  `var tab4 = new Array(4);`
  ou `var tab4 = [] ; tab4.length = 4;`
  tab4 : *length* → *4*
  tab4[0], tab4[1]… sont évalués à undefined (idem pour tab[72] !)

  - Attention : indiquer la taille sous la forme d'un nombre entier !
    `var tabUnÉlém = new Array("42");` revient à
    `var tabUnÉlém = ["42"];`

  - Pour un tableau à une case contenant un entier :
    `var tabUnEntier = [42];`
    `var tabUnEntier = new Array (42);` // tableau de 42 cases

Les objets prédéfinis : Array

- Accès aux éléments (lecture ou écriture)

```
var jour = new Array ("lun", "mar", "mer", "jeu", "ven");
jour[2]
for (var i = 0 ; i < 5 ; i++) console.log(jour[i]);

var matrice = [[1, 2, 3], [4, 5], [6, 7, 8, 9]];
matrice[1][2]
```

  - Il faut que le tableau soit déclaré avant de pouvoir y écrire une valeur : pas de `t[4] = 3` sans `t = []` ou autre déclaration préalable.

  - Accès hors-borne (indice >= length)
    - En lecture : renvoie undefined
    - En écriture : modifie la longueur !

Les objets prédéfinis : Array

- Taille d'un tableau (lecture ou écriture)
  - Pour obtenir la taille d'un tableau :
    `tab.length` ou `tab["length"]`

- Il s'agit d'une propriété modifiable (taille dynamique) !

```
var tab = new Array (1, 2, 3, 4, 5);
console.log(tab.length);      // 5
console.log(tab[3]);          // 4
tab[8] = 6;
console.log(tab.length);      // 9
tab.length = 3;
console.log(tab[3]);          // undefined
tab.length = 6;
console.log(tab[3]);          // undefined
```

  - Quand on diminue la longueur, on supprime les associations numériques inutiles (les valeurs sont perdues définitivement).

Les objets prédéfinis : Array

- Boucle sur un tableau
  - Première version

```
var tab = ["un", "deux", "trois"];
for (var i = 0 ; i < tab.length ; i++)
  console.log(tab[i]);
```

  - For-in : la propriété "length" est non énumérable !

```
var tab = ["un", "deux", "trois"];
for (x in tab) console.log(tab[x]);
```

    - Problème si on a ajouté d'autres propriétés !

  - "Mapping" (inspiré de la programmation fonctionnelle)

```
function affiche (x) { console.log(x); }
tab.forEach(affiche);
```

    - L'argument de la fonction correspond aux valeurs tab[i].

Les objets prédéfinis : Array

- Tableaux à trous (certaines cases sans valeur)
  - c'est-à-dire : pas d'association pour certains indices < length
  - Exemple

```
var tabTrou = new Array[6];
tabTrou[0] = 1; tabTrou[1] = 2; tabTrou[4] = 5;

var tabTrou = [1, 2, , , 5, ,];
```

(dernière virgule est ignorée)

tabTrou : *0 → 1, 1 → 2, 4 → 5, length → 6*

  - Différent de :

```
var tab = [1, 2, undefined, undefined, 5, undefined];
```

tab : *0 → 1, 1 → 2, 2 → undef, 3 → undef, 4 → 5, 5 → undef, length → 6*

  - Dans un for-in ou un forEach : les trous sont ignorés !

```
for (x in tabTrou) console.log(x);
tabTrou.forEach(function (v) { console.log(v); } );
```

Les objets prédéfinis : Array

- Opérations (renvoient un nouveau tableau)
  - `tab.concat(e1,e2…)` : ajoute les valeurs e1, e2… à la fin de tab
  - `tab.concat(t1, t2…)` : ajoute les éléments des tableaux t1, t2… à la fin de tab
    - *On peut aussi utiliser concat avec un mélange d'éléments et de tableaux.*

- Transformations (modifient directement tab)
  - `tab.reverse()` : inverse l'ordre des éléments dans tab
  - `tab.sort()` : trie les éléments de tab (par défaut, les éléments sont convertis en string puis triés selon l'ordre lexicographique)
  - `tab.sort(f)` : trie les éléments en se basant sur la fonction f
    - Interprétation : x < y si f(x,y) < 0
    - Pour un tri numérique : tab.sort(function (x,y) { return x-y; })

- Conversion (vers une chaîne de caractères)
  - `tab.join()` : éléments de tab (écrits selon toString) séparés par des virgules
  - `tab.join(sep)` : idem mais avec le séparateur indiqué

Les objets prédéfinis : Array

- Recherche d'un élément (renvoie -1 s'il n'est pas présent)
  - `tab.indexOf(e)` : position de l'élément e
  - `tab.indexOf(e, i)` : idem en commençant la recherche à la position i
  - `tab.lastIndexOf(e)` : idem en commançant la recherche à la fin

- Sous-tableaux (extraction)
  - `tab.slice(i1)` : sous-tableau commençant à la position i1 (jusqu'à la fin)
  - `tab.slice(i1,i2)` : sous-tableau allant de la position i1 à la position i2 – 1
  - Un indice négatif repère une position à partir de la fin (-1 = dernière case, -2 = avant-dernière, etc.)

- Sous-tableaux (ajout, suppression, remplacement)
  - `tab.splice(i)` : supprime les éléments de l'indice i à la fin
  - `tab.splice(i, nb)` : supprime nb éléments à partir de l'indice i
  - `tab.splice(i, nb, e1, …)` : idem et les remplace par elem1…
  - splice agit directement sur le tableau et renvoie les éléments supprimés sous la forme d'un tableau.

Les objets prédéfinis : Array

- Tableaux en tant que piles / files
  - `tab.shift()` : enlève le 1er élément de tab et le renvoie
  - `tab.unshift(e)` : ajoute e en tête de tab (et renvoie la nouvelle longueur)
  - `tab.pop()` : enlève le dernier élément de tab et le renvoie
  - `tab.push(e)` : ajoute e à la fin de tab (et envoie la nouvelle longueur)
  - push et unshift peuvent également ajouter plusieurs éléments à la fois

- Programmation fonctionnelle sur les tableaux
  - `tab.forEach(f)` : exécute f(x) pour chaque valeur x de tab
  - `tab.map(f)` : renvoie une copie de tab où chaque valeur x a été remplacée par f(x)
  - `tab.filter(f)` : renvoie un tableau composé des éléments de tab pour lesquels f(x) = true
  - `tab.every(f)` : renvoie true ssi f(x) = true pour tous les éléments de tab
  - `tab.some(f)` : renvoie true ssi f(x) = true pour au moins un élément de tab
  - La fonction f s'applique aux valeurs contenues dans le tableau.
  - À chaque appel f(x, i, tab), elle reçoit 3 arguments : la valeur, sa position dans le tableau et le tableau lui-même.

Les objets prédéfinis : Function

- Définition d'une fonction
  - Syntaxe classique

    ```
    function somme (x,y) { return x + y; }
    ```

    - Plus efficace car pré-compilée
  - Syntaxe "affectation" (lambda-expression)

    ```
    var somme = function (x) { return x + y; }
    ```

  - Syntaxe "orienté objet"

    ```
    var somme = new Function (x, y, "return x + y;");
    ```

    - Moins efficace car le code est évalué à chaque appel
  - `f.length` : nombre d'arguments attendus par la fonction
    - Attention : aucune vérification n'est effectuée !

Les objets prédéfinis : Function

- Deux méthodes héritées de Function.prototype
  - `f.call(argThis, arg1, arg2…)` : exécute la fonction f sur les arguments arg1, arg2… dans un contexte où this se réfère à argThis
  - `f.apply(argThis, [arg1, arg2…])` : idem mais les arguments sont donnés sous la forme d'un tableau

  - Exemple

```
var nombres = [17, 33, 15, 12, 5];
var max = Math.max.apply(null, nombres);
```

  - L'exécution d'une méthode `obj.meth(…)` revient en fait à `meth.call(obj, …)`.

Les objets prédéfinis : Function

- Utilisation du tableau arguments
  - Tableau accessible dans le corps d'une fonction et contenant l'ensemble des arguments
  - Permet de simuler la surcharge

  - Exemple

```
function afficheDouble () {
  for (var i = 0 ; i < arguments.length ; i++)
    console.log(arguments[i] * 2);
}
afficheDouble(1);
afficheDouble(1, 2, 3, 4, 5, 6);
```

Les objets prédéfinis : Function

- Autre exemple

```
function somme () {
  var total = 0;
  for (var i = 0 ; i < arguments.length ; i++)
    total += arguments[i];
  return total;
}

alert(somme(3,4,5));
alert(somme(10,2,4,6,8));
alert(somme());
```

- Exercice : concaténation de chaînes avec un séparateur donné.

Les objets prédéfinis : Math

- Constantes prédéfinies
  - `Math.E`, `Math.PI`, `Math.SQRT2`, `Math.LN2`, `Math.LN10`…
- Fonctions prédéfinies
  - Valeur absolue : `Math.abs`
  - Min/max : `Math.max(v1, v2…)`, `Math.min(v1, v2…)`
  - Arrondis : `Math.ceil`, `Math.floor`, `Math.round`
  - Racine carrée : `Math.sqrt`
  - Fonctions trigonométriques : `Math.cos`, `Math.asin`, `Math.tan`…
  - Exponentielle et logarithme : `Math.exp`, `Math.log`, `Math.pow(base,exp)`
  - Génération de nombres aléatoires : `Math.random()` donne une valeur dans]0,1[

Les objets prédéfinis : Date

- Création d'objets (mois : jan = 0, fév = 1, mar = 2, …, déc = 11)
  - `new Date ()` date/heure actuel
  - `new Date (n)` n millisecondes après le 1er janvier 1970 minuit
  - `new Date (s)` chaîne de caractères décrivant un moment
  - `new Date (an, mois, jour, h, min, sec, millisec)`

- Formatage de la date (en string) :
  - `d.toDateString()` format anglais "Day Mon dd yyyy"
  - `d.toISOString()` format ISO "yyyy-mm-dd**T**hh:mm:ss.sss**Z**"
  - `d.toLocaleDateString()` date au format local
  - `d.toLocaleTimeString()` heure au format local
  - `d.toLocaleString()` les deux ensemble

Les objets prédéfinis : Date

- Extraction d'information : `d.getXXX()`

  où XXX peut être FullYear (année), Month, Date (jour du mois), Hours, Minutes, Seconds ou Milliseconds

  - `d.getDay()` donne le jour de la semaine (0 = dim, 1 = lun…)
  - `d.getTime()` donne le nombre de millisecondes depuis le 1/1/1970

- Modification d'information : `d.setXXX(valeur)`

  où XXX peut être FullYear (année), Month (0-11), Date (1-31), Hours, Minutes, Seconds ou Milliseconds

  - On peut aussi fournir plusieurs paramètres en une fois : `d.setFullYear(an, mois, j)` ; `d.setHours(h, min, sec)`
  - `d.setTime(msec)` avec le nombre de millisecondes