1 Applications web en javascript

1.1 Plan

- Bases solides de javascript
  - types, expressions, instructions de base
  - objets
  - fonctions,
  - constructeurs, prototypage
- Manipulation du navigateur et de son contenu
  - rappels et points importants de HTML
  - objet `window`
  - Document Object Model
- Interactions avec l'utilisateur
  - modèle événementiel
  - contrôles et formulaires

2 Introduction

2.1 Page Web, application Web

- une page Web est composée de trois aspects :

  1. HTML : contenu sémantique structuré
  2. CSS : mise en page, aspect
  3. javascript : comportement (interactions avec l'utilisateur)

- javascript sert à :

  - améliorer une page Web : la page Web doit quand même être fonctionnelle sans javascript (amélioration progressive)
  - faire une application web : application/programme qui tourne dans un navigateur (et ne peut fonctionner sans javascript)

3 Introduction

- 1995 : Livescript, langage de script dans le navigateur Netscape pour faire de la validation côté client (navigateur) plutôt que serveur, renommé javascript
- implémentations ensuite incompatibles dans les différents navigateurs
- 1997 : standard ECMA-262 définissant ECMAScript, un « langage de script à utilité générale cross-plateforme et indépendant des vendeurs »
- depuis les navigateurs essaient d'utiliser ECMAScript comme base d'implémentation de javascript
- décembre 2009 : ECMAScript version 5, supportée complètement par Firefox >= 4, IE >= 9, Safari >=6 Chrome >= 23
- juin 2015 : ECMAScript version 6, en cours de support
- compilateur vers une version antérieure : babel (`babeljs.io`), traceur (`github.com/google/traceur-compiler`)

- javascript :
  - ECMAScript : langage de base, noyau de javascript
  - DOM : méthodes et interfaces pour manipuler le contenu d'une page web
  - BOM : méthodes et interfaces pour interagir avec le navigateur
- javascript est aussi de plus en plus utilisé côté serveur (écosystème Node.js)

- Javascript (EcmaScript) est un langage vaguement typé dynamique à objets par prototypage :
  - vaguement typé : quelques types de base, pas vraiment de type utilisateur, pas de vérification de type avant utilisation, les variables peuvent changer de type
  - dynamique :
    - tout est effectué à l'exécution : vérification de l'existence des données et fonctions utilisées, résolution de noms, ...
    - les propriétés et comportements des objets peuvent être changés en cours d'exécution
  - à objets : à part quelques types primitifs (convertibles en objets), tout est objet (y compris les fonctions p.ex.)
  - par prototypage : pas de classe, les objets héritent d'un objet (dit prototype)

3.1 Javascript dans le navigateur

- le code javascript est chargé par une page Web grâce aux éléments `<script>` :
  - soit écrit directement dans l'élément dans le fichier HTML
  - soit chargé depuis un fichier javascript externe (attribut `src`)
- les éléments `<script>` sont interprétés dans l'ordre dans lequel ils apparaissent
- le code dans un élément `<script>` doit être complètement interprété avant que la lecture de la page web continue

→ mettre ces éléments plutôt à la fin pour ne pas freiner le chargement de la page

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">  <!-- encodage -->
    <title><!-- titre page --></title>
  </head>
  <body>
    <!-- contenu de la page -->
    <!-- code dans la page web -->
    <script>
      function hello() { alert("Hello"); }
    </script>

    <!-- code dans un fichier externe -->
    <script src="file.js"></script>
  </body>
</html>
```

3.2 Développement en javascript

- pour développer des programmes en javascript :
  - IDE lourd (Netbeans, Eclipse)
  - éditeur spécialisé (Komodo edit `komodoide.com/komodo-edit/`, Brackets `brackets.io`)
  - un bon éditeur (Notepad++)
- pour tester : utiliser un debogger dans le navigateur
  - avec Firefox : utiliser Firebug (`getfirebug.com`)
  - ouvrir avec F12, accès au script, au DOM, à une console (écrire dedans avec `console.log(msg)`), à un interpréteur javascript
- écrire du javascript en mode strict : mettre `"use strict";` en début de script ou en début de fonction → signale plus d'erreurs
- outils de vérification du code : JSLint (`jslint.com`), JSHint (`jshint.com`)

4 Types et expressions

type		`typeof x`
primitif : immuable manipulé par valeur	nombre	`"number"`
	booléen	`"boolean"`
	chaîne	`"string"`
	`undefined`	`"undefined"`
	`null`	`"object"`
objet : manipulé par référence	ordinaire : ensemble de paires (nom,valeur)	`"object"`
	tableau : valeurs numérotées	`"object"`
	fonction : ordinaire	`"function"`
	constructeur	

On peut tester le type d'une variable `x` avec `typeof x`

4.1 Nombres

- réels au format IEEE 754, conversion automatique en objet de type `Number`
- « pas un nombre » : `NaN`, testable avec `isNaN(x)`
- opérations arithmétiques usuelles :
  - convertissent leur opérandes en nombre (`null` → 0, `undefined` → NaN)
  - en particulier : `+` unaire convertit en nombre
  - ! sauf `+` binaire si l'un des opérandes est une chaîne
- fonctions mathématiques usuelles : méthodes de l'objet `Math`
  `Math.sqrt(3.1)`, `Math.sin(3.14)`, ...
- conversion depuis chaîne de caractères : fonctions `parseInt(s)`, `parseFloat(s)`
- conversion en chaîne de caractères : méthodes `toString()`, `toFixed(`*`taille`*`)`, `toExponential(`*`taille`*`)`, `toPrecision(`*`taille`*`)`

4.2 Chaînes de caractères

- chaînes de caractères au format UTF-16 : suite de valeurs en lecture seule sur 16 bits, conversion automatique en objet de type `String`
- longueur : `s.length`, caractère i (1^{er} en 0) : `s.charAt(i)` ou `s[i]`
- valeurs littérales : entre `""` (`"texte"`) ou entre `' '` (`'texte'`)
- `'\udddd'` : caractère unicode (numéro hexadécimal 4 chiffres)
- concaténation avec `+` : si l'un des opérandes de `+` est une chaîne, l'autre est converti en chaîne et le résultat est une nouvelle chaîne, concaténation des deux
- pour concaténer plusieurs chaînes : les mettre dans un tableau et fusionner le tableau (plus efficace)
- conversion en chaîne de caractères : les autres types ont une méthode `toString()` et `toLocaleString()`

- recherche (-1 si pas trouvé) :
  - `s.indexOf(chaîne[,début])` : recherche depuis début
  - `s.lastIndexOf(chaîne[,début])` : recherche depuis la fin
- sous-chaîne :
  - `s.slice(from,to)` et `s.substring(from,to)` : renvoient une chaîne contenant les caractères aux positions `[from,to[` (`slice` : peuvent être négatives)
  - `s.split(délimiteur[,limite])` : renvoie un tableau contenant les sous-chaînes séparées suivant le délimiteur
  - `s.trim()` : renvoie la chaîne sans les espaces en tête et en fin de chaîne

4.3 Booléens

- valeurs `false` et `true`
- conversions :
  - les valeurs `undefined`, `null`, `0.0`, `NaN` et `""` sont *fausses* et converties en `false`
  - toutes les autres valeurs sont *vraies* et convertie en `true`
- non logique : `!`, convertit son opérande en booléen et renvoie l'opposé
- comparaison :
  - *identité* : `===` (resp. `!==`) renvoie vrai (resp. faux) si les opérandes sont de même type et ont la même valeur
  - *égalité* : `==` (resp. `!=`) compare ses opérandes *après* conversion éventuelle (`3.14 == "3.14"` → `true`)
    à ne pas utiliser en général

– opérateurs logiques et `&&` et `||` : ne renvoient pas des booléens mais des valeurs vraies ou fausses en fonction de la vérité ou de la fausseté des leurs opérandes :

A	B	A et B	A `&&` B
V	V	V	B
V	F	F	B
F	V	F	A
F	F	F	A

A	B	A ou B	A `||` B
V	V	V	A
V	F	V	A
F	V	V	B
F	F	F	B

– évaluation de gauche à droite en court-circuit

→ première valeur existante : `val = valeur1 || valeur2 || valeur3;`

→ valeur par défaut : `s = flight.status || "unknown";`

→ tester existence avant de lire : `t = flight.start && flight.start.hour;`

4.4 `null` et `undefined`

- seules valeurs de leur type
- `undefined` : n'existe pas
  - variable non déclarée ou non initialisée
  - propriété, paramètre non existant
  - valeur de retour des fonctions ne renvoyant pas de valeur
- `null` : existe mais n'a pas de valeur
  - pointe sur un objet non existant

  → ! `typeof null === "object"`

5 Instructions

– les instructions sont terminées par un point-virgule « `;` »

– s'il n'y en a pas, javascript en rajoute un dès qu'il peut (mais pas forcément à l'endroit attendu)

→ terminer systématiquement une instruction par `;`

5.1 Variables

– le nom d'une variable doit commencer par une lettre, `'_'` (données « privées ») ou `'$'` et ne doit pas contenir de ponctuation

– il ne doit pas être un mot réservé :

```
abstract boolean break byte case catch char class const continue
debugger default delete do double else enum export extends false
final finally float for function goto if implements import in
instanceof int interface long native new null package private
```

```
protected public return short static super switch synchronized
this throw throws transient true try typeof var void volatile
while with
```

– une variable est déclarée avec `var` et peut-être initialisée (sinon elle vaut `undefined`)

```
var x;              //undefined
var y = 2.1;
var p = null;
```

– une variable n'est pas typée : elle peut contenir des valeurs de types différents au cours du programme

```
var x = 3.1;      //Number
x = "bonjour";    //String
```

- une variable peut être :
  - globale (déclarée en dehors d'une fonction)
  - locale à une fonction (portée de fonction)
  - il n'y a pas de portée de bloc
- la déclaration est « remontée » au début de la portée (mais pas l'initialisation)
- on peut re-déclarer une variable avec `var` (l'initialisation devient une affectation)
- affectation : *`variable`* **`=`** *`valeur`*`;`
- affectation d'une variable non déclarée : erreur en mode strict, mais crée une variable globale sinon

  $\rightarrow$ ne pas oublier `var` quand on déclare une variable
- les variables objets sont des références sur des objets

5.2 Instructions conditionnelles

– **si sinon** :

```
if (condition) { instructions vrai }
[ else { instruction faux } ]
```

– si la condition est vraie, les instructions du `if` sont exécutées, si la condition est fausse, les instructions du `else` (s'il existe) sont exécutées

```
if (x)
{
  if (isNaN(x))
  { alert("n'est pas un nombre"); }
}
else
{ alert("rentrez une valeur"); }
```

– **choix multiple** :
`switch (condition) { case value: instructions ; ...}`

– les valeurs des `case` sont évaluées et testées jusqu'à tomber sur la valeur égale à celle du `switch`

– les instructions sont exécutées à partir de là jusqu'à la fin

→ ne pas oublier le `break;`

– `default:` : si aucun `case` n'a été trouvé

```
switch(c)
{
  case "a": case "e": case "i": case "o": case "u": case "y":
            lettre = "voyelle";
            break;
  default:  lettre = "consonne";
};
```

5.3 Instructions itératives

– boucle **tant que** :
`while (condition) { instructions }`

– teste la condition, si elle est vrai exécute les instructions et recommence, sinon passe à la suite

```
var log2 = 0;
while (n>1) { ++log2; n/=2; }
```

– boucle **faire tant que** :
`do { instructions } while (condition);`

– exécute d'abord les instructions, teste la condition, si elle est vrai recommence, sinon passe à la suite

```
var nbchiffres=0;
do {
  ++nbchiffres;  n/=10;
} while (n>0);
```

– boucle **pour** :

```
for (init; continuation; m.à.j.)  { instructions }
```

```
var fact = 1;
for (var i=2; i<=n; ++i) { fact *=i; }
```

– le compteur n'est pas local à la boucle mais global dans la portée où se trouve la boucle

– mais on peut re-déclarer le compteur dans chaque boucle sans poser de problème

6 Types objets

6.1 Objets

- un objet est une collection de *propriétés*, c.-à-d. de couples *(nom,valeur)*
- expression objet : entre accolades `{}`, liste des propriétés `nom: valeur` séparées par des virgules (nom entre `" "` si nom de variable incorrect ou mot réservé)

```
var p =  { x: 2.1, y: -3.5 };
var p2 = { "le x": 2.1, "le y": -3.5 };
```

- accès à une propriété d'un objet :
  - `o.pte` ou `o["pte"]` (le nom de la propriété peut alors être calculé)

```
console.log("p x=" + p.x); console.log("p2 x=" + p2["le x"]);
var coord = "le ";         console.log("p2 x=" + p2[coord+"x"]);
```

- les propriétés d'un objet sont librement accessibles : préfixer les noms des propriétés « privées » par `"_"` (juste une convention, ou utiliser les configurations des propriétés)
- la structure des objets est dynamique et variable
- quand on affecte une valeur à une propriété d'un objet :
  - si elle existe, la nouvelle valeur lui est affectée
  - si elle n'existe pas, elle est créée et initialisée avec la valeur
- on peut effacer une propriété d'un objet : `delete`

```
var p = { x: 2.1, y: -3.5 };
p.x = -5.1;     // p : x:-5.1, y:-3.5
p.z = 7.4;      // p : x:-5.1, y:-3.5, z:7.4
delete p.y;     // p : x:-5.1, z:7.4
```

– on peut créer un objet au fur et à mesure à partir d'un objet vide `{}`

– préférer si possible un objet littéral complet

```
var p1 = {};
p1.x = 3.14;
p1.y = -2.72;
var p2 = { x: 2.1, y: -3.5 };  //plus efficace
```

– on ne peut pas effacer une variable déclarée avec `var`

- les variables sont des pointeurs/références sur les valeurs objets
- les objets existent indépendamment des variables qui permettent d'y accéder
- les variables sont copiées, comparées et passées en paramètres par valeur
- mais si elles référencent des *objets*, ces objets sont copiés, comparés, passés en paramètre par *référence*
- rajouter éventuellement des méthodes pour comparer par valeur ou faire des copies

```
var p1 = { x: 2.1, y: -3.5 };
var p2 = { x: 2.1, y: -3.5 };
p1 === p2;          // -> false
var p3 = p2;
p3 === p2;          // -> true
p3.y = 0.1;         // p2 : x: 2.1, y: 0.1
```

- les valeurs gérées par javascript sont en mémoire dynamique et gérées par un ramasse-miettes
- elles ne peuvent être détruites que lorsqu'il n'y a plus de référence active dessus

→ libérer les références gardées en mémoire dans des variables non locales pour que les valeurs pointées puissent être détruites : `x = null;`

- notamment pour les objets non gérés directement par le moteur javascript du navigateur (objets du DOM p.ex.)
- quand javascript s'exécute, il existe un objet global (l'objet `window` de la fenêtre dans un navigateur)
- les données globales sont en fait des propriétés de cet objet global

6.2 Tableaux

- les tableaux sont des objets spéciaux dont les propriétés ont des numéros pour nom (commençant à 0)
- cases accessibles avec `t[`*`numéro`*`]` (`numéro` expression entière)
- valeur littérale tableau : `[`*`elt0, elt1,...`*` ]`
  `elta, ,eltb` laisse un trou (c.-à-d. `undefined`) entre `elta` et `eltb`
- constructeur tableau : `new Array(`*`taille, elt0, elt1, ...`*`)`
- propriété spéciale `length` :
  - $\geq maxindice + 1$, $maxindice$ plus grand indice occupé dans le tableau

  $\rightarrow$ égale au nombre d'éléments dans un tableau plein (sans trou)
  - mise à jour automatiquement quand on ajoute des valeurs au tableau
  - tableau mis à jour quand on la diminue (valeurs en trop effacées)

– parcourir un tableau plein :

```
for(var i = 0, len = t.length; i < len; ++i) {
// utiliser t[i]
}
```

– les tableaux peuvent être hétérogènes (les cases peuvent contenir des valeurs de types différents)

```
var c = [ "Dupont", 25, "Martin", 103, "Machin", 54, "Toto" , 2 ];
for (var i=0, len=c.length; i<len; i+=2) {
  console.log(c[i]+" classé "+(i/2+1)+"eme avec "+c[i+1]+" points");
}
```

– tableaux à plusieurs dimensions :

  – représenter le tableau à plat ligne par ligne dans un tableau à une dimension et calculer les indices correspondants

  – ou créer un tableau de tableaux

Modifier un tableau

- `delete t[i]` : efface la case du tableau en laissant un trou (`undefined`) sans changer `length` → tableau creux
- parcourir un tableau creux :

```
for(var i = 0, len = t.length; i < len; ++i) {
  if (t[i]===undefined) continue; //Skip undefined + nonexistent elements
// loop body here
}
```

- `t[i]=valeur` : si l'élément existe, change sa valeur, sinon l'ajoute dans le tableau et met éventuellement `length` à jour
  (→ `t[length]=valeur` ajoute la valeur à la fin du tableau)
- `t.length=len` : efface les éléments en trop du tableau (éléments d'indice>`len`)

- opérations de pile/file :
  - `t.push(x)`/`t.unshift(x)` : ajoute `x` à la fin/au début du tableau
  - `t.pop(x)`/`t.shift(x)` : enlève le dernier/premier élément du tableau et le renvoie
- `t.splice(`*`begin`*`[,`*`nb`*`[,`*`elems`*`...]])` :
  - enlève du tableau les `nb` éléments (jusqu'à la fin si `nb` non donné) à partir de l'indice `begin`,
  - insère les éléments `elems` donnés ensuite à partir de la position `begin`
  - et renvoie un tableau contenant les éléments enlevés
- `t.reverse()`/`t.sort()` : inverse/trie le tableau et le renvoie

Autres méthodes

- `t.join([delim])` : renvoie la chaîne concaténation des valeurs du tableau séparées par le délimiteur (`,` par défaut), inverse de `s.split()` de `String`
- `t.toString()` : pareil que `t.join()`
- `t.concat(elem)` : renvoie tableau contenant les éléments du tableau `t` et ceux passé en paramètre (tableaux en paramètres aplatis)
- `t.slice(begin,end)` : renvoie un tableau contenant les éléments d'une « tranche » de `t`
  - `t.slice(begin,end)` : éléments d'indices compris dans `[begin,end[`
  - `t.slice(begin)` : éléments de `begin` jusqu'à la fin
  - indices négatifs : indices comptés à partir de la fin du tableau

- pseudo-tableaux :
  - objets ayant des valeurs accessibles par `t[`*`numéro`*`]` et une propriété `length` correspondante
  - mais ces valeurs ne sont pas effaçables/insérables comme pour un tableau
  - et n'ayant pas les méthodes d'un tableau
- appliquer une méthode de tableau non modificatrice à un objet pseudo-tableau `t` : `Array.prototype.`*`méthode`*`.call(t,`*`paramètres`*`)`

```
console.log(Array.prototype.join.call("bonjour")); //b,o,n,j,o,u,r
```

6.3 Autres Objets utiles

- javascript contient d'autres objets utiles : dates `Date`, expressions régulières `RegExp`, erreurs `Error`

7 Fonctions

– on peut déclarer une fonction avec une *instruction* de définition de fonction :

```
function distance(x1, y1, x2, y2) {
  var dx = x2-x1, dy = y2-y1;
  return Math.sqrt(dx*dx+dy*dy);
}
```

– `return` interrompt la fonction et renvoie la valeur

– sans `return`, ou avec `return;` : la fonction renvoie `undefined`

– ! le début de l'expression renvoyée par `return` doit être sur la même ligne

– les paramètres sont non typés et non vérifiés :

  – s'il a trop d'arguments à l'appel, ceux en trop sont ignorés

  – s'il en manque, les paramètres restants sont initialisés à `undefined`

```
function distance(x1, y1, x2, y2, n)
{
  var dx = x2-x1, dy = y2-y1;
  if (n === undefined)              //distance(x1,y1,x2,y2) = L2
  //ou mieux ici : if (isNan(n))
  { return Math.sqrt(dx*dx+dy*dy); }
  else                              //distance(x1,y1,x2,y2,n) = Ln
  { return Math.pow(Math.pow(dx,n)+Math.pow(dy,n),1.0/n); }
};
```

→ on ne peut pas surcharger une fonction

– paramètres par défaut : utiliser p.ex. `||`

```
function distance(x1, y1, x2, y2)
{
//distance(x,y) = distance(x,y,0.0,0.0)
  x2 = x2 || 0.0; y2 = y2 || 0.0;
  var dx = x2-x1, dy = y2-y1;
  return Math.sqrt(dx*dx+dy*dy);
};
```

– regrouper plusieurs valeurs dans un objet plutôt que de les passer dans un ordre précis dans des paramètres séparés

```
function aireRectangle1(x, y, width, height)
{
//rectangle: x,y,width,height, but x and y not used here
  return width*height;
};

function aireRectangle(r)
{
  return r.width*r.height;
};

var r1 = { x:2, y:3.1; width:120, height:140 };
var aire = aireRectangle(r1);
var area = aireRectangle({ width:140, height:120 });
```

- les types primitifs sont passés par valeur
- les objets sont passés par référence
- mais les variables elles-mêmes sont passées par valeur

```
function testParam(n, p1, p2) {
  n = 0;
  p1.x = 0;
  p2 = { x:0, y:0 };
}

var n = 10;
var p1 = { x:10, y:10 };
var p2 = { x:20, y:20 };
testParam(n, p1, p2);
//n=10, p1 = { x:0, y:10 }, p2 = { x:20, y:20 }
```

- les variables déclarées à n'importe quel endroit dans une fonction (hors fonctions internes) sont des variables *locales* à la fonction :
  - elles ne sont utilisables que dans la portée de la fonction (le code situé à l'intérieur de la fonction)
  - mais sont utilisables dans **toute** la portée de la fonction (dans tout le code situé à l'intérieur de la fonction :
    - leur déclaration est remontée au début de la fonction
  - elles cachent les variables des portées englobantes
- une fonction a accès à toutes les variables qui lui sont accessibles au moment de sa création (et non de son utilisation)

```
//never write such a code!
function f() {
  a = 1.1;
  b = 2.1;
  c = 3.1;
  d = 4.1;
  var a;
} ;

var a = -1, b = -2;
f();
var c= -3, d;
console.log("a="+a+" b="+b+" c="+c+" d="+d);
//a=-1 b=2.1 c=-3 d=4.1
```

– les fonctions sont en fait des valeurs objets anonymes, éventuellement référencées par des variables pour être utilisées

– on peut créer une fonction avec une *expression* de définition de fonction qui crée un objet fonction anonyme et le renvoie :

```
var distance = function (p1, p2)
{
  var dx = p2.x-p1.x, dy = p2.y-p1.y;
  return Math.sqrt(dx*dx+dy*dy);
} ;
```

– un nom peut être donné à la fonction (dans l'expression après `function`) : le nom n'est utilisable que dans le corps de la fonction (pour fonction récursive)

– une *expression* de définition de fonction peut être utilisée partout

- une instruction de déclaration de fonction
  - est une instruction de haut niveau (non utilisable dans une instruction conditionnelle, itérative)
  - qui déclare une variable globale de nom la fonction, et l'initialise avec l'objet fonction créé correspondant
  - la déclaration de la variable et son initialisation étant remontée en début de portée

```
//début de portée                          //début de portée
                                           var f = function() { ... };
...                                        ...
function f() { ... };          =>
...                                        ...

//début de portée                          //début de portée
                                           var f;
...                                        ...
var f = function() { ... }; =>             f = function() { ... };
...                                        ...
```

– plusieurs variables peuvent pointer sur la même fonction

– on peut changer la fonction pointée par une variable

```
var pdistance = distance;
//pdistance === distance
var p1 = { x:2.1, y:-3.2 },  p2 = { x:-3.2, y:5.4 };
var dd = distance(p1,p2);
var dp = pdistance(p1,p2);  //dd === dp

distance = function (x1, y1, x2, y2)
{
  var dx = x2-x1, dy = y2-y1;
  return Math.sqrt(dx*dx+dy*dy);
};
//distance is now another function
dd = distance(p1.x, p1.y, p2.x, p2.y);
```

7.1 Méthodes

– la valeur d'une propriété d'un objet peut être une fonction : la fonction est alors une *méthode*

  – on l'appelle sur l'objet

  – l'objet sur lequel la méthode est appelée est accessible dans la fonction par le mot clé `this`

  – `this` doit être utilisé explicitement pour accéder aux propriétés de l'objet courant,
  sinon on accède aux variables locales et globales et aux paramètres

```
var p = { x: 2.1,  y: -3.5,
          d: function() {
               return Math.sqrt(this.x*this.x+this.y*this.y);
             }
        };
var pn = p.d();
```

– les méthodes sont des objets fonctions comme les autres, elles n'ont pas besoin d'être définies dans les objets sur lesquels elles sont appelées

```
function norme(n) {
  n = n || 2; return Math.pow(Math.pow(Math.abs(this.x),n)+
                              Math.pow(Math.abs(this.y),n),1.0/n);
};
var p = { x: 2.1, y: -3.5, d: norme };
var pn = p.d(3);
```

– on peut appeler une méthode sur un objet dont elle n'est pas une propriété

  – *fonction*.**call**(*obj, argument1, ...*)
  – *fonction*.**apply**(*obj, tableau des arguments*)

– `this` désigne alors l'objet `obj` passé à `apply` ou `call`

```
var p = { x: 2.1, y: -3.5 };
var pn = norme.call(p,3);
```

– on peut passer `null` comme objet, cela permet de passer les valeurs contenues dans un tableau en paramètres à une fonction

```
fonction.apply(null, tableau des arguments)
```

```
function somme(x,y) { return x+y; };
var s = somme.apply(null,[1,2])); //s = 3
```

– `this` ne dépend pas de la fonction, mais dépend du contexte d'**appel** de la fonction :

  – fonction appelée comme fonction globale, **`f()`** : `this` désigne `undefined` en mode strict, sinon l'objet global (`window` dans navigateur)

  – fonction appelée comme méthode, **`obj.f()`** : `this` désigne l'objet `obj` sur lequel la méthode est appelée

  – fonction appelée indirectement avec **`f.apply`** ou **`f.call`** : `this` désigne l'objet passé à `apply` ou `call`

7.2 Accès aux données

- accéder aux variables locales coute moins cher que d'accéder aux
  - variables globales
  - propriétés de certains objets

→ mettre en mémoire dans des variables locales les données globales ou certaines propriétés souvent utilisées

```
function longeurPolygone(poly) { //tableau des sommets
  var dist = Geom.2d.dist;
  var len = poly.length;
  var l = 0;
  for (var i=1 ; i<len; ++i) {
    l += dist(poly[i-1],poly[i]);
  }
  l += dist(poly[0],poly[len-1]);
  return l;
};
```

8 « Classes » en javascript

8.1 Créer des objets : fonction usine

– pour créer des objets de « même type » (mêmes propriétés et mêmes méthodes) sans ajouter les propriétés (notamment les méthodes) à la main par copier-coller : fonction usine

```
function makePoint(x, y) {
  return {
    x: x, y: y,
    distance: function(p) { var dx=p.x-this.x, dy=p.y-this.y;
                            return Math.sqrt(dx*dx+dy*dy); },
    move: function(dx,dy) { this.x += dx; this.y += dy; },
    toString: function()  { return "("+this.x+","+this.y+")"; }
  };
};

var p = makePoint(3.1,-2.4); console.log(""+p); p.move(1,2);
```

– inconvénient : les fonctions sont recalculées et recréées pour chaque objet créé

→ créer ces fonctions à l'avance et les ranger dans un objet, p.ex. l'objet fonction usine :

```
function makePoint(x, y) {
  return {
    x:x, y:y,
    distance = makePoint.distance,
    move     = makePoint.move,
    toString = makePoint.toString
  };
};
makePoint.distance = function(p) {
   var dx = p.x-this.x, dy = p.y-this.y;
   return Math.sqrt(dx*dx+dy*dy);
};
makePoint.move = function(dx,dy) { this.x += dx; this.y += dy; };
makePoint.toString = function() { return "("+this.x+","+this.y+")"; };
```

8.2 Prototype

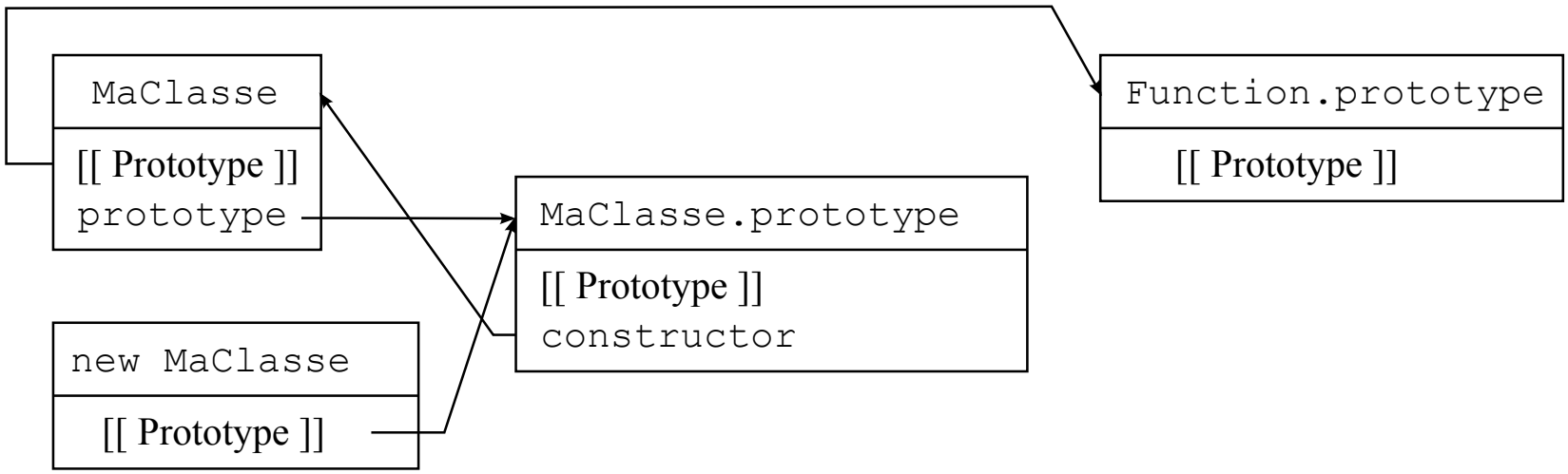
- inconvénients :
  - on doit mettre les méthodes partagées par les objets à la main dans l'objet
  - objet qui contient donc une propriété supplémentaire pour chaque méthode
  - on ne peut distinguer les objets créés avec la fonction usine de ceux créés autrement
- tout objet a un *objet prototype*
- même l'objet prototype d'un objet a un lui-même un objet prototype ... $\rightarrow$ chaîne de prototypes
- en javascript, un objet hérite de son objet prototype

- quand on lit une propriété d'un objet `o.pty` :
  - si l'objet `o` contient la propriété `pty`, alors cette propriété est utilisée
  - sinon, c'est la propriété `pty` de l'objet prototype de l'objet `o` qui est utilisée
  - sinon, la recherche continue en remontant la chaîne de prototypes
  - si finalement elle n'existe pas : `undefined`
- quand on affecte une valeur à une propriété d'un objet `o.pty = value` :
  - si l'objet `o` contient lui-même la propriété `pty`, alors la valeur est affectée à la propriété
  - sinon, la propriété `pty` est créée dans l'objet `o` et initialisée avec la valeur (et cache donc éventuellement la propriété du prototype)

8.3 Fonction constructeur

- tout objet fonction a une propriété `prototype` pointant sur un objet créé automatiquement lors de la création de la fonction
- cet objet a une propriété `constructeur` pointant à son tour sur l'objet fonction
- on peut invoquer une fonction avec `new` :
  - elle crée un objet vide ayant comme objet prototype l'objet de sa propriété `prototype`
  - cet objet est désigné par `this` dans la fonction
  - s'il n'y a pas de `return`, la fonction renvoie automatiquement cet objet
- si la fonction n'est pas invoquée avec `new`, `this` désigne l'objet global !

→ un objet créé avec `new F(...)` hérite des propriétés de `F.prototype` (et en plus `var f = new F(...); //f.constructor === F`)



→ « classe » en javascript :

- fonction constructeur qui initialise les propriétés propres de l'objet (convention : nom commence par une majuscule pour indiquer qu'il faut l'invoquer avec `new`)
- données partagées (méthodes et données « de classe ») mises dans l'objet pointé par la propriété `prototype` du constructeur

– créer une « classe » point

```
//constructor function
function Point(x, y) {  //set properties of the object
  this.x = x;
  this.y = y;
};
//methods of Point
Point.prototype.distance = function(p) {
                  var dx = p.x-this.x, dy = p.y-this.y
                  return Math.sqrt(dx*dx+dy*dy);
                };
Point.prototype.move = function(dx,dy) {
                  this.x += dx; this.y += dy;
                };
Point.prototype.toString = function() {
                  return "("+this.x+","+this.y+")";
                };

var p = new Point(3.1,-2.4);  p.move(3,4);
```

8.4 Typage

– javascript étant totalement dynamique, il n'y a pas besoin de se reposer sur un typage statique de classe pour pouvoir utiliser les objets

– on peut utiliser comme typage la « canardisation » (duck typing, James Whitcomb) :

> When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.

```
function norm(p) {
//p must have properties x and y
  return Math.sqrt(p.x*p.x+p.y*p.y);
};

function Point(x, y) {  //constructor
  this.x = x;
  this.y = y;
};

var n1 = norm({ x:3.2, y:-8.3 });
var p3d = { x:2.2, y:-7.3 z:5.5 };
var n2 = norm(p3d); //norm in the plane
var n3 = norm(new Point(3.1,-2.4));
```

- il peut être utile dans certains cas de tester le type d'un objet :
  - `o instanceof C` : vrai si `o` hérite de `C.prototype`, c.-à-d. si `o` a été créé avec le constructeur `C`

```
function norm(p) {
  if (!(p instanceof Point))
  { throw New TypeError("norm needs a Point"); }
  return Math.sqrt(p.x*p.x+p.y*p.y);
};
```

  - `p.isPrototypeOf(o)` : vrai si `p` est dans la chaîne de prototypes de `o`

- sinon simplement tester la présence et le type des propriétés utilisées

```
function norm(p) {
  if (!(typeof p.x === "number" && typeof p.x === "number"))
  { throw New TypeError("norm needs p.x and p.y"); }
  return Math.sqrt(p.x*p.x+p.y*p.y);
};
```

8.5 Types dynamiques

- on peut modifier un objet prototype en cours d'exécution du programme

→ change le comportement de tous les objets dérivés de ce prototype, même de ceux déjà existants

- en général, permet de rajouter des propriétés à des objets
- éviter de le faire pour les objets « standard » javascript
- sauf pour ajouter des propriétés « standard » inexistantes dans l'implémentation
- pour créer directement un objet ayant pour objet prototype un objet donné : `Object.create(`*`prototype`*`)`

9 Fonctions pour créer des modules

9.1 Exécution sans laisser de traces : IIEF

- les variables locales d'une fonction sont
  - créées à chaque appel de la fonction
  - inaccessibles depuis l'extérieur de la fonction après son exécution
  - mais toujours accessibles aux fonctions internes à la fonction créées à ce moment là
- pour exécuter du code sans laisser de traces des variables et fonctions utilisées :
  - ranger ce code dans une fonction et l'exécuter
  - pour ne même pas laisser trace de cette fonction : créer une fonction anonyme temporaire et l'exécuter dans la foulée :
    Immediately Invoqued Expression Function (IIEF)

– `(function() { })` : transforme l'instruction de déclaration de fonction en une expression de déclaration de fonction, objet fonction que l'on peut appeler ensuite et qui sera ensuite détruit :

```
(function() {
  //code à exécuter
}) ();  //IIEF
```

– exemple

```
(function () {
function dessus(event) { ... }
function dehors(event) { ... }
function init(event)  {
  var p = document.querySelectorAll("p.phrase");
  if (p[0]) { p[0].addEventListener("mouseover", dessus, false);
              p[0].addEventListener("mouseout", dehors, false); }
}
window.addEventListener("load", init, false);
})();
```

9.2 Objets modules/espaces de noms

- il faut éviter les données (variables et fonctions) globales qui polluent la portée globale
- un objet est une collection de propriétés (données et fonctions)
- ces propriétés, au lieu de représenter les propriétés d'un « objet », peuvent représenter les propriétés d'un module

→ utiliser un objet comme objet espace de noms pour regrouper les éléments d'un module

9.3 Création directe

- on peut créer directement cet objet s'il n'existe pas
- et rajouter dedans les propriétés

- inconvénients :
  - ne pas oublier de qualifier les appels internes à l'espace de noms
  - tout est publique

```
var Geom = Geom || {};    //reuse if exists already

//add content

Geom._distxy = function (x1,y1,x2,y2) { //should be private
    var dx = x2-x1, dy = y2-y1;
    return Math.sqrt(dx*dx+dy*dy); };

Geom.dist = function (p1,p2) {           //don't forget Geom. inside
    return Geom._distxy(p1.x, p1.y, p2.x, p2.y); };

Geom.lTriangle = function (p1,p2,p3) {
    return Geom.dist(p1,p2)+Geom.dist(p2,p3)+Geom.dist(p2,p3); };

//test
console.log(Geom.lTriangle({x:0,y:0}, {x:0,y:4}, {x:3,y:0});
```

9.4 IIEF pour créer un module

- programmer un module avec des données globales
- ranger le code du module dans une IIEF qui cache tout
- à la fin, renvoyer un objet ne contenant et n'exposant que la partie « publique » du module

```
var GEOM = (function () {
  function distxy(x1,y1,x2,y2) { ... }
  function Point(x,y) { ... }
  Point.prototype.dist = function(p) { return distxy(this.x,this.y,p.x,p.y);
  ...
  function determinant(p1,p2,p3) { return ...; }
  function aGauche(p1,p2,p3)     { return determinant(p1,p2,p3)>0; }

  //export public data
  return { Point: Point,
           aGauche: aGauche };
})();
```

- inconvénient : efface l'espace de noms s'il existait déjà
- passer le module en paramètre de l'IIEF et lui ajouter les propriétés « publiques »

```
var GEOM = GEOM || {};

(function (exports) {
  function distxy(x1,y1,x2,y2) { ... }
  function Point(x,y) { ... }
  Point.prototype.dist = function(p) { return distxy(this.x,this.y,p.x,p.y);
  function determinant(p1,p2,p3) { return ...; }
  function aGauche(p1,p2,p3)     { return determinant(p1,p2,p3)>0; }

  //add public data to exported module
  exports.Point = Point;
  exports.aGauche =  aGauche ;
})(GEOM);
```