JavaScript

Programmation Web - Avancé

Aspects administratifs

- Représente 50% de l'UE Programmation Web - Avancé
- Evaluation : à l'examen uniquement
  - Sur machine, mini-projet à réaliser
  - Les dernières semaines de cours consisteront en un mini-projet préparatoire à l'examen
- Vous aurez largement l'occasion d'utiliser vos compétences JS dans le projet du second semestre

Organisation du cours

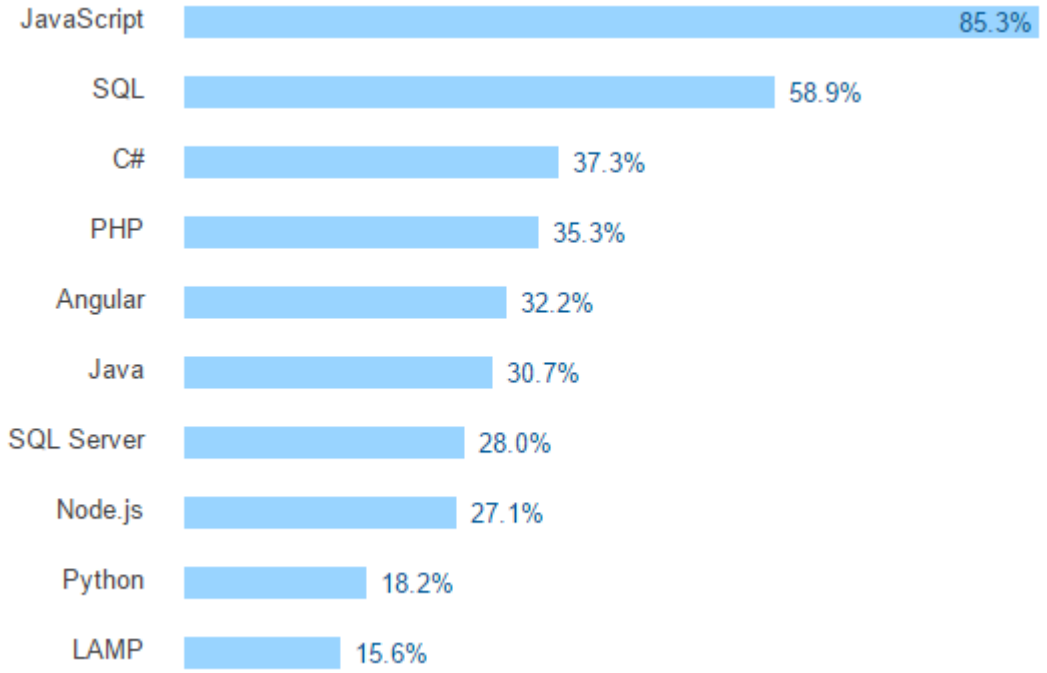
- Semaines 1 à 6 : 6 X 4h/semaine
- Semaines 7 à fin : une plage horaire de JS est recyclée en ErgoWeb : 6 x 2h/semaine

Calendrier (approximatif)

Semaine	Sujet
S1	JS : introduction au langage
S2	HTML, DOM & JS
S3	jQuery
S4	Formulaires & JSON
S5	HTTPServlet & Genson
S5	Ajax
S6	Frameworks JS
S7-S12	Exercices récapitulatifs

JavaScript : pourquoi ?

- http://stackoverflow.com/research/developer-survey-2016



JavaScript : langage d’avenir

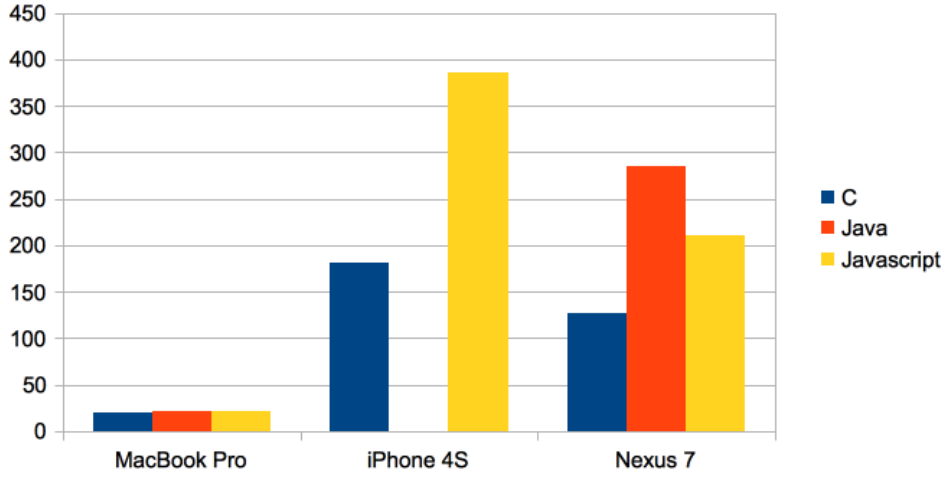
- Langage pour le front-end Web
- Diversification au back-end Web : Node.js
- Diversification aux applications desktop : Windows 8 introduit le Windows Runtime App
- Vous pouvez créer des applications Windows universelles en utilisant les langages de programmation les plus répandus, comme JavaScript

JavaScript : mauvaise réputation

- Argument 1 : performances médiocres
  - Vu l'importance du langage dans le monde web, une énergie considérable est investie par Google, Mozilla et Microsoft pour optimiser au maximum.

http://www.stefankrause.net/wp/?p=144

Axe vertical : le temps



JavaScript : mauvaise réputation

- « Quirks » : le langage est « sale »
  - Ceci couvre des spécificités du langage qui sont parfois inhabituelles, parfois vraiment étranges.
  - Beaucoup de manières de faire la même chose, et ce, de manière vraiment différente.
  - Pour éviter que cela ne devienne un souci, il faut s'en tenir à ce qui fonctionne bien et rester discipliné dans son développement
    Ceci est en fait vrai pour tous les langages !
  - Nous verrons ces choses en temps utile.

JavaScript : les bons côtés

- Beaucoup de manières de faire la même chose de manières différentes :
  - Parmi toutes ces manières, il en existe qui sont simples à comprendre et à réaliser, qui utilisent les spécificités du langage à bon escient.
    - Ceci ne signifie pas que cela sera similaire à du Java !
  - C'est sur cela que ce cours veut se concentrer.

JavaScript : langage interprété

- Il n'y a pas de phase de compilation.
  - (au plus bas niveau, ceci n'est pas totalement exact, mais cela fonctionne comme s'il n'y en avait pas).
- C'est donc le code source qui est directement utilisé lors de l'exécution.

JavaScript : introduction

- Langage vaguement inspiré de la famille des langages C (comme Java, C++, Objective-C,…).

```
for (i=0; i<10; i++) {
    somme+=i;  // commentaire
}
/* commentaire
   multi
   lignes
*/
```

Similarités

- ```
  while (i < 10) {
      text += "The number is " + i;
      i++;
  }
  ```
- ```
  do {
      text += "The number is " + i;
      i++;
  }
  while (i < 10);
  ```
- ```
  if (hour < 18) {
      greeting = "Good day";
  } else {
      greeting = "Good evening";
  }
  ```
  ```

Similarités

```
switch (new Date().getDay()) {
    case 1:
    case 2:
    case 3:
    default:
        text = "Looking forward to the Weekend";
        break;
    case 4:
    case 5:
       text = "Soon it is Weekend";
        break;
    case 0:
    case 6:
       text = "It is Weekend";
}
```

JavaScript : langage **dynamiquement** typé

- Le type des variables est déterminé à l'exécution, et n'est même pas à spécifier.
  - Le mot clef var déclare les variables.
  - Le type peut changer en cours d'exécution !

```
var somme=0;
for (var i=0; i<10; i++) {
    somme+=i;
}
var toto="bonjour toto";
toto=42;
```

JavaScript : types http://www.w3schools.com/js/js_datatypes.asp

```
var i=1; // Number
var f=1.0; // Number
var s="chaîne"; // String
var s2='chaîne'; // String
var s3=new String('chaîne'); // String
var a=[1,2.0,"3"]; // Array
var b=true; // Boolean
var u; // undefined
var n=null; // null
```

Notez la différence : undefined => la variable n'a pas été affectée, son type est inconnu.
null => cas particulier des objets

- (il manque encore les objets associatifs et les fonctions)

Javascript : opérateur instanceof

instanceof	Number	String	Array	Object	Boolean	Function
1						
new Number(1)	■			■		
"a"						
new String("a")		■		■		
[1] // Array			■	■		
new Array()			■	■		
{} // Object				■		
new Object()				■		
function()				■		■
true						
new Boolean(true)				■	■	
null						
undefined						

```
var color1 =
new String("green");
color1 instanceof String;
// returns true
var color2 = "coral";
color2 instanceof String;
// returns false (color2 is not a String object)
```

1 et new Number(1) sont fonctionnellement identiques en Javascript. Cependant en interne, 1 correspond à un int Java, tandis que new Number(1) correspond à un Integer. Ceci explique le comportement bizarre d'instanceof.

Javascript : typeof

- L'opérateur typeof v : renvoie une chaîne de caractère décrivant le type de v

	typeof
1	"number"
new Number(1)	"object"
"a"	"string"
new String("a")	"object"
[1] // Array	"object"
new Array()	"object"
{} // Object	"object"
new Object()	"object"
function()	"function"
true	"boolean"
new Boolean(true)	"object"
null	"object"
undefined	"undefined"

Pour être vraiment certain d'un type (p.ex String), il faudra donc utiliser typeof et instanceof :

typeof s=="string" || s instanceof String

JavaScript : à propos du typage

- Dynamiquement typé == le type de la variable est connu à l'exécution uniquement.
  - Contraire : statiquement typé (ex. : Java).
- Faiblement typé == la variable contient une certaine valeur qui peut être interprétée suivant plusieurs types.

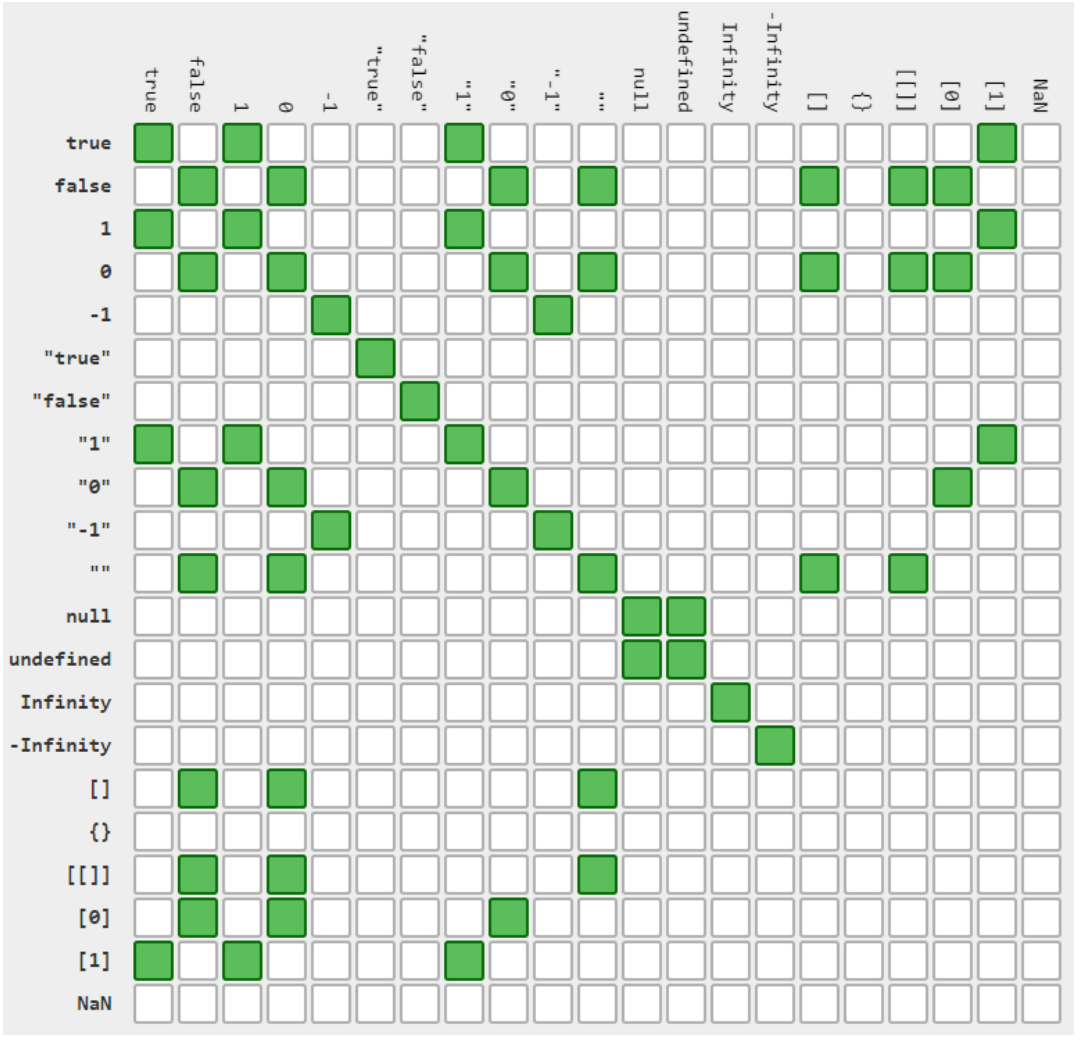
  10 == la chaîne 10 == l'entier 10 == le flottant 10.0
  - Contraire : fortement typé (ex. : Java).
- Dynamiquement typé != faiblement typé.

JavaScript n’est pas faiblement typé

- Mais certains opérateurs se comportent comme si c’était le cas !
- Une transtypage est alors automatiquement appliqué.

JavaScript et ==

- L'opérateur == effectue un tel transtypage.
- Ceci donne des résultats parfois surprenants.
- En général : c'est une mauvaise idée de l'utiliser.



JavaScript et ===

- === est l'opérateur d'égalité qui n'effectue pas de transtypage.
  - En général, c'est celui-ci qui doit vraiment être utilisé, pas ==
- Idem !== et !=

JavaScript : langage fonctionnel

- Une fonction, c'est comme une méthode, mais sans la classe.

```
function add(a,b) {
  return a+b;
}
var total=add(2,3);
```

Fonction : citoyen de premier ordre !

- Contrairement au Java classique, les fonctions sont assignables aux variables.
  - Une variable peut donc être du type "function".

```
function add(a,b) {
  return a+b;
}
var add2=add;
var total2=add2(3,4);
```

Fonctions anonymes

- Comme une fonction peut être affectée à des variables de noms différents, le nom de la fonction n'a pas réellement d'importance (contrairement aux méthodes en Java).
- On peut même ne pas nommer une fonction, dans ce cas on parle d'une fonction anonyme :

```
var add3 = function(a,b) {return a+b;}

var total3 = add3(7,8)
```

Exécution d'une fonction

- C'est la présence ou l'absence des parenthèses qui détermine si on parle de la référence de la fonction, ou de son exécution.

```
function copy(getter,setter) {
    setter(getter());
}
var a;
copy(function() {return 2;}, function(p) {a=p;});
```

Portée lexicale des variables

- La portée lexicale d'une variable en JavaScript **n'est pas** les accolades englobantes.
- La portée lexicale d'une variable en JavaScript **est la fonction** englobante.
- Ceci est un quirks de JavaScript, généralement anodin, mais parfois source de bugs très mesquins.

Attention aux variables globales

```
var i=1;
function toto() { // toto cherche à renvoyer la valeur de i
    return i;
}
toto(); // renvoie 1
i=5;
toto(); // renvoie 5
```

- Au moment de la définition de la fonction, c'est la référence de la variable globale i qui est considérée dans toto(), pas sa valeur !

  $\Rightarrow$si i change, toto() renvoie la dernière valeur assignée à i.

Complexifions un peu

```
function test() {
  var i=1;
  function toto() { // toto cherche à renvoyer la valeur de i
    return i;
  }
  i=5;
  return toto;
}
var  monToto=test(); // (1)
monToto() // (2) mais que renvoie ceci ???
```

Le i de toto() est le i déclaré par test().

- A la fin de (1), test() a fini de s'exécuter, on pourrait croire que son i disparaît de la mémoire.
- Mais alors quand on fait (2), de quel i parle la fonction monToto() ?

Closure

http://www.w3schools.com/js/js_function_closures.asp

```
function test() {
  var i=1;
  function toto() { // toto cherche à renvoyer la valeur de i
    return i;
  }
  i=5;
  return toto;
}
var  monToto=test(); // (1)
monToto() // (2) renvoie la dernière valeur de i => 5
```

- toto(), a besoin de garder une référence à i pour pouvoir fonctionner.
- Ce que la fonction test() renvoie, c'est la fonction toto **accompagnée de son environnement.**
  - Ici son environnement, c'est la variable i.
  - Une fonction qui embarque ainsi son environnement est appelée une fermeture transitive (closure en anglais).

Autre exemple de closure

```
function compteur() {
        var cpt=0;
        function incr() {
                cpt++;
                return cpt;
        }
        return incr;
}
var compteur1=compteur();
var compteur2=compteur();
compteur1(); // renvoie 1
compteur1(); // renvoie 2
compteur1(); // renvoie 3
compteur2(); // renvoie 1
```

- compteur1 et compteur2 sont deux closures qui embarquent chacune leur propre variable cpt.
- Elles comptent donc indépendamment l'une de l'autre !

Ex. de problème lié à la portée lexicale

```
function foo() {
      for(var i=0; i<10; i++) {
            ... // des trucs
      }
      .. // des trucs
      var i;
      .. // d'autres trucs qui changent i ou pas
      if (i===undefined) // problème !
}
```

Comme on est dans la même fonction, c'est le même i. Le redéclarer (var) n'y change rien !

Correction du problème précédent

```
function foo() {
  var i; // la portée lexicale étant globale,
         // on rend la déclaration globale aussi.
  for(i=0; i<10; i++) {
     ... // des trucs
  }
  ... // d'autres trucs
  i=undefined;// réaffectation de i à undefined
  ... // d'autres trucs qui changent i ou pas
  if (i===undefined) { ... }
}
```

Autre exemple de problème

```
for(var i=0; i<10; i++)
    a.push(function() {return i;});
    // http://www.w3schools.com/jsref/jsref_push.asp

a[3]() === 10 // true !
```

- La variable i n'est pas redéfinie dans la fonction anonyme (pas de var i), c'est donc le i englobant qui est utilisé. Ce i finit par valoir 10, et toutes les fonctions de l'Array a renverront donc cette valeur 10.
- Notez que ce qui est mis dans le tableau a, ce sont des **closures** : chacune de ces fonctions embarque son environnement défini par la variable i.

Correction du problème précédent

```
for(var i=0; i<10; i++)
   a.push((function(i) {
      return function() {return i;}
   })(i));
```

- la fonction anonyme englobante prends un i en paramètre, ce i est unique à chaque fois qu'on l'appelle.
  - Cette fonction n'est donc pas une closure.
- la fonction anonyme interne utilise ce i, qui reste cette fois unique.
  - Cette fonction-ci par contre défini une closure.
- la fonction englobante retourne la fonction interne, cette fonction est appelée immédiatement.

Technique pour réintroduire une portée lexicale classique

```
(function() {
    var a,b,c;
...
})();
```

- La fonction anonyme est immédiatement appliquée (exécutée).
- Les variables à l'intérieur de la fonction ont une portée lexicale propre à cette fonction.
- Notez les parenthèses autours de la fonction, elles sont syntaxiquement nécessaires.

OO en JavaScript

- JavaScript supporte aussi la programmation orientée-objet
  - `new Object(), new Array(), ...`
  - La manière dont cela fonctionne est très fortement différent d'un Java ou autres langages de programmation OO.
  - Nous parlerons des aspects OO plus tard.

OO en JavaScript

- Indépendamment de la programmation OO, les Objets JavaScript agissent aussi comme des Maps :
  - Association d'une clé (une String) à une valeur (de n'importe quel type)
  - Ceci s'appelle des objets associatifs.

```
var o={"nom":"Atreides",
       prenom:"Paul", // les " de la clé
                      // sont facultatifs
      "age":30,
      "truc":function(p1,p2) {...}
 }
```

Gestion des attributs

- Valeur d'un attribut

```
o.nom          // renvoie "Atreides"
o["nom"]     // renvoie "Atreides"
```

La seconde syntaxe est basée sur une String et peut contenir des espaces et caractères spéciaux; la première pas.

- Changer un attribut

```
o.nom="Muad'dib";
o["nom"]="Kwisatz Haderach";
```

- Supprimer un attribut

```
delete o.nom;
delete o["prenom"];
```

Parcours sur les clés

`Object.keys(o)`

renvoie le tableau des clés `["nom","prenom",...]`

```
for(var k in o) {
      // k vaudra successivement les clés de o
}
```

Objets et héritage

- Nous n'avons pas parlé de l'héritage, juste un petit écart :
  - Les attributs d'un objet peuvent provenir d'un parent de l'objet.
  - `Object.keys` ne renvoie pas les attributs parents.
  - `for(... in ...)` par contre itère bien sur les attributs parents.
  - La méthode `hasOwnProperty()` retourne un booléen indicant si l'objet possède l'attribut spécifié.
    - `o.hasOwnProperty("attribut")`

Voilà, nous en connaissons suffisamment pour programmer proprement en JavaScript

- Si si !
- Comment on organise le code sans OO ?

Pseudo objet en JavaScript fonctionnel

```
var o=(function() { // création d'une portée
lexicale
      var a,b,c; // des variables restant privées
      function truc(p1,p2) { // aussi privé
      ...
      }
      function machin() { ... }
      return {
            getA:function() {return a;}
            "machin":machin // machin rendue publique
      }
})(); // exécution immédiate de la fonction (self-invoking)
```

Utilisation

```
o.getA(); // renvoie la valeur de a
o.truc(1,2); // appel de la fonction
```

Variation : Pseudo Classe

```
var createPerson=function(name,surname) {
        var age,address;
        function setAddress(a) { address=a;}
        function setAge(a) { age=a;}
        var self={
                getName:function() {return name;},
                getSurname:function() {return surname;},
                getAge:function() {return age;},
                getAddress:function() {return address;},
                setAge:setAge,
                setAddress:setAddress
        }
        return self;
};
```

- Nous utilisons une variable pour retenir la pseudo-instance retournée. Nous l'appelons ici self, this étant un mot-clef réservé.

Utilisation : Pseudo Classe

```
var toto=createPerson("Toto","Blague");
toto.setAddress("rue de la blague, 10");
toto.setAge(7);
var jean=createPerson("Jean","Valjean");
jean.setAge(42);
```

Variation : Pseudo Héritage

```
var createEtudiant=function(nom,prenom) {
      var self=createPerson(nom,prenom);
      var noma;
      self.setNoma=function(n) {noma=n;}
      self.getNoma=function() {return noma;}
      return self;
}
```