M1IF03
Conception d'applications Web

RAPPELS JAVASCRIPT

LIONEL MÉDINI
NOVEMBRE 2023

Plan du cours

- Introduction
  - Rich Internet Applications (RIA)
  - Historique
  - JavaScript aujourd'hui
- Rappels JavaScript
  - Programmation fonctionnelle
  - Programmation orientée-objet
  - Programmation événementielle
- Références

Introduction

- Objectif : concevoir des applications Web « riches »
  - Web-based
    - Paradigme client-serveur, HTTP
      - ➔ Programmation côté serveur et côté client
  - Expérience utilisateur proche des applications natives
    - Interface utilisateur fluide, ergonomique, dynamique
      - ➔ Traitement de l'interface côté client (JavaScript, CSS , DOM)
      - ➔ Échanges client-serveur asynchrones (AJAX)
  - Logique métier complexe
    - Outils « évolués » de modélisation, conception, développement
      - ➔ IDE, POO, UML, design patterns, méthodes agiles, XP…

➔ Où placer la logique métier ? La couche données ?

Introduction

- L'offre côté client (rappel)
  - Moteur de rendu : *cf*. cours 1
  - Moteur de scripting
    - Objectif : dynamiser les applications Web
    - Propriétés
      - Manipulation de la page
      - Échanges de données asynchrones
      - Logique applicative côté client
      - Mécanismes de bas niveau
    - Premières versions :
      - JavaScript : Netscape Navigator 2.0 (mars 1996)
      - JScript : MS Internet Explorer 3.0 (août 1996)
      - Standard commun : ECMAScript-262 (juin 1997)
        - Officiellement supporté par JavaScript, JScript et ActionScript

Introduction

- ECMAScript-262 : historique des versions
  - 1st Edition (06/1997)
  - Edition 2 (06/1998)
  - Edition 3 (12/1999)
  - Edition 4 : abandonnée (2008)
  - Edition 5 (12/2009), puis 5.1 (06/2011) → JavaScript 1.8
  - ECMAScript Harmony = ES 2015 = ES6 (06/2015) → JavaScript 2.0
  - ES7 (06/2016)
  - ES.Next… ES10
- TypeScript
  - Développé par MicroSoft en 2012
  - Utilisé dans plusieurs bibliothèques et frameworks
- Nouvelles fonctionnalités à chaque édition

Source :
https://en.wikipedia.org/wiki/ECMAScript

Introduction

- Moteurs ECMAScript / JavaScript / JScript

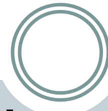
➞ 1995 : SpiderMonkey/TraceMonkey/JägerMonkey/IonMonkey (Gecko) : FireFox
  ➥ 2017 : Quantum : Servo

➞ 2000 : KJS (KHTML) : Konqueror
  ➥ 2002 : WebCore + JavaScriptCore (Webkit) : Chrome, Safari
    ➥ 2008 : SquirrelFish Extreme (= Nitro) : Chrome, Safari
      ➥ 2013 : Blink : Chrom[e|ium], Opera

➞ 2003 -> 2013 : Linear B/Futhark/Carakan (Presto) : Opera

➞ 1996 -> 2015 : JScript/Chakra (Trident) : Internet explorer
  ➥ 2015 : Chakra (JavaScript) : MicroSoft Edge

Introduction



- Caractéristiques des moteurs actuels (côté client)
  - Interprétés
    - Compilation JIT en bytecode
  - Implémentent un sous-ensemble de ES6
- JavaScript côté serveur
  - Moteur V8 (Google)
  - Implémentation : NodeJS
- Nouvelles versions (ES6, ES7, TS)
  - Objectif
    - Permettre la programmation d'applications structurées côté client
  - Nouveaux éléments syntaxiques
  - Transpilables en ES5

Introduction

- ECMAScript 6
  - Classes & modules
  - Iterators & for/of loops
  - Python-style generators & generator expressions
  - Arrow functions
  - Typed arrays
  - Collections (maps, sets and weak maps)
  - Promises
  - Reflection
  - …
- ECMASCript 7 (puis 8, 9, … , 14)
  - Exponentiation operator (**)
  - Array.prototype.includes
- … ES Next (TC39 process)

Source : https://en.wikipedia.org/wiki/ECMAScript

Introduction

- TypeScript
  - Superset de ES6
  - Orienté-objet
  - Typage statique
  - Syntaxe spécifique
  - Fonctionnalités
    - Type annotations and compile-time type checking
    - Type inference
    - Type erasure
    - Interfaces
    - Enumerated type
    - Mixin
    - Generic
    - Namespaces
    - Tuple
    - Await

Source : https://en.wikipedia.org/wiki/TypeScript

Introduction

- Fonctionnalités en lien avec la spécification HTML5
- Philosophie
  - Rapprocher les fonctionnements des navigateurs de ceux des OS
- Exemples de fonctionnalités
  - Sélecteurs CSS : accès standardisé aux contenus de la page
  - Workers : threads
  - WebSockets : streaming, server push, connexion avec d'autres clients (P2P)
  - WebStorage : émulation BD pour stockage des données de session (sessionStorage) ou d'une application (localStorage)
  - GeoLocation
  - Device APIs…
- Implémentations variables selon les moteurs/navigateurs
- Utilisation simplifiée par de nombreuses bibliothèques
- plus de détails : http://html5demos.com/

- JavaScript : généralités
  - Conçu pour être « simple »
    - Typage dynamique
      - Déclarations : var
    - Flexible & permissif
      - Conversions de types implicites
      - Utilisation de variables hors scope
      - Pas/peu d'erreurs à l'exécution
      - ➔ Résultats parfois surprenants
        https://www.toptal.com/javascript/10-most-common-javascript-mistakes

- JavaScript : généralités
  - …Pour les applications complexes
    - Déclarations : utiliser `const` et `let`
    - Gérer les erreurs avec `try` et `catch`
    - Utiliser les fonctionnalités des versions récentes
    - Utiliser le mode strict

https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Strict_mode

Rappels JavaScript / ECMAScript

- Caractéristiques du langage
  - Fonctionnel
    - Fonctions = « citoyens de première classe »
  - Orienté prototype
    - « object-based » plutôt qu'« object-oriented »
    - Typage dynamique : types associés aux instances et non aux classes
  - Événementiel
    - Mécanismes de « callback »
    - Pattern observer : eventListener
- Fonctionnalités
  - Reflection, E4X : ECMAScript for XML (ECMA-357), JSON…

Rappels JavaScript / ECMAScript

- Déclarations

```
const nombre = 1;

let dictionnaire = {name: "foo", one: 1, two: 2};

let tableau = [1, "toto", dictionnaire];

const maFonction = function(x) { return x+1; };
```

Rappels JavaScript / ECMAScript

- JavaScript Object Notation (JSON)
  - Spécification liée à ECMAScript – RFC 4627
  - Sérialisation/désérialisation de données
    - Alternative à XML
  - Implémentée par tous les navigateurs
  - Indépendante du langage de programmation
    - Natif en JS
    - Bibliothèques dans les autres langages
  - ➔ Permet les échanges de données entre serveur et client

Rappels JavaScript / ECMAScript

- JavaScript Object Notation (JSON)
  - Définit des types de données de façon simple
    - Dictionnaires (aka objets)
    - Tableaux
  - Syntaxe : des inclusions
    - d'objets sous forme d'une liste de membres
      { nommembre1 : valmembre1, nommembre2: valmembre2, … }
    - de tableaux sous forme d'une liste de valeurs
      [valeur1, valeur2, valeur3, …]
  - Primitives JS simples
    - `JSON.parse( maVariableString );`
    - `JSON.stringify ( maVariableJS );`

Rappels JavaScript / ECMAScript

- JavaScript Object Notation (JSON)
  - Exemple de fichier au format JSON :

```
{ "menu": "Fichier", "commandes":
[ { "title": "Nouveau",
"action":"CreateDoc" }, {
"title": "Ouvrir", "action":
"OpenDoc" }, { "title": "Fermer",
"action": "CloseDoc" } ] }
```

  - Equivalence en XML :
  - Source : http://www.xul.fr/ajax-format-json.html

```
<?xml version="1.0" ?>
<root>
  <menu>Fichier</menu>
  <commands>
     <item>
         <title>Nouveau</value>
         <action>CreateDoc</action>
     </item>
     <item>
         <title>Ouvrir</value>
         <action>OpenDoc</action>
     </item>
     <item>
         <title>Fermer</value>
         <action>CloseDoc</action>
     </item>
  </commands>
</root>²
```

Rappels JavaScript / ECMAScript

- Rappels de programmation fonctionnelle
  - S'appuie sur une pile d'appels imbriqués (contexte)
  - Fermeture (closure)
    - Permet de capturer l'environnement d'une fonction

```
function makeFunc() {
  var name = "Mozilla";
  function displayName() {
    alert(name);
  }
  return displayName;
}
```

```
var myFunc= makeFunc();
myFunc();
```

Sources

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures

https://developer.mozilla.org/fr/docs/JavaScript/Guide/Closures

Rappels JavaScript / ECMAScript

- Rappels de programmation fonctionnelle
  - S’appuie sur une pile d’appels imbriqués (contexte)
  - Function factory
    - Permet de passer des paramètres au moment de la création d'une fonction

```
function makeAdder(x) {
  return function(y) {
    return x + y;
  };
}
```

```
var add5 = makeAdder(5);
var add10 = makeAdder(10);

console.log(add5(2));  // 7
console.log(add10(2)); // 12
```

Sources

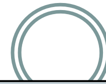
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Closures

https://developer.mozilla.org/fr/docs/JavaScript/Guide/Closures

Rappels JavaScript / ECMAScript

- Programmation orientée-prototype (jusqu'à ES5)
  - POO sans classe : on ne manipule que des objets
  - Objets représentés sous forme de dictionnaires (tableaux associatifs)
  - Propriétés
    - Pas de distinction entre les propriétés (attributs/méthodes) d'un objet
    - On peut remplacer le contenu des propriétés et en ajouter d'autres
  - Réutilisation des comportements (héritage)
    - se fait en clonant les objets existants, qui servent de prototypes
  - Sources
    - http://fr.wikipedia.org/wiki/Programmation_orient%C3%A9e_prototype
    - http://en.wikipedia.org/wiki/Prototype-based_programming

Rappels JavaScript / ECMAScript



```
// Example of true prototypal inheritance style in JavaScript.
// "ex nihilo" object creation using the literal object notation {}.

var foo = {name: "foo", one: 1, two: 2};
var bar = {three: 3};

// For the sake of simplicity, let us pretend that the following line works
regardless of the engine used:
// bar.__proto__ = foo;                     }
// bar.[[ prototype ]] = foo                }  // foo is now the prototype of bar.
bar = Object.create( foo ); // JS 1.8.5     }

// If we try to access foo's properties from bar from now on, we'll succeed.
bar.one                    // Resolves to 1.

// The child object's properties are also accessible.
bar.three                  // Resolves to 3.

// Own properties shadow prototype properties
bar.name = "bar";
foo.name                   // unaffected, resolves to "foo"
bar.name                   // Resolves to "bar"
```

Source : http://en.wikipedia.org/wiki/Prototype-based_programming

Rappels JavaScript / ECMAScript

- Comment programmer « proprement » de l'OO (ES5)
  - Plutôt « object-based » qu'« object-oriented »
  - ➔ Pour programmer en objet, il faut simuler des objets
    - Créer des constructeurs
    - Encapsuler les données (avec « `this` »)
    - Utiliser des « inner functions » à l'intérieur du constructeur
  - Exemple
    http://www.sitepoint.com/article/oriented-programming-1/

Rappels JavaScript / ECMAScript

- Programmer de l'OO (ES6)
  - Exemple
    https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Classes

```
class Rectangle {
  constructor(height, width) {
    this.height = height;
    this.width = width;
  }
  // Getter
  get area() {
    return this.calcArea();
  }
  // Method
  calcArea() {
    return this.height * this.width;
  }
}
const square = new Rectangle(10, 10);
console.log(square.area); // 100
```

Rappels JavaScript / ECMAScript

- Programmer de l'OO (ES6)
  - Mots-clés « classiques »
    - class, constructor, static, this, extends, super
  - Les propriétés sont publiques
  - Pas de classes abstraites ou d'interfaces (pour l'instant)

    → voir TypeScript
  - Liens vers les spécifications

Rappels JavaScript / ECMAScript

- Fonctions de rappel (callback)
  - Définition
    - Fonction qui est passée en paramètre à une autre fonction afin que cette dernière puisse en faire usage
  - Exemple : soient une fonction A et une fonction B
    - Lors de l'appel de A, on lui passe en paramètre la fonction B : A(B)
    - Lorsque A s'exécutera, elle pourra exécuter la fonction B
  - Intérêt : faire exécuter du code
    - Sans savoir ce qu'il va faire (défini par un autre programmeur)
    - En suivant une interface de programmation qui définit
      - Le nombre et le type des paramètres en entrée
      - Le type de la valeur en sortie
  - Source : http://www.epershand.net/developpement/algorithmie/explication-utilite-fonctions-callback

Rappels JavaScript / ECMAScript

- Fonctions de rappel (callback)
  - La fonction qui reçoit une callback en paramètre doit respecter son interface

    ```
    fonctionNormale(fonctionCallBack) {… fonctionCallback(argument); …}
    ```

  - 2 syntaxes pour le passage d'une fonction callback en argument d'une autre fonction
    - Sans paramètre : directement le nom de la fonction

      ```
      fonctionNormale(fonctionCallback);
      ```

    - Avec paramètre : encapsulation dans une fonction anonyme

      ```
      fonctionNormale(function() { fonctionCallback(arg1); });
      ```

Rappels JavaScript / ECMAScript

- Programmation événementielle
  - L'objet Event
    - Dénote un changement d'état de l'environnement
      - Peut être provoqué par l'utilisateur ou par l'application
    - Peut être intercepté à l'aide de code JavaScript
    - Possède un **flux d'événement** : propagation dans l'arbre DOM
      - Capture : du nœud Document au nœud visé par l'événement
      - Cible : sur le nœud visé
      - Bouillonnement (bubling) : remontée jusqu'au nœud document
    - Principales propriétés
      - **type** : type de l'événement ("click", "load", "mouseover"…)
      - **target** : élément cible (élément a pour un lien cliqué)
      - **stopPropagation** : arrête le flux d'un événement
      - **preventDefault** : empêche le comportement par défaut (navigation quand un lien est cliqué)

Source : http://www.alsacreations.com/article/lire/578-La-gestion-des-evenements-en-JavaScript.html

Rappels JavaScript / ECMAScript

- Programmation événementielle
  - Deux processus en parallèle
    - Principale : déroulement des traitements et association des événements à des fonctions de callback
    - Callbacks : récupèrent et traitent les événements
  - Deux syntaxes
    - DOM 0 : attributs HTML / propriétés JavaScript spécifiques
      onclick, onload…
      (http://www.w3.org/TR/html4/interact/scripts.html#h-18.2.3)
    - DOM 2 : ajout d'eventListeners en JavaScript
      monElement.addEventListener("click", maFonctionCallback, false);
      - Remarques :
        - Le troisième paramètre indique le type de propagation dans l'arbre DOM
        - Internet Explorer utilise la méthode attachEvent() au lieu de addEventListener()

Source : http://www.alsacreations.com/article/lire/578-La-gestion-des-evenements-en-JavaScript.html

Rappels JavaScript / ECMAScript

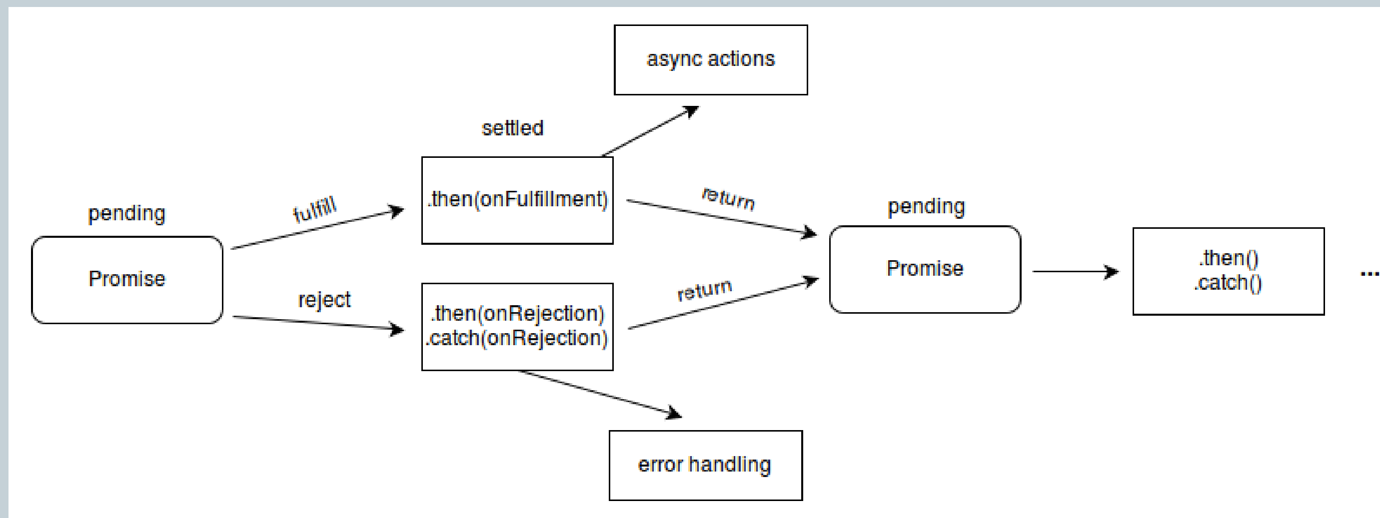
- Programmation asynchrone
  - Deux alternatives
    - Programmation événementielle : event handlers + callbacks
      - Voir slide précédent

Rappels JavaScript / ECMAScript

- Programmation asynchrone
  - Deux alternatives
    - Programmation événementielle : event handlers + callbacks
    - Promesses
      - Une promesse permet de :
        - Lancer du code asynchrone ou différé (deferred)
          - requête AJAX, entrée utilisateur, calcul complexe…
        - Réagir spécifiquement
          - en cas de succès
          - en cas d'erreur
          - à la fin de l'exécution
        - En conservant des performances acceptables
          - multithread

Rappels JavaScript / ECMAScript

- Programmation asynchrone
  - Deux alternatives
    - Programmation événementielle : event handlers + callbacks
    - Promesses
      - Workflow (rappel M1IF01)



Source : https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

Rappels JavaScript / ECMAScript

- Programmation asynchrone
  - Deux alternatives
    - Programmation événementielle : event handlers + callbacks
    - Promesses
      - Statuts : une promesse est dans l'un des états suivants
        - Pending : état initial
        - Fulfilled : l'opération s'est terminée correctement
        - Rejected : l'opération s'est terminée avec une erreur
        - Settled : l'opération s'est terminée, peu importe comment

Rappels JavaScript / ECMAScript

- Programmation asynchrone
  - Deux alternatives
    - Programmation événementielle : event handlers + callbacks
    - Promesses
      - Exemple : création simple

```
const promise = new Promise((resolve, reject) => {
    try {
        const pi = computePiIn30seconds();
        resolve(pi);
    } catch(e) {
        reject("machine too slow!");
    }
});
```

Rappels JavaScript / ECMAScript

- Programmation asynchrone
  - Deux alternatives
    - Programmation événementielle : event handlers + callbacks
    - Promesses
      - Exemple : création avec paramètres (closure)

```
function maFonctionAsynchrone(nbDecimales) {
    return promise = new Promise((resolve, reject) => {
        try {
            const pi = computePiIn30seconds(nbDecimales);
            resolve(pi);
        } catch(e) {
            reject("machine too slow!");
        }
    });
}
```

Rappels JavaScript / ECMAScript

- Programmation asynchrone
  - Deux alternatives
    - Programmation événementielle : event handlers + callbacks
    - Promesses
      - Exemple : utilisation

```
maFonctionAsynchrone(150).then(
   (result) => { console.log(result); }
).catch(
   (error) => { console.err(error); }
).finally(
   () => { alert("Résultat disponible dans la console"); }
);
```

Rappels JavaScript / ECMAScript

- Programmation asynchrone
  - Deux alternatives
    - Programmation événementielle : event handlers + callbacks
    - Promesses
      - « Re-synchroniser » une promesse
        - Mots-clés async/await

```
async function maFonctionSynchrone() {
    try {
        const result = await maFonctionAsynchrone(150);
        console.log(resultat);
    } catch(error) {
        console.err(error);
    }
    alert("Résultat disponible dans la console");
}
```

Quelques références

- Spécifications
  - En règle générale, la vérité est ici : http://www.w3.org
  - …Sauf quand elle est ailleurs :
    - http://www.ecmascript.org/
    - http://www.typescriptlang.org/
    - http://json.org/
- Documentation et tutoriels
  - https://perso.liris.cnrs.fr/romuald.thion/dokuwiki/doku.php?id=enseignement:lifap5:start
  - https://developer.mozilla.org/fr
  - http://msdn.microsoft.com/en-us/library/hbxc2t98(VS.85).aspx
  - http://www.xul.fr/ajax-format-json.html
  - http://es6-features.org/
  - http://eloquentjavascript.net/