LIFAP5 – Programmation fonctionnelle pour le WEB

CM4 – programmation fonctionnelle et asynchrone en `javascript`

Licence informatique UCBL – Printemps 2017–2018

https://perso.liris.cnrs.fr/romuald.thion/dokuwiki/doku.php?id=enseignement:lifap5:start

Plan

1. Interlude sur les objets `javascript`
2. Événements asynchrones dans le navigateur
3. Les promesses en `javascript`
4. Application chargement asynchrone avec `fetch` (et `XMLHttpRequest`)

1. Interlude sur les objets `javascript`
2. Événements asynchrones dans le navigateur
3. Les promesses en `javascript`
4. Application chargement asynchrone avec `fetch` (et `XMLHttpRequest`)

Interlude sur les objets javascript

Constructeur

On définit les propriétés et méthodes d'un objet en définissant une fonction qui sera utilisée par la suite pour construire l'objet souhaité. Source

⚠ Convention : les constructeurs commencent par une Majuscule ⚠

Exemple

```
function Car(make, model, year) {
  this.make  = make;
  this.model = model;
  this.year  = year;
}

var mycar = new Car('Eagle', 'Talon TSi', 1993);
console.log(mycar.model)         //'Talon TSi'
console.log(mycar.constructor)   //function Car()
```

Interlude sur les objets javascript

L'opérateur **new** permet de créer une instance d'un certain « type » à partir du constructeur de celui-ci. Source

On crée ainsi un nouveau contexte `this`

```
Car('Eagle', 'Talon TSi', 1993);
console.log(this);          //Window -> https://.../
console.log(this.make);     //Eagle

o = {}; Car.call(o, 'Ford', 'Fiesta', 1993);
console.log(o);
  //{ make: "Ford", model: "Fiesta", year: 1993 }

o = new Car('Fiat', '500', 1960);
console.log(o);
  //{ make: "Fiat", model: "500", year: 1960 }
```

Interlude sur les objets javascript

Exemple, avec une fonction

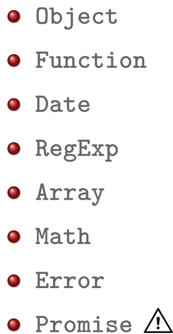
```
function displayCar() {
  var result = 'A Beautiful ' + this.year + ' ' + this
      .make + ' ' + this.model;
  console.log(result);
}

function Car(make, model, year) {
  this.make = make;
  this.model = model;
  this.year = year;
  this.displayCar = displayCar;
}

var o = new Car ('Eagle', 'Talon TSi', 1993);
o.displayCar(); //A Beautiful 1993 Eagle Talon TSi
```

Interlude sur les objets javascript

Quelques constructeurs standards



- Object
- Function
- Date
- RegExp
- Array
- Math
- Error
- Promise ⚠

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

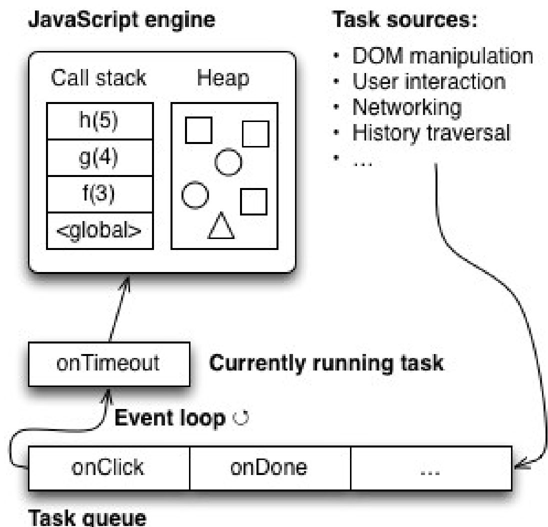
1 Interlude sur les objets javascript

2 Événements asynchrones dans le navigateur

3 Les promesses en javascript

4 Application chargement asynchrone avec fetch (et XMLHttpRequest)

La boucle d'événement du navigateur



Exemple sur le TP1

LIFAP5-TP1.html

```
<span id="output1"></span>
<p>
  <input type="text" id="input1" value="2"/>
  <input type="button" id="eval1" value="Bottles"/>
</p>
```

LIFAP5-TP1.js

```
let output1  = document.getElementById("output1");
let input1   = document.getElementById("input1");

document.getElementById("eval1").onclick =
  function() {
    output1.innerHTML = bottles(input1.value);
  };
```

Gestion d'événements asynchrones

Des actions pour plus tard : des fonctions en paramètres

- Passage de *callbacks*
  la fonction passée en paramètre de `XMLHttpRequest`
- Transformation en « promesses » (*promises*)
  pour éviter la pyramid of doom

De la maîtrise des fonctions

- Les fonctions de gestions des événements[a] prennent des fonctions en paramètres
- Qu'il faut pouvoir créer, souvent avec des fermetures

Ce style de programmation – fonctionnel – motive l'étude du λ-calcul : le proto-langage fonctionnel pur.

a. Voir aussi `Node.js`, e.g. https://nodejs.org/api/fs.html

La boucle d'événement du navigateur

Exemple avec `setTimeout`

```
setTimeout(function () {      // (A)
  console.log('Second');
}, 0);
console.log('First');         // (B)

//First
//Second
```

La fonction de la ligne `(A)` est ajoutée à la file des tâches à traiter (du navigateur), puis la fonction de la ligne `(B)` est immédiatement exécutée. Dans un tour suivant, quand l'évènement `onTimeout` arrivera, alors `(A)` sera exécutée.

La boucle d'événement du navigateur

Ajout d'événement dans la file

```
(function() {
  console.log('this is the start');

  setTimeout(function () {
    console.log('this is a msg from call back');
  }, 0);

  console.log('this is just a message');

  setTimeout(function c() {
    console.log('this is a msg from call back1');
  }, 0);

  console.log('this is the end');
})();
```

La boucle d'événement du navigateur

Affichage dans la console

```
this is the start
this is just a message
this is the end
undefined
this is a msg from call back
this is a msg from call back1
```

L'IIFE est invoquée : pendant son exécution elle ajoute deux événements dans la pile qui ne seront évalués qu'après le retour de la fonction (ici l'IIFE renvoie `undefined` car il n'y a pas de `return`). Comme la durée du `setTimeout` est nulle, ces événements sont évalués *immédiatement*.

La boucle d'événement du navigateur

Code bloquant

```
const s = new Date().getSeconds();
setTimeout(function() {
  console.log("Ran after " + (new Date().getSeconds()
     - s) + " seconds");
}, 500);
while(true) {
  if(new Date().getSeconds() - s >= 2) {
    console.log("Good, looped for 2 seconds");
    break;
  }
}
```

Affiche `Good, looped for 2 seconds` puis `Ran after 2 seconds` car pendant que la boucle `while(true)` s'exécute, aucun événement n'est défilé.

1. Interlude sur les objets javascript

2. Événements asynchrones dans le navigateur

3. Les promesses en javascript

4. Application chargement asynchrone avec fetch (et XMLHttpRequest)

Les promesses en javascript

Définition

L'interface Promise représente un intermédiaire (proxy) vers une valeur qui n'est pas nécessairement connue au moment de sa création. [...] Ainsi, des méthodes asynchrones renvoient des valeurs comme les méthodes synchrones, la seule différence est que la valeur retournée par la méthode asynchrone est une promesse (d'avoir une valeur plus tard). Source

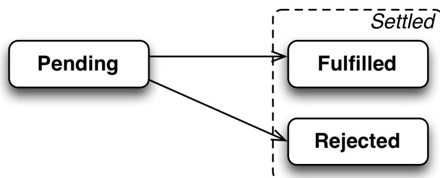
Ce qu'on veut éviter : *callback hell*

```
doSomething(function(result) {
  doSomethingElse(result, function(newResult) {
    doThirdThing(newResult, function(finalResult) {
      console.log('Result: ' + finalResult);
    }, failureCallback);
  }, failureCallback);
}, failureCallback);
```

Les promesses en `javascript`

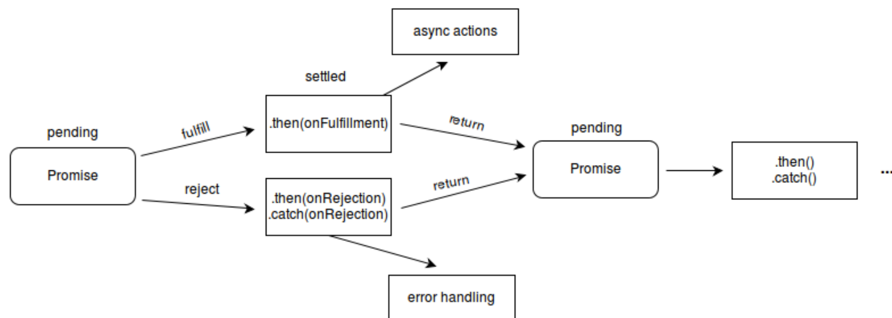
États des promesses

- *pending* (en attente) : état initial, ni remplie, ni rompue ;
- *fulfilled* (tenue) : l'opération a réussi ;
- *rejected* (rompue) : l'opération a échoué ;
- *settled* (acquittée) : la promesse est tenue ou rompue mais elle n'est plus en attente.



Les promesses en javascript

Les méthodes `then()` et `catch()` renvoient des promesses et peuvent ainsi être chaînées.



Création de promesse

Construction de Promise

```
const promise = new Promise(
  function (resolve, reject) { // (A)
      ...
      if (...) {
          resolve(value); // success
      } else {
          reject(reason); // failure
      }
  });
```

Consommation de Promise

```
promise
  .then(value => { /* fulfillment */ })
  .catch(error => { /* rejection */ });
```

Les promesses en javascript

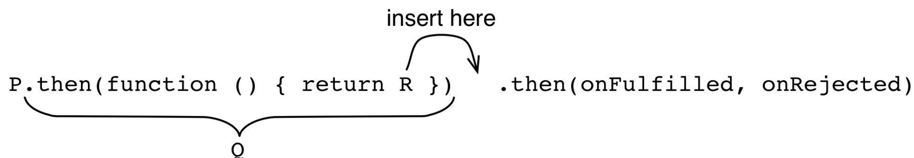
Exemple

```
var p = new Promise((resolve, reject) => {
    setTimeout(() => resolve("Success!"), 1000);
});

p.then((str) => console.log("Yay! " + str));
//Affiche "Yay! Success!" après 1 seconde
```

Les promesses permettent de simplifier l'écriture et la gestion des programmes asynchrones.

Enchaînement de promesses



`Promise.prototype.then(fn)`

- Renvoie une promesse
  - soit `fn` retourne une promesse `R` dont le résultat utilisé quand elle sera résolue par la suite,
  - soit `fn` retourne une valeur `R` qui sera transformée en une promesse immédiatement résolue,
- Dans le cas où `fn` n'est pas une fonction, alors c'est la promesse d'origine qui est renvoyée.

Enchaînement de promesses

```
PA                                  //(PA)
.then(function (value1) {
    return asyncFunc1();            //(PF1)
})                                  //(PB)
.then(function (value2) {
    return asyncFunc2()             //(PF2)
})                                  //(PC)
```

Exécution : boucle d'événements

Loop #1	*Loop #2*	*Loop #3*	*Loop #4*
(PB) créée	(PA) ⇝ filled	(PF1) ⇝ filled	(PF2) ⇝ filled
(PC) créée	asyncFunc1()	(PB) ⇝ filled	(PC) ⇝ filled
	(PF1) créée	/w val. de (PF1)	/w val. de (PF2)
		asyncFunc2()	
		(PF2) créée	

Enchaînement de promesses

```
PA                                  //(PA)
.then(function (value1) {
    return asyncFunc1();            //(PF1)
})                                  //(PB)
.then(function (value2) {
    return asyncFunc2()             //(PF2)
})                                  //(PC)
```

Exécution : boucle d'événements

Loop #1	*Loop #2*	*Loop #3*	*Loop #4*
(PB) créée	(PA) ⇝ filled	(PF1) ⇝ filled	(PF2) ⇝ filled
(PC) créée	asyncFunc1()	(PB) ⇝ filled	(PC) ⇝ filled
	(PF1) créée	/w val. de (PF1)	/w val. de (PF2)
		asyncFunc2()	
		(PF2) créée	

Enchaînement de promesses

```
PA                                  //(PA)
.then(function (value1) {
    return asyncFunc1();            //(PF1)
})                                  //(PB)
.then(function (value2) {
    return asyncFunc2()             //(PF2)
})                                  //(PC)
```

Exécution : boucle d'événements

Loop #1	*Loop #2*	*Loop #3*	*Loop #4*
(PB) créée	(PA) ⇝ filled	(PF1) ⇝ filled	(PF2) ⇝ filled
(PC) créée	asyncFunc1()	(PB) ⇝ filled	(PC) ⇝ filled
	(PF1) créée	/w val. de (PF1)	/w val. de (PF2)
		asyncFunc2()	
		(PF2) créée	

Enchaînement de promesses

```
PA                                  //(PA)
.then(function (value1) {
    return asyncFunc1();            //(PF1)
})                                  //(PB)
.then(function (value2) {
    return asyncFunc2()             //(PF2)
})                                  //(PC)
```

Exécution : boucle d'événements

Loop #1	*Loop #2*	*Loop #3*	*Loop #4*
(PB) créée	(PA) ⇝ filled	(PF1) ⇝ filled	(PF2) ⇝ filled
(PC) créée	asyncFunc1()	(PB) ⇝ filled	(PC) ⇝ filled
	(PF1) créée	/w val. de (PF1)	/w val. de (PF2)
		asyncFunc2()	
		(PF2) créée	

Enchaînement de promesses

```
PA                                  //(PA)
.then(function (value1) {
    return asyncFunc1();            //(PF1)
})                                  //(PB)
.then(function (value2) {
    return asyncFunc2()             //(PF2)
})                                  //(PC)
```

Exécution : boucle d'événements

Loop #1	*Loop #2*	*Loop #3*	*Loop #4*
(PB) créée	(PA) ⇝ filled	(PF1) ⇝ filled	(PF2) ⇝ filled
(PC) créée	asyncFunc1()	(PB) ⇝ filled	(PC) ⇝ filled
	(PF1) créée	/w val. de (PF1)	/w val. de (PF2)
		asyncFunc2()	
		(PF2) créée	

Les promesses en javascript

```
let count = 0;
function testPromise() {
  let thisCount = ++count;
  console.log(thisCount + ') Started');
  let p1 = new Promise((resolve, reject) => {
      console.log(thisCount + ') Promise');
      setTimeout(() => resolve(thisCount), Math.random
          () * 2000 + 1000);
    });
  p1.then((val) => console.log(val + ') Fulfilled'))
    .catch((res) => console.warn('Rejected'));
  console.log(thisCount + ') Promise made');
} //END testPromise()
testPromise();testPromise();testPromise();
```

Démonstration

Gestion de promesses multiples

`Promise.all()`

The `Promise.all()` method returns a single `Promise` that resolves when all of the promises in the iterable argument have resolved, or rejects with the reason of the first promise that rejects. Source

`Promise.race()`

The `Promise.race` method returns a promise that resolves or rejects as soon as one of the promises in the iterable resolves or rejects, with the value or reason from that promise. Source

Nouvelle méthode async/await

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

```
function resolveAfter2Seconds() {
  return new Promise(resolve => {
    setTimeout(() => {
      resolve('resolved');
    }, 2000);
  });
}

async function asyncCall() {
  console.log('calling');
  var result = await resolveAfter2Seconds();
  console.log(result);
  // expected output: "resolved"
}
```

1 Interlude sur les objets `javascript`

2 Événements asynchrones dans le navigateur

3 Les promesses en `javascript`

4 Application chargement asynchrone avec `fetch` (et `XMLHttpRequest`)

Introduction à XMLHttpRequest

https://developer.mozilla.org/fr/docs/Web/Guide/AJAX

Asynchronous JavaScript + XML, while not a technology in itself, is a term coined in 2005 by Jesse James Garrett, that describes a "new" approach to using a number of existing technologies together, including HTML or XHTML, Cascading Style Sheets, JavaScript, The Document Object Model, XML, XSLT, and most importantly the XMLHttpRequest object. When these technologies are combined in the Ajax model, web applications are able to make quick, incremental updates to the user interface without reloading the entire browser page. This makes the application faster and more responsive to user actions.
Although X in Ajax stands for XML, JSON is used more than XML nowadays because of its many advantages such as being lighter and a part of JavaScript. Both JSON and XML are used for packaging information in Ajax model.

Exemple XMLHttpRequest avec *callback*

```
function ajaxCB(url, callback) {
    console.log(`ajaxCB [${url}] ...`);
    let request = new XMLHttpRequest();
    request.open("GET", url);
    request.overrideMimeType("text/json");
    request.onload = function() {
      if (request.status === 200) {
        console.log("Done [" + url + "]");
        callback(request.responseText, undefined);
      } else {
        callback(undefined, Error(`Network error on [$
           {url}] : ${request.statusText}` ));
      }
    };
    request.onerror = () => callback(undefined, Error
       (`Network error on [${url}] ...`));
    request.send();
}
```

Exemple XMLHttpRequest avec Promise

```
function ajaxPromise(url) {
  return new Promise(function(resolve, reject) {
    console.log(`ajaxPromise [${url}] ...`);
    let request = new XMLHttpRequest();
    request.open("GET", url);
    request.overrideMimeType("text/json");
    request.onload = function() {
      if (request.status === 200) {
        console.log(`Done [${url}] ...`);
        resolve(request.response);
      } else
        reject(Error(`Network error on [${url}] : ${
            request.statusText}` ));
    };
    request.onerror = () => reject(Error(`Network
       error on [${url}] ...`));
    request.send();
  });
```

L'api fetch

https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API

The Fetch API provides an interface for fetching resources (including across the network). It will seem familiar to anyone who has used XMLHttpRequest, but the new API provides a more powerful and flexible feature set.

Exemple simple

```
var myImage = document.querySelector('img');

fetch('flowers.jpg').then(function(response) {
  return response.blob();
}).then(function(myBlob) {
  var objectURL = URL.createObjectURL(myBlob);
  myImage.src = objectURL;
});
```

Exemple de chargement asynchrone de données `json`

Démonstration

Références

Pour ce cours, les ressources suivantes ont été utilisées (cliquer pour suivre) :

- Le standard des promesses et une implémentation
- ECMAScript 6 promises (1/2): foundations et ECMAScript 6 promises (2/2): the API
- Promises for asynchronous programming
- We have a problem with promises