

LIFAP5 – Programmation fonctionnelle pour le WEB

CM4 – programmation fonctionnelle et asynchrone en JavaScript

Licence informatique UCBL – Printemps 2020–2021

<http://emmanuel.coquery.pages.univ-lyon1.fr/enseignement/lifap5/>



Plan

- 1 Interlude sur les objets JavaScript
- 2 Événements et tâches asynchrones dans le navigateur
- 3 Les promesses en JavaScript
- 4 Et le λ -calcul dans tout ça ?

- 1 Interlude sur les objets JavaScript
- 2 Événements et tâches asynchrones dans le navigateur
 - Principe de la programmation événementielle/asynchrone
 - La boucle d'événements et la file des tâches du navigateur
 - Exemples avec `setTimeout`
 - *The pyramid of doom*
- 3 Les promesses en JavaScript
 - Les promesses : définition
 - Enchaînement de promesses
 - Remarques diverses
- 4 Et le λ -calcul dans tout ça ?

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;  
}  
  
let 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
```

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

```
let 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() {  
  const 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;  
}
```

```
let 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>

Rappel sur la “fat arrows”, a.k.a. les lambdas

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

Une fonction sans `this` à elle

```
(param1, param2, ..., paramN) => { statements }
(param1, param2, ..., paramN) => expression
// Equivalent to: => { return expression; }
```

```
(singleParam) => { statements }
singleParam => { statements }
// Parentheses are optional when there's only one
// parameter name
```

```
() => { statements }
// Function with no parameters needs ()
```


- 1 Interlude sur les objets JavaScript
- 2 Événements et tâches asynchrones dans le navigateur
 - Principe de la programmation événementielle/asynchrone
 - La boucle d'événements et la file des tâches du navigateur
 - Exemples avec `setTimeout`
 - *The pyramid of doom*
- 3 Les promesses en JavaScript
 - Les promesses : définition
 - Enchaînement de promesses
 - Remarques diverses
- 4 Et le λ -calcul dans tout ça ?

Principe de la programmation événementielle/asynchrone

Des actions pour plus tard : des fonctions en paramètres

- Passage de *callbacks*

la fonction passée en paramètre de `fetch` ou de `setTimeout`

- Transformation en « promesses » (*promises*)

pour éviter la « pyramid of doom » ou « `callback hell` »

- Les fonctions de gestions des **événements** (les *handlers*) prennent des **fonctions en paramètres**,
- Les **fonctions asynchrones** prennent aussi des fonctions en paramètre (les *callbacks*)
- Il faut donc pouvoir **créer** des fonctions, généralement avec des **fermetures**

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

Principe de la programmation événementielle/asynchrone

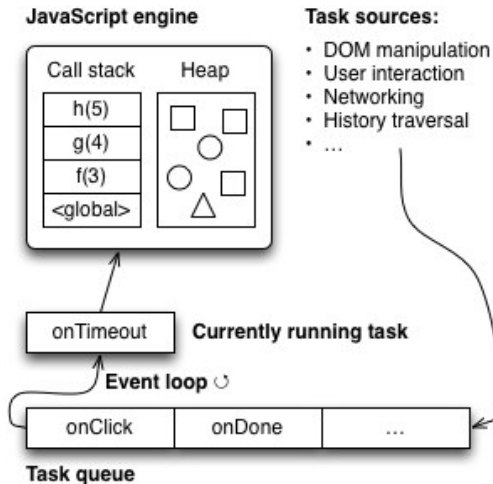
L'omniprésence des fonctions asynchrones en JavaScript

- Les gestionnaires d'événements pour [HTML](#)`Element`, `Document`, ou `Window` et la programmation événementielle plus généralement.
- Les applications AJAX (Asynchronous JavaScript + XML) `Fetch API` (et son ancêtre `XMLHttpRequest`) (XHR)
- L'essentiel de la bibliothèque standard Node.js comme `File System`, `HTTP` (programmation JavaScript côté serveur) ou `Stream`

Ces APIs imposent une façon de penser et de structurer le code

Dans ce style de programmation, où on passe en paramètre « la suite des traitements » (*Continuation Passing Style* – CPS), **on ne peut plus utiliser l'instruction `return` pour renvoyer un résultat !**

La boucle d'événements et la file des tâches du navigateur



Voir [What the heck is the event loop anyway?](#) et [In The Loop](#)

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 =
  () => output1.innerHTML = bottles(input1.value);
```

La boucle d'événements et la file des tâches du navigateur

Exemple avec `setTimeout` voir la documentation

```
setTimeout (                // (A)
  () => console.log('Second') // (B)
  , 0);                      // (A, suite)
console.log('First');       // (C)
//First
//Second
```

- 1 (A) ajoute un *handler* à l'évènement `onTimeout`
- 2 la ligne (C) est *immédiatement* exécutée.
- 3 l'évènement `onTimeout` arrive après 0 milliseconde
 - La fonction de la ligne (B) est *ajoutée* à la file des tâches
- 4 Au prochain tour de la boucle d'événements, (B) est exécutée.

La boucle d'événements et la file des tâches du navigateur

Exemple avec `setTimeout` voir la documentation

```
setTimeout (                // (A)
  () => console.log('Second') // (B)
  , 0);                      // (A, suite)
console.log('First');       // (C)
//First
//Second
```

- 1 (A) ajoute un *handler* à l'évènement `onTimeout`
- 2 la ligne (C) est **immédiatement** exécutée.
- 3 l'évènement `onTimeout` arrive après 0 milliseconde
 - La fonction de la ligne (B) est **ajoutée** à la file des tâches
- 4 Au prochain tour de la boucle d'événements, (B) est exécutée.

Exemples avec setTimeout

```
console.log('Start'); // (A)
setTimeout( // (T1)
  () => console.log('Call back #1') // (CB1)
  , 0);
console.log('Middle'); // (B)
setTimeout( // (T2)
  () => console.log('Call back #2') // (CB2)
  , 0);
console.log('End'); // (C)

//Start
//Middle
//End
//undefined
//Call back #1
//Call back #2
```


Exemples avec setTimeout

```
console.log('Start'); // (A)
setTimeout( // (T1)
  () => console.log('Call back #1') // (CB1)
  , 0);
console.log('Middle'); // (B)
setTimeout( // (T2)
  () => console.log('Call back #2') // (CB2)
  , 0);
console.log('End'); // (C)

//Start
//Middle
//End
//undefined
//Call back #1
//Call back #2
```

Exemples avec setTimeout

Attention au code bloquant !

```
const s = new Date().getSeconds();
setTimeout(function() {
  console.log("Ran after " + (new Date().getSeconds()
    - s) + " seconds");
}, 0);
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 tant que la boucle `while(true)` s'exécute, on reste dans la même tâche.

Exemples avec setTimeout

Attention au code bloquant !

```
const s = new Date().getSeconds();
setTimeout(function() {
  console.log("Ran after " + (new Date().getSeconds()
    - s) + " seconds");
}, 0);
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 tant que la boucle `while(true)` s'exécute, on reste dans la même tâche.

The pyramid of doom

```
function logger_cb(str, cb){
  return function () {
    console.log(str);
    if (typeof cb === "function")
      cb();          // /\ PAS DE RETURN /\
  }
}

setTimeout(
  logger_cb('#1s', () => {
    setTimeout(
      logger_cb('#2s', () => {
        setTimeout(
          logger_cb('#3s')
            , 1000)}})
    , 1000)}})
, 1000);
```

The pyramid of doom : transfert de résultats

```
function makeAsync(func){
  return function (arg, cb){
    function handler(){
      const res = func(arg);
      if (typeof cb === "function")
        cb(res);          // !\ PAS DE RETURN !\
    };
    setTimeout(handler, 0);
  }
}

let logger = makeAsync((str) => {
  console.log(str);
  return str.length;
});

const x1 = logger('#1s');
const x2 = logger('#2s:' + x1);
const x3 = logger('#3s:' + x2);
// !\ ERREUR CLASSIQUE !\
```

The pyramid of doom : transfert de résultats

```
function makeAsync(func){
  return function (arg, cb){
    function handler(){
      const res = func(arg);
      if (typeof cb === "function")
        cb(res);          // !\ PAS DE RETURN !\
    };
    setTimeout(handler, 0);
  }
}

let logger = makeAsync((str) => {
  console.log(str);
  return str.length;
});

logger('#1s',
  (x) =>  logger('#2s:' + x,
    (x) =>  logger('#3s:' + x, undefined)
  )
); // Traitements successifs == imbrications des fonctions
```

- 1 Interlude sur les objets JavaScript
- 2 Événements et tâches asynchrones dans le navigateur
 - Principe de la programmation événementielle/asynchrone
 - La boucle d'événements et la file des tâches du navigateur
 - Exemples avec `setTimeout`
 - *The pyramid of doom*
- 3 Les promesses en JavaScript
 - Les promesses : définition
 - Enchaînement de promesses
 - Remarques diverses
- 4 Et le λ -calcul dans tout ça ?

Les promesses : définition

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](#)

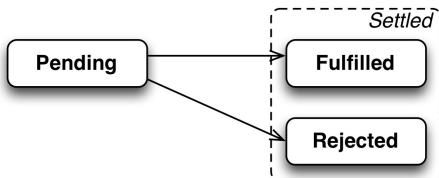
Mais ce n'est pas « magique »

- il faut *toujours* passer des fonctions en paramètre ;
- créer ces fonctions avec des fermetures ;
- *in fine*, c'est surtout **la syntaxe** change ;
- mais c'est très pratique et (désormais) standard.

Les promesses : définition

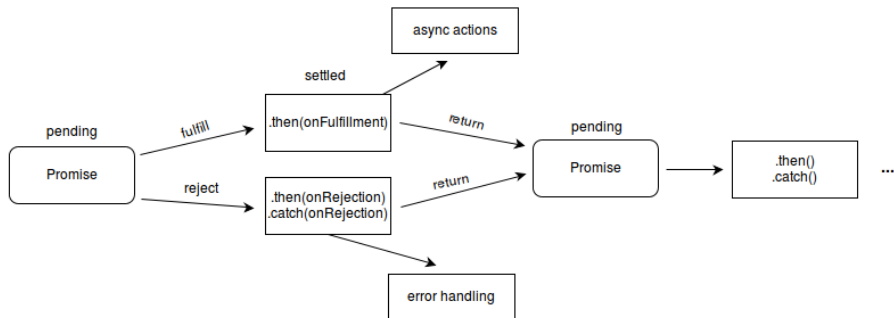
États des promesses

- *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 n'est plus en attente.



Les promesses : définition

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



C'est ce qui va permettre un enchaînement *linéaire* et non *imbrication* des traitements successifs.

Les promesses : définition

Construction de Promise

```
const promise = new Promise(  
  function (resolve, reject) {  
    if (...) {  
      resolve(value); // succès  
    } else {  
      reject(reason); // échec  
    }  
    // /\ pas d'instruction RETURN ! /\  
  });
```

Consommation de Promise

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

Les promesses : définition

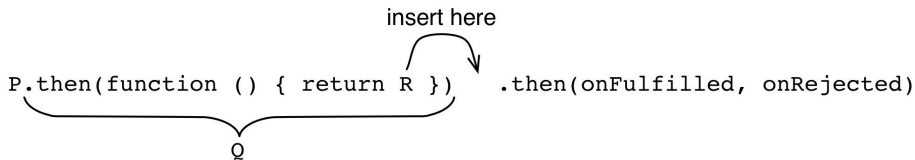
Exemple

```
let p = new Promise((resolve, reject) =>
  setTimeout(() => resolve("Success!"), 3000)
  //NB pas de "return" ici !
);

p.then((str) => console.log("Yay! " + str))
//Affiche "Yay! Success!" après 3 secondes
```

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

Enchaînement de promesses

`P.then(function () { return R })`  `.then(onFulfilled, onRejected)`

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 de la valeur qui sera renvoyée par `fn`,
- ⚠ Au cas où `fn` n'est **pas** une fonction, alors c'est *la promesse d'origine qui est renvoyée*. ⚠

Enchaînement de promesses

```

PA                                     //(PA)
  .then(
    (v1) => async1(v1)                 //(PF1)
  )                                     //(PB)
  .then(
    (v2) => async2(v2)                 //(PF2)
  )                                     //(PC)

```

Exécution : file des (micro)tâches

<i>Loop #1</i>	<i>Loop #2</i>	<i>Loop #3</i>	<i>Loop #4</i>
(PB) créée	(PA) \rightsquigarrow vA	(PF1) \rightsquigarrow vF1	(PF2) \rightsquigarrow vF2
(PC) créée	async1(vA)	(PB) \rightsquigarrow vF1	(PC) \rightsquigarrow vF2
	(PF1) appelée	async2(vF1)	
		(PF2) appelée	

Enchaînement de promesses

```

PA                                     //(PA)
  .then(                               //(PF1)
    (v1) => async1(v1)                 //(PB)
  )
  .then(                               //(PF2)
    (v2) => async2(v2)                 //(PC)
  )

```

Exécution : file des (micro)tâches

<i>Loop #1</i>	<i>Loop #2</i>	<i>Loop #3</i>	<i>Loop #4</i>
(PB) créée	(PA) \rightsquigarrow vA	(PF1) \rightsquigarrow vF1	(PF2) \rightsquigarrow vF2
(PC) créée	async1(vA)	(PB) \rightsquigarrow vF1	(PC) \rightsquigarrow vF2
	(PF1) appelée	async2(vF1)	
		(PF2) appelée	

Enchaînement de promesses

```

PA                                     //(PA)
  .then(
    (v1) => async1(v1)                //(PF1)
  )                                     //(PB)
  .then(
    (v2) => async2(v2)                //(PF2)
  )                                     //(PC)

```

Exécution : file des (micro)tâches

<i>Loop #1</i>	<i>Loop #2</i>	<i>Loop #3</i>	<i>Loop #4</i>
(PB) créée	(PA) \rightsquigarrow vA	(PF1) \rightsquigarrow vF1	(PF2) \rightsquigarrow vF2
(PC) créée	async1(vA)	(PB) \rightsquigarrow vF1	(PC) \rightsquigarrow vF2
	(PF1) appelée	async2(vF1)	
		(PF2) appelée	

Enchaînement de promesses

```

PA                                     //(PA)
  .then(
    (v1) => async1(v1)                 //(PF1)
  )                                     //(PB)
  .then(
    (v2) => async2(v2)                 //(PF2)
  )                                     //(PC)

```

Exécution : file des (micro)tâches

<i>Loop #1</i>	<i>Loop #2</i>	<i>Loop #3</i>	<i>Loop #4</i>
(PB) créée	(PA) \rightsquigarrow vA	(PF1) \rightsquigarrow vF1	(PF2) \rightsquigarrow vF2
(PC) créée	async1(vA)	(PB) \rightsquigarrow vF1	(PC) \rightsquigarrow vF2
	(PF1) appelée	async2(vF1)	
		(PF2) appelée	

Enchaînement de promesses

```

PA                                     //(PA)
  .then(                               //(PF1)
    (v1) => async1(v1)                 //(PB)
  )
  .then(                               //(PF2)
    (v2) => async2(v2)                 //(PC)
  )

```

Exécution : file des (micro)tâches

<i>Loop #1</i>	<i>Loop #2</i>	<i>Loop #3</i>	<i>Loop #4</i>
(PB) créée	(PA) \rightsquigarrow vA	(PF1) \rightsquigarrow vF1	(PF2) \rightsquigarrow vF2
(PC) créée	async1(vA)	(PB) \rightsquigarrow vF1	(PC) \rightsquigarrow vF2
	(PF1) appelée	async2(vF1)	
		(PF2) appelée	

Enchaînement de promesses

```
let log = console.log;
function make(delay, val) {
  log('make (${delay}, ${val})');
  return new Promise((resolve, reject) => {
    log('Promise ${val}');
    setTimeout(() => {
      log('Fullfilled: ${val}');
      resolve(val);
    }, delay*1000);
  });
}
let PA = make(1, "PA");
PA
  .then((v1) => {log("PF1"); return make(1, "PF1"+v1);})
  .then((v2) => {log("PF2"); return make(1, "PF2"+v2);});
```

Enchaînement de promesses, avec exceptions

```
let PA = new Promise(  
  (resolve, reject) => setTimeout(() => resolve("Error  
    1"), 0));  
PA  
  .then(v => Promise.reject(v))           //rejet  
  .then(console.log)                     //pas exécuté  
  .catch(e => console.error("Catch " + e)) //catch  
  .then(() => console.log("OK"));         //exécuté
```

Les promesses rejetées sont transmises jusqu'au prochain `Promise.prototype.catch` qui les traitera et renverra une promesse à son tour, qui pourra être suivie d'un `Promise.prototype.then`.

Remarques diverses, concurrence

Comment contrôler la réussite (et l'échec) de plusieurs promesses lancées simultanément ?

`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](#)

Remarques diverses, tâches et micro-tâches

Exemple : quel est l'ordre d'affichage ?

```
console.log("Start...");
let P1 = Promise.resolve("Resolved");
let P2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Success"), 0);
});
P2.then(console.log);
P1.then(console.log);
console.log("...End");
```

Les promesses produisent des *micro-tâches* qui sont *exécutées immédiatement* à la suite de la (macro-)tâche en cours, avant de passer à la tâche suivante dans la file des tâches. ([stackoverflow](#))

Remarques diverses, tâches et micro-tâches

Exemple : quel est l'ordre d'affichage ?

```
console.log("Start...");
let P1 = Promise.resolve("Resolved");
let P2 = new Promise((resolve, reject) => {
  setTimeout(() => resolve("Success"), 0);
});
P2.then(console.log);
P1.then(console.log);
console.log("...End");
```

Les promesses produisent des **micro-tâches** qui sont *exécutées immédiatement* à la suite de la (macro-)tâche en cours, avant de passer à la tâche suivante dans la file des tâches. ([stackoverflow](#))

Remarques diverses, async/await (hors programme)

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

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

```
// !\ DU SUCRE SYNTAXIQUE SUR LES PROMESSES !\  
async function asyncCall () {  
  console.log('calling');  
  const result = await resolveAfter2Seconds ();  
  console.log(result);  
}
```


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*

```
"use strict";
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

```
"use strict";
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

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

Exemple de chargement asynchrone de données json

Démonstration

- 1 Interlude sur les objets JavaScript
- 2 Événements et tâches asynchrones dans le navigateur
 - Principe de la programmation événementielle/asynchrone
 - La boucle d'événements et la file des tâches du navigateur
 - Exemples avec `setTimeout`
 - *The pyramid of doom*
- 3 Les promesses en JavaScript
 - Les promesses : définition
 - Enchaînement de promesses
 - Remarques diverses
- 4 Et le λ -calcul dans tout ça ?

Et le λ -calcul dans tout ça ?

Une « plomberie » bien connue : Continuation Passing Style (CPS)

- On ne renvoie en fait **jamais** de valeur. . .
- . . . mais on donne la fonction à exécuter **ensuite**
- En λ -calcul, cela permet de *fixer une stratégie d'évaluation*, d'imposer **dans quel ordre** on va évaluer une expression $(\lambda x.M) N$

On peut décrire « le style CPS » de façon systématique

- Au lieu d'un type A , on veut un type $^a (A \rightarrow R) \rightarrow R$: on attend « une suite » qui prendra une donnée de type A et rendra un R
- Toute donnée peut être transformée avec `ret` : $A \rightarrow T A$
- On peut enchaîner des étapes :
 - Si j'ai un $T A$ et une fonction $A \rightarrow T B$
 - On peut les enchaîner avec `bind` : $T A \rightarrow (A \rightarrow T B) \rightarrow T B$

a. Qu'on note $T A = (A \rightarrow R) \rightarrow R$

Et le λ -calcul dans tout ça ?Quelles définitions en λ -calcul ?

- $\text{ret} : A \rightarrow (A \rightarrow R) \rightarrow R$
 $\text{ret } a = \lambda k.k \ a$
- $\text{bind} : T A \rightarrow (A \rightarrow T B) \rightarrow T B$
 $\text{bind} : ((A \rightarrow R) \rightarrow R) \rightarrow (A \rightarrow ((B \rightarrow R) \rightarrow R)) \rightarrow ((B \rightarrow R) \rightarrow R)$
 $\text{bind } ta \ f = \lambda k.ta \ (\lambda x.(f \ x \ k))$

Pour, qu'en un sens, ça se passe bien, on veut

- $\text{bind} (\text{ret } a) \ f = f \ a$
 ret est neutre à gauche pour bind
- $\text{bind } ta \ \text{ret} = ta$
 ret est neutre à droite pour bind
- $\text{bind} (\text{bind } ta \ f) \ g = \text{bind } ta \ (\lambda x.\text{bind} (f \ x) \ g)$
 bind est associative

Et le λ -calcul dans tout ça ? Le lien en JavaScript

La traduction avec Promise

On lit $T A$ comme « *une promesse qui renvoie une valeur de type A* »

- ret c'est en fait `Promise.resolve`
- bind c'est en fait `Promise.prototype.then`

Promise respecte les spécifications

```
const inc = x => Promise.resolve(x + 1)
const dec = x => Promise.resolve(x - 1)
let P1 = Promise.resolve(1)
  .then(inc)
  .then(dec);
let P2 = Promise.resolve(1)
  .then(x => inc(x))
    .then(y => dec(y))
  );
```

Références

- (recommandé) [The JavaScript language: Promises, async/await](#)
- [ECMAScript 6 promises \(1/2\): foundations](#) et [ECMAScript 6 promises \(2/2\): the API](#)
- (vidéo) [What the heck is the event loop anyway? Philip Roberts – JSConf.EU 2014](#)
- (vidéo) [In The Loop. Jake Archibald – JSConf.Asia 2018](#) et le blogpost associé [Tasks, microtasks, queues and schedules. Jake Archibald.](#)