LIFAP5 – Programmation fonctionnelle pour le WEB

CM4 – programmation fonctionnelle et asynchrone en `javascript`

Licence informatique UCBL – Printemps 2018–2019

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

Plan

1. Interlude sur les objets `javascript`
2. Événements et tâches 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 et tâches asynchrones dans le navigateur
3. Les promesses en `javascript`
   - Les promesses : définition
   - Enchaînement de promesses
   - Et le λ-calcul dans tout ça ?
   - Les promesses : ouverture
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

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

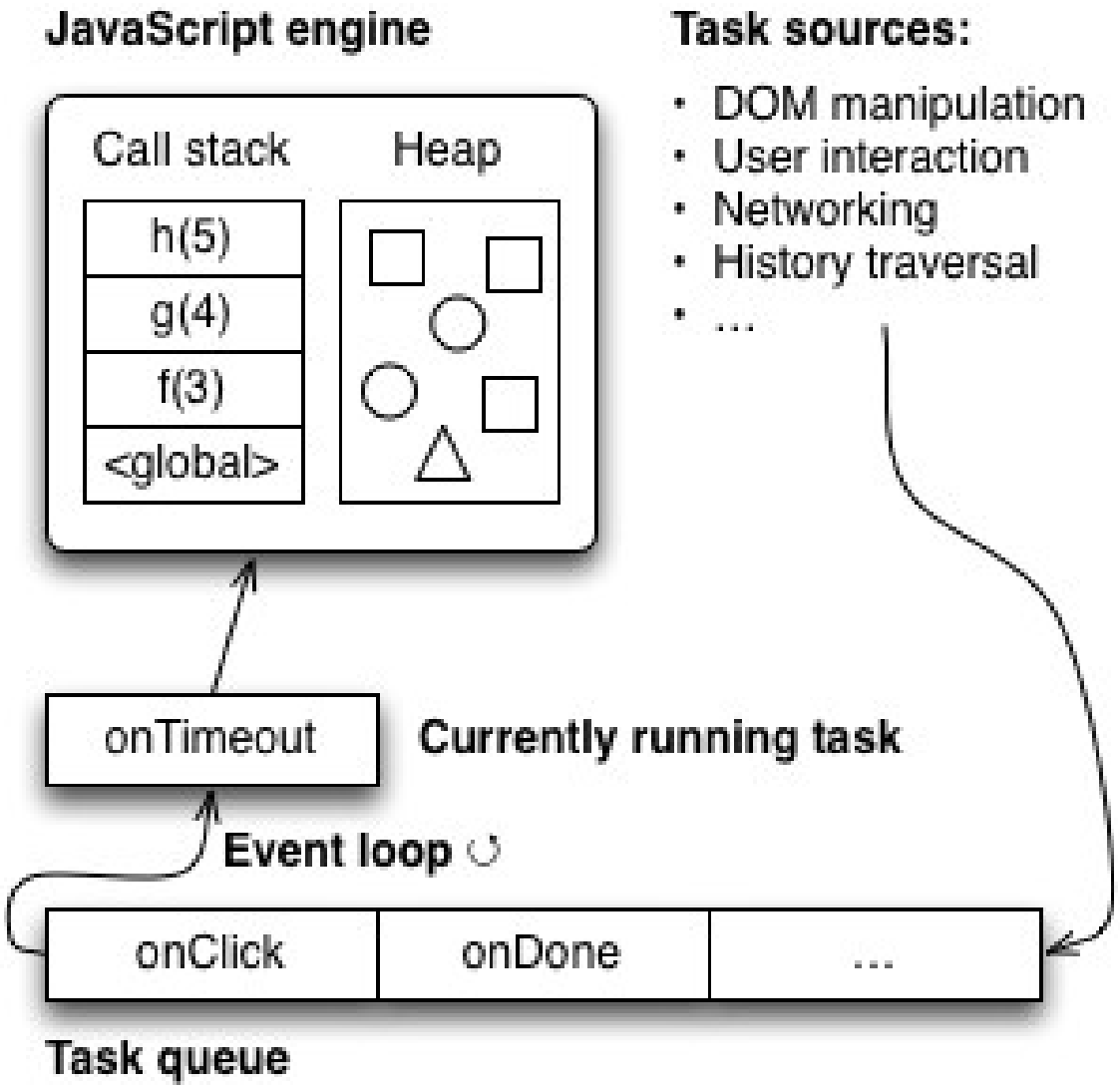
```
(param1, param2, ..., paramN) => { statements }
(param1, param2, ..., paramN) => expression
// Equivalent to: => { return expression; }

(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
3. Les promesses en `javascript`
   - Les promesses : définition
   - Enchaînement de promesses
   - Et le λ-calcul dans tout ça ?
   - Les promesses : ouverture
4. Application chargement asynchrone avec `fetch` (et `XMLHttpRequest`)

La boucle d'événements et la file des tâches



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);
```

Gestion d'événements asynchrones

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

De la maîtrise des fonctions

- Les fonctions de gestions des événements (les *handlers*) prennent des fonctions en paramètres
- Il faut donc pouvoir créer des fonctions, souvent avec des fermetures

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

La boucle d'événements 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 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 du navigateur

```
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
```

La boucle d'événements du navigateur

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.

La boucle d'événements du navigateur

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.

1. Interlude sur les objets javascript
2. Événements et tâches asynchrones dans le navigateur
3. Les promesses en javascript
   - Les promesses : définition
   - Enchaînement de promesses
   - Et le λ-calcul dans tout ça ?
   - Les promesses : ouverture
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

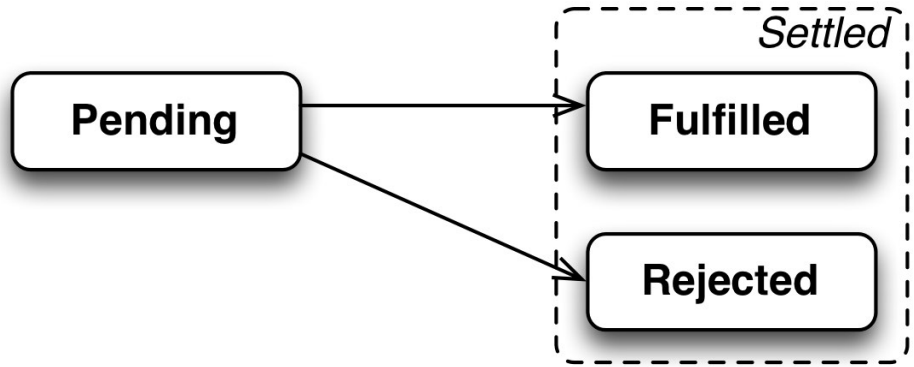
Ce qu'on veut éviter : *callback hell*

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

Les promesses en javascript

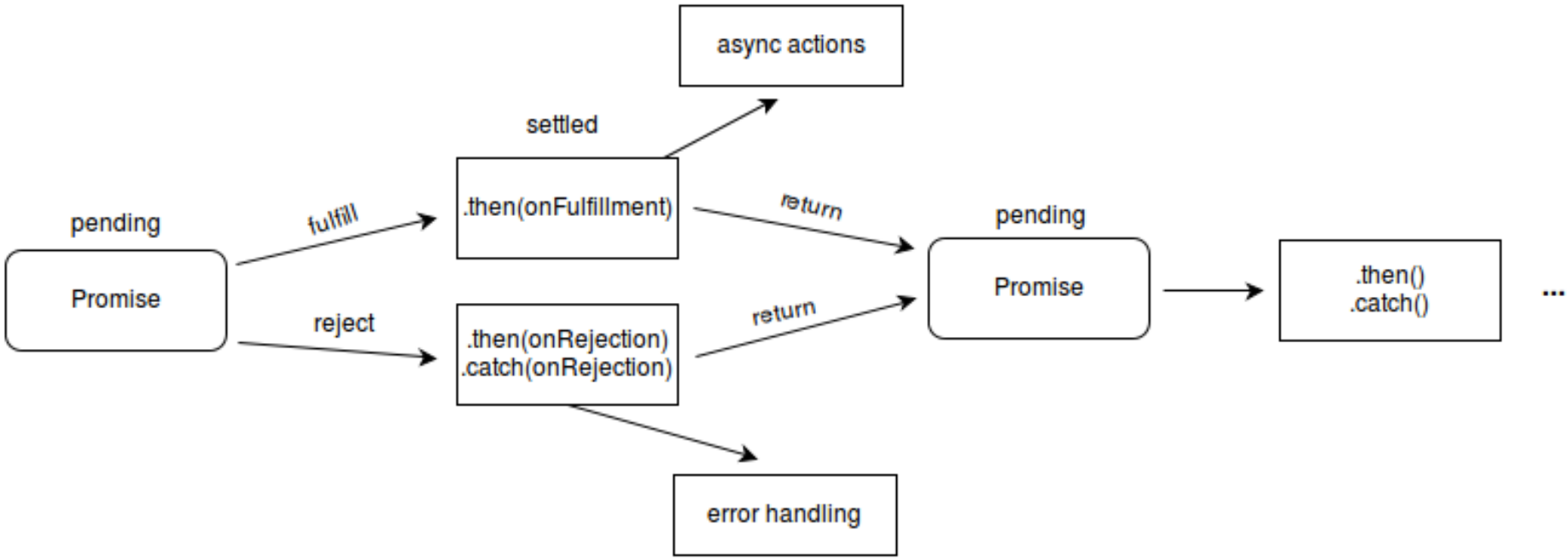
États des promesses

- *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) {
      if (...) {
          resolve(value); // success
      } else {
          reject(reason); // failure
      }
      // no return statement!
  });
```

Consommation de Promise

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

Les promesses en javascript

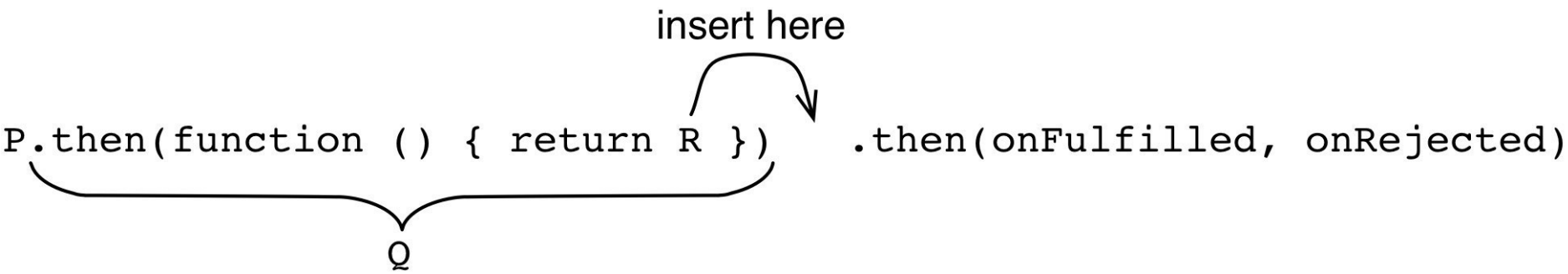
Exemple

```
var 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



`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,
- ⚠ 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

Loop #1	*Loop #2*	*Loop #3*	*Loop #4*
(PB) créée	(PA) $\rightsquigarrow$ vA	(PF1) $\rightsquigarrow$ vF1	(PF2) $\rightsquigarrow$ vF2
(PC) créée	async1(vA)	(PB) $\rightsquigarrow$ vF1	(PC) $\rightsquigarrow$ vF2
	(PF1) créée	async2(vF1)	
		(PF2) créée	

Enchaînement de promesses

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

Exécution : file des (micro)tâches

Loop #1
(PB) créée
(PC) créée

Loop #2
(PA) ⇝ vA
async1(vA)
(PF1) créée

Loop #3
(PF1) ⇝ vF1
(PB) ⇝ vF1
async2(vF1)
(PF2) créée

Loop #4
(PF2) ⇝ vF2
(PC) ⇝ vF2

Enchaînement de promesses

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

Exécution : file des (micro)tâches

Loop #1	*Loop #2*	*Loop #3*	*Loop #4*
(PB) créée	(PA) $\rightsquigarrow$ vA	(PF1) $\rightsquigarrow$ vF1	(PF2) $\rightsquigarrow$ vF2
(PC) créée	async1(vA)	(PB) $\rightsquigarrow$ vF1	(PC) $\rightsquigarrow$ vF2
	(PF1) créée	async2(vF1)	
		(PF2) créée	

Enchaînement de promesses

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

Exécution : file des (micro)tâches

Loop #1	*Loop #2*	*Loop #3*	*Loop #4*
(PB) créée	(PA) $\rightsquigarrow$ vA	(PF1) $\rightsquigarrow$ vF1	(PF2) $\rightsquigarrow$ vF2
(PC) créée	async1(vA)	(PB) $\rightsquigarrow$ vF1	(PC) $\rightsquigarrow$ vF2
	(PF1) créée	async2(vF1)	
		(PF2) créée	

Enchaînement de promesses

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

Exécution : file des (micro)tâches

Loop #1
(PB) créée
(PC) créée

Loop #2
(PA) $\rightsquigarrow$ vA
async1(vA)
(PF1) créée

Loop #3
(PF1) $\rightsquigarrow$ vF1
(PB) $\rightsquigarrow$ vF1
async2(vF1)
(PF2) créée

Loop #4
(PF2) $\rightsquigarrow$ vF2
(PC) $\rightsquigarrow$ vF2

Enchaînement de promesses

```
var 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);
  });
}
var PA = make(1,"PA");
PA
.then((v1) => {log("PF1"); return make(1,"PF1"+v1);})
.then((v2) => {log("PF2"); return make(1,"PF2"+v2);});
```

Démonstration

Enchaînement de promesses avec exceptions

```
var 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"));                  //executé
```

Démonstration

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`.

Tâches et micro-tâches

Exemple : quel est l'ordre d'affichage (Démonstration) ?

```
console.log("Start...");
var P1 = Promise.resolve("Resolved");
var 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)

Tâches et micro-tâches

Exemple : quel est l'ordre d'affichage (Démonstration) ?

```
console.log("Start...");
var P1 = Promise.resolve("Resolved");
var 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)

Les promesses en javascript : exemple final

```
let count = 0;

function testPromise() {
  let thisCount = ++count;  // /!\ closure
  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

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] $(A \to R) \to R$: on attend « une suite » qui prendra une donnée de type A et rendra un R
- Toute donnée peut être transformée avec ret : $A \to T\ A$
- On peut enchaîner des étapes :
  - Si j'ai un $T\ A$ et une fonction $A \to T\ B$
  - On peut les enchaîner avec bind : $T\ A \to (A \to T\ B) \to T\ B$

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

Et le λ-calcul dans tout ça ?

Quelles définitions en λ-calcul ?

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

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

- bind (ret a) $f = f\ a$
  ret est neutre à gauche pour bind
- bind ta ret $= ta$
  ret est neutre à droite pour bind
- bind (bind $ta\ f$) g = bind ta $(\lambda x.$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)
var P1 = Promise.resolve(1)
.then(inc)
.then(dec);
var P2 = Promise.resolve(1)
.then(x => inc(x)
           .then(y => dec(y))
     );
```

Pourquoi tout ça ?

Maintenant vous pourrez lire The Promise of a Promise ou Monads in JavaScript et remarquer que tout ça à entre 30 et 50 ans...

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 (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);
  });
}

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

1. Interlude sur les objets `javascript`
2. Événements et tâches asynchrones dans le navigateur
3. Les promesses en `javascript`
   - Les promesses : définition
   - Enchaînement de promesses
   - Et le λ-calcul dans tout ça ?
   - Les promesses : ouverture
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

- 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
- What the heck is the event loop anyway? Philip Roberts – JSConf.EU 2014
- In The Loop. Jake Archibald – JSConf.Asia 2018
- Tasks, microtasks, queues and schedules. Jake Archibald.
- Google Tech Talk January 15, 2013: Monads and Gonads. Douglas Crockford. (at 37')
- Programmer = démontrer ? La correspondance de Curry-Howard aujourd'hui. Xavier Leroy