Javascript

Notions avancées

En fait tu crois que tu sais et puis non...

Types : Tout est objet ! Et ouais...

Types Javascript

Boolean,

Number,

String,

Array,

Object,

Function (aie la tête)

Undefined

Fonctions - Notions avancées

Callback

Une fonction peut être un paramètre d’ une autre fonction. On nomme cela un “callback” ou fonction de rappel.

Note : Cette fonction de rappel peut être anonyme et définie à la volée.

```
function fin() {
  console.log("Fini !");
}

function traitement(cb) {
  console.log("Traitement...");
  cb();
}

traitement(fin);
```

Fonctions Asynchrones - Exemples

SetTimeout (function, delay);

```
function Hello() {
  console.log("Hello World");
}
setTimeout(hello, 2000);
```

SetInterval (function, delay);

```
var hndl = setTimeout(function() {
  console.log("Hello World");
}, 500);

setTimeout(function() {
  clearInterval(hndl);
}, 5000);
```

Encapsulation

Permet de définir et d'exécuter immédiatement une fonction.

Javascript n’autorise pas l’appel function(){}();

Il faut “enrober” la définition de la fonction avec ().

```
(function encapsulation() {
  // Code à exécuter immédiatement
})();
```

Closure

But : Permet de rendre privé (inaccessible) un morceau de code ou variable.

Ceci permet de retrouver un comportement “privé” fort.

```
function ajouteur(val1) {
  function ajoute(val2) {
    return val1 + val2;
  }
  return ajoute;
}
var add10 = ajouteur(10);
console.log(add10(5));
```

Call me !!

.call execute une fonction sur un objet

maFonction.call(monObjet, param1, param2)

.apply idem avec un tableau de paramètres

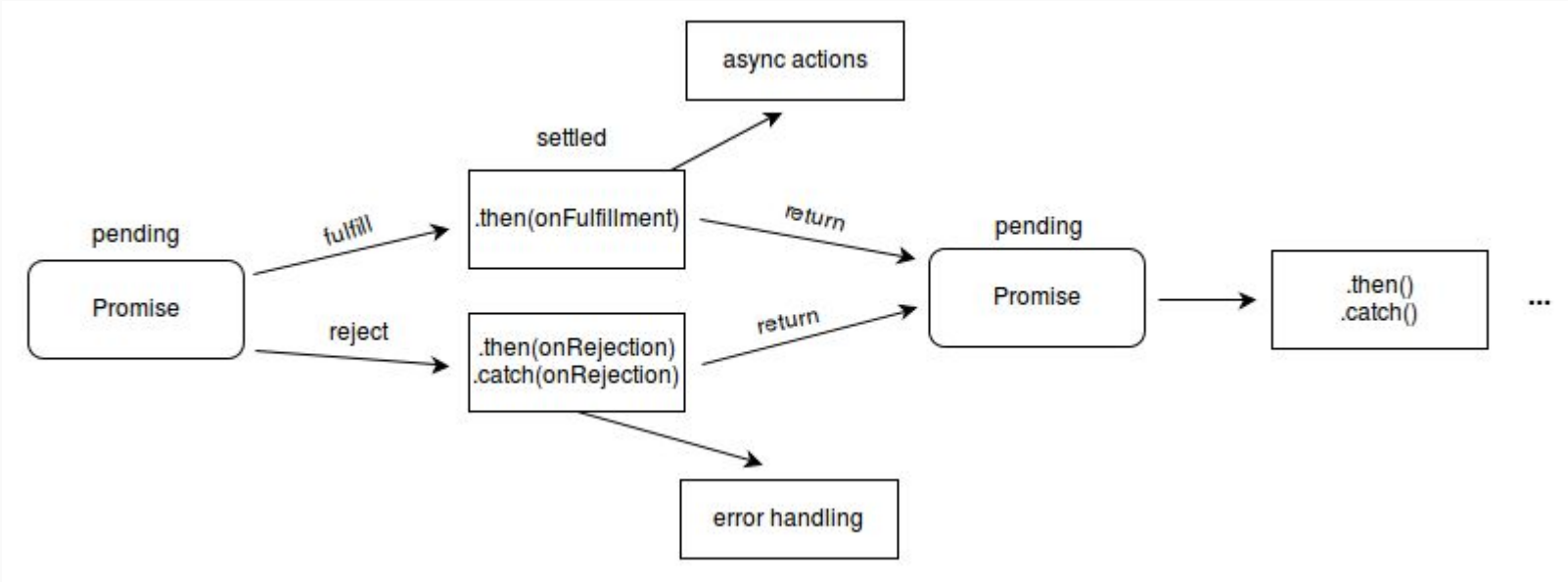
maFonction.apply(monObjet, [param1, param2]

Promesses

Promesses

- Permet d’éviter le callback-hell (imbrication de callback)
- Permettre une lecture séquentielle d’un code asynchrone
- Ne traiter les erreurs qu’une seule fois.

Promesses - Fonctionnement



Promesses - Écriture & Appel

```
var promise = new Promise(function(resolve, reject) {
  // faire un truc, peut-être asynchrone, puis...

  if (/* tout a bien marché */) {
    resolve("Ces trucs ont marché !");
  }
  else {
    reject(Error("Ça a foiré"));
  }
});
```

```
promise.then(function(result) {
  console.log(result); // "Ces trucs ont marché !"
}, function(err) {
  console.log(err); // Error: "Ça a foiré"
});
```

Promesses - Méthodes

`Promise.all(itérable)`

Renvoie une promesse qui est tenue lorsque toutes les promesses de l'argument itérables sont tenues.

`Promise.race(itérable)`

Renvoie une promesse qui est tenue ou rompue dès que l'une des promesses de l'itérable est tenue ou rompue avec la valeur ou la raison correspondante.

`Promise.reject(raison)`

Renvoie un objet `Promise` qui est rompu avec la raison donnée.

`Promise.resolve(valeur)`

Renvoie un objet `Promise` qui est tenue (résolue) avec la valeur donnée. Si la valeur possède une méthode `then`, la promesse renvoyée « suivra » cette méthode pour arriver dans son état, sinon la promesse renvoyée sera tenue avec la valeur fournie. Généralement, quand on veut savoir si une valeur est une promesse, on utilisera `Promise.resolve(valeur)` et on travaillera avec la valeur de retour engendrée.

Objets

JSON

- JSON : Javascript Object Notation
- Peut contenir des fonction (utilisation objet)
- Utilisé dans l'échange de données (flexible / peu verbeux)
  - Les fonctions ne sont pas transmises lors d'un échange de données. Cela deviens un type.

```
var json = {
  a : "Toto",
  b : 5,
  c : [4,2]
};
console.log(json.a); // affiche "toto"
```

var x = { property:value };

Objet JSON

```
function definePerson(prenom, nom, age) {
  this.prenom = prenom;
  this.nom    = nom;
  this.age    = age;
}

var myself = new definePerson("Mathieu", "LALLEMAND", 36);
console.log(myself.age); // 36
```

Objet JSON + Fonction

```
function fooObject(i) {
  this.myI = i;
  this.compte = function() {
    for (var k=0; k<this.myI; k++) {
      console.log(k);
    }
  }
}

var o = new fooObject(10);
o.compte(); // compte jusqu'à 10;
```

Objet Maléable

```
function definePerson(prenom, nom, age) {
  this.prenom = prenom;
  this.nom    = nom;
  this.age    = age;
}

var myself = new definePerson("Mathieu", "LALLEMAND", 36);
console.log(myself.age); // 36

myself.ageProchain = new function() {return this.age+1;}
console.log(myself.ageProchein()); // 37
```

Getter / Setter

Permet de forcer le comportement
lors des assignation ou lectures de valeurs
en définissant un accesseur (getter) ou un
mutateur (setter) sur un propriété

```
var o = {
  a: 7,
  get b() {
    return this.a + 1;
  },
  set c(x) {
    this.a = x / 2
  }
};

console.log(o.a); // 7
console.log(o.b); // 8
o.c = 50;
console.log(o.a); // 25
```

Prototypage Objet

Prototypage Objet ?

Dès lors qu'on aborde l'héritage, JavaScript n'utilise qu'un seul concept : les objets. Chaque objet possède un lien, interne, vers un autre objet, appelé **prototype**. Cet objet prototype possède lui aussi un prototype et ainsi de suite, jusqu'à ce que l'on aboutisse à un prototype `null`. `null`, n'a, par définition, aucun prototype et forme donc le dernier maillon de la **chaîne des prototypes**.

Souvent considéré comme l'un des points faibles de JavaScript, le modèle d'héritage des prototypes se révèle plus puissant que le modèle habituel basé sur les classes. On peut par exemple construire assez facilement un modèle classique à partir d'un modèle basé sur des prototypes, l'inverse en revanche, est beaucoup plus complexe.

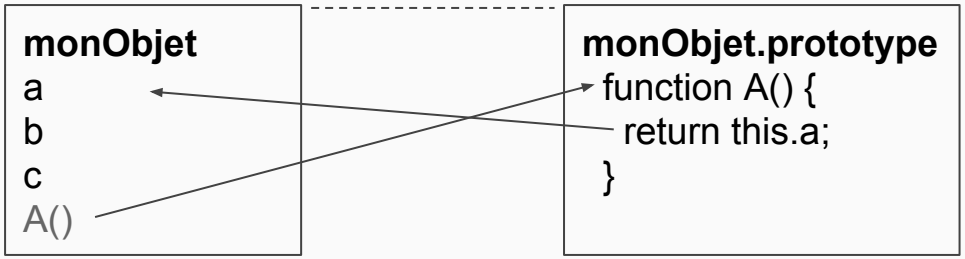
Source : Mozilla ©

Objet prototypés

- Créé par le mot clé “new”

```
function vacataire(prenom, nom, annee) {
  if (!(this instanceof vacataire)) {
    throw new Error("Cet objet doit être construit avec le mot clé 'new'");
  }
  this.prenom = prenom;
  this.nom    = nom;
  this.anne   = annee;
}
var mat = new vacataire("Mathieu", "LALLEMAND", "2015-2016");
console.log(mat.annee);
```

Prototype Schéma




```
monObjet = function(a,b,c) {
  this.a=a,
  this.b=b,
  this.c=c
}
monObjet.prototype.somme = function() {
  return this.a + this.b + this.c;
}
var test = new monObjet(1,2,3);
test.somme(); // 6
```

```
Object.defineProperty(monObjet, "somme", {
  get: function() {return return this.a + this.b + this.c; }
});
```

Utilisation du prototypage

```
function m1() { return "un"; }
function m2() { return "deux"; }
function Base() {}

Base.prototype.y = m1;
alpha = new Base();
console.log(alpha.y()); // "un"

Base .prototype.y = m2;
beta = new Base();
console.log(alpha.y(), beta.y()); // "deux", "deux";
```

Héritage sans ES6

```
function heritier(enfant, parent) {
  var tmp = function() {};               // Création d'un objet générique temporaire
  tmp.prototype = parent.prototype;      // Copie du prototype de parent sur le prototype de temporaire
  enfant.prototype = new tmp();          // Création d'un objet enfant héritant de parent mais sans lien direct
  enfant.prototype.constructor = enfant; // Recallage du constructeur de l'objet enfant.
}
```

Héritage avec ES6 (babel.js)

```
"use strict";

class Polygone {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }
}
```

```
class Carré extends Polygone {
  constructor(longueurCote) {
    super(longueurCote, longueurCote);
  }
  get aire() {
    return this.hauteur * this.largeur;
  }
  set longueurCote(nouvelleLongueur) {
    this.hauteur = nouvelleLongueur;
    this.largeur = nouvelleLongueur;
  }
}

var carré = new Carré(2);
```

Construction par vol (Call / Apply)

```
function person(prenom, nom) {
  this.prenom = prenom;
  this.nom    = nom;
}
person.prototype.greeting = function () {
    return "Mon nom est "+this.nom;
  }
}
mePerson = new person("James","BOND");
console.log( mePerson.greeting() );
```

```
function secretAgent(prenom, nom) {
  person.call(this, prenom, nom);
  //person.apply(this, [prenom, nom]);
}
secretAgent.prototype.greeting = function () {
  var old = person.prototype.greeting.call(this);
  return old + ", " + this.prenom + " " + this.nom;
}
meSecret = new secretAgent("James","BOND");
console.log( meSecret.greeting() );
```

Require

Require

```
// demo.js

module.exports = {

  function myFunc() {}

  function urFunc() {}

}
```

```
var toto = require(“demo”);

toto.myFunc();

toto.urFunc();
```

Require

```
//square.js

var foo = function(myVar) {

  return myVar*myVar;

}

module.exports=foo;
```

```
var nb4 = require('square')(2);
```