# Atelier ES6

Création d'un framework MVC en ES6 Décrire tout ce que nous allons faire Bien sûr nous ne verrons pas tout si nous n'arrivons pas au bout : pinguer moi ici : ph.charriere@gmail.com

Notre framework s'appelera "skeleton"

# ES6 - Partie 1

*Ou, tout ce dont vous avez besoin pour le 1er exercice.*

## Ce que nous attendions tous: les Classes

class Dog {  
 constructor (name="cookie") { /\* mot-clé constructor + valeurs par défaut \*/  
 this.name = name; /\* propriétés définies dans le constructeur \*/  
 }  
 wouaf () { /\* pas de mot-clé function \*/  
 console.log(this.name + ": wouaf! wouaf!");  
 }  
}  
  
let wolf = new Dog();  
wolf.wouaf()

## Bien sûr, on peut "hériter"

class Animal {  
 constructor(name) {  
 this.name = name;  
 }  
}  
  
class Dog extends Animal {  
 constructor (name="cookie") {  
 super(name) /\* on appelle le constructeur de la classe mère \*/  
 }  
 wouaf () {  
 console.log(this.name + ": wouaf! wouaf!");  
 }  
}  
  
let wolf = new Dog();  
wolf.wouaf()

## Import / Export: ou la modularisation facile

### Classe Animal dans Animal.js

class Animal {  
 constructor(name) {  
 this.name = name;  
 }  
}  
export default Animal;

### Classe Dog dans Dog.js

import Animal from './Animal'; /\* pas d'extension .js \*/  
  
class Dog extends Animal {  
 constructor (name="cookie") {  
 super(name) /\* on appelle le constructeur de la classe mère \*/  
 }  
 wouaf () {  
 console.log(this.name + ": wouaf! wouaf!");  
 }  
}  
export default Dog;

### Utilisation de Dog dans main.js

import Dog from './Dog'  
let wolf = new Dog();  
wolf.wouaf()

## Fat Arrow =>

/\* Avant \*/  
var sayHello = function(name) { return "hello " + name; }  
  
/\* Après \*/  
var sayHello = (name) => "hello " + name  
// ou var sayHello = (name) => { return "hello " + name; }  
sayHello("Bob Morane");

Remarques:

* pas "newable"
* pas d'objet arguments, à la place : "rest parameters"

var sayHello = (...people) => people.forEach((somebody) => console.log("Hello", somebody));  
  
sayHello("Bob Morane", "John Doe", "Jane Doe");

### Fat Arrow + Lexical this binding

La valeur de this est déterminée par l'endroit où se trouve la "Arrow function"

/\* Avant \*/  
function Animal(friends) {  
 this.friends = friends;  
 this.hello = function(friend) {  
 console.log("hello " + friend);  
 }  
 this.helloAll = function() {  
 this.friends.forEach(function(friend) {  
 this.hello(friend); /\* error \*/  
 });  
 }  
}  
  
var wolf = new Animal(["rox", "rookie"]);  
wolf.helloAll();  
  
  
/\* Correction : bind \*/  
function Animal(friends) {  
 this.friends = friends;  
 this.hello = function(friend) {  
 console.log("hello " + friend);  
 }  
 this.helloAll = function() {  
 this.friends.forEach(function(friend) {  
 this.hello(friend);  
 }.bind(this)); // ou var that = this  
 }  
}  
  
var wolf = new Animal(["rox", "rookie"]);  
wolf.helloAll();  
  
  
/\* Après \*/  
class Animal {  
 constructor (friends=[]) {  
 this.friends = friends;  
 }  
 hello(friend) { console.log("hello " + friend); }  
 helloAll() {  
 this.friends.forEach((friend) => this.hello(friend));  
 }  
}

## let versus var

### var

var bob = {  
 firstName:"Bob", lastName:"Morane"  
}  
  
var bob = { foo:"foo"}

### let : c'est plus propre

let bob = {  
 firstName:"Bob", lastName:"Morane"  
}  
  
let bob = { foo:"foo"} /\* Duplicate declaration, bob \*/

## Exécuter du code ES6: Préparation de index.html

<script src="node\_modules/traceur/bin/traceur.js"></script>  
  
<script>  
 traceur.options.experimental = true;  
</script>  
  
<script>  
 System.import('js/main').catch(function (e) {console.error(e);});  
</script>

**PS**: il y a d'autres méthodes, mais pour apprendre, c'est la plus simple

# ES6 - Exercice 1: Models

## Le minimum pour commencer

### Côté serveur

* on fonctionne en "mode http", il nous faut donc un serveur http, j'ai pris node + express
* j'utilise NeDb pour "simuler" une base de données, c'est un MongoDb-like avec du fichier plat
* j'ai besoin de npm

#### package.json

{  
 "name": "es6",  
 "description": "es6",  
 "version": "0.0.0",  
 "dependencies": {  
 "body-parser": "1.0.2",  
 "express": "4.1.x",  
 "nedb": "0.10.5"  
 }  
}

Et je vous ai préparé un ficihier app.js avec les API REST qui vont bien pour faire du CRUD

### Côté client

Notre webapp sera dans le répertoire /public

Elle sera composée de:

* index.html dans /public, déjà préparée pour vous
* main.js dans public/js qui contiendra le code principal de notre application (essentiellement des tests qui sont déjà **codés**)
* Model.js dans public/js/skeleton : **vous allez devoir le coder** pour que les tests fonctionnent
* Human.js dans public/js/app/models : **vous allez devoir le coder** pour que les tests fonctionnent

Là aussi j'ai besoin de npm pour installer les dépendances :

* **Traceur**, qui va donc nous permettre d'exécuter du code ES6 (nous allons faire de la "transpilation online")
* **QUnit**, pour faire nos tests unitaires

#### package.json

{  
 "name": "es6",  
 "description": "es6 formation",  
 "author": "@k33g\_org",  
 "license": "MIT",  
 "dependencies": {  
 "traceur": "0.0.65"  
 },  
 "devDependencies": {  
 "qunitjs": "^1.15.0"  
 }  
}

## Exercice

Créer nos 1ers modèles

* public/js/skeleton/Model.js
* public/js/app/models/Human.js

**Remarque**: les spécifications sont décrites dans les fichiers

cd 01-models  
node app.js  
http://localhost:3000

# Correction

## Model.js

/\*--- model ---\*/  
  
/\* === Spécifications ===  
une classe Model  
  
Paramètres du constructeur:  
  
- fields, valeur par défaut {}, contiendra les "champs" du model, ex: {firstName:"Bob", lastName:"Morane"}  
- observers, valeur par défaut []  
  
Propriétés:  
  
- fields : initialisé par le paramètre correspondant du constructeur  
- observers : initialisé par le paramètre correspondant du constructeur  
  
Méthodes:  
  
- addObserver (observer)  
- notifyObservers (context)  
- get (fieldName), va lire la valeur d'un champ dans fields  
- set (fieldName, value), va modifier la valeur d'un champ dans fields  
- toString (), retourne une représentation json de fields  
  
un observer est juste un objet avec une méthode update  
donc notifyObservers execute la méthode update de tous les observers avec context en paramètre  
  
\*/  
  
class Model {  
 constructor (fields={}, observers=[]) {  
 this.fields = fields;  
 this.observers = observers;  
 }  
  
 addObserver (observer) {  
 this.observers.push(observer);  
 }  
  
 notifyObservers (context) {  
 this.observers.forEach((observer) => {  
 observer.update(context)  
 })  
 }  
  
 get (fieldName) {  
 return this.fields[fieldName];  
 }  
  
 set (fieldName, value) {  
 this.fields[fieldName] = value;  
 return this;  
 }  
  
 toString () {  
 return JSON.stringify(this.fields)  
 }  
  
}  
  
export default Model;

## Human.js

/\* === Spécifications ===  
 une classe Human qui hérite de Model  
   
 Paramètres du constructeur:  
  
 - fields, valeur par défaut {firstName:"John", lastName:"Doe"},   
   
 Propriétés:  
   
 - initialiser la propriété fields de la classe mère avec le paramètre du constructeur  
   
 Méthodes:  
  
 - sans objet  
  
 \*/  
import Model from '../../skeleton/Model';  
  
class Human extends Model {  
 constructor (fields = {firstName:"John", lastName:"Doe"}) {  
 //superclass's constructor invocation  
 super(fields);  
  
 }  
}  
  
export default Human;

# ES6 - Partie 2

*RAS: même chose que pour la partie 1.*

# ES6 - Exercice 2: Collections

## Exercice

Créer notre 1ère collection :

* public/js/skeleton/Collection.js
* public/js/app/models/Humans.js

**Remarque**: les spécifications sont décrites dans les fichiers

cd 02-collections  
node app.js  
http://localhost:3000

# Correction

## Collection.js

/\*--- collection ---\*/  
  
/\* === Spécifications ===  
 une classe Collection  
  
 Paramètres du constructeur:  
  
 - model : ce sera le type de la collection (une classe qui héritera de Model), pas de valeur par défaut  
 - models : un tableau, contiendra les instances de modèles, valeur par défaut : []  
 - observers, valeur par défaut []  
  
 Propriétés:  
  
 - model : initialisé par le paramètre correspondant du constructeur  
 - models : initialisé par le paramètre correspondant du constructeur  
 - observers : initialisé par le paramètre correspondant du constructeur  
  
 Méthodes:  
  
 - addObserver (observer)  
 - notifyObservers (context)  
 - toString (), retourne une représentation json de la propriété models  
 - add (model), ajoute un model à models et notifie les observers avec un "contexte" égal à {event: "add", model: model}  
 - each (callbck) : parcourir les models et exécuter callbck pour chacun (et passer le modèle en paramètre à callbck)  
 - filter (callbck) : retourner un tableau de modèle filtré selon callbck  
 - size () : retourner le nombre de modèles dans la collection  
   
 un observer est juste un objet avec une méthode update  
 donc notifyObservers execute la méthode update de tous les observers avec context en paramètre  
  
 \*/  
  
  
class Collection {  
 constructor (model, models = [], observers = []) {  
 this.model = model;  
 this.models = models;  
 this.observers = observers;  
 }  
  
 toString () {  
 return JSON.stringify(this.models);  
 }  
  
 addObserver (observer) {  
 this.observers.push(observer);  
 }  
  
 notifyObservers (context) {  
 this.observers.forEach((observer) => {  
 observer.update(context)  
 })  
 }  
  
 add (model) {  
 this.models.push(model);  
 this.notifyObservers({event: "add", model: model});  
 return this;  
 }  
  
 each (callbck) {  
 this.models.forEach(callbck)  
 }  
  
 filter (callbck) {  
 return this.models.filter(callbck)  
 }  
  
 size () { return this.models.length; }  
  
}  
  
export default Collection;

## Humans.js

/\* === Spécifications ===  
une classe Humans qui hérite de Collection  
  
Paramètres du constructeur:  
  
- humans, un tableau de modèles  
  
Propriétés:  
  
- initialiser la propriété model de la classe mère avec le type Human  
- initialiser la propriété models de la classe mère avec le paramètre humans du constructeur  
  
Méthodes:  
  
- sans objet  
\*/  
  
import Collection from '../../skeleton/Collection';  
import Human from './Human';  
  
class Humans extends Collection{  
  
 constructor (humans) {  
 super(Human,humans);  
 }  
}  
  
export default Humans;

# ES6 - Partie 3

*RAS: même chose que pour la partie 1.* #ES6 - Exercice 3: Observables

## Exercice

La collection et le modèle sont tous les 2 "observables", factorisez ce comportement :

* public/js/skeleton/Collection.js
* public/js/skeleton/Model.js
* public/js/skeleton/Observable.js

**Remarque**: les spécifications sont décrites dans les fichiers

cd 03-observables  
node app.js  
http://localhost:3000

# Correction

## Observable.js

class Observable {  
  
 constructor (observers=[]) {  
 this.observers = observers;  
 }  
  
 addObserver (observer) {  
 this.observers.push(observer);  
 }  
  
 notifyObservers (context) {  
 this.observers.forEach((observer) => {  
 observer.update(context)  
 })  
 }  
}  
  
export default Observable;

## Model.js

import Observable from './Observable';  
  
class Model extends Observable {  
  
 constructor (fields={}, observers=[]) {  
 this.fields = fields;  
 super(observers);  
 }  
  
 get (fieldName) {  
 return this.fields[fieldName];  
 }  
  
 set (fieldName, value) {  
 this.fields[fieldName] = value;  
 return this;  
 }  
  
 toString () {  
 return JSON.stringify(this.fields)  
 }  
  
}  
  
export default Model;

## Collection.js

import Observable from './Observable';  
  
class Collection extends Observable {  
 constructor (model, models = [], observers = []) {  
 this.model = model;  
 this.models = models;  
 super(observers);  
 }  
  
 toString () {  
 return JSON.stringify(this.models);  
 }  
  
  
 add (model) {  
 this.models.push(model);  
 this.notifyObservers({event: "add", model: model});  
 return this;  
 }  
  
 each (callbck) {  
 this.models.forEach(callbck)  
 }  
  
 filter (callbck) {  
 return this.models.filter(callbck)  
 }  
  
 size () { return this.models.length; }  
  
}  
  
export default Collection;

# ES6 - Partie 4

## Interpolations de chaînes

### Template strings

let firstName = "Bob", lastName = "Morane";  
console.log(`Hello I'm ${firstName} ${lastName}`); // Hello I'm Bob Morane

### Multiline strings

let firstName = "Bob", lastName = "Morane";  
console.log(`  
Hello I'm   
 ${firstName}   
 ${lastName}  
`  
);   
/\*  
Hello I'm   
 Bob   
 Morane  
\*/

### Tagged template strings

let upper = (strings, ...values) => {  
 console.log(strings); // ["Hello I'm ", " ", "", raw: Array[3]]  
 console.log(values); // ["Bob", "Morane"]   
 let result = "";  
 for(var i = 0; i < strings.length; i++) {  
 result = result + strings[i];  
 if (i < values.length) {  
 result = result + values[i];  
 }  
 }  
 return result.toUpperCase();  
}  
let firstName = "Bob", lastName = "Morane";  
console.log(upper `Hello I'm ${firstName} ${lastName}`)  
/\*  
HELLO I'M BOB MORANE  
\*/

## Mixin d'objets

let tonyStark = {  
 firstName:"Tony", lastName:"Stark"  
};  
let armorAbilities = {  
 fly:() => console.log("I'm flying")  
};  
Object.assign(tonyStark, armorAbilities);  
  
tonyStark.fly(); // I'm flying

## Array.from

Exemple:

Avant pour parcourir comme un tableau le résultat d'un document.querySelectorAll, il fallait d'abord transformer ce résultat en tableau :

var items = [].slice.apply(document.querySelectorAll("li"));  
// ou var items = Array.prototype.slice.apply(document.querySelectorAll("li"));  
items.forEach(function(item) { ... });

Maintenant :

Array.from(document.querySelectorAll("li")).forEach((item) => {})

# ES6 - Exercice 4: Views

## Remarque : selector.js

vous avez un mini sélecteur "à la jquery" ici : public/js/skeleton/selector.js

let q = (selector) => {  
  
 var nodes = Array.from(document.querySelectorAll(selector));  
  
 if (nodes.length == 1) {  
 nodes = nodes[0];  
 } else {  
  
 Object.assign(nodes, {  
 first () { return this[0]; },  
 last () { return this[this.length-1]; }  
 });  
 }  
  
 nodes.find = q;  
  
 return nodes;  
}  
  
export default q;

qui s'utilise comme ceci:

import $q from 'js/skeleton/selector';  
  
$q("h1");  
  
$q("form").find("button").innerHTML = "HELLO";  
  
/\* etc. ... \*/

## Exercice

* complétez public/js/skeleton/View.js pour que public/js/app/views/Title.js s'affiche
* complétez public/js/skeleton/View.js pour que public/js/app/views/HumanForm.js s'affiche
* complétez public/js/app/views/HumansList.js pour afficher la liste des humains
* testez public/js/app/views/HumanForm.js pour vérifier que la liste des humains se met à jour

**Remarque**: les spécifications sont décrites dans les fichiers

cd 04-views  
node app.js  
http://localhost:3000

# Correction

## View.js

class View {  
  
 constructor (options={}) {  
 /\*  
 exemple:  
 options: {model,collection,element}  
 \*/  
 Object.assign(this, options);  
 }  
  
 html (code) {  
 this.element.innerHTML = code;  
 }  
  
 // transformer la vue en observer  
 listen (observable, callback) {  
 observable.addObserver(this);  
 // la vue devient un observer  
 this.update = callback;  
 }  
  
}  
  
export default View;

## Title.js

import View from '../../skeleton/View';  
import $q from '../../skeleton/selector';  
  
class Title extends View {  
  
 constructor (title) {  
  
 super({  
 element: $q("#my-title")  
 });  
  
 this.title = title;  
  
 // afficher le titre  
 this.render();  
 }  
  
 template (title) {  
 return `<h1 style="color:green;">${title}</h1>`;  
 }  
  
 render () {  
 this.html(this.template(this.title));  
 }  
  
}  
  
export default Title;

## HumansList.js

import View from '../../skeleton/View';  
import $q from '../../skeleton/selector';  
  
class HumansList extends View {  
  
 constructor (humansCollection) {  
  
 super({  
 collection: humansCollection,  
 element: $q("#humans-list")  
 });  
  
 // afficher la liste lorsque la collection est "chargée"  
 // ou lorsque l'on ajoute un modèle  
 this.listen(humansCollection, (context) => {  
 if (context.event == "loaded" || context.event == "add") {  
 this.render();  
 }  
 })  
  
 }  
  
 template (humans) {  
 return `  
 <ul>${  
 humans.models.map(  
 (human) => `<li>${human.get("firstName")}, ${human.get("lastName")}</li>`  
 ).join("")  
 }</ul>  
 `;  
 }  
  
 render () {  
 this.html(this.template(this.collection));  
 }  
  
}  
  
export default HumansList;

# ES6 - Partie 5

## Les Promises!

let doSomeThing = new Promise((resolve, reject) => {  
  
 // faites quelque chose (asynchrone)  
  
 let allisfine = true; // essayez avec false  
   
 if (allisfine) {  
 resolve("Hello World!");  
 }  
 else {  
 reject(Error("Ouch"));  
 }  
});  
  
doSomeThing  
 .then((data) => { console.log(data); })  
 .catch((err) => { console.log(err); });

**Voir cet article**: <http://www.html5rocks.com/en/tutorials/es6/promises/>

# ES6 - Exercice 5: Request

## Remarque : Request.js

vous avez une classe pour faire des requêtes ajax : public/js/skeleton/Request.js

qui s'utilise comme ceci:

let request = new Request("/about").get()  
 .then((data) => {  
 console.log("data", data)  
 })  
 .catch((error) => {  
 console.log("error", error)  
 })  
   
new Request("/humans").post({firstName:"JOHN", lastName:"DOE"})  
 .then((data) => {  
 console.log("data", data)  
 })  
 .catch((error) => {  
 console.log("error", error)  
 })

## Exercice

* complétez public/js/skeleton/Model.js en complétant la méthode save()
* complétez public/js/app/views/HumansList.js pour afficher la liste des humains quand la collection est modifiée
* testez public/js/app/views/HumanForm.js pour vérifier que la liste des humains se met à jour

**Remarque**: les spécifications sont décrites dans les fichiers

cd 05-sync  
node app.js  
http://localhost:3000

# Correction

## Model.js

import Observable from './Observable';  
import Request from './Request';  
  
class Model extends Observable {  
  
 // ajout de l'url  
 constructor (fields={}, url="/", observers=[]) {  
  
 this.fields = fields;  
 this.url = url;  
  
 super(observers);  
 }  
  
 get (fieldName) {  
 return this.fields[fieldName];  
 }  
  
 set (fieldName, value) {  
 this.fields[fieldName] = value;  
 return this;  
 }  
  
 toString () {  
 return JSON.stringify(this.fields)  
 }  
  
 /\*--- sync ---\*/  
  
 id() { return this.get("\_id");}  
  
 save () {  
  
 if (this.id() == undefined) {  
 // create (insert)  
 return new Request(this.url).post(this.fields)  
 .then((data) => {  
 this.fields = data;  
 this.notifyObservers({event: "create", model: this});  
 return data;  
 })  
 .catch((error) => error)  
 } else {  
 // update  
 return new Request(`${this.url}/${this.id()}`).put(this.fields)  
 .then((data) => {  
 this.fields = data;  
 this.notifyObservers({event: "update", model: this});  
 return data;  
 })  
 .catch((error) => error)  
 }  
  
 }  
  
 fetch (id) {  
  
 if (id == undefined) {  
 return new Request(`${this.url}/${this.id()}`).get()  
 .then((data) => {  
 this.fields = data;  
 this.notifyObservers({event: "fetch", model: this});  
 return data;  
 })  
 .catch((error) => error)  
 } else {  
 return new Request(`${this.url}/${id}`).get()  
 .then((data) => {  
 this.fields = data;  
 this.notifyObservers({event: "fetch", model: this});  
 return data;  
 })  
 .catch((error) => error)  
 }  
  
 }  
  
 delete (id) {  
 if (id == undefined) {  
 return new Request(`${this.url}/${this.id()}`).delete()  
 .then((data) => {  
 this.fields = data;  
 this.notifyObservers({event: "delete", model: this});  
 return data;  
 })  
 .catch((error) => error)  
 } else {  
 return new Request(`${this.url}/${id}`).delete()  
 .then((data) => {  
 this.fields = data;  
 this.notifyObservers({event: "delete", model: this});  
 return data;  
 })  
 .catch((error) => error)  
 }  
 }  
  
  
}  
  
export default Model;

## Collection.js

import Observable from './Observable';  
import Request from './Request';  
  
class Collection extends Observable {  
 // ajout de l'url  
 constructor (model, url="/", models = [], observers = []) {  
 this.model = model;  
 this.models = models;  
 this.url = url;  
 super(observers);  
 }  
  
 toString () {  
 return JSON.stringify(this.models);  
 }  
  
  
 add (model) {  
 this.models.push(model);  
 this.notifyObservers({event: "add", model: model});  
 return this;  
 }  
  
 each (callbck) {  
 this.models.forEach(callbck)  
 }  
  
 filter (callbck) {  
 return this.models.filter(callbck)  
 }  
  
 size () { return this.models.length; }  
  
  
 /\*--- sync ---\*/  
  
 fetch () {  
 return new Request(this.url).get().then((models) => {  
 this.models = []; /\* empty list \*/  
  
 models.forEach((fields) => {  
 this.add(new this.model(fields));  
 });  
  
 this.notifyObservers({event: "fetch", models:models});  
 return models;  
 })  
 .catch((error) => error)  
  
 }  
  
  
}  
  
export default Collection;

# ES6 - Partie 6

## Map

let map = new Map();  
map.set("one",{firstName:"John", lastName:"Doe"});  
map.set("two",{firstName:"Jane", lastName:"Doe"});  
  
console.log(map.has("one")); // true  
console.log(map.get("one")); // Object {firstName: "John", lastName: "Doe"}  
console.log(map.size); // 2  
  
for (let key of map.keys()) {  
 console.log("Key: %s", key);  
}  
/\*  
 Key: one  
 Key: two  
\*/  
  
for (let value of map.values()) {  
 console.log("Value: %s %s", value.firstName, value.lastName);  
}  
/\*  
 Value: John Doe  
 Value: Jane Doe  
\*/  
  
for (let item of map) {  
 console.log("Key: %s, Value: %s", item[0], item[1].firstName, item[1].lastName);  
}  
/\*  
 Key: one, Value: John Doe  
 Key: two, Value: Jane Doe  
\*/  
  
/\* Et aussi : \*/  
  
let myOtherMap = new Map([  
 ["one",{firstName:"John", lastName:"Doe"}],  
 ["two",{firstName:"Jane", lastName:"Doe"}],  
 ["three",{firstName:"Bob", lastName:"Morane"}]  
]);  
  
myOtherMap.delete("three")  
  
myOtherMap.forEach((item)=>{  
 console.log(item)  
})  
/\*  
 Object {firstName: "John", lastName: "Doe"}  
 Object {firstName: "Jane", lastName: "Doe"}  
\*/

## Remarque

Vos classes peuvent hériter des types javascript (mais pas tout le temps :( )

# ES6 - Exercice 6: Router

Le routeur a plusieurs routes

Routes : une url danssle navigateur que l'on associe à un traitement (ex afficher une vue) Du coup les fonctionnalités de ma webapp sont bookmarkables

Ma classe Router a une propriété routes qui est une Map

Donc si je veux créer un routeur et rajouter des routes :

let router = new Router();  
  
 router.routes.set("humans", (args) => {  
 // faire quelque chose  
 });  
  
 router.routes.set("animals", (args) => {  
 // faire quelque chose  
 });  
  
 router.routes.set("/", (args) => {  
 // faire quelque chose  
 });

* Ajouter une méthode add(uri, action) à la classe Router
* Ajouter une méthode match(uri) qui servira à vérifier les urls saisies dans le navigateur ou les liens cliqués et déclencher les méthodes associées

uri peut prendre différents types de valeurs :

* #/humans
* #/humans/1234
* etc...

Donc match(uri) va "retraiter" uri : enlever le #/ et:

* #/humans devient humans
* #/humans/1234 devient humans/1234

Ensuite

* "spliter" uri : on sépare tous les éléments entre les "/"
* on obtient un tableau que l'on filtre pour ne garder que les éléments dont la taille > 0
* **le 1er élément du tableau** devient **la clé** à rechercher dans la map
* les éléments restants représentent **les paramètres** que l'on passera à la méthode correspondant à la clé

Par exemple si je saisis #/humans/bob/morane, cela deviendra :

["humans", "bob", "morane"]

*et j'irais chercher dans mon instance de Router la clé "humans" qui va me retourner une méthode à laquelle je passerais (si elle existe) les paramètres ["bob", "morane"] (donc sous forme d'un tableau) et je l'exécuterais*

*(allez voir main.js pour les exemples)*

**J'ai ajouté une méthode listen()**

Elle va une 1ère fois au lancement vérifier l'url pour déterminer s'il y a un traitement à lancer, la méthode match du routeur sera appelé avec window.location.hash en paramètre.

window.location.hash : retourne la partie de l'url qui correspond à "anchor" : http://localhost:3000/#/humans donnera #/humans

Cette 1ère vérification est utile si on a bookmarké une fonctionnalité de l'application

Ensuite on va s'abonner à l'évènement onpopstate (déclenché lorsque l'utilisateur "navigue", ultilise le bouton back, etc...)

et à chaque fois que l'évènement onpopstate sera déclenché, la méthode match du routeur sera appelé avec window.location.hash en paramètre.

## Exercice

* complétez public/js/skeleton/Router.js en complétant les méthodeq add() et match()

**Remarque**: les spécifications sont décrites dans les fichiers

cd 06-router  
node app.js  
http://localhost:3000

# Correction

## Router.js

class Router {  
  
 constructor () {  
 this.routes = new Map();  
 }  
  
 add (uri, action) {  
 this.routes.set(uri, action);  
 return this;  
 }  
  
 match (uri) {  
  
 // on retraite l'uri: enlever les #/  
 uri = uri.replace("#\/","");  
  
 // splitter uri avec "/" et ne garder que les éléments non vides  
 let uriParts = uri.split("/").filter((part)=>part.length>0);  
  
 // clé à chercher  
 let route = uriParts[0];  
 // paramètres à passer à la méthode  
 let params = uriParts.slice(1);  
  
 // récupérer la méthode  
 let method = this.routes.get(route);  
  
 // exécuter la méthode  
 if (method) {  
 method(params)  
 } else {  
 this.routes.get("/")(params)  
 }  
 }  
  
 listen () {  
 // une fois le routeur en mode écoute  
 // lui faire vérifier une 1ère fois l'url pour déterminer quoi faire  
 this.match(window.location.hash);  
  
 /\* s'abonner à onpopstate \*/  
 window.onpopstate = (event) => {  
 this.match(window.location.hash);  
 };  
 }  
  
}  
  
export default Router