ECMAScript 2015

Les générateurs

Itérateurs

Qu’est ce qu’un itérateur ?

- Il s’agit d’une structure de données dont on peut parcourir les éléments un par un
- On dit d’une telle structure qu’elle est « iterable »
- Certains types sont itérable :
  - Arrays
  - Strings
  - Maps
  - Sets
  - DOM node

Utiliser un itérateur

- Les itérateurs sont à la base de certains choses que nous avons vu :
  - Les boucles for-of
  - Le destructuring
  - Le spreading
  - Array.from()
  - Promise.all() et Promise.race()
  - Les constructeurs de Map et Set

Créer un itérateurs

- Un itérateur est un objet qui possède une méthode **next()** qui retourne un objet **{value,done}**
- **Value** contient la prochaine valeur de la collection
- **Done** est un boolean permettant de savoir quand il n'y a plus de nouvelle valeur

Un exemple en ES5

```
function createIntegerIterator(max) {
    var curInt = 0;
    return {
        next: function() {
            curInt++;
            if(curInt <= max)
                return {value: curInt, done: false};
            else
                return {done: true};
        }
    }
};

var it = createIntegerIterator(2);
console.log(it.next().value); // 1
console.log(it.next().value); // 2
console.log(it.next().done);  // true
```

Les itérateurs ES2015

- En ES2015, les itérateurs sont des objets un peu particuliers
- En plus des types que l'on a vu plus tôt il est possible de créer ses propres itérateurs
- Un itérateur est un objet classique avec un Symbole **Symbol.Iterator** comme propriété
- La valeur de cette propriété est une fonction retournant un itérateur

Un exemple en ES2015

```
function IntergerIterable (max) {
  this[Symbol.iterator] = function() {
    var curInt = 0;
    return {
      next: function() {
        curInt++;
        if(curInt <= max)
          return {
            value: curInt,
            done: false
          };
        else
          return {done: true};
      }
    }
  };
};
```

Ce qui permet

```
var myIterable = new IntegerIterable(2);
for(var i of myIterable) {
  console.log(i);
}


spread(...myIterable);


var it = myIterable[Symbol.iterator]();

console.log(it.next());
console.log(it.next());
console.log(it.next());
```

Les générateurs

Qu’est ce qu’un générateur ?

- Un générateur est une fonction un peu spéciale qui renvoie un itérateur
- Le code contenu dans un générateur peut être mis en pause et reprendre son exécution plus tard

Créer un générateur

- Un générateur est une fonction à laquelle on ajoute le caractère * :

```
function* myGenerator() {}

let myGenerator = function*() {};

class MyClass {
  *generatorMethod() {}
}
```

Utiliser un générateur

- Il s'agit d'une fonction retournant TOUJOURS un itérateur :

```
function* myGenerator() {}

let it = myGenerator();

for(let val of it) {}


var elem = it.next();
if(!elem.done) {
  console.log(elem.value);
}
```

Générer des valeurs

- Le principe d'un générateur est de générer des valeurs à la demande
- La prochaine valeur est retournée en utilisant le mot clef **yield**
- Il est possible de mettre autant de **yield** que vous le souhaitez
- Lorsqu'il n'y a plus de valeur vous terminer l'itérateur en le laissant finir son exécution ou avec **return**

Exemples

```
function* gen1() {
  yield 1;
}

function* gen1to3() {
  yield 1;
  yield 2;
  yield 3;
}

function* genInfiniteLoop() {
  while(true) {
    yield 1;
  }
}

function* genReturn() {
  let val = 0;
  while(true) {
    yield val++;
    if(val > 10) return;
  }
}
```

La mise en pause

- Lorsque vous appelez un générateur le code à l'intérieur ne s'exécute pas immédiatement
- A chaque appel de **next()** le code s'exécute jusqu'au prochain **yield**, renvois la valeur suivant le yield puis se met en pause

Step by step

```
var iterator = myGenerator();
```

```
function* myGenerator()
{
    yield 1;
    yield 2;
    yield 3;
}
```

Step by step

```
var iterator = myGenerator();
var elem = iterator.next();
```

```
function* myGenerator()
{
    yield 1;
    yield 2;
    yield 3;
}
```

```
elem = {
    value: 1,
    done: false
}
```

Step by step

```
var iterator = myGenerator();
var elem = iterator.next();
elem = iterator.next();
```

```
function* myGenerator()
{
    yield 1;
    yield 2;
    yield 3;
}
```

```
elem = {
    value: 2,
    done: false
}
```

Step by step

```
var iterator = myGenerator();
var elem = iterator.next();
elem = iterator.next();
elem = iterator.next();
```

```
function* myGenerator()
{
    yield 1;
    yield 2;
    yield 3;
}
```

```
elem = {
    value: 3,
    done: false
}
```

Step by step

```
var iterator = myGenerator();
var elem = iterator.next();
elem = iterator.next();
elem = iterator.next();
elem = iterator.next();
```

```
function* myGenerator()
{
    yield 1;
    yield 2;
    yield 3;
}
```

```
elem = {
    value: undefined,
    done: true
}
```

Rappelez-vous…

```
function IntergerIterable (max) {
  this[Symbol.iterator] = function() {
    var curInt = 0;
    return {
      next: function() {
        curInt++;
        if(curInt <= max)
          return {
            value: curInt,
            done: false
          };
        else
          return {done: true};
      }
    }
  };
};
```

En générateur

```
function* integerGenerator(max) {
  var curInt = 0;
  while(curInt < max) {
    curInt++;
    yield curInt;
  }
};
```

Arrêter un générateur

- Il est possible de terminer un générateur de trois façons :
  - Il n'y a plus de **yield** a atteindre → {done:true}
  - Une instruction **return** est atteinte → {done:true,value:<return value>}
  - Une exception est levée → pas de retour

```
function* gen1() {
  yield 1;
  return 2;
}

function* gen2() {
  yield 1;
  throw new Error('arrrrg');
}
```

Communiquer avec un générateur

- La méthode **next()** peut prendre un paramètre qui sera passé au générateur
- La valeur est renvoyée par le **yield** sur lequel le générateur est arrêté
- Il n'est pas possible de passer de valeur au premier appel à **next()**

```
function* consoleLog() {
  while(true) {
    var value = yield;
    console.log(`received : ${value}`);
  }
}
```

Step by step

```
var it = consoleLog();
it.next();
```

```
function* consoleLog() {
    while(true) {
        var value = yield;
        console.log(…);
    }
}
```

Step by step

```
var it = consoleLog();
it.next();
it.next(2);
```

```
function* consoleLog() {
    while(true) {
        var value = yield;
        console.log(…);
    }
}
```

Step by step

```
var it = consoleLog();
it.next();
it.next(2);
```

```
function* consoleLog() {
    while(true) {
        var value = yield;
        console.log(…);
    }
}
```

"received : 2"

Step by step

```
var it = consoleLog();
it.next();
it.next(2);
```

```
function* consoleLog() {
    while(true) {
        var value = yield;
        console.log(…);
    }
}
```

"received : 2"

Step by step

```
var it = consoleLog();
it.next();
it.next(2);
it.next(3);
```

```
function* consoleLog() {
    while(true) {
        var value = yield;
        console.log(…);
    }
}
```

"received : 2"

Step by step

```
var it = consoleLog();
it.next();
it.next(2);
it.next(3);
```

```
function* consoleLog() {
    while(true) {
        var value = yield;
        console.log(…);
    }
}
```

"received : 3"

Autres interactions

- Il est possible d'intéragir avec le générateur depuis l'itérateur
- Il est possible de forcer une terminaison avec **return(value)**
- Il est possible de forcer une erreur avec **throw(error)**

```
let it = consoleLog();
it.return();
it.throw(new Error("arg"));
```

Imbriquer les générateurs

- Il est possible de « yield » un itérateur avec **yield***
- Fonctionne avec tous les itérateurs
- Tous les appels sont transmis au second itérateur avec leurs valeurs

```
function* gen123() {
  yield 1;
  yield 2;
  yield 3;
}

function* gen123456() {
  yield* gen123();
  yield 4;
  yield 5;
  yield 6;
}
```

Générateur et asynchrone

Co-routines

- Il est possible d'utiliser les générateurs pour faire de l'asynchrone
- Le générateur est suspendu en attendant la résolution de la tâche
- C'est ce qu'on appel une coroutine
- Une coroutine prend en paramètre un générateur qui yield des promesses
- Lorsque la promesse est résolue la méthode **next()** du générateur est appelée avec la valeur de résolution

Le code

```
function co(genFunc) {
  const genObj = genFunc();
  step(genObj.next());

  function step({value,done}) {
    if (!done) {
      // A Promise was yielded
      value
        .then(result => {
          step(genObj.next(result)); // (A)
        })
        .catch(error => {
          step(genObj.throw(error)); // (B)
        });
    }
  }
}
```

Utiliser une coroutine

```
co(function* () {
    const value = yield getFilePromise();
    console.log(value);
});
```