ECMAScript 2015

Les fonctions

Paramètres par défaut

Avant

```
function testParams(param1, param2) {
  if(param1 === undefined) {
    param1 = "";
  }
   …
}
```

Ou (bien plus dangereux)

```
function testParams(param1, param2) {
  param1 = param1?param1:"";
}
```

En ES2015

- Il est possible de définir des valeurs par défaut dans la signature :

```
function testParams(param1 = "", param2) {}
```

- N'importe quel paramètre peut avoir une valeur par défaut :

```
function testParams(param1, param2 = "") {}
```

- La valeur par défaut peut être celle d'un autre paramètre

```
function testParams(param1, param2 = param1) {}
```

- Il est recommandé de mettre les paramètres optionnels en dernier

Destructuration dans les paramètres

Rappelez-vous…

```
var {prop1, prop2, prop3} = myObject;
```

Cela marche également dans les signatures de fonctions !

```
function convertPrice(price,settings) {
  return price * settings.change;
}
```

Devient

```
function convertPrice(price,{change}) {
  return price * change;
}
```

Exemples avancés

Mixé avec les paramètres par défaut

```
function convertPrice(price,{change=1}) {
  return price * change;
}
```

Mixé avec les template strings

```
function
buildUrl({protocol='http',host,port=80,context}) {
  return `${protocol}://${host}:${port}/${context}`;
}
```

Destructuring de tableaux

- Cela marche également avec les tableaux :

```
function join(char,[first,...others]) {
  return first + others.reduce(
    function(joined,value) {
      return `${joined}${char}${value}`;
    },'');
}


join(',',['Mathieu','Jean','Robert']);
// "Mathieu,Jean,Robert"
```

Rest operator

Gérer un nombre inconnu de paramètres

- En ES5 nous avons la variable **arguments** définie dans chaque fonction qui permet de manipuler les paramètres sans en connaître le nombre
- Son utilisation est un peu fastidieuse
- En ES2015, le Rest parameter '…' permet de gérer cela bien plus facilement :

```
function rested(param1,...params) {}

rested('hello'); // param1='hello', params=[]
rested('hello','world'); // param1='hello',
                         params=['world']
rested('hello','world','!'); // param1='hello',
                             params=['world','!']
```

Exemple plus complexe

```
function sum(a, b, ...rest) {
  return a + b + rest.reduce(function(initial,
value) {
      return initial + value;
    },0);
}

sum(1,2); // 3
sum(1,2,3); // 6
sum(1,2,3,4); // 10


let values = [1,2,3];
sum(values); // "1,2,3undefined0"
sum(...values); // 6
```

Fat arrow functions

Fat arrow functions

- Nouvelle écriture simplifié pour les fonctions passées en paramètre :

```
() => "hello world";
```

Equivaut à

```
function() {
  return "hello world";
}
```

Si la fat arrow function est monoligne, les accolades sont facultatives et le return est implicite

S'il y a un seul paramètre, les parenthèse autour sont facultatives

Maintient du contexte

- En javascript, le context est perdu lorsqu'une fonction anonyme est passée en paramètre :

```
function MyClass() {
  this.value1 = 'val1';
}

MyClass.prototype.myMethod = function(array) {
  array.map(function(value) {
      console.log(this.value1); // undefined
  });
};
```

Solution 1 : that = this

- On créé une variable contenant le contexte :

```
function MyClass() {
  this.value1 = 'val1';
}

MyClass.prototype.myMethod = function(array) {
  var that = this;
  array.map(function(value) {
      console.log(that.value1); // "val1"
  });
};
```

➔ Source d'erreurs en mélangeant *this* et *that*

Solution 2 : binding

- On utilise la fonction *bind()* :

```
function MyClass() {
  this.value1 = 'val1';
}

MyClass.prototype.myMethod = function(array) {
  array.map(function(value) {
      console.log(this.value1); // "val1"
  }.bind(this));
};
```

Solution 3 : ES2015 !

- Les fat arrow functions maintiennent naturellement le contexte :

```
function MyClass() {
  this.value1 = 'val1';
}

MyClass.prototype.myMethod = function(array) {
    array.map(value => console.log(this.value1));
};
```