

Module 7

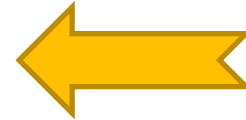
Compléments sur les fonctions

Technologies web
HENALLUX — IG2

Compléments sur les fonctions

➤ Gestion des paramètres de fonctions

- Paramètres optionnels, valeurs par défaut
- Surcharge
- Paramètres en nombre indéterminé
- Déstructuration




➤ Le mot-clef "this"

➤ Fonctions comme objets de premier ordre

➤ Pour aller plus loin : les closures (*hors cours*)

Gestion des paramètres

- 
- Rappels
 - Paramètres optionnels
 - Surcharge
 - Paramètres en nombre indéterminé
 - Déstructuration

Ensuite : *Le mot-clef "this"*

Rappels

- Sur les paramètres : **aucun contrôle de type**
 - Aucune déclaration de type non plus
 - *Comment surcharger une fonction ?*
- Sur les paramètres : **aucun contrôle de nombre**
 - On peut appeler une fonction avec "pas assez" ou "trop" d'arguments.
 - **Pas assez** : on complète avec des valeurs undefined.
 - **Trop** : on ignore les paramètres supplémentaires.

Paramètres optionnels

- Cas #1 : **paramètre optionnel simple**

```
function créePara (texte, classe) {  
  let code = "<p";  
  if (classe !== undefined)  
    code += " class='" + classe + "'";  
  code += ">" + texte + "</p>";  
  return code;  
}
```

- Conversion de "classe" en booléen (`undefined` → `false`)
 - *Rappel* : donne "true" sauf pour les valeur falsy
 - *Valeurs falsy* : `false`, `null`, `undefined`, `0`, `NaN`, `""`
- **Attention** : uniquement si aucune valeur falsy n'est valide !
- Protection : tester `if (classe === undefined)`

Paramètres optionnels

- Cas #2 : **paramètre optionnel avec valeur par défaut (1/2)**

```
function saluer (nom) {  
  if (nom) {  
    alert("Hi, " + nom + "!");  
  } else {  
    alert("Hi, anonyme!");  
  }  
}
```

```
function saluer (nom) {  
  nom = nom || "anonyme";  
  alert("Hi, " + nom + "!");  
}
```

- Conversion de nom en booléen (`undefined` → `false`)
 - *Rappel* : donne "true" sauf pour les valeurs falsy
 - *Valeurs falsy* : `false`, `null`, `undefined`, `0`, `NaN`, `""`
- **Attention** : uniquement si aucune valeur falsy n'est valide !

Paramètres optionnels

- Cas #2 : **paramètre optionnel avec valeur par défaut (2/2)**

```
function salue (nom = "anonyme") {  
    alert("Hi, " + nom + "!");  
}
```

- Seul `undefined` déclenche le calcul et l'utilisation de la valeur par défaut (sans ça, la valeur par défaut n'est même pas évaluée).
- On peut utiliser les paramètres précédents comme valeur par défaut :

```
function créeLien (url, texte = url) {  
    return `\${texte}</a>`;   
}
```

Surcharge

- Si on définit deux fois une fonction (même nom), seule la dernière définition compte.

- Il faut donc **traiter tous les cas en une seule fonction** !

```
function retardTrain (retard) {  
    if (typeof retard == "number")  
        return retard + " minute(s)";  
    if (typeof retard == "string")  
        return retard;  
}
```

- [Clean Code] Rien ne vous empêche de définir plusieurs fonctions de noms différents pour séparer les cas.

```
function retardTrain (retard) {  
    if (typeof retard == "number") return retardNum(retard);  
    if (typeof retard == "string") return retardStr(retard);  
}
```


Nb de paramètres variable

- Fonction qui peut être appelée avec un **nombre quelconque d'arguments** (exemple : `Math.min`)

```
function moyenne (...valeurs) {  
  let somme = 0;  
  for (let valeur of valeurs) somme += valeur;  
  return somme / valeurs.length;  
}
```
- Le **"rest" parameter** `...valeurs` rassemble tous les autres arguments en un tableau.
- Il ne peut y avoir qu'un seul paramètre "rest" par fonction et celui-ci doit se trouver en dernière position !
- *Ancienne méthode (pré-ES6)* : dans la fonction, utiliser la variable locale prédéfinie "arguments" = tableau des paramètres effectifs.

Nb de paramètres variable

- Le pendant du "rest" parameter est le "**spread operator**".
 - Rest : conversion liste d'arguments -> tableau
 - Spread : conversion tableau -> liste d'arguments
- Exemples :

```
let tempSemaine1 = [15, 17, 16, 14, 13, 12, 15];
moyenne(...tempSemaine1);

moyenne(16, ...tempSemaine1);


let tempSemaine2 = [18, 15, 13, 14, 11, 16, 12];
moyenne(...tempSemaine1, ...tempSemaine2);
```
- Peut également s'utiliser dans un littéral de tableau :

```
let temp = [...tempSemaine1, ...tempSemaine2];
```
- Peut s'utiliser sur n'importe quel objet itérable.

Déstructuration

- La **déstructuration** permet d'accéder directement aux composantes d'une valeur structurée.
 - *Valeur structurée* = objets et tableaux (ou autres éléments similaires comme les collections).
 - *Accéder directement* = les associer à un nom de variable
- Exemples (tableaux) : `afficheHeure([13, 37, 30]);`

```
function afficheHeure (tHeure) {  
  console.log `${tHeure[0]}:${tHeure[1]}:${tHeure[2]}`;  
}
```



Pattern pour un tableau

```
function afficheHeure ([h, m, s]) {  
  console.log `${h}:${m}:${s}`;  
}
```


Faire correspondre une valeur avec un pattern = "pattern matching"

Déstructuration

- Exemples (objets) :

```
afficheHeure({heure: 13, min: 37, sec: 30});
```

```
function afficheHeure (oHeure) {  
  console.log `${oHeure.heure}:${oHeure.min}:${oHeure.sec}`;  
}
```



Pattern pour un objet

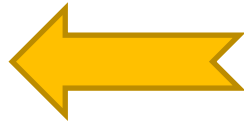
```
function afficheHeure ({heure, min, sec}) {  
  console.log `${heure}:${min}:${sec}`;  
}
```

Compléments sur les fonctions

➤ Gestion des paramètres de fonctions

- Paramètres optionnels, valeurs par défaut
- Surcharge
- Paramètres en nombre indéterminé
- Déstructuration


➤ Le mot-clef "this"



➤ Fonctions comme objets de premier ordre

➤ Pour aller plus loin : les closures

Le mot-clef "this"

- 
- Signification de "this"
 - Un exemple
 - Gestion "manuelle" de "this"

Ensuite : *Les fonctions comme objets de premier ordre*

Signification de "this"

- En Javascript, une fonction (ou "objet exécutable") peut être
 - une **fonction simple**
 - Appel : `afficheHeure(h, m);`
 - Exécution : standard (`this` = l'objet global `window` généralement)
 - un **méthode** (d'un objet)
 - Appel : `heureDébut.afficheHeure();`
 - Exécution : on considère que `this` = l'objet concerné
 - un **constructeur**
 - Appel : `new Heure (15, 30);`
 - Exécution : on crée un nouvel objet lié au prototype associé à la fonction constructrice, on considère que `this` = cet objet
- 3 cas mais même syntaxe et même représentation Javascript.
- Il s'agit juste de différences d'utilisation. C'est la manière dont on appelle une fonction qui détermine ce que `this` signifie.

Un exemple (1/2)

- Un compteur intégré sous la forme d'un objet.

```
let compteur = {  
  val : 0,  
  inc () { this.val++; }  
};
```

- Utilisation du compteur

```
compteur.val;  
compteur.inc();  
compteur.val;
```

- À tester

```
let ajoute = compteur.inc;  
ajoute();  
compteur.val;
```


Un exemple (2/2)

- Un compteur intégré sous la forme d'un objet.

```
let compteur = {  
  val : 0,  
  inc () { this.val++; }  
};
```

- Utilisation de setTimeout (rappels)

```
function saluer () { console.log("Hello !"); }  
setTimeout(saluer, 2000);
```

- À tester

```
setTimeout(compteur.inc, 2000);  
// attendre  
compteur.val;
```

Gestion manuelle de "this"

1^{re} méthode : demander **l'exécution directe**

- `fonc.call(argThis, arg1, arg2, ...)`
- `fonc.apply(argThis, tabArgs)`
- On exécute la fonction en utilisant `this = argThis` et les arguments donnés.
- `obj.fn(a1,a2) ≡ fn.call(obj,a1,a2) ≡ fn.apply(obj,[a1,a2])`

Exemple :

```
function présente () {  
    return `Je suis ${this.nom}.`;  
}  
présente.call(homer);  
homer.présente(); // seulement si homer a accès à cette propriété
```

Gestion manuelle de "this"

2^e méthode : lier `this` pour une **exécution future** :

- `fonc.bind(argThis, arg1, arg2, ...)`
- Renvoie une fonction qui pourra être exécutée plus tard.

Exemple :

```
let action = compteur.inc.bind(compteur);  
setTimeout(action, 2000);
```

```
function facture (article, prix, nb) {  
  return `${article} x ${nb} : ${prix * nb} Euros.`;  
}
```

```
let factureDago = facture.bind(this, "Dago", 2.5);  
factureDago(10);    // Dago x 10 : 25 Euros.
```

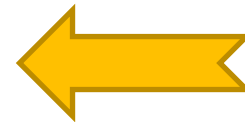
Compléments sur les fonctions

➤ Gestion des paramètres de fonctions

- Paramètres optionnels, valeurs par défaut
- Surcharge
- Paramètres en nombre indéterminé
- Déstructuration


➤ Le mot-clef "this"

➤ Fonctions comme objets de premier ordre



➤ Pour aller plus loin : les closures

Fonctions = objets de 1^{er} ordre

- 
- Fonctions anonymes (une nouvelle syntaxe)
 - Objets de 1^{er} ordre
 - Quelques exemples
 - Application

Ensuite : *Les fonctions comme objets de premier ordre*

Fonctions anonymes

- Utiliser une valeur (numérique) sans lui attribuer de nom.

```
let produit = x * y;  
console.log(produit);  console.log(x * y);
```

- Utiliser une valeur (fonctionnelle) sans lui attribuer de nom.

```
let logDouble = function (x) { console.log(x + x); }  
logDouble("hello");
```



```
(function (x) { console.log(x + x); })("hello");
```

ou

```
((x) => { console.log(x + x); })("hello");
```

IIFE

Immediately
Invoked
Function
Expression

Fonctions anonymes

- Syntaxe "**Big arrow**" : *argument(s) => résultat*
 - **Argument(s)** :
 - `()` aucun argument
 - `x` un seul argument
 - `(x,y)` plusieurs arguments
 - **Résultat** :
 - `x * 2` valeur à renvoyer
 - `{ ... }` code à exécuter
- Exemples :
 - `() => { alert("Une erreur s'est produite !"); }`
 - `(article, prix) => `${article} coûte ${prix} Euros``
 - `nom => { console.log("Salut, " + nom + " !"); }`

Fonctions anonymes

- Deux cas particuliers de la syntaxe "**Big arrow**"
 - Seul cas où on peut omettre les parenthèses autour des arguments : quand il s'agit d'un simple identificateur

```
(x,y) => x * y    // plusieurs arguments
```
 - ```
([x,y]) => x * y // un argument mais déstructuration
```
  - ```
(x = 10)          // un argument mais valeur par défaut
```
 - Si la fonction retourne un objet décrit par un littéral { ... }, il faut l'entourer de parenthèses (sinon, les accolades sont interprétées comme un bloc)

```
hhmm => ({ hr : Math.floor(hhmm/100), min : hhmm % 100 })
```
- *Note.* Les "big arrow" sont plus qu'une simple syntaxe alternative : contrairement aux fonctions définies via `function`, les "big arrow" ne cachent pas la valeur de `this`.

Objets de premier ordre

Cela signifie que :

- On peut placer une valeur fonctionnelle dans une variable.

```
let carré = function (x) { return x * x; };  
let carré = x => x * x;
```
- On peut passer une fonction comme argument d'une fonction.

```
tabNombres.forEach(x => console.log(2 * x));  
tabValeurs.forEach(créeTD);
```
- Une fonction peut renvoyer une fonction comme résultat.

```
function actionAfficher (msg) {  
  return function () { alert(msg); }  
}  
bt.onclick = actionAfficher("Vous avez cliqué ?");
```

Fonctions de haut niveau

- Quelques méthodes de haut niveau sur les tableaux
 - `tab.forEach(f)` : exécute la fonction `f` sur chacun des éléments
 - `tab.map(f)` : renvoie le tableau obtenu en remplaçant chaque élément `x` du tableau par `f(x)`
 - `tab.every(f)` : indique si tous les éléments vérifient `f`
 - `f` : fonction à valeur booléenne (prédicat)
 - valeur renvoyée = $\forall x \in \text{tab} : f(x)$
 - `tab.some(f)` : indique si au moins un élément vérifie `f`
 - valeur renvoyée = $\exists x \in \text{tab} : f(x)$
 - `tab.filter(f)` : renvoie les éléments qui vérifient `f`

Application

- **Objets** pour représenter un réseau d'amis.
 - Chaque objet = une personne
 - **Attributs** : nom, amis (= tableau de personnes)
 - **Méthode** : ajouteAmi, listeAmis (cite les amis d'une personne)
- Exemple
 - Homer a pour amis Marge et Ned. (lors de la création de l'objet)
 - Lenny et Carl sont également ses amis. (à ajouter)
 - listeAmis devrait afficher :
 - Homer a pour ami Marge.
 - Homer a pour ami Ned.
 - Homer a pour ami Lenny.
 - Homer a pour ami Carl.

Application

```
function Personne (nom, amis) {  
  this.nom = nom; this.amis = amis;  
}  
Personne.prototype.ajouteAmi = function (ami) {  
  this.amis.push(ami);  
}  
Personne.prototype.listeAmis = function () {  
  for (let ami of this.amis) {  
    console.log(`${this.nom} a pour ami ${ami}.`);  
  }  
}  
let h = new Personne ("Homer", ["Marge", "Ned"]);  
h.ajouteAmi("Carl");  
h.ajouteAmi("Lenny");  
h.listeAmis();
```

- Écrire une version utilisant `.forEach` au lieu d'une boucle `for of` ?

Application

- Version avec "function"


```
Personne.prototype.listeAmis = function () {  
  this.amis.forEach(function(ami) {  
    console.log(`${this.nom} a pour ami ${ami}.`);  
  });  
}  
h.listeAmis();
```



KO car cache "this"

- Version "big arrow"

```
Personne.prototype.listeAmis = function () {  
  this.amis.forEach(ami => {  
    console.log(`${this.nom} a pour ami ${ami}.`);  
  });  
}  
h.listeAmis();
```



OK car => ne cache pas "this"