

Semaine 12

Autres structures de contrôle et création d'éléments HTML

Intro. à la programmation



❖ Boucles

- ◆ for

❖ Conditions

- ◆ switch
- ◆ Condition ternaire

❖ Création d'éléments HTML

- ◆ Créer un élément
- ◆ Ajouter l'élément dans le DOM (Dans la page Web)
- ◆ Personnaliser l'élément
- ◆ Supprimer un élément HTML

❖ null et undefined

- ◆ Vérifier si un élément HTML existe avant de le supprimer



❖ Il y a plusieurs types de **boucles**

◆ ... mais nous avons surtout pratiqué avec **while**

★ **while**

```
let x = 3;

while(x < 10){
  x += 5;
}

console.log(x);
// x vaut 13
```

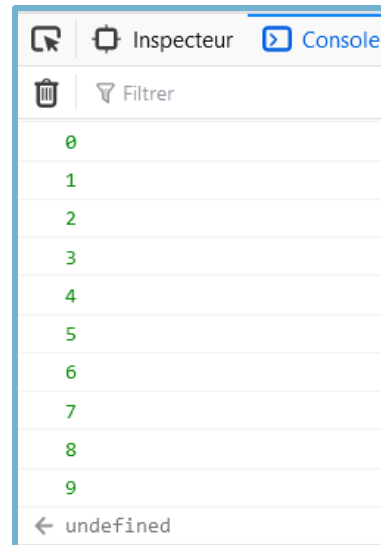
do ... while

```
let y = 20;
do{
  y -= 5;
}while(y > 30);

console.log(y);
// y vaut 15
```

for

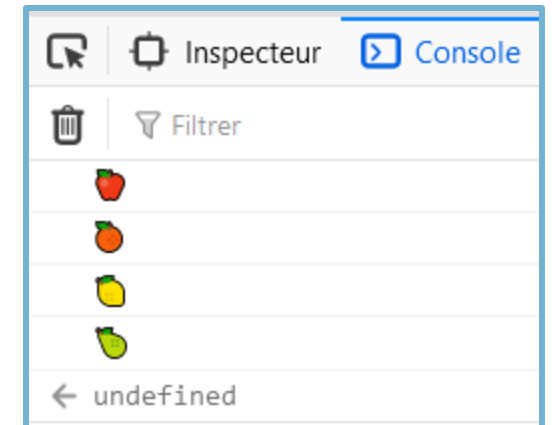
```
for(let i = 0; i < 10; i+= 1){
  console.log(i);
}
```



for-of (for pour tableau)

```
let fruits = ["🍎", "🍊", "🍋", "🍌"];

for(let f of fruits){
  console.log(f);
}
```



Vous verrez cette boucle dans un autre cours !



- ❖ Nous avons peu pratiqué **do ... while** et **for** 🤔
 - ◆ Les boucles **for** sont généralement utilisées dans **99% des cas**.

1% des cas



```
while(...) {  
      
}
```

```
do {  
      
} while(...);
```

```
for (..; ..; ..) {  
      
}
```

99% des cas



Avoir peu pratiqué **do while** n'est pas grave du tout. Cela dit, on va devoir pratiquer **for** !



❖ Fonctionnement de la boucle **for**

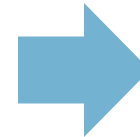
- ◆ Pour les boucles **for**, une variable et une condition sont **intégrées directement** dans la structure de la boucle. C'est idéal quand on sait d'avance le **nombre d'itérations** qu'on souhaite faire.

- La variable **i** est déclarée **avant la boucle**.
- La valeur de la variable **i évolue** à l'intérieur de la boucle. (**i += 1**)

- Avec la boucle **for**, la déclaration de **i**, la **condition** et **l'évolution de la variable** sont tous intégrés dans la déclaration de la boucle.
- **i** commence à **1**, avant chaque itération, on vérifie que **i** est **< 5** et après chaque itération, on **augmente i** de **1**.

```
let i = 1;

while(i < 5){
  console.log(`i vaut ${i}`);
  i += 1;
}
```



Console	
🗑️	🔍 Filtrer
i	vaut 1
i	vaut 2
i	vaut 3
i	vaut 4

```
for(let i = 1; i < 5; i += 1){
  console.log(`i vaut ${i}`);
}
```



Console	
🗑️	🔍 Filtrer
i	vaut 1
i	vaut 2
i	vaut 3
i	vaut 4

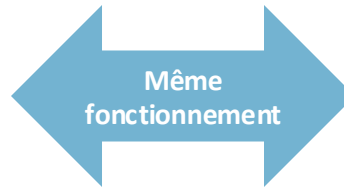


❖ Les boucles **while** et **for** sont interchangeables

- ◆ Les deux peuvent faire le même travail ! (Dans la majorité des cas)
- ◆ La principale différence est la **syntaxe**.

```
let i = 0;

while(i < 10){
  console.log(i);
  i += 1;
}
```



```
for(let i = 0; i < 10; i += 1){
  console.log(i);
}
```

- La syntaxe de la boucle **for** est généralement considérée comme **plus élégante**. (Et on a moins de chances d'oublier de mettre le **i += 1** !)
- Quand on connaît d'avance le **nombre d'itérations** que doit faire la boucle, (ici, **10**, car on va de **0** à **9**) la boucle à favoriser est la boucle **for**. (Rien ne vous empêche d'utiliser **while** si vous y tenez absolument)



❖ Boucles **while**

- ◆ On doit essayer de les utiliser surtout **quand on ne sait pas d'avance combien d'itérations la boucle doit faire.**

- Exemple : J'achète des potions tant que j'ai assez d'argent pour en acheter.

(On aurait pu faire un calcul mathématique à la place d'utiliser une boucle, mais à notre niveau les exemples pertinents sont limités pour une boucle while)

```
let nbPotionsObtenues = 0;

while(gPortefeuille > gPrixPotion){

    gPortefeuille -= gPrixPotion;
    nbPotionsObtenues += 1;

}
```



- ◆ Pourquoi avoir principalement utilisé les boucles **while** jusqu'ici dans ce cas ?
 - Car ce sont les plus **flexibles**. (Avec **for**, on veut connaître le nombre d'itérations d'avance)
 - Elles sont un **bon point de départ pour apprendre**.



❖ Bref, pratiquons les boucles **for** !

Calculer la **somme** des nombres de **1** à **100**

```
let sommeDe1a100 = 0;

for(let i = 1; i < 101; i += 1){
  |   sommeDe1a100 += i;
}
```

```
alert(sommeDe1a100);
```



Donner une largeur de **200 pixels** aux éléments avec les classes **.chien1**, **.chien2**, **.chien3**, ..., **.chien16**

```
for(let i = 1; i < 17; i += 1){
  |   document.querySelector(`.chien${i}`).style.width = "200px";
}
```





❖ La boucle **for** peut parcourir des tableaux et les modifier

◆ Exemple 1 : Augmenter toutes les valeurs de **4**.

```
let nombres = [11, 17, 3, 8, 12];  
  
for(let i = 0; i < nombres.length; i += 1){  
  nombres[i] += 4;  
}
```

```
>> nombres
```

```
← ► Array(5) [ 15, 21, 7, 12, 16 ]
```

- La condition d'exécution doit toujours être **`i < tableau.length`** !

- Par exemple, ici **`nombres.length`** vaut **5**. Ça tombe bien, on peut utiliser les index **0, 1, 2, 3** et **4** pour ce tableau, donc on s'arrête juste avant **5**.


Index	0	1	2	3	4
Valeur	11	17	3	8	12



❖ La boucle **for** peut parcourir des tableaux et les modifier

◆ Exemple 2 : Retirer tous les **"rat"** dans le tableau.

```
let gAnimaux = ["chat", "rat", "rat", "chien", "chat"];  
  
for(let i = gAnimaux.length - 1; i >= 0; i -= 1){  
    if(gAnimaux[i] == "rat"){  
        gAnimaux.splice(i, 1);  
    }  
}
```



Index	0	1	2	3	4
Valeur	"chat"	"rat"	"rat"	"chien"	"chat"

- Remarquez qu'on parcourt le tableau à l'envers ! En effet, on commence par l'index 4 (`gAnimaux.length - 1` donne 4) et on diminue l'index de 1 jusqu'à ce qu'on atteigne 0.

- Pourquoi ? Car si on parcourait le tableau **dans l'ordre**, supprimer un élément avec `splice` déplacerait tous les éléments suivants restants vers la gauche... mais lors de la prochaine itération, `i` va augmenter de 1 et on va « sauter » (skip) le prochain élément. (Pas grave si vous ne comprenez pas totalement)

Index 0 ● ["chat", "rat", "rat", "chien", "chat"]



❖ Jusqu'ici nous avons utilisé **if**, **else**, et **else if**.

```
if(gArgent >= 20){  
    gArgent -= 20;  
    console.log("Repas acheté ! 🍲");  
}
```

```
if(gAge >= 18){  
    console.log("Je bois du vin 🍷👤");  
}  
else{  
    console.log("Je bois un p'tit jus 🥤👤");  
}
```

```
if(gDate == "31 octobre"){  
    console.log("C'est l'Olowigne ! 🎃");  
}  
else if(gDate == "14 février"){  
    console.log("C'est la Saint-Valentin ! ❤️");  
}  
else{  
    console.log("C'est une journée. 🤔");  
}
```



- ❖ Nous allons voir deux nouvelles structures conditionnelles
 - ◆ Les **switch**
 - ◆ Les **conditions ternaires**
- ◆ Ces deux structures conditionnelles sont **utilisées très fréquemment** et il est important de les maîtriser au même titre que les **if / else**.
 - Les **if / else** peuvent déjà gérer toutes les situations, mais les **switch** et les **conditions ternaires** permettent de gérer certains problèmes plus **efficacement** ou **élégamment**.



❖ Les **switch**

- ◆ Permettent d'exécuter un bloc de code selon la valeur reçue

- Dans les parenthèses du `switch(...)`, on met généralement **une simple valeur**. (Plutôt qu'une **condition** !)
- Chaque **case** contient le **code à exécuter** si la valeur correspond. Par exemple, si la valeur vaut **3**, le code à exécuter est « `alert("C'est un 3 !");` »

```
let gNombre = 3;

switch(gNombre){

    case 1 : alert("C'est un 1 !"); break;
    case 2 : alert("C'est un 2 !"); break;
    case 3 : alert("C'est un 3 !"); break;

}
```



❖ Les **switch**

- ◆ L'équivalent avec des **if** / **else if** est moins élégant !

```
let gNombre = 3;

switch(gNombre){

    case 1 : alert("C'est un 1 !"); break;
    case 2 : alert("C'est un 2 !"); break;
    case 3 : alert("C'est un 3 !"); break;
}
```



Revient au même

```
let gNombre = 3;

if(gNombre == 1){

    alert("C'est un 1 !");

}
else if(gNombre == 2){

    alert("C'est un 2 !");

}
else if(gNombre == 3){

    alert("C'est un 3 !");

}
```



❖ Les **switch**

◆ À quoi servent les **break** ?

- Avec un **switch**, lorsqu'un **case** est choisi, tout le reste des **case** suivants sont exécutés ... à moins qu'un **break** soit rencontré !

```
let gNombre = 2;

switch(gNombre){

  case 1 : console.log("C'est un 1 !");
  case 2 : console.log("C'est un 2 !");
  case 3 : console.log("C'est un 3 !");
}
```

Inspecteur Console

Filtrer

C'est un 2 !

C'est un 3 !

← undefined

- Puisque **gNombre** vaut **2**, on **exécute** le code associé au **case 2**.
- Comme il n'y a pas de **break**, on continue d'**exécuter** le code du **case** suivant !



❖ Les **switch**

◆ À quoi servent les **break** ?

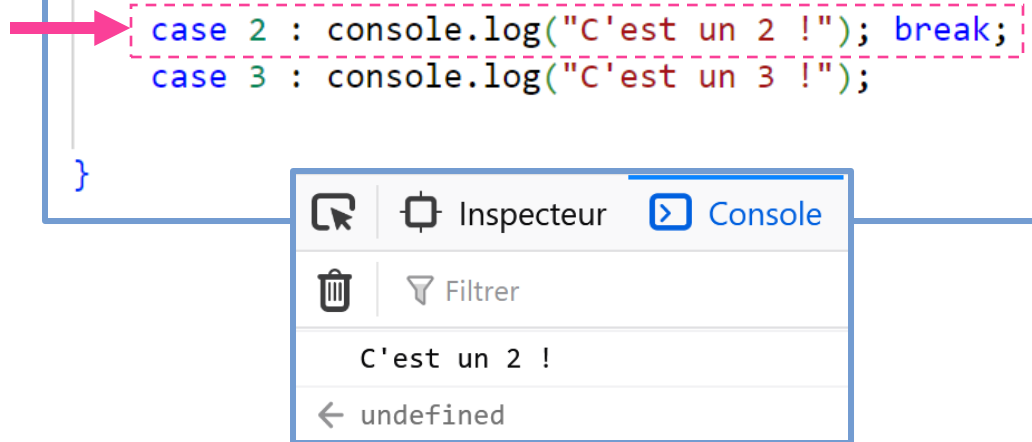
- Avec un **switch**, lorsqu'un **case** est choisi, tout le reste des **case** suivants sont exécutés ... à moins qu'un **break** soit rencontré !

```
let gNombre = 2;

switch(gNombre){

  case 1 : console.log("C'est un 1 !"); break;
  case 2 : console.log("C'est un 2 !"); break;
  case 3 : console.log("C'est un 3 !");
}


```



- Puisque **gNombre** vaut **2**, on **exécute** le code associé au **case 2**.
- Comme un **break** est rencontré, on s'arrête !



❖ Les switch

◆ Cas par défaut

- Si on veut, on peut ajouter un « cas par défaut ». Si aucun case n'est validé, ce sera le code associé au bloc default qui va s'exécuter.

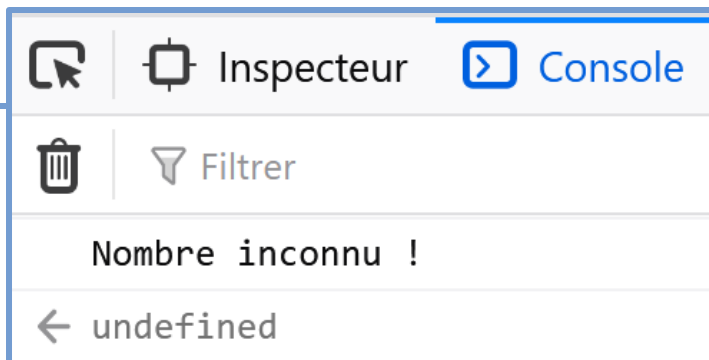
```
let gNombre = -4;

switch(gNombre){

    case 1 : console.log("C'est un 1 !"); break;
    case 2 : console.log("C'est un 2 !"); break;
    case 3 : console.log("C'est un 3 !"); break;
    default : console.log("Nombre inconnu !");

}
```

- Comme -4 ne correspond ni à 1, ni à 2, ni à 3, on exécute le code associé au bloc default.





❖ Les switch

- ◆ On peut aussi utiliser des chaînes de caractères.

```
let gTypeProjectile = "Boule de feu";

switch(gTypeProjectile){

    case "Balle de plomb" : gPointsDeVie -= 70; break;
    case "Boule de feu" : gPointsDeVie -= 60; break;
    case "Jet d'eau" : gPointsDeVie -= 2; break;
    case "Piano" : gPointsDeVie -= 200; break;
    default : gPointsDeVie -= 10;

}
```



❖ Les conditions ternaires

- ◆ Grossièrement, ce sont des « if ... else » miniatures pour choisir une valeur selon une condition.

Ces symboles sont importants

```
let message = gNote >= 60 ? "Tu passes le cours !" : "Hmm... mauvaise nouvelle.;"
```

Condition Valeur si true Valeur si false

- Ceci est l'équivalent si on avait utilisé un if ... else.

```
let message;  
  
if(gNote >= 60){  
|   message = "Tu passes le cours !";  
}  
else{  
|   message = "Hmm... mauvaise nouvelle.;"  
}
```



❖ Les conditions ternaires

◆ Autre exemple

- On peut se servir d'une condition ternaire pour retourner une valeur.

```
function minimum(x, y){  
    return x < y ? x : y;  
}
```

Revient au
même

```
function minimum(x, y){  
    if(x < y){  
        return x;  
    }  
    else{  
        return y;  
    }  
}
```



❖ Créer un élément HTML

◆ Étape 1 : Le préparer

- Pour le moment, c'est un `<p></p>` vide qui n'a pas encore été ajouté dans la page Web.

```
let nouveauP = document.createElement("p");
```

◆ Étape 2 : Le personnaliser

- On peut lui ajouter du texte, des attributs, des classes, etc.

```
nouveauP.textContent = "J'aime vraiment beaucoup les chaises";  
nouveauP.classList.add("louche");  
nouveauP.title = "Phrase louche";
```

Ici il faut mettre le **type d'élément** :
"p", "img", "div", etc...

◆ Étape 3 : L'ajouter dans la page Web

- Pour cela, il faut le « glisser dans un **élément parent** de notre choix ».

```
document.querySelector(".container").appendChild(nouveauP);
```

Élément parent



❖ Créer un élément HTML

◆ Pour récapituler :

```
// Étape 1 : Créer l'élément
let nouveauP = document.createElement("p");

// Étape 2 : Le personnaliser
nouveauP.textContent = "J'aime vraiment beaucoup les chaises";
nouveauP.classList.add("louche");
nouveauP.title = "Phrase louche";

// Étape 3 : L'ajouter dans la page
document.querySelector(".container").appendChild(nouveauP);
```

Élément parent

```
<div class="container">
```

Tada ! 🤖

```
<p class="louche" title="Phrase louche">J'aime vraiment beaucoup les chaises</p>
</div>
```



❖ Créer un élément HTML

- ◆ Exemple pour créer un élément ``

```
let uneImage = document.createElement("img");  
  
uneImage.classList.add("monImage");  
uneImage.src = "images/crotteDeFromage.png";  
uneImage.alt = "Crotte de fromage";  
  
document.querySelector(".galerie").appendChild(uneImage);
```



```
<div class="galerie">  
    
</div>
```



❖ Créer un élément HTML

◆ Précision pour la fonction `appendChild()`

- `appendChild()` permet d'ajouter le nouvel élément à la fin de l'élément parent choisi.

```
let animal = document.createElement("p");  
  
animal.classList.add("animal4");  
animal.textContent = "Lézard";  
  
document.querySelector(".animaux").appendChild(animal);
```

```
<div class="animaux">  
  <p class="animal1">Chien</p>  
  <p class="animal2">Vache</p>  
  <p class="animal3">Ornithorynque</p>  
  <p class="animal4">Lézard</p>  
</div>
```

Il y avait déjà trois éléments dans `.animaux`. Le nouvel élément (`.animal4`) a été ajouté en-dessous de ceux-ci.



❖ Supprimer un élément HTML

◆ On peut aussi **supprimer** des éléments HTML

- Il suffit de trouver l'élément HTML avec **querySelector** et d'utiliser la fonction **.remove()**.
- L'élément HTML sera alors **retiré de la page Web**. (Donc du **DOM**)

```
let element = document.querySelector(".mario");  
element.remove();
```

Par exemple, je supprime l'élément avec la classe **.mario**

```
let elements = document.querySelectorAll(".animal");  
elements[3].remove();  
elements.splice(3, 1);
```



Par exemple, j'obtiens un tableau avec tous les éléments qui possèdent la classe **.animal**, puis je supprime celui à l'**index 3** dans le tableau. (Donc le 4^e dans la page Web) Attention de ne pas oublier de supprimer l'élément du tableau avec **splice()** ensuite ! On ne veut pas garder un élément HTML « zombie 🧟 » dans notre tableau.



❖ Les variables « vides »

- ◆ Lorsqu'une variable est déclarée, mais qu'aucune valeur ne lui est affectée, elle est « **undefined** » :

```
>> let x;  
← undefined  
  
>> x  
← undefined
```

Comme on n'a mis aucune valeur dans la variable **x**, elle est **undefined**.

```
>> let y = 2;  
← undefined  
  
>> y  
← 2
```

On a mis une valeur dans la variable **y**, elle n'est **pas undefined**.

Généralement, on n'aime pas qu'une variable soit **undefined**. On doit lui donner une valeur dès que possible pour éviter que cela génère des problèmes !

- Par exemple, ici, on additionne **x** et **y** alors que **x** est **undefined**.
- On obtient **NaN** (Not a Number), qui n'était sûrement pas le résultat qu'on espérait.

```
>> let x;  
    let y = 1;  
← undefined  
  
>> x + y  
← NaN
```



❖ Les variables « vides »

- ◆ Il existe également des variables « **null** ». Contrairement à une variable **undefined**, lorsqu'une variable est **null**, c'est généralement volontaire plutôt que d'être une **maladresse**.

- Par exemple, parmi les **variables globales**, on a déclaré **gPlanifJeu**, qui servira à stocker un **planificateur à intervalles**. Initialement, on lui donne la valeur « **null** », pour indiquer clairement qu'il n'y a pas encore de **planificateur** stocké dans cette variable.

- Au moment de créer et stocker le **planificateur**, on pourrait commencer par vérifier que **gPlanifJeu** est **null** (donc vide) pour s'assurer de ne pas écraser un autre **planificateur** existant.

```
let gPlanifJeu = null;
```

```
function lancerJeu(){  
    ....if(gPlanifJeu == null){  
    ....|   gPlanifJeu = setInterval(gPlanifJeu, 50);  
    ....}  
    ....else{  
    ....|   console.log("Il y a déjà un planificateur de stocké !");  
    ....}  
    ....  
}
```

Supprimer un élément HTML



- ❖ Vérifier qu'un élément HTML existe avant de le supprimer
 - ◆ Nous avons vu comment supprimer un élément HTML d'une page Web :

```
let element = document.querySelector(".mario");  
element.remove();
```

Par exemple, je supprime l'élément avec la classe `.mario`

- ◆ Cela dit, il arrive qu'on essaye de supprimer un élément qui n'existe pas :

- Ici, on tente d'aller récupérer l'élément avec la classe `.mario`, mais on réalise ensuite que la variable `element` contient `null` : ça veut dire qu'**aucun élément avec cette classe n'existe** ! (Ou qu'on a mal écrit la `classe`)

```
>> let element = document.querySelector(".mario");  
← undefined  
  
>> element  
← null
```

- Si on tente de le supprimer, cela provoque une `erreur`.

```
>> element.remove();  
! ▶ Uncaught TypeError: element is null
```



- ❖ Vérifier qu'un élément HTML existe avant de le supprimer
 - ◆ Pour éviter ce problème, lorsqu'on n'est pas sûr qu'un élément existe, on peut le supprimer comme ceci :

- Grâce au **if**, on tente seulement de supprimer l'élément s'il n'est pas **null**, donc seulement s'il existe bel et bien.

```
let element = document.querySelector(".etoile");  
  
if(element != null){  
    element.remove();  
}
```