JS

Les fondamentaux du javascript

Boucles : while et for

Boucles : while et for

Nous avons souvent besoin d'effectuer des actions similaires plusieurs fois de suite.
Par exemple, lorsque nous devons extraire des marchandises d'une liste les unes à la suite des autres. Ou exécutez simplement le même code pour chaque numéro de 1 à 10.
Les boucles permettent de répéter plusieurs fois la même partie du code.

La boucle "while"
La boucle **while** a la syntaxe suivante :

```
while (condition) {
  // code
  // appelé "loop body" ("corps de boucle")
}
```

Tant que la condition est vraie, le code du corps de la boucle est exécuté.

Par exemple, la boucle ci-contre affiche **i** tant que **i < 3** :

```
let i = 0;
while (i < 3) { // affiche 0, puis 1, puis 2
  alert( i );
  i++;
}
```

Boucles : while et for

Une unique exécution du corps de la boucle est appelée une itération.
La boucle dans l'exemple ci-dessus fait trois itérations.

S'il n'y avait pas d'**i++** dans l'exemple ci-dessus, la boucle se répèterait (en théorie) pour toujours.
En pratique, le navigateur fournit des moyens d'arrêter ces boucles, et pour JavaScript côté serveur, nous pouvons tuer le processus.

Toute expression ou variable peut être une condition de boucle, pas seulement une comparaison. Ils sont évalués et convertis en un booléen par **while**.

Par exemple, le moyen le plus court d'écrire **while (i != 0)** pourrait être **while (i)** :

```
let i = 0;
while (i) { // quand i devient 0, la condition devient fausse et la boucle s'arrête
  alert( i );
  i--;
}
```

Boucles : while et for

Les accolades ne sont pas requis pour un corps à une seule ligne
Si le corps de la boucle a une seule déclaration, nous pouvons omettre les accolades {...} :

```
let i = 0;
while (i)  alert( i-- );
```

La boucle "do...while"
La vérification de la condition peut être déplacée sous le corps de la boucle en utilisant la syntaxe do..while

```
do {
  // corps de la boucle
} while (condition);
```

La boucle exécute d'abord le corps, puis vérifie la condition et, tant que c'est vrai, l'exécute encore et encore.

Par exemple:

```
let i = 0;
do {
  alert( i );
  i++;
} while (i < 3);
```

Boucles : while et for

Cette forme de syntaxe est rarement utilisée, sauf lorsque vous souhaitez que le corps de la boucle s'exécute au moins une fois, quelle que soit la condition. Habituellement, l'autre forme est préférée : while(...) {...}.

La boucle "for"

La boucle for est plus complexe, mais c'est aussi la boucle la plus utilisée.
Cela ressemble à ceci :

```
for (début; condition; étape) {
  // ... corps de la boucle ...
}
```

Apprenons la signification de ces parties par l'exemple.
La boucle ci-dessous exécute alert(i) pour i en partant de 0 jusqu'à 3 (mais non compris) :

```
for (let i = 0; i < 3; i++) { // affiche 0, puis 1, puis 2
  alert(i);
}
```

Boucles : while et for

Examinons la déclaration for partie par partie :

partie		
début	let i = 0	Exécute une fois en entrant dans la boucle.
condition	i < 3	Vérifié avant chaque itération de la boucle, en cas d'échec, la boucle s'arrête.
corps	alert(i)	Exécute encore et encore tant que la condition est vraie
étape	i++	Exécute après le corps à chaque itération

L'algorithme de boucle général fonctionne comme ceci :

Exécuter le début
→ (si condition → exécuter le corps et exécuter l'étape)
→ (si condition → exécuter le corps et exécuter l'étape)
→ (si condition → exécuter le corps et exécuter l'étape)
→ ...

C'est-à-dire que begin est exécuté une fois, puis itéré : après chaque test de condition, body et step sont exécutés.

Boucles : while et for

Si vous débutez dans les boucles, il pourrait être utile de revenir à l'exemple et de reproduire comment elle s'exécute pas à pas sur une feuille de papier.
Voici ce qui se passe exactement dans notre cas :

```
// for (let i = 0; i < 3; i++) alert(i)
// exécute début
let i = 0
// si condition → exécuter le corps et exécuter l'étape
if (i < 3) { alert(i); i++ }
// si condition → exécuter le corps et exécuter l'étape
if (i < 3) { alert(i); i++ }
// si condition → exécuter le corps et exécuter l'étape
if (i < 3) { alert(i); i++ }
// ... fini, parce que maintenant i == 3
```

Déclaration de variable en ligne

Ici, la variable "counter" i est déclarée directement dans la boucle. Cela s'appelle une déclaration de variable "en ligne". De telles variables ne sont visibles que dans la boucle.

Boucles : while et for

Si vous débutez dans les boucles, il pourrait être utile de revenir à l'exemple et de reproduire comment elle s'exécute pas à pas sur une feuille de papier.
Voici ce qui se passe exactement dans notre cas :

```
for (let  i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
alert(i); // erreur, pas de variable
```

Au lieu de définir une variable, nous pouvons en utiliser une existante :

```
let i = 0
for (let  i = 0; i < 3; i++) {
  alert(i); // 0, 1, 2
}
alert(i); // 3, visible, car déclaré en dehors de la boucle
```

Boucles : while et for

Sauter des parties

Toute partie de for peut être ignorée.

Par exemple, nous pouvons omettre le début si nous n'avons rien à faire au début de la boucle.

Comme ici :

```
let i = 0; // nous avons i déjà déclaré et assigné
for (; i < 3; i++) {
  alert(i); // 0, 1, 2
}
```

Nous pouvons également supprimer la partie étape :

```
let i = 0; // nous avons i déjà déclaré et assigné
for (; i < 3;) {
  alert(i++);
}
```

La boucle est devenue identique à while (i < 3).

Nous pouvons tout supprimer, créant ainsi une boucle infinie :

Boucles : while et for

```
for (;;) {
 // répète sans limite
}
```

Veuillez noter que les deux les points-virgules ; de for doivent être présents, sinon ce serait une erreur de syntaxe.

Briser la boucle

Normalement, la boucle sort quand la condition devient fausse.

Mais nous pouvons forcer la sortie à tout moment. Il y a une directive spéciale appelée break pour cela.

Par exemple, la boucle ci-dessous demande à l'utilisateur une série de chiffres, mais "se casse" quand aucun numéro n'est entré :

```
let sum = 0;
while (true) {
  let value = +prompt("Entrez un nombre", '');
  if (!value) break; // (*)
  sum += value;
}
alert( 'Sum: ' + sum );
```

Boucles : while et for

La directive break est activée sur la ligne (*) si l'utilisateur entre une ligne vide ou annule l'entrée. Il arrête la boucle immédiatement, en passant le contrôle à la première ligne après la boucle. À savoir, alert.

La combinaison "boucle infinie + break au besoin" est idéale pour les situations où la condition doit être vérifiée non pas au début / à la fin de la boucle, mais au milieu, voire à plusieurs endroits du corps.

Continuer jusqu'à la prochaine itération

La directive continue est une "version plus légère" de break. Cela n'arrête pas toute la boucle. Au lieu de cela, elle arrête l'itération en cours et force la boucle à en démarrer une nouvelle (si la condition le permet).

Nous pouvons l'utiliser si nous avons terminé l'itération en cours et aimerions passer à la suivante.

La boucle ci-dessous utilise continue pour ne produire que des valeurs impaires :

Boucles : while et for

```
for (let i = 0; i < 10; i++) {
  // si vrai, saute le reste du corps
  if (i % 2 == 0) continue;
  alert(i); // 1, ensuite 3, 5, 7, 9
}
```

Pour les valeurs paires de i, la directive continue arrête l'exécution du corps en passant le contrôle à la prochaine itération de for (avec le nombre suivant). Donc, l'alert n'est appelée que pour les valeurs impaires.

La directive `continue` aide à réduire le niveau d'imbrication

Une boucle affichant des valeurs impaires pourrait ressembler à ceci :

```
for (let i = 0; i < 10; i++) {
  if (i % 2 ) {
    alert( i );
  }
}
```

D'un point de vue technique, c'est identique à l'exemple du dessus.
Certes, nous pouvons simplement envelopper le code dans un bloc if au lieu de continue.
Mais comme effet secondaire, nous avons obtenu un niveau d'imbrication supplémentaire (l'appel de l'alert à l'intérieur des accolades). Si le code à l'intérieur du if est plus long que quelques lignes, la lisibilité globale peut en être réduite.

JS

Exercices

Boucles : while et for

Extraire les nombres pairs dans la boucle

Utilisez la boucle for pour afficher les nombres pairs de 2 à 10.

```
for (let  i = 2; i <= 10; i++) {
  if (i % 2 == 0) {
    alert( i );
  }
}
```

Boucles : while et for

Remplacer "for" par "while"

Réécrivez le code en modifiant la boucle for en while sans modifier son comportement (la sortie doit rester la même).

```
for (let i = 0; i < 3; i++) {
  alert( `number ${i}!` );
}
```

```
let i = 0;
while (i < 3) {
  alert( `number ${i}!` );
  i++;
}
```

JS

Les fondamentaux du javascript

La déclaration "switch"

La déclaration "switch"

Une instruction switch peut remplacer plusieurs vérification if.
Cela donne un moyen plus descriptif de comparer une valeur avec plusieurs variantes.

La syntaxe

Le switch a un ou plusieurs blocs case (cas) et une valeur par défaut facultative.

Cela ressemble à ceci :

```
switch(x) {
  case 'value1':  // si (x === 'value1')
    ...
    [break]
  case 'value2':  // si (x === 'value2')
    ...
    [break]
  default:
    ...
    [break]
}
```

La déclaration "switch"

La valeur de x est vérifiée pour une égalité stricte avec la valeur du premier case (c'est-à-dire, value1), puis du second (value2) et ainsi de suite.
Si l'égalité est trouvée, switch commence à exécuter le code à partir du case correspondant, jusqu'au prochain break (ou jusqu'à la fin du switch).
Si aucun cas ne correspond, le code par défaut (default) est exécuté (s'il existe).

Un exemple
Un exemple de switch (le code exécuté est mis en évidence):
Ici, le switch commence à comparer a avec le premier case dont la valeur est 3. La correspondance échoue.

Ensuite 4, c'est une correspondance. L'exécution commence donc à partir du case 4 jusqu'au prochain break.

```
let a = 2 + 2;
switch (a) {
  case 3:
    alert( 'Trop petit' );
    break;
  case 4:
    alert( 'Exact !' );
    break;
  case 5:
    alert( 'Trop grand' );
    break;
  default:
    alert( 'Connais pas cette valeur ?' );
}
```

La déclaration "switch"

S'il n'y a pas de break, l'exécution continue avec le case suivant sans aucun contrôle.

Un exemple sans break:

```
let a = 2 + 2;
switch (a) {
  case 3:
    alert( 'Trop petit' );
  case 4:
    alert( 'Exact !' );
  case 5:
    alert( 'Troo grand' );
  default:
    alert( 'Connais pas cette valeur ?' );
}
```

Dans l'exemple ci-dessus, nous verrons l'exécution séquentielle de trois alert :

```
    alert( 'Exact !' );
    alert( 'Trop grand' );
    alert( 'Connais pas cette valeur ?' );
```

La déclaration "switch"

Toute expression peut être un argument switch/case
Switch et case permettent des expressions arbitraires.

Par exemple :

```
let a = "1";
let b = 0;

switch (+a) {
 case b + 1:
  alert("Passe, car +a est 1, et égal à b+1");
  break;

 default:
  alert("Ne passe pas");
}
```

Ici +a donne 1, qui est comparé à b + 1 dans le case, et le code correspondant est exécuté.

La déclaration "switch"

Groupement de “case”

Plusieurs variantes de case partageant le même code peuvent être regroupées.

Par exemple, si nous voulons que le même code soit exécuté pour les case 3 et case 5 :

```
let a = 2 + 2;
switch (a) {
  case 4:
    alert('Right!');
    break;
  case 3: // (*) grouped two cases
  case 5:
    alert('Wrong!');
    alert("Why don't you take a math class?");
    break;
  default:
    alert('Le résultat est étrange');
}
```

La déclaration "switch"

Maintenant, les 3 et 5 affichent le même message.
La possibilité de "grouper" les case est un effet secondaire de la façon dont le switch/case fonctionne sans break. Ici, l'exécution du case 3 commence à partir de la ligne (*) et passe par le case 5, car il n'y a pas de break.

Le type compte
Soulignons que le contrôle d'égalité est toujours strict.
Les valeurs doivent être du même type pour correspondre.

Par exemple, considérons le code suivant :

1. Pour 0, 1, la première alert est exécutée.
2. Pour 2, la deuxième alert est exécutée.
3. Mais pour 3, le résultat du prompt est une chaîne de caractères "3", ce qui n'est pas strictement égal === au chiffre 3. Nous avons donc un code mort dans le case 3 ! La variante par défaut sera donc exécutée.

```
let arg = prompt("Enter a value?");
switch (arg) {
  case '0':
  case '1':
    alert( 'One or zero' );
    break;
  case '2':
    alert( 'Two' );
    break;
  case 3:
    alert( 'Never executes!' );
    break;
  default:
    alert( 'An unknown value' );
}
```

JS

Exercices

Boucles : while et for

Réécrire le "if" dans un "switch"

Réécrivez le code ci-dessous en utilisant une seule instruction switch :

```
let a = +prompt('a?', '');

if (a == 0) {
  alert( 0 );
}
if (a == 1) {
  alert( 1 );
}

if (a == 2 || a == 3) {
  alert( '2,3' );
}
```

Boucles : while et for

Réécrire le "if" dans un "switch"
Les deux premiers contrôles se transforment en deux case. Le troisième contrôle est divisé en deux case :

Remarque: *le break en bas n'est pas requis. Mais nous le mettons pour rendre le code à l'épreuve du temps.*

Dans le futur, il est possible que nous voulions ajouter un case supplémentaire, par exemple le case 4.
Et si nous oublions d'ajouter un break avant, à la fin du case 3, il y aura une erreur. *C'est donc une sorte d'assurance.*

```
let a = +prompt('a?', '');

switch(a) {
  case 0:
    alert( 0 );
    break;
  case 1:
    alert( 1 );
    break;
  case 2:
  case 3:
    alert( '2,3'  );
    break;
}
```

JS

Les fondamentaux du javascript

Fonctions

Fonctions

Très souvent, nous devons effectuer une action similaire à plusieurs endroits du script.
Par exemple, nous devons afficher un beau message lorsqu'un visiteur se connecte, se déconnecte et peut-être ailleurs.
Les fonctions sont les principales "composantes" du programme. Ils permettent au code d'être appelé plusieurs fois sans répétition.
Nous avons déjà vu des exemples de fonctions intégrées, telles que alert(message), prompt(message, default) et confirm(question). Mais nous pouvons aussi créer nos propres fonctions.

Déclaration de fonction
Pour créer une fonction, nous pouvons utiliser une déclaration de fonction.
Cela ressemble à ceci :

```
function showMessage() {
  alert( 'Hello everyone!' );
}
```

Fonctions

Le mot-clé function commence en premier, puis le nom de la fonction, puis une liste de paramètres entre les parenthèses (séparés par des virgules, vides dans l'exemple ci-dessus, nous verrons des exemples plus tard) et enfin le code de la fonction, également appelé "le corps de la fonction", entre des accolades.

```
function name(param1,param2,... paramN) {
  // Body
}
```

Notre nouvelle fonction peut être appelée par son nom : showMessage().
Par exemple :

```
function showMessage() {
  alert( 'Hello everyone!' );
}

showMessage()
showMessage()
```

L'appel showMessage() exécute le code de la fonction. Ici, nous verrons le message deux fois, parce qu'on l'appelle deux fois.
Cet exemple illustre clairement l'un des principaux objectifs des fonctions: *éviter la duplication de code.*
Si nous devons un jour modifier le message ou son affichage, il suffit de modifier le code à un endroit: la fonction qui le renvoie.

Fonctions

Variables locales

Une variable déclarée à l'intérieur d'une fonction n'est visible qu'à l'intérieur de cette fonction. Par exemple :

```
function showMessage() {
 let message = 'Hello le JS' );
 // Variable locale
  alert( message );
}
showMessage() // 'Hello le JS'
 alert( message ); // Erreur
```

Variables externes

Une fonction peut également accéder à une variable externe. Par exemple :

La fonction a un accès complet à la variable externe. Cela peut aussi la modifier.

Par exemple :

```
let userName = 'Jean';
function showMessage() {
 let message = 'Hello' + userName;
 alert( message );
}
showMessage() // 'Hello Jean'
```

Fonctions

```
let userName = 'Jean';
function showMessage() {
userName = 'Pierre';  // (1) Change la variable externe
 let message = 'Hello' + userName;
 alert( message );
}
alert( userName ); // Jean avant l'appel de fonction
showMessage()
alert( userName ); // Pierre, la valeur a été modifiée par la fonction
```

La variable externe n'est utilisée que s'il n'y a pas de variable locale.

Si une variable du même nom est déclarée à l'intérieur de la fonction, elle eclipsera la variable externe. Par exemple, dans le code ci-dessous, la fonction utilise le nom userName local. L'externe est ignoré :

Fonctions

```
let userName = 'Jean';
function showMessage() {
 let userName = 'Pierre';  // Déclare une variable locale
 let message = 'Hello' + userName; // Pierre
 alert( message );
}
// la fonction créera et utilisera son propre userName
showMessage()
alert( userName ); // Jean, inchangé, la fonction n'a pas accédé à la variable locale
```

Variables globales

Les variables déclarées en dehors de toute fonction, telle que userName externe dans le code ci-dessus, sont appelées globales. Les variables globales sont visibles depuis n'importe quelle fonction (sauf si elles sont masquées par les variables locales).

C'est une bonne pratique de minimiser l'utilisation de variables globales. Le code moderne a peu ou pas de variable globales. La plupart des variables résident dans leurs fonctions. Parfois, cependant, ils peuvent être utiles pour stocker des données au niveau du projet.

Fonctions

Arguments

Nous pouvons transmettre des données arbitraires à des fonctions à l'aide de paramètres.
Dans l'exemple ci-dessous, la fonction a deux paramètres: from et text.

```
function showMessage( from, text ) { // arguments : from, text
 alert( from + ': ' + text );
}
showMessage('Anne','Bonjour !')
showMessage('Anne','Quoi de neuf ?')
```

Lorsque la fonction est appelée dans les lignes (*) et (**), les valeurs données sont copiées dans les variables locales from et text. Ensuite, la fonction les utilise.
Voici un autre exemple: nous avons une variable from et la transmettons à la fonction. Remarque : la fonction change from, mais le changement n'est pas visible à l'extérieur, car une fonction obtient toujours une copie de la valeur :

```
function showMessage( from, text ) {
 from = '*' + from + '*'; // "from" amélioré
 alert( from + ': ' + text );
}
let from = 'Anne';
showMessage(from ,'Bonjour !');
// la valeur de "from" est la même, la fonction
a modifié une copie locale
alert( from ); // Anne;
```

Fonctions

Lorsqu'une valeur est passée en tant que paramètre de fonction, elle est également appelée argument.
En d'autres termes, pour mettre ces termes au clair :
Un paramètre est la variable répertoriée entre parenthèses dans la fonction déclaration (c'est un terme du temps de la déclaration).
Un argument est la valeur qui est transmise à la fonction lorsqu'elle est appelée (c'est un terme du temps de l'appel).
Nous déclarons des fonctions en listant leurs paramètres, puis les appelons en passant des arguments.
Dans l'exemple ci-dessus, on pourrait dire : "la fonction showMessage est déclarée avec deux paramètres, puis appelée avec deux arguments : from et "Hello".

Les valeurs par défaut

Si une fonction est appelée, mais qu'aucun argument n'est fourni, alors la valeur correspondante devient **undefined**.
Par exemple, la fonction showMessage(from, text) mentionnée précédemment peut être appelée avec un seul argument :

```
showMessage('Anne')
```

Fonctions

Ce n'est pas une erreur. Un tel appel produirait "*Ann*: undefined". Comme la valeur de text n'est pas transmise, elle devient undefined.
Nous pouvons spécifier la valeur dite "par défaut" (à utiliser si omise) pour un paramètre dans la déclaration de fonction, en utilisant **=** :

```
function showMessage(from, text = "no param") {
  alert( from + ": " + text );
}
showMessage('Anne') // Anne: no param
```

Maintenant, si le paramètre text n'est pas passé, il obtiendra la valeur "no param".
La valeur par défaut saute également si le paramètre existe, mais est strictement égal à undefined, comme ceci :

```
showMessage('Anne', undefined ) // Anne: no param
```

Ici, "no param" est une chaîne de caractères, mais il peut s'agir d'une expression plus complexe, qui n'est évaluée et affectée que si le paramètre est manquant. Donc, cela est également possible :

Fonctions

```
function showMessage( from, text = anotherFunction()) {
  // anotherFunction() est exécuté uniquement si aucun texte n'est fourni
  // son résultat devient la valeur de text
}
```

Évaluation des paramètres par défaut

En JavaScript, un paramètre par défaut est évalué chaque fois que la fonction est appelée sans le paramètre correspondant.

Dans l'exemple ci-dessus, anotherFunction() n'est pas du tout appelé, si le paramètre text est fourni.

D'un autre côté, il est appelé indépendamment à chaque fois que text est manquant.

Paramètres par défaut dans l'ancien code JavaScript

Il y a plusieurs années, JavaScript ne prenait pas en charge la syntaxe des paramètres par défaut.

Les gens ont donc utilisé d'autres moyens pour les spécifier.

De nos jours, on peut les croiser dans d'anciens scripts.

Par exemple, une vérification explicite pour undefined :

```
function showMessage( from, text ) {
  if( text === undefined ){
    text = 'no text given';
  }
  alert( from + ": " + text );
}
```

Fonctions

...Ou en utilisant l'opérateur || :

```
function showMessage( from, text ) {
// Si la valeur du texte est fausse, attribuez la valeur par défaut
// cela suppose que text == "" est identique à pas de texte du tout
  text = text || 'no param';
  ...
}
```

Paramètres par défaut alternatifs

Il est parfois judicieux de définir des valeurs par défaut pour les paramètres non pas dans la fonction déclaration, mais à un stade ultérieur, lors de son exécution. Nous pouvons vérifier si le paramètre est passé lors de l'exécution de la fonction, en le comparant avec undefined :

```
function showMessage( text ) {
  // ...
  if( text === undefined ){ // si le paramètre est manquant
    text = 'Message vide';
  }
  alert( text );
}
showMessage( ); // Message vide
```

Fonctions

Renvoyer une valeur

Une fonction peut renvoyer une valeur dans le code appelant en tant que résultat.
L'exemple le plus simple serait une fonction qui additionne deux valeurs :

```
function sum(a,b) {
  return a + b
}
let result = sum(1,2);
alert( result ); // 3
```

La directive **return** peut être n'importe où dans la fonction. Lorsque l'exécution le permet, la fonction s'arrête et la valeur est renvoyée au code appelant (affecté à result ci-dessus).
Il peut y avoir plusieurs occurrences de return dans une seule fonction.
Par exemple :

```
function checkAge(age) {
if( age >= 18 ){
  return true
} else {
  return confirm('Avez vous la permission de vos parents ?')
}
}
let age = +prompt('Quel est votre age ?','18');
if( checkAge( age ) ){
  alert( 'Accès autorisé' )
} else {
  alert( 'Accès refusé' )
}
```

Fonctions

Il est possible d'utiliser return sans valeur. Cela entraîne la sortie immédiate de la fonction.
Par exemple :
Dans le code ci-dessus, si **checkAge(age)** renvoie **false**,
alors **ShowMovie** n'effectuera pas l'alert.

```
function showMovie(age) {
 if( !checkAge(age) ){
   return;
 }
 alert( 'Voici le film ' ) // (*)
}
```

Une fonction avec un return vide ou rien dedans retourne undefined

```
function doNothing() { /* Vide */ }
 alert( doNothing() === undefined ); // true
```

Un **return** vide est également identique à un **return undefined** :

```
function doNothing() { return }
 alert( doNothing() === undefined ); // true
```

Fonctions

Nommer une fonction

Les fonctions sont des actions. Donc, leur nom est généralement un verbe. Il convient de décrire brièvement, mais aussi précisément que possible, le rôle de la fonction. Pour qu'une personne qui lit le code reçoive le bon indice.

C'est une pratique répandue de commencer une fonction avec un préfixe verbal qui décrit vaguement l'action. Il doit exister un accord au sein de l'équipe sur la signification des préfixes.

Par exemple, les fonctions qui commencent par "show" affichent généralement quelque chose.

Fonction commençant par...
"get..." – retourne une valeur,
"calc..." – calcule quelque chose,
"create..." – créer quelque chose,
"check..." – vérifie quelque chose et retourne un booléen, etc.

Exemples de quelques noms :

```
showMessage(..) // Affiche un message
calcSum(..)     // Calucle une somme
createForm(..)  // Créé un formulaire
checkPermission(..) // Vérifie une permission, retourne vrai/faux
```

Fonctions

Nommer une fonction

Les fonctions sont des actions. Donc, leur nom est généralement un verbe. Il convient de décrire brièvement, mais aussi précisément que possible, le rôle de la fonction. Pour qu'une personne qui lit le code reçoive le bon indice.

C'est une pratique répandue de commencer une fonction avec un préfixe verbal qui décrit vaguement l'action. Il doit exister un accord au sein de l'équipe sur la signification des préfixes.

Par exemple, les fonctions qui commencent par "show" affichent généralement quelque chose.

Fonction commençant par...
"get..." – retourne une valeur,
"calc..." – calcule quelque chose,
"create..." – créer quelque chose,
"check..." – vérifie quelque chose et retourne un booléen, etc.

Exemples de quelques noms :

```
showMessage(..) // Affiche un message
calcSum(..)     // Calucle une somme
createForm(..)  // Créé un formulaire
checkPermission(..) // Vérifie une permission, retourne vrai/faux
```

JS

Exercices

Boucles : while et for

Réécrivez la fonction en utilisant '?' ou '||'

La fonction suivante renvoie true si le paramètre age est supérieur à 18.
Sinon, il demande une confirmation et renvoie le résultat.

```
function checkAge(age) {
  if (age > 18) {
    return true;
  } else {
    return confirm('Vos parents sont ils ok ?');
  }
}
```

Boucles : while et for

Réécrivez la fonction en utilisant '?' ou '||'

En utilisant un opérateur point d'interrogation '?' :

```
function checkAge( age ) {
    return (age > 18) ? true : confirm('Vos parents sont ils ok ?');
}
```

En utilisant OU || (la variante la plus courte) :

```
function checkAge( age ) {
    return (age > 18) || confirm('Vos parents sont ils ok ?');
}
```

Notez que les parenthèses autour de age > 18 ne sont pas obligatoires ici. Elles existent pour une meilleure lisibilité.

JS

Les fondamentaux du javascript

Fonctions Expressions

Fonctions Expressions

En JavaScript, une fonction n'est pas une "structure de langage magique", mais un type de valeur particulier.
La syntaxe utilisée précédemment s'appelle une déclaration de fonction :

```
function sayHi( ) {
  alert('Bonjour !');
}
```

Il existe une autre syntaxe pour créer une fonction appelée Expression de Fonction. Cela nous permet de créer une nouvelle fonction au milieu de n'importe quelle expression.
Par exemple :

```
let sayHi = function( ) {
  alert('Bonjour !');
};
```

Ici, nous pouvons voir une variable **sayHi** obtenir une valeur, la nouvelle fonction, créée en tant que **function() { alert("Hello"); }**.
Comme la création de la fonction se produit dans le contexte de l'expression d'affectation (à droite de **=**), il s'agit d'une Fonction Expression. Veuillez noter qu'il n'y a pas de nom après le mot clé **function**. L'omission d'un nom est autorisée pour les fonctions expressions. Ici, nous l'assignons immédiatement à la variable, donc la signification de ces exemples de code est la même : "créer une fonction et la mettre dans la variable sayHi". Dans des situations plus avancées, que nous verrons plus tard, une fonction peut être créée et immédiatement appelée ou planifiée pour une exécution ultérieure, non stockée nulle part, restant ainsi anonyme.

Fonctions Expressions

La fonction est une valeur

Répétons-le : quelle que soit la manière dont la fonction est créée, une fonction est une valeur. Les deux exemples ci-dessus stockent une fonction dans la variable sayHi.

La signification de ces exemples de code est la même : "créer une fonction et la placer dans la variable sayHi".

Nous pouvons même afficher cette valeur en utilisant alert :

```
function sayHi( ) {
  alert('Bonjour !');
}
alert( sayHi ); // affiche le code de la fonction
```

Veuillez noter que la dernière ligne n'exécute pas la fonction, car il n'y a pas de parenthèses après sayHi. Il y a des langages de programmation où toute mention d'un nom de fonction provoque son exécution, mais JavaScript n'est pas comme ça.

En JavaScript, une fonction est une valeur, nous pouvons donc la traiter comme une valeur. Le code ci-dessus montre sa représentation sous forme de chaîne de caractères, qui est le code source.

Certes, une fonction est une valeur spéciale, en ce sens que nous pouvons l'appeler comme cela sayHi().

Mais c'est toujours une valeur. Nous pouvons donc travailler avec comme avec d'autres types de valeurs.

Fonctions Expressions

Nous pouvons copier une fonction dans une autre variable :

```
function sayHi( ) {  // (1) créer
  alert('Bonjour !');
}
let func = sayHi;    // (2) copier
func();  // Bonjour  // (3) exécuter la copie (ça fonctionne)!
sayHi(); // Bonjour  // cela fonctionne toujours aussi (pourquoi pas)
```

Voici ce qui se passe ci-dessus en détail :

1. La Déclaration de Fonction (1) crée la fonction et la place dans la variable nommée sayHi.
2. La ligne (2) la copie dans la variable func. Veuillez noter à nouveau : il n'y a pas de parenthèses après sayHi. S'il y en avait, alors func = sayHi() écrirait le résultat de l'appel sayHi() dans func, et non la fonction sayHi elle-même.
3. Maintenant, la fonction peut être appelée à la fois en tant que sayHi() et func().

Fonctions Expressions

Nous aurions aussi pu utiliser une Fonction Expression pour déclarer sayHi, à la première ligne :

```
let sayHi = function( ) { // (1) créer
  alert('Bonjour !');
};
let func = sayHi;    // (2) copier
func();  // Bonjour  // (3) exécuter la copie (ça fonctionne)!
sayHi(); // Bonjour  // cela fonctionne toujours aussi (pourquoi pas)
```

Tout fonctionnerait de la même manière.

1. La Déclaration de Fonction (1) crée la fonction et la place dans la variable nommée sayHi.
2. La ligne (2) la copie dans la variable func. Veuillez noter à nouveau : il n'y a pas de parenthèses après sayHi. S'il y en avait, alors func = sayHi() écrirait le résultat de l'appel sayHi() dans func, et non la fonction sayHi elle-même.
3. Maintenant, la fonction peut être appelée à la fois en tant que sayHi() et func().

Fonctions Expressions

Fonctions callback (de rappel)

Examinons plus d'exemples de fonctions passées en tant que valeurs et utilisant des expressions de fonction. Nous allons écrire une fonction ask(question, yes, no) avec trois paramètres :
question -> Texte de la question
yes -> Fonction à exécuter si la réponse est "Yes"
no -> Fonction à exécuter si la réponse est "No"

La fonction doit poser la question et, en fonction de la réponse de l'utilisateur, appeler **yes()** ou **no()** :

```
function ask(question, yes, no) {
  if ( confirm(question) ) yes();
  else no();
}
function showOk() {
  alert( "Validé" );
}
function showCancel() {
  alert( "Annulé" );
}

// utilisation: les fonctions showOk, showCancel
sont transmises en tant qu'arguments à ask
ask("D'accord ?", showOk, showCancel);
```

Fonctions Expressions

Fonctions callback (de rappel)

En pratique, ces fonctions sont très utiles. La principale différence entre une demande réelle (ask) et l'exemple ci-dessus est que les fonctions réelles utilisent des moyens d'interagir avec l'utilisateur plus complexes que la simple confirmation (confirm). Dans le navigateur, une telle fonction dessine généralement une belle fenêtre de questions. Mais c'est une autre histoire.

Les arguments showOk et showCancel de ask s'appellent des fonctions callback (fonctions de rappel) ou simplement des callbacks (rappels).

L'idée est que nous passions une fonction et attendions qu'elle soit "rappelée" plus tard si nécessaire. Dans notre cas, showOk devient le rappel pour la réponse "oui" et showCancel pour la réponse "non".

Fonctions Expressions

Quand choisir la fonction déclaration par rapport à la fonction expression ?

En règle générale, lorsque nous devons déclarer une fonction, la première chose à prendre en compte est la syntaxe de la fonction déclaration.
Cela donne plus de liberté dans l'organisation de notre code, car nous pouvons appeler de telles fonctions avant qu'elles ne soient déclarées.

C'est également meilleur pour la lisibilité, car il est plus facile de rechercher la fonction f(...) {...} dans le code que let f = function(...) {...};. Les fonction déclarations sont plus "accrocheuses".

... Mais si une déclaration de fonction ne nous convient pas pour une raison quelconque (nous en avons vu un exemple ci-dessus), alors il convient d'utiliser une Fonction Expression.

JS

Les fondamentaux du javascript

Fonctions fléchées, les bases

Fonctions fléchées, les bases

Il existe une syntaxe plus simple et concise pour créer des fonctions, c'est souvent mieux que les Expressions de Fonction.Les "fonctions fléchées" sont appelées ainsi pour leur syntaxe :

```
let func = (arg1, arg2, ..., argN) => expression;
```

Cela va créer une function func qui accepte les arguments arg1...argN, puis évalue l'expression sur le côté droit et retourne le résultat. C'est donc la version raccourcie de :

```
let func = function(arg1, arg2, ..., argN) {
  return expression;
};
```

Voyons un exemple concret :

```
let sum = (a, b) => a + b;
/* Cette fonction fléchée est la forme
raccourcie de :
let sum = function(a, b) {
  return a + b;
};*/
alert( sum(1, 2) ); // 3
```

Fonctions fléchées, les bases

Comme vous pouvez le voir (a, b) => a + b représente une fonction qui accepte 2 arguments nommés a et b. Lors de l'exécution, elle évalue l'expression a + b et retourne le résultat.
Pour un argument unique, alors les parenthèses autour du paramètre peuvent être omises, rendant la fonction encore plus courte. Par exemple:

```
let double = n => n * 2;
// Similaire à : let double = function(n) { return n * 2 }
alert( double(3) ); // 6
```

S'il n'y a pas d'arguments, les parenthèses seront alors vides, mais elles doivent êtres présentes :

```
let sayHi = () => alert("Hello!");
sayHi();
```

Les fonctions fléchées peuvent être utilisées de la même manière que les Expressions de Fonction.
Par exemple pour créer une fonction dynamiquement :

Fonctions fléchées, les bases

```
let age = prompt("Votre age?", 18);
let welcome = (age < 18) ?
  () => alert("Hello !") :
  () => alert("Salut !");
welcome(); // ok
```

Les fonctions fléchées peuvent paraître étranges et peu lisibles au début, mais cela change rapidement avec les yeux s'habituant à cette structure.
Elles sont très utiles pour des actions sur une ligne et que l'on est juste paresseux d'écrire d'autres mots.

Les fonctions fléchées multiligne

Les fonctions fléchées que nous avons vues jusqu'à présent étaient très simples. Elles ont pris des arguments à gauche de =>, les ont évalués et ont renvoyé l'expression de droite avec elles.
Parfois nous avons besoin de plus de complexité, comme des expressions multiples ou des déclarations. Cela est possible avec des accolades les délimitant. Il faut ensuite utiliser un return à l'intérieur de celles-ci.

Fonctions fléchées, les bases

```
let sum = (a, b) => {  // Les accolades ouvre une fonction multiligne
  let result = a + b;
  return result; // si nous utilisons des accolades,
                 // nous avons besoin d'un "return" explicite
};

alert( sum(1, 2) ); // 3
```

JS

Qualité du code

Débogage dans le navigateur

Débogage dans le navigateur

Avant d'écrire un code plus complexe, parlons de débogage.

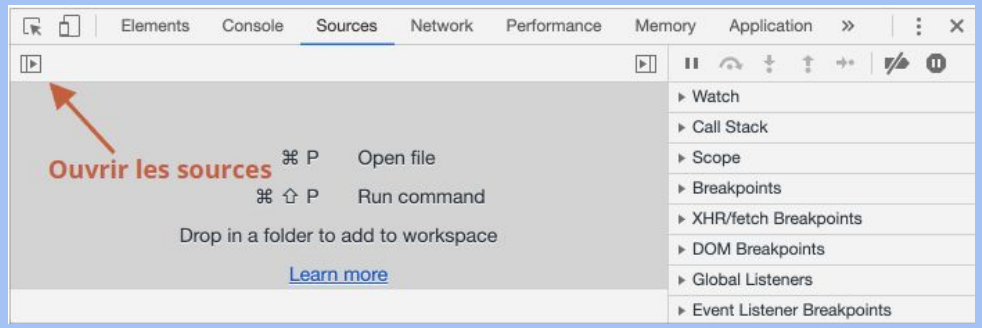
Le Debugging est le processus de recherche et de correction des erreurs dans un script. Tous les navigateurs modernes et la plupart des autres environnements prennent en charge les outils de débogage – une interface utilisateur spéciale dans les outils de développement facilitant grandement le débogage. Cela permet également de tracer le code étape par étape pour voir ce qui se passe exactement.
Nous allons utiliser Chrome ici, car il possède suffisamment de fonctionnalités, la plupart des autres navigateurs utilisent un processus similaire.

Le volet "Sources"
Votre version de Chrome peut sembler un peu différente, mais vous devez tout de même savoir ce qui est là.

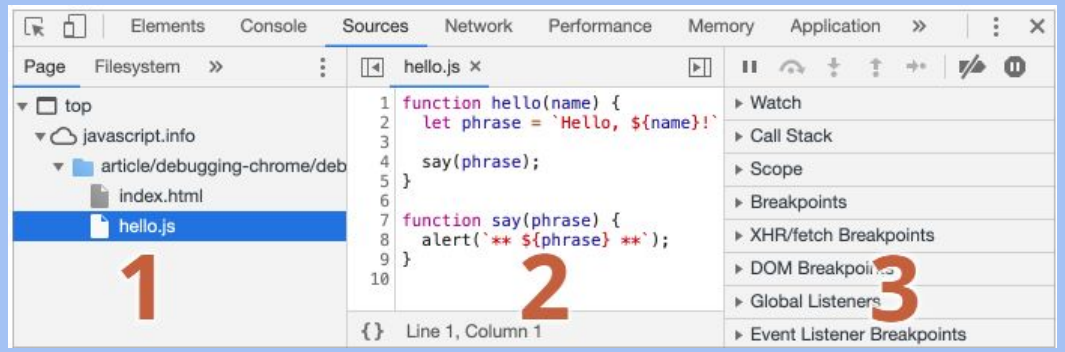
- Ouvrez la page d'exemple dans Chrome, disponible sur le Github SCAP.
- Activer les outils de développement avec F12 (Mac: Cmd+Opt+I).
- Séléctionner le volet Sources.

Débogage dans le navigateur

Voici ce que vous devriez voir si vous le faites pour la première fois :



Le bouton ▶ ouvre le volet avec les fichiers.
Cliquez dessus et sélectionnez hello.js dans l'arborescence. Voici ce qui devrait apparaître :



Débogage dans le navigateur

Ici nous pouvons voir 3 parties :

1. Le volet explorateur de fichiers répertorie les fichiers HTML, JavaScript, CSS et autres fichiers, y compris les images jointes à la page. Des extensions Chrome peuvent également apparaître ici.
2. Le volet Editeur de Code affiche le code source.
3. Le volet Débugueur JavaScript est pour le débogage, nous l'explorerons bientôt.

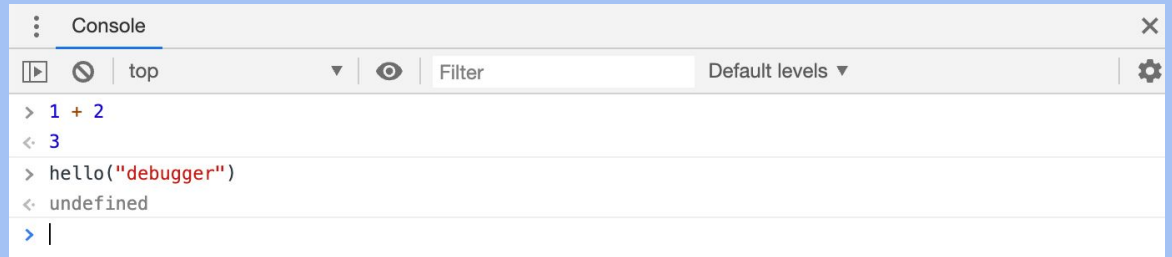
Maintenant, vous pouvez cliquer sur le même bouton à nouveau pour masquer la liste des ressources et laisser un peu d'espace au code.

Console

Si nous appuyons sur Esc, une console s'ouvre ci-dessous. Nous pouvons taper des commandes ici et appuyer sur Entrée pour les exécuter.

Une fois une instruction exécutée, son résultat est présenté ci-dessous.

Par exemple, ici 1+2 donne 3, tandis que l'appel de fonction hello("debugger") ne renvoie rien, donc le résultat est undefined :



Débogage dans le navigateur

Breakpoints

Examinons ce qui se passe dans le code de la page d'exemple. Dans hello.js, cliquez sur le numéro de ligne 4. Oui, sur le chiffre 4, pas sur le code.

Félicitations ! Vous avez défini un point d'arrêt. Veuillez également cliquer sur le numéro correspondant à la ligne 8. Cela devrait ressembler à ceci (le bleu est l'endroit où vous devez cliquer) :

Un breakpoint est un point dans le code où le débogueur mettra automatiquement en pause l'exécution de JavaScript. Pendant que le code est en pause, nous pouvons examiner les variables actuelles, exécuter des commandes dans la console, etc. En d'autres termes, nous pouvons le déboguer.

Débogage dans le navigateur

Nous pouvons toujours trouver une liste de points d'arrêt dans le volet de droite. C'est utile lorsque nous avons plusieurs points d'arrêt dans divers fichiers. Ça nous permet de :

- Sauter rapidement au point d'arrêt dans le code (en cliquant dessus dans le volet de droite).
- Désactiver temporairement le point d'arrêt en le décochant.
- Supprimer le point d'arrêt en cliquant avec le bouton droit de la souris et en sélectionnant Supprimer.
- ... Et ainsi de suite.
-

Points d'arrêt conditionnels

Clic droit sur le numéro de ligne permet de créer un point d'arrêt conditionnel. Il ne se déclenche que lorsque l'expression donnée, que vous devez fournir lors de sa création, est vraie.

C'est pratique lorsque nous devons nous arrêter uniquement pour une certaine valeur de variable ou pour certains paramètres de fonction.

Débogage dans le navigateur

La commande “debugger”

Nous pouvons également suspendre le code en utilisant la commande debugger, comme ceci :

```
function hello(name) {
  let phrase = `Hello, ${name}!`;

  debugger;  // <-- le débogueur s'arrête ici

  hello(phrase);
}
```

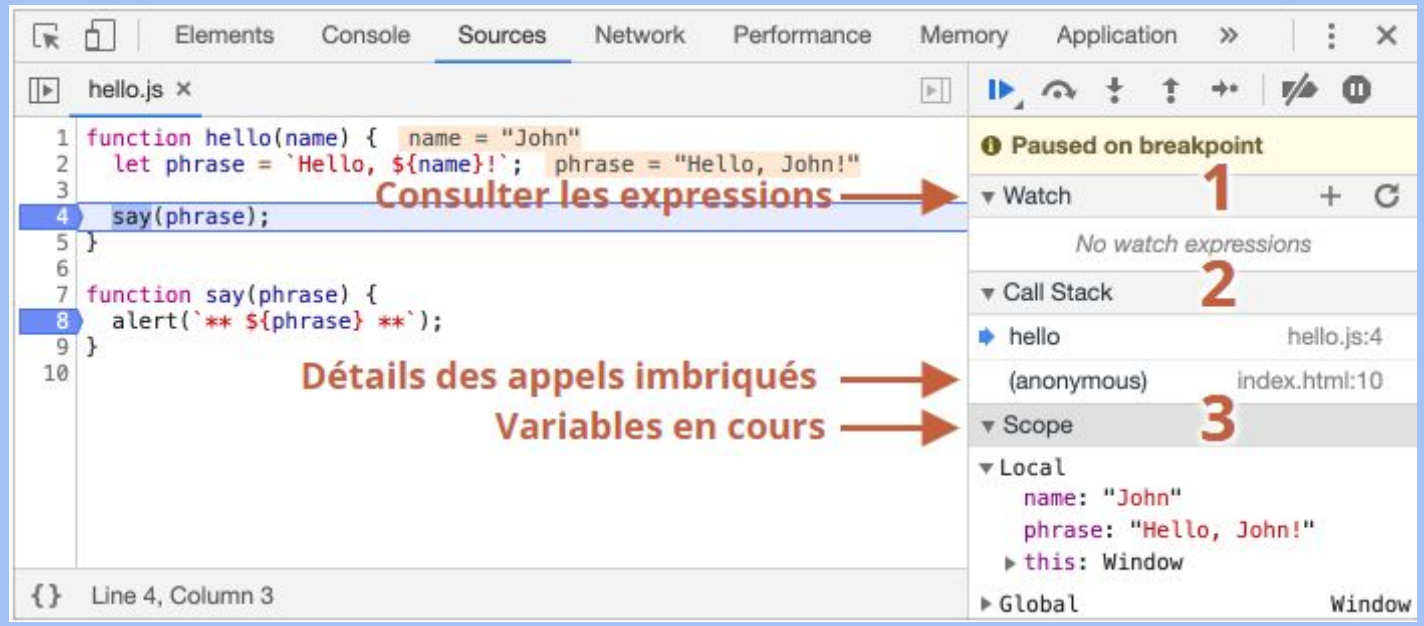
Une telle commande ne fonctionne que lorsque les outils de développement sont ouverts, sinon le navigateur l’ignore.

Pause et regarder autour

Dans notre exemple, hello() est appelé lors du chargement de la page, donc le moyen le plus simple d’activer le débogueur (après avoir défini les points d’arrêt) est de recharger la page. Appuyez donc sur F5 (Windows, Linux) ou sur Cmd+R (Mac).

Débogage dans le navigateur

Lorsque le point d'arrêt est défini, l'exécution s'interrompt à la 4ème ligne :



Veuillez ouvrir les menus déroulants d'information à droite (indiqués par des flèches). Ils vous permettent d'examiner l'état du code actuel :

Débogage dans le navigateur

1. **Watch – affiche les valeurs actuelles pour toutes les expressions.**
   Vous pouvez cliquer sur le plus + et saisir une expression. Le débogueur affichera sa valeur, la recalculant automatiquement dans le processus d'exécution.

2. **Call Stack – affiche la chaîne des appels imbriqués.**
   À ce moment précis, le débogueur se trouve dans l'appel hello(), appelé par un script dans index.html (aucune fonction n'est appelée, elle est donc appelée "anonyme").
   Si vous cliquez sur un élément de la pile (ex: "anonymous"), le débogueur passe au code correspondant, et toutes ses variables peuvent également être examinées.

3. **Scope – variables actuelles.**
   Local affiche les variables de fonction locales. Vous pouvez également voir leurs valeurs surlignées directement sur la source.
   Global a des variables globales (en dehors de toutes fonctions).
   Il y a aussi le mot-clé this que nous n'avons pas encore étudié, mais nous le ferons bientôt.

Débogage dans le navigateur

Tracer l'exécution

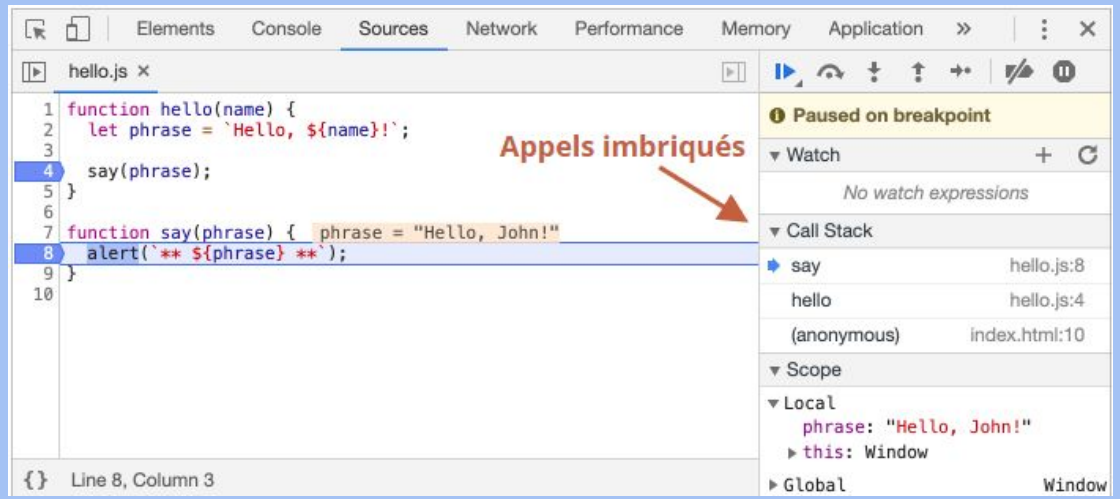
Il est maintenant temps de tracer le script.

Il y a des boutons pour cela en haut du volet de droite. Actionnons-les.

- "Reprendre" : continue l'exécution, raccourci clavier **F8.**

Reprend l'exécution. S'il n'y a pas de points d'arrêt supplémentaires, l'exécution continue et le débogueur perd le contrôle.

Voici ce que nous pouvons voir après un clic dessus :



Débogage dans le navigateur

L'exécution a repris, atteint un autre point d'arrêt à l'intérieur de say() et s'y est arrêtée. Jetez un coup d'œil à "Call stack" à droite. Il a augmenté d'un appel supplémentaire. Nous sommes à l'intérieur say() maintenant.

- "Step": lance la commande suivante, raccourci clavier F9.
  Exécute la prochaine déclaration. Si nous cliquons dessus maintenant, alert sera affiché.
  En cliquant dessus encore et encore, vous parcourrez toutes les instructions de script une par une.
- "Step over": lance la commande suivante, mais n'entre pas dans une fonction, raccourci clavier F10. :
  Similaire à la commande "Step" précédente, mais se comporte différemment si l'instruction suivante est un appel de fonction (pas une fonction intégrée, comme alert, mais une fonction qui nous est propre).
  Si nous les comparons, la commande "Step" entre dans un appel de fonction imbriqué et interrompt l'exécution à sa première ligne, tandis que "Step over" exécute l'appel de fonction imbriqué de manière invisible pour nous, en sautant les fonctions internes.
  L'exécution est alors suspendue immédiatement après cette fonction.
  C'est bien si nous ne sommes pas intéressés à voir ce qui se passe dans l'appel de fonction.

Débogage dans le navigateur

-  **"Step into",** raccourci clavier F11.
  Cela ressemble à "Step", mais se comporte différemment dans le cas d'appels de fonctions asynchrones. Si vous commencez seulement à apprendre le JavaScript, vous pouvez alors ignorer la différence, car nous n'avons pas encore d'appels asynchrones.
  Pour le futur, il suffit de noter que la commande "Step" ignore les actions asynchrones, telles que setTimeout (appel de fonction planifiée), qui s'exécutent ultérieurement. Le "Pas à pas" entre dans leur code, les attend si nécessaire. Voir DevTools manual pour plus de détails.
-  **"Step out":** continuer l'exécution jusqu'à la fin de la fonction en cours, raccourci clavier Shift+F11.
  Continue l'exécution et l'arrête à la toute dernière ligne de la fonction en cours. C'est pratique lorsque nous avons accidentellement entré un appel imbriqué en utilisant , mais cela ne nous intéresse pas et nous voulons continuer jusqu'au bout le plus tôt possible.
-  **active / désactive** tous les points d'arrêt.
  Ce bouton ne déplace pas l'exécution. Juste un ensemble de on/off pour les points d'arrêt.
-  **active/désactive** la pause automatique en cas d'erreur.
  Lorsqu'il est activé, si les outils de développement sont ouverts, une erreur lors de l'exécution du script le met automatiquement en pause. Ensuite, nous pouvons analyser les variables dans le débogueur pour voir ce qui n'a pas fonctionné. Donc, si notre script s'arrête avec une erreur, nous pouvons ouvrir le débogueur, activer cette option et recharger la page pour voir où il s'arrête et quel est le contexte à ce moment-là.

Débogage dans le navigateur

Logging

Pour afficher quelque chose sur la console depuis notre code, utilisez la fonction console.log.
Par exemple, cela affiche les valeurs de 0 à 4 sur la console :

```
// ouvrir la console pour visualiser
for (let i = 0; i < 5; i++) {
  console.log("value,", i);
}
```

Les internautes ne voient pas cette sortie, elle se trouve dans la console. Pour la voir, ouvrez l'onglet Console des outils de développement ou appuyez sur Esc lorsque vous vous trouvez dans un autre onglet : la console en bas s'ouvre.

Si nous avons assez de logging dans notre code, nous pouvons voir ce qui se passe dans les enregistrements, sans le débogueur.

JS

Qualité du code

Commentaires

Les commentaires

Comme nous le savons du chapitre Structure du code, les commentaires peuvent être simples : à partir de // et multiligne : /* ... */.
Nous les utilisons normalement pour décrire comment et pourquoi le code fonctionne.
De prime abord, les commentaires peuvent sembler évidents, mais les novices en programmation les utilisent souvent à tort.

Mauvais commentaires
Les novices ont tendance à utiliser des commentaires pour expliquer "ce qui se passe dans le code". Comme ceci :

```
// Ce code fera cette chose (...) et cette chose (...)
// ...Et qui sait quoi d'autre...
code;
complexe;
```

Mais en bon code, le nombre de ces commentaires "explicatifs" devrait être minime.
Sérieusement, le code devrait être facile à comprendre sans eux.
Il existe une excellente règle à ce sujet: "Si le code est si peu clair qu'il nécessite un commentaire, il devrait peut-être être réécrit".

Les commentaires

Bons commentaires

Ainsi, les commentaires explicatifs sont généralement mauvais. Quels commentaires sont bons ?

Décrivez l'architecture

Fournissez une vue d'ensemble des composants, de leurs interactions, de ce que sont les flux de contrôle dans diverses situations...

Documenter les paramètres de fonction et leur utilisation

Il y a une syntaxe spéciale **JSDoc** pour documenter une fonction : utilisation, paramètres, valeur renvoyée.

Par exemple :

```
/**
 *  Renvoie x élevé à la n-ième puissance.
 * @param {number} x Le nombre à augmenter.
 * @param {number} n L'exposant doit être un nombre naturel.
 * @return {number} x élevé à la n-ème puissance.
 */
function pow(x, n) {
  ...
}
```