insy2s

Programmer avec JavaScript (JS)

Sommaire

Intégrer JavaScript dans vos pages Web

Pour intégrer du JavaScript dans vos pages Web, vous devez d'abord créer un ou plusieurs fichiers avec l'extension **.js**.

Dans notre cours, nous utiliserons un seul fichier .js que nous appellerons *test.js*.

Une fois le fichier créé, vous devez permettre à votre ou vos navigateurs (Chrome, Firefox, Opera, Brave, Safari, etc) d'interpréter votre code JavaScript.

Vous devez donc créer une liaison entre vos fichiers HTML et vos fichiers JS.

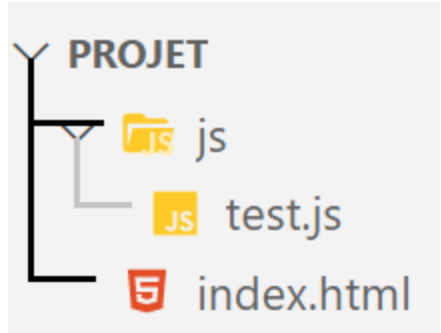
Pour cela vous devez éditer votre ou vos pages HTML avec un éditeur de texte comme Sublime Text ou VS Code, puis, à l'exemple du CSS, ajouter une balise <script></script> (au lieu de <link /> pour le CSS) avec la source du fichier .js

Intégrer JavaScript dans vos pages Web

La ligne de code HTML devra ressembler à ceci :

```
<script src="js/test.js"></script>
```

L'attribut ***src*** devra avoir pour valeur le chemin du fichier test.js à partir de la position du fichier html.



Dans notre exemple on aura un sous-dossier **js** au même niveau qu'index.html, et le fichier test.js se trouvera à l'intérieur pour que la liaison fonctionne.

Pour des raisons de confort et de pratique, il est recommandé de mettre cette ou ces lignes de liaisons <script></script> tout en bas du contenu de votre page web juste avant la balise de fermeture du body dans votre HTML.

Il n'y a plus qu'à ouvrir le fichier test.js avec votre éditeur de texte et vous pouvez commencer à coder en JavaScript!

La variable

Une **variable** est un contenant utilisé pour enregistrer une donnée spécifique dont votre programme a besoin pour travailler. Un nom d'utilisateur, le nombre de billets restants pour un vol, la disponibilité ou non d'un certain produit en stock, toutes ces données sont enregistrées dans des variables.

Une donnée placée dans une variable s'appelle une **valeur**. Si on reprend l'analogie du "carton dans un entrepôt", des cartons différents peuvent enregistrer des valeurs différentes. Vous pouvez par exemple utiliser un carton pour stocker de l'argent pour des dépenses courantes, et un autre pour des économies pour une occasion particulière, par exemple un voyage. Vous pouvez aussi vider les cartons ou modifier leur contenu, en ajoutant par exemple de l'argent ou en en prélevant.

La variable

Pour savoir à quoi sert chaque carton, vous devez les étiqueter. En programmation, c'est la même chose : vous attribuez un **nom** à votre variable. Le nom d'une variable doit indiquer ce qui se trouve à l'intérieur, tout comme les étiquettes sur les cartons.

Voici quelques règles générales pour la création de noms :

- utilisez des noms descriptifs dans l'ensemble de votre code.
- Évitez d'utiliser des abréviations ou de raccourcir des mots à chaque fois que c'est possible.
- La convention de nommage la plus courante est **camelCase**. Dans cette convention, les noms sont constitués de plusieurs mots dont l'initiale est en capitale.

Créer une variable

En JavaScript, la déclaration d'une variable se limite au mot clé let, suivi du nom de variable choisi :

```
let numberOfCats = 2;
let numberOfDogs = 4;
```

Ici, nous déclarons (créons) et initialisons (donnons une valeur à) deux variables : numberOfCats et numberOfDogs .

Exercice 1

- Déclarer des variables et leurs donner une valeur.

Modifier la valeur d'une variable

La façon la plus simple de modifier la valeur d'une variable est simplement de la réaffecter :

```
let numberOfCats = 3;
numberOfCats = 4;
```

Ici, nous déclarons la variable **numberOfCats** et l'initialisons à la valeur 3. Nous lui réaffectons ensuite la valeur 4 (**sans le mot clé let**, par ce que la variable a déjà été déclarée).

Néanmoins, si c'était la seule façon de modifier une valeur de variable, vous ne pourriez pas en faire grand-chose. Voyons maintenant quelques **opérateurs**.

Modifier la valeur d'une variable

Opérateurs arithmétiques : addition et soustraction

Pour ajouter deux variables, utilisez le signe + :

```
let totalCDs = 67;
let totalVinyls = 34;
let totalMusic = totalCDs + totalVinyls;
```

À l'inverse, la soustraction utilise le signe - :

```
let cookiesInJar = 10;
let cookiesRemoved = 2;
let cookiesLeftInJar = cookiesInJar - cookiesRemoved;
```

Modifier la valeur d'une variable

Opérateurs arithmétiques : addition et soustraction

Pour ajouter ou soustraire un nombre d'une variable, vous pouvez utiliser les opérateurs `+=` et `-=` :

```
let cookiesInJar = 10;
/* manger deux cookies */
cookiesInJar -= 2;  // il reste 8 cookies
/* cuisson d'un nouveau lot de cookies */
cookiesInJar += 12; // il y a maintenant 20 cookies dans la
boîte
```

Modifier la valeur d'une variable

Opérateurs arithmétiques : addition et soustraction

Enfin, vous pouvez utiliser ++ ou -- pour ajouter ou soustraire 1 (incrément ou décrément) :

```
let numberOfLikes = 10;
numberOfLikes++; // cela fait 11
numberOfLikes--; // et on revient à 10
```

Modifier la valeur d'une variable

Opérateurs arithmétiques : multiplication et division

Les opérations de multiplication et de division utilisent les opérateurs * et / :

```
let costPerProduct = 20;
let numberOfProducts = 5;
let totalCost = costPerProduct * numberOfProducts;
let averageCostPerProduct = totalCost / numberOfProducts;
```

Modifier la valeur d'une variable

Opérateurs arithmétiques : multiplication et division

Comme pour l'addition et la soustraction, il existe aussi les opérateurs *= et /= pour multiplier ou diviser un nombre :

```
let numberOfCats = 2;
numberOfCats *= 6;  // numberOfCats vaut maintenant 2*6 = 12;
numberOfCats /= 3;  // numberOfCats vaut maintenant 12/3 = 4;
```

Exercice 2

- Utilisation de opérateurs

Modifier la valeur d'une variable

Mutabilité

Une variable est de base **mutable** c'est-à-dire qu'elle peut changer au cours du temps. On la déclare avec le mot clé **let** suivi d'un nom de variable et on lui affecte une valeur de départ. Par la suite, on va pouvoir changer la valeur de cette variable autant de fois que l'on souhaite.

```
let compteur = 0;
compteur++;
compteur = 10;
```

Les constantes

Dans de nombreux programmes, certaines données ne seront pas modifiées pendant l'exécution du programme. C'est le cas par exemple du nom d'une entreprise, de la date de naissance d'un utilisateur, ou du nombre d'heures dans une journée. Pour s'assurer de ne pas réaffecter par inadvertance de nouvelles valeurs à ces données, vous allez utiliser des **constantes**.

Ce sont simplement des variables qui ne seront **pas mutables**. On donnera une valeur de départ et on ne pourra plus changer la valeur par la suite. Ainsi s'il y a une erreur de logique dans votre code changeant la valeur du variable (constante) qui ne devait pas changer, javascript retournera une erreur.

```
const nombrePostParPage = 20;
nombrePostParPage = 30; // Retournera une erreur dans la console car on
ne peut plus changer sa valeur
```

Exercice 3

- Calcul de nombre de jours avec des constantes.

En résumé

Pour déclarer des **variables**, on utilise le mot clé ***let***, auquel on donne un nom en « **camelCase** ».

On peut initialiser les variables avec l'opérateur **=**.

On peut modifier la **valeur** d'une variable en la réaffectant ou avec des **opérateurs** (addition, soustraction, multiplication, division, incrémentation, décrémentation).

On doit utiliser des **constantes** avec le mot clé ***const*** pour éviter le remplacement d'éléments de données essentiels.

Les types

Le **type** d'une variable ou d'une constante est tout simplement le genre des données qu'elle enregistre. En JavaScript, il y a trois types primitifs principaux :

- **number** (nombre) ;
- **string** (chaîne de caractères) ;
- **boolean** (valeur logique).

Les **types primitifs** sont les briques de base de chaque structure de données en JavaScript. Peu importe la complexité finale de votre application, à sa base se trouveront ces trois types primitifs.

Les types

JavaScript est un langage dit à **types dynamiques** et à **typage faible**. Cela signifie que vous pouvez initialiser une variable en tant que nombre, puis la réaffecter comme chaîne, ou tout autre type de variable.

Ceci offre une grande souplesse, mais peut aussi conduire à un comportement inattendu si vous opérez sans précaution.

Prenez garde aux types de vos variables, et en général, utilisez des constantes chaque fois que c'est possible.

Le type « number »

Toutes les variables que vous avez créées jusqu'à maintenant dans ce cours étaient du type **number** (nombre, en français). Comme vous l'avez vu, elles peuvent être manipulées de nombreuses façons.

Les variables de type **number** peuvent être positives ou négatives. Elles peuvent aussi être des nombres entiers (1, 2, 3, etc.) ou décimaux (1.4 ; 67.34 ; etc.).

En programmation, les nombres entiers sont aussi appelés **entiers** ou **integers** ; les nombres avec des chiffres après la virgule sont aussi appelés nombres en **virgule flottante** ou **floating-point**.

Le type « string »

Les **chaînes de caractères** (chaînes, ou **strings**, en anglais) sont la façon d'enregistrer du texte dans des variables JavaScript. On peut enregistrer dans une variable de type string n'importe quelle chaîne de caractères, allant d'une seule lettre à un très grand nombre de lettres (plus de 134 millions, même dans des navigateurs anciens).

Les variables de type string sont encadrées par des guillemets simples ou doubles 'text' ou "text" :

```
let firstName = "Hilal";
let lastName = 'Wahim';
```

Le type « string »

Les chaînes peuvent aussi être **concaténées** (ajoutées à la fin l'une de l'autre) par l'opérateur + :

```
let wholeName = firstName + " " + lastName;  // valeur : “Hilal Wahim"
```

Le type « string »

Il est possible depuis quelques années d'utiliser une nouvelle écriture qui simplifie la concaténation des variables et des chaînes de caractère : **la string interpolation**.

Pour créer une string interpolation on écrit du texte encadré par l'accent grave `text` et si on veut injecter une variable dans ce code on utilise l'expression ${maVariable}.

```
const myName = `Wahim`;
const salutation = `Bienvenue sur mon site ${myName}!`;
console.log(salutation);   //retournera "Bienvenue sur mon site
Wahim!"
```

Le type « boolean »

Les **valeurs logiques** (**booleans**) sont le plus simple des types primitifs : elles ne peuvent avoir que deux valeurs, **true** ou **false** (vrai ou faux). Elles s'utilisent dans toutes sortes de cas : pour indiquer si un utilisateur est connecté ou non, si une case est cochée ou non, ou si un ensemble de conditions particulières est réuni.

```
let userIsSignedIn = true;
let userIsAdmin = false;
```

Exercice 4

- Les types de données

En résumé

les trois principaux types de données primitifs en JavaScript :

- number (nombre) ;
- boolean (valeur logique) ;
- string (chaîne de caractères).

Il existe d'autres types de données plus complexes.

Techniquement, il y a trois autres types de données primitifs dans JavaScript : **null** , **undefined** et **symbol** .

Les objets

Exemple du monde réel :

Les livres => un titre, un nombre de pages, un auteur, etc.

Ici, **l'objet** sera « *le livre* », **les clés** seront « *le titre* », « *le nombre de pages* », « *l'auteur* », etc., et **les valeurs** seront les valeurs que l'on donnera à nos clés.

Les objets

Les objets JavaScript sont écrits en **JSON (JavaScript Object Notation)**. Ce sont des séries de paires **clés-valeurs** séparées par des virgules, entre des accolades. Les objets peuvent être enregistrés dans une variable :

```
let myBook = {
  title: «La Communauté de l’Anneau»,
  author: 'J.R.R. Tolkien',
  numberOfPages: 736,
  isAvailable: true
};
```

Chaque clé est une chaîne (title, author, numberOfPages...), et les valeurs associées peuvent avoir tout type de données (nombre, chaîne, etc.).

Les objets

Construire des objets présente un avantage essentiel : cela permet de regrouper les attributs d'une chose unique à un même emplacement, que ce soit un livre, un profil d'utilisateur ou la configuration d'une application, par exemple.

Exercice 5

- La création d’objet

Les objets

Une fois l'objet créé en JavaScript, on va pouvoir accéder aux données de l'objet avec la **notation pointée** (*dot notation*)

```
let myBook = {
  title: 'La Communauté de l'Anneau',
  author: 'J.R.R. Tolkien',
  numberOfPages: 736,
  isAvailable: true
};
let bookTitle = myBook.title; // "La Communauté de l'Anneau"
let bookPages = myBook.numberOfPages; // 736
```

Les objets (bracket notation)

Les **brackets notation** permettent d'accéder à un sous élément.

```
let myBook = {
  title: 'La Communauté de l'Anneau',
  author: 'J.R.R. Tolkien',
  numberOfPages: 736,
  isAvailable: true
};
let bookTitle = myBook["title"]; // "La Communauté de l'Anneau"
let bookPages = myBook["numberOfPages"]; // 736
```

Les objets (bracket notation)

L'intérêt c'est qu'on va pouvoir mettre entre bracket une variable qui contient en string le nom de la propriété que l'on souhaite atteindre

```
let myBook = {
  title: 'La Communauté de l'Anneau',
  author: 'J.R.R. Tolkien',
  numberOfPages: 736,
  isAvailable: true
};
let propertyToAccess = "title";
let bookTitle = myBook[propertyToAccess]; // 'La Communauté de
l'Anneau'
```

Exercice 6

- Récupération de valeurs depuis un objet

Les classes

une classe est un **modèle** pour un objet dans le code. Elle permet de construire plusieurs objets du même type (appelés **instances** de la même classe) plus facilement, rapidement et en toute fiabilité.

Pour créer une classe dans JavaScript, utilisez le mot clé **class** , suivi par un nom. Encadrez ensuite le code de la classe entre accolades :

```
class Book {}
```

Pour cette classe, nous souhaitons que chaque *Book* ait un titre, un auteur et un nombre de pages. Pour cela, vous utilisez ce qu'on appelle un **constructor**.

Les classes

Le constructor d'une classe est la fonction qui est appelée quand on crée une nouvelle instance de cette classe avec le mot clé *new.*

```
class Book {
  constructor(title, author, pages) {}
}
```

Il y a un ensemble d'instructions à suivre à l'intérieur du constructor pour créer une instance de la classe *Book* .

Les classes

Pour attribuer le titre, l'auteur et le nombre de pages reçus à cette instance, utilisez le mot clé **this** et la notation dot.

```
class Book {
  constructor(title, author, pages) {
    this.title = title;
    this.author = author;
    this.pages = pages;
  }
}
```

Ici, le mot clé *this* fait référence à la nouvelle instance. Donc, il utilise la notation dot pour attribuer les valeurs reçues aux clés correspondantes.

Les classes

Maintenant que la classe est terminée, vous pouvez créer des instances par le mot clé **new** :

```
let myBook = new Book("La Communauté de l’Anneau", "J.R.R. Tolkien",
736);
// Cette ligne crée l'objet suivant :
let Book = {
  title: "La Communauté de l’Anneau",
  author: "J.R.R. Tolkien",
  pages: 736
}
```

Avec une classe *Book* , vous pouvez créer facilement et rapidement de nouveaux objets *Book* .

Exercice 7

- Les classes

En résumé

Les objets avec les paires **clés-valeurs** en notation **JSON** permettent d'enregistrer plusieurs éléments de données associés dans une même variable ;

La **notation pointée** (dot) donne accès aux valeurs d'un objet et à la possibilité de les modifier ;

L'utilisation de **classes** peut vous permettre de créer des objets plus facilement et de façon plus lisible.

Le tableau (array)

Pour créer un tableau vide et l'enregistrer dans une variable, utilisez une paire de crochets.

Une fois le tableau créé, on le remplit en plaçant les éléments voulus à l'intérieur de ces crochets.

Vous pouvez ensuite accéder aux éléments de ce tableau par leur indice en commençant à l'indice 0 :

```
let guests = ["Jovany", "Wahim", "David"];
let firstGuest = guests[0]; // “Jovany"
let thirdGuest = guests[2]; // “David"
let undefinedGuest = guests[12] // undefined
```

Exercice 8

- Création d’un tableau (array)

Valeur et référence

En JavaScript, les types primitifs tels que les nombres (**number**), les valeurs logiques (**boolean**) et les chaînes (**string**) passent par des **valeurs**.

```
let numberOfGuests = 20;
let totalNumberOfGuests = numberOfGuests; // 20
```

20 est la valeur qui est copiée dans la nouvelle variable (totalNumberOfGuests), et aucun lien n'est maintenu entre les deux variables.

Valeur et référence

En revanche les **objets** et les **tableaux** passent par des **références.**

```
let formerProfile = {
  name: "Wahim",
  age: 39,
  available: true
};
let allProfiles = [formerProfile]; // nouveau tableau contenant l'objet
ci-dessus
formerProfile.available = false; // modification de l'objet
console.log(allProfiles) // affiche { nom: “Wahim", âge: 39,
disponible: false }
```

Quand on utilise des tableaux et des objets, on passe des **références** aux **objets** plutôt que la valeur des données qu'ils contiennent. Les variables *formerProfile* et *allProfiles* présentées ci-dessus contiennent des références à l'objet et au tableau en mémoire.

Compter, ajouter et supprimer les éléments d'un tableau (array)

La propriété *length* d'un tableau indique le nombre d'éléments qu'il contient.

```
let guests = ["Jovany", "Wahim", "David"];
let howManyGuests = guests.length; // 3
```

Pour ajouter un élément à la fin d'un tableau, on utilise *push.*

```
guests.push("Mohamed"); // ajoute "Mohamed" à la fin de notre tableau
guests
```

Compter, ajouter et supprimer les éléments d'un tableau (array)

Pour ajouter votre élément au début du tableau plutôt qu'à la fin, utilisez la méthode *unshift*

```
guests.unshift("Mohamed"); // ajoute "Mohamed" au début de notre tableau guests
```

Pour supprimer le dernier élément d'un tableau, utilisez la méthode *pop* , sans passer aucun argument

```
guests.pop(); // supprimer le dernier élément du tableau
```

Exercice 9

- Travailler avec un tableau (array)

En résumé

Nous avons pris connaissance des **collections.**

La collection la plus courante en JavaScript est le **tableau** ou **array.**

Les types primitifs utilisent des **valeurs**, les collections utilisent des **références**.

Le déroulement du programme

Le **déroulement du programme** est un terme général qui décrit l'ordre dans lequel s'exécutent vos lignes de code. Cela signifie que certaines lignes seront lues une seule fois, certaines plusieurs fois, et d'autres complètement ignorées, selon la situation.

Les instructions conditionnelles if/else

L'instruction **if / else** est une des plus universelles en programmation. Qu'il s'agisse de réagir à une saisie de l'utilisateur, aux résultats de calculs ou de simplement vérifier si quelqu'un est connecté ou non, vous aurez souvent à utiliser des instructions if/else.

Exemple :

IF (*SI*) l'utilisateur est connecté, ouvrir sa page d'accueil

ELSE (*SINON*) revenir à la page de connexion

C'est ce qu'on appelle une **instruction conditionnelle**, parce qu'elle vérifie si certaines conditions sont réunies, et réagit en conséquence.

if/else avec les valeurs bouléens (boolean)

En JavaScript, si on utilise des boolean (bouléens, en français) simples pour les instructions if / else , la syntaxe se présente comme suit :

```
if (myBoolean) {
  // réaction à la valeur vraie de myBoolean
} else {
  // réaction à la valeur fausse de myBoolean
}
```

if/else avec les valeurs bouléens (boolean)

Exemple :

```
let UserLoggedIn = true;
if (UserLoggedIn) {
  console.log("Utilisateur connecté!");
} else {
  console.log("Alerte, intrus!");
}
```

Dans le cas ci-dessus, nous aurons sur la console "Utilisateur connecté!" , car le boolean userLoggedIn a la valeur true.

S'il avait la valeur false, nous aurions à la place "Alerte, intrus!" .

Les expressions

Dans une instruction conditionnelle, nous pouvons utiliser une simple variable logique mais aussi ce que nous appelons des **expressions de comparaison** qui compare des valeurs entre elles.

Pour cela nous utilisons les opérateurs suivants :

- < inférieur à
- <= inférieur ou égal à
- == égal à
- >= supérieur ou égal à
- > supérieur à
- != différent de

Les expressions

Exemple :

```
const numberOfSeats = 30;
const numberOfGuests = 25;
if (numberOfGuests < numberOfSeats) {
  // autoriser plus d'invités
} else {
  // ne pas autoriser de nouveaux invités
}
```

Les expressions

Il est possible de chaîner les instructions if / else

```
if (numberOfGuests == numberOfSeats) {
  // tous les sièges sont occupés
} else if (numberOfGuests < numberOfSeats) {
  // autoriser plus d'invités
} else {
  // ne pas autoriser de nouveaux invités
}
```

Exercice 10

- If / else

L'égalité == ou ===

Il y a deux façons de vérifier si deux valeurs sont égales en JavaScript : == et ===, aussi appelées égalité **simple** et égalité **stricte.**

L'égalité simple vérifie la **valeur**, mais pas le type, alors que l'égalité stricte vérifie à la fois la **valeur** mais aussi le **type.**

Exemple :

- 2 == "2" => ceci enverra la valeur *true* car seule la valeur est vérifiée
- 2 === "2" => ceci enverra la valeur *false* car nous comparons une valeur de type ***number*** avec une valeur de type ***string*** (présence des guillemets) soit deux types différents

Nous aurons la même distinction pour l'opérateur d'inégalité, *!=* pour une **inégalité simple**, *!==* pour une **inégalité stricte**.

Les conditions multiples

Dans certaines situations, nous pouvons vérifier plusieurs conditions pour un même résultat ; par exemple dans la même instruction if. Pour cela, il existe des **opérateurs logiques** :

- *&&* – ET logique – pour vérifier si deux conditions sont **toutes les deux** vraies ;
- *||* – OU logique – pour vérifier si **au moins une condition** est vraie ;
- *!* – NON logique – pour vérifier si une condition **n'est pas** vraie.

Les conditions multiples

Exemple :

```
let userLoggedIn = true;
let UserHasPremiumAccount = true;
let userHasMegaPremiumAccount = false;

userLoggedIn && userHasPremiumAccount; // true
userLoggedIn && userHasMegaPremiumAccount; // false

userLoggedIn || userHasPremiumAccount; // true
userLoggedIn || userHasMegaPremiumAccount; // true

!userLoggedIn; // false
!userHasMegaPremiumAccount; // true
```

Exercice 11

- Les opérateurs logiques

Le scope des variables

En JavaScript, les variables créées par *let* ou *const* ne peuvent être vues ou utilisées qu'à l'intérieur du **bloc de code** dans lequel elles sont déclarées.

Un bloc de code, aussi souvent appelé **bloc** tout court, est une section de code incluse entre accolades {}.

Ce phénomène est appelé **portée des variables** ou **block scope** (en anglais).

Le scope des variables

Exemple :

```
let userLoggedIn = true;

if (userLoggedIn) {
    let welcomeMessage = 'Welcome back!';
} else {
    let welcomeMessage = 'Welcome new user!';
}

console.log(welcomeMessage); // renvoie une erreur
```

Chaque bloc est représenté par une couleur différente.

il n'y a pas de variable welcomeMessage de déclaré en dehors des blocs, ce qui remonte une erreur

Le scope des variables

Pour cet exemple, une bonne méthode sera de déclarer la variable dans la portée extérieure ou ce que nous appelons le scope parent, puis de la modifier à l'intérieur des blocs if / else

```
let userLoggedIn = true;
let welcomeMessage = ''; // déclarer la variable ici

if (userLoggedIn) {
welcomeMessage = 'Welcome back!'; // modifier la variable extérieure
} else {
welcomeMessage = 'Welcome new user!'; // modifier la variable
extérieure
}

console.log(welcomeMessage); // imprime 'Welcome back!'
```

Les instructions switch

Pour vérifier la valeur d'une variable par rapport à une liste de valeurs attendues, et réagir en conséquence, nous utilisons l'instruction ***switch***.

Exemple : Supposons que vous ayez quelques objets utilisateurs. Vous souhaitez vérifier le type de compte de chaque utilisateur, pour envoyer un message personnalisé.

Les instructions switch

```
let firstUser = {
  name: "Wahim",
  age: 39,
  accountLevel: "normal",
};
let secondUser = {
  name: "Graziella",
  age: 31,
  accountLevel: "premium",
};
let thirdUser = {
  name: "Farid",
  age: 40,
  accountLevel: "mega-premium",
};
```

Les instructions switch

Nous pouvons utiliser une instruction *switch*, qui prend la variable à vérifier et une liste de valeurs, comme différents cas :

```
switch (firstUser.accountLevel) {
  case "normal":
    console.log("You have a normal account!");
    break;

  case "premium":
    console.log("You have a premium account!");
    break;

  case "mega-premium":
    console.log("You have a mega premium account!");
    break;

  default:
    console.log("Unknown account type!");
}
```

Les instructions switch

L'instruction ***case*** reprend chaque cas pour mettre le message approprié, l'instruction ***break*** permet d'éviter l'effet **cascade**, ainsi il interrompt l'exécution des cas suivants si le cas est vrai.

Le cas ***default*** sera exécuté que si la variable que nous vérifions ne correspond à aucune valeur répertoriée.

En résumé

Nous avons vu :

- le fonctionnement des instructions **if/else**
- les différents types de conditions pouvant être utilisés pour les instructions **if/else**
- comment regrouper les différentes conditions avec des opérateurs logiques
- la **portée des variables** ou le **scope**
- l'instruction switch pour comparaison à une liste de valeurs attendues.

La boucle for

En programmation, il est possible d'utiliser des ensembles d'instructions à répéter plusieurs fois sans forcément connaître le nombre de fois mais à répéter jusqu'à atteindre une certaine condition.

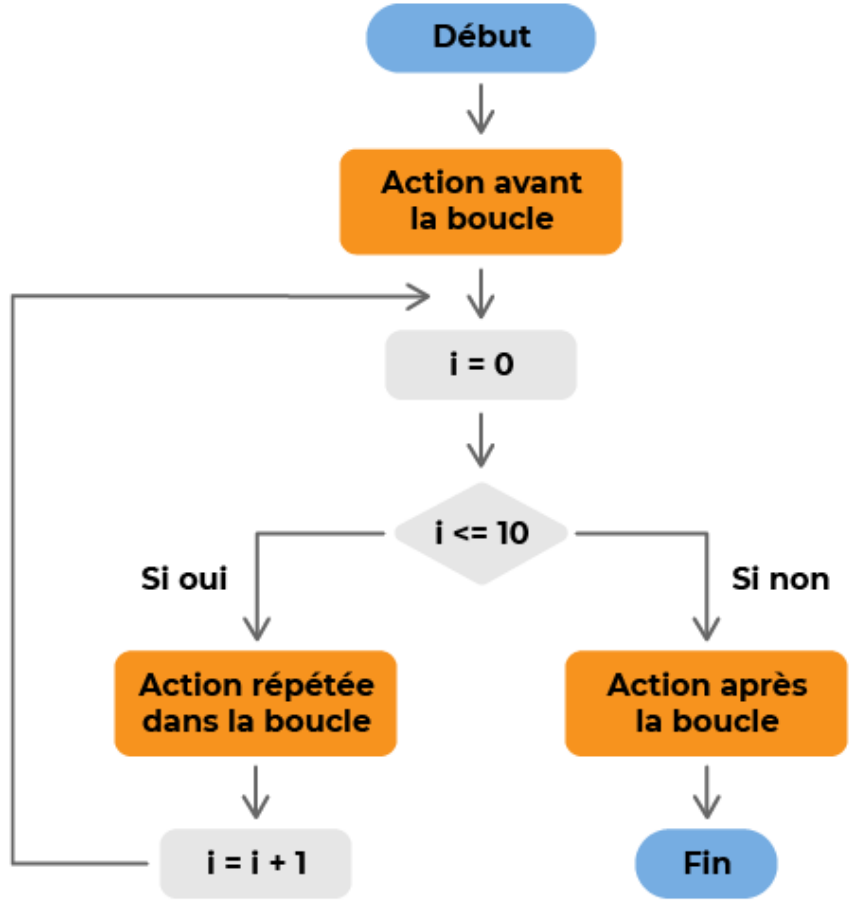
Pour cela, nous utiliserons des **boucles**.

Pour la boucle ***for*** , nous devons tout d'abord créer une variable d'indice appelée *i* qui servira de compteur pour le nombre d'exécutions de la boucle.

Cet indice démarrera à 0.

Dans la suite de la parenthèse, nous fixons la condition de poursuite de la boucle, dès que la condition n'est plus respectée, dès qu'elle est fausse, nous quittons la boucle.

La boucle for



La boucle for

Exemple :

```
const numberOfPassengers = 10;
for (let i = 0; i < numberOfPassengers; i++) {
  console.log("Passager embarqué !");
}

Console.log("Tous les passagers ont embarqués !");
```

Dans cet exemple, nous avons déclaré 10 passagers au total à embarquer, donc tant qu'il y a des passagers à embarquer, la boucle fonctionne, une fois les 10 passagers embarqués, la boucle doit s'arrêter, ceci est représenté par *i < numberOfPassengers.*

i++ demande à la boucle for d'incrémenter i de 1 à chaque exécution.

For…in et for…of

La boucle ***for…in*** est très comparable à l'exemple de boucle for normale, mais elle est plus facile à lire, et effectue tout le travail d'itération pour vous.

For... in et for... of

```
const passengers = ["Wahim", "Jovany'", "David", "Alex"];

for (let i in passengers) {
  console.log("Embarquement du passager " + passengers[i]);
}
```

i démarre automatiquement à zéro et s'incrémente à chaque boucle jusqu'à terminer l'itération sur tous les passagers.

For... in et for... of

Pour les cas où l'indice précis d'un élément n'est pas nécessaire pendant l'itération, vous pouvez utiliser une boucle *for... of*

```
const passengers = ["Wahim", "Jovany'", "David", "Alex"];

for (let passenger of passengers) {
  console.log("Embarquement du passager " + passenger);
}
```

Ceci produit exactement le même résultat, mais de façon plus lisible, car vous n'avez pas à vous inquiéter des indices et des tableaux : vous recevez simplement chaque élément dans l'ordre.

Exercice 12

- Les boucles *for*

La boucle while

Une boucle ***while*** vérifie si une condition est vraie. Si c'est le cas, la boucle se poursuit, sinon elle s'arrête. Exemple :

```
let seatsLeft = 10;
let passengersStillToBoard = 8;
let passengersBoarded = 0;

while (seatsLeft > 0 && passengersStillToBoard > 0) {
passengersBoarded++; // un passager embarque
passengersStillToBoard--; // donc il y a un passager de moins à embarquer
seatsLeft--; // et un siège de moins
}

console.log(passengersBoarded); // imprime 8, car il y a 8 passagers pour 10
sièges
```

Cette boucle ***while*** poursuit son exécution jusqu'à ce que l'un des nombres seatsLeft et passengersStillToBoard atteigne zéro.

En résumé

En JavaScript, il existe deux façons de répéter les tâches :

- la boucle ***for*** , pour un nombre d'itérations fixe ;
- la boucle ***while*** , quand le nombre d'itérations nécessaires est inconnu.

Les fonctions

Une **fonction** est un bloc de code auquel vous attribuez un nom. Quand vous appelez cette fonction, vous exécutez le code qu'elle contient.

Par exemple ***console.log()*** , qui contient du code permettant d'imprimer sur la console.

```
// On définit la fonction
function afficherDeuxValeurs(valeur1, valeur2) {
  console.log("Première valeur : " + valeur1);
  console.log("Deuxième valeur : " + valeur2);
}
// On exécute la fonction
afficherDeuxValeurs(12, "Bonjour");
// On obtient dans la console
// > Première valeur : 12
// > Deuxième valeur : Bonjour
```

Les fonctions

Beaucoup de fonctions ont besoin de variables pour effectuer leur travail.

Quand vous créez ou **déclarez** une fonction, vous indiquez la liste des variables dont elle a besoin pour effectuer son travail : vous définissez les **paramètres** de la fonction.

Ensuite, à l'appel de la fonction, vous lui attribuez des valeurs pour ses paramètres.

Les valeurs sont les **arguments** d'appel.

Enfin, votre fonction peut vous donner un résultat : une **valeur de retour**.

Supposons que vous ayez une fonction qui compte le nombre de mots dans une chaîne :

- le paramètre sera une chaîne dont vous allez compter les mots ;
- l'argument sera toute chaîne attribuée à votre fonction quand vous l'appelez ;
- la valeur de retour sera le nombre de mots.

Les fonctions

Exemple :

```
// déclaration de la fonction somme avec les paramètres
a et b
function somme(a, b) {
// la valeur de retour, ici a + b
  return a + b;
};
// On appelle la fonction avec les arguments d'appel 12
pour a et 5 pour b
console.log(somme(12, 5));
```

Les fonctions fléchées

Une **fonction fléchée** (**arrow function** en anglais) permet d'avoir une syntaxe plus courte que les expressions de fonction.

Les fonctions fléchées sont souvent anonymes et ne sont pas destinées à être utilisées pour déclarer des méthodes.

Les fonctions fléchées n'ont pas besoin du couple d'accolades classique aux fonctions pour fonctionner et n'ont pas besoin non plus d'une expression return puisque celles-ci vont automatiquement évaluer l'expression à droite du signe **=>** et retourner son résultat.

Les fonctions fléchées

Exemple :

```
// Fonction classique
function somme(a, b) {
  return a + b;
};
console.log(somme(12, 5);

// Équivalent en fonction fléchée : ES6
let somme = (a, b) => a + b;
console.log(somme(12, 5));
```

Exercice 13

- Calculer une moyenne

En résumé

Nous avons vu :

- ce qu'est une **fonction**, comment en **déclarer** une et comment en **appeler** une
- ce que sont les **paramètres**, les **arguments** et les **valeurs de retour**
- à écrire une fonction qui parcourt un tableau de nombres pour calculer leur moyenne

Exercice final

https://api.next.tech/api/v1/publishable_key/2A9CAA3419124E3E8C3F5AFCE5306292?content_id=e8174309-e22c-48d8-a368-1154d36270cc

Vocabulaire avec le glossaire

Variable

Une variable permet d'associer un nom à une valeur.

La valeur peut prendre plusieurs formes (texte, nombre, boolean, etc.).

On la déclare avec le mot clé ***let*** .

Exemple :

```
let maVariable = "Contenu de la variable";
```

Vocabulaire avec le glossaire

Constante

Une constante est une variable qui a la particularité de ne pas pouvoir changer de valeur dans le temps.

On la déclare avec le mot clé ***const*** .

Exemple :

```
const maConstant = "Production";
```

Vocabulaire avec le glossaire

Type

Un type correspond à la nature des valeurs que peut prendre une donnée (string, number, array, etc).

Exemple :

```
let monTableau = ['a','b','c']; let monNombre = 123;
```

Selon son type on peut appliquer divers actions sur une variable :

- Les variables **number** pourront s'additionner, multiplier, etc.
- Les variables **string** pourront est converti en majuscule avec la fonction *maString.toUpperCase()*
- Les autres types ont chacun des fonctionnalités associées.

Vocabulaire avec le glossaire

Programmation orientée Objet (POO)

La programmation orientée objet est un concept de programmation qui consiste à définir et faire interagir des éléments qu'on appelle "objets".

Un "objet" est un concept, une représentation, une idée qui se rattache au monde physique.

Par exemple, un livre, une page de livre, une lettre, etc.

Vocabulaire avec le glossaire

Classe

En POO, une classe regroupe les méthodes et propriétés (attributs) qui se rapportent à un objet.

Elle définit ce qu'est un objet et ce que l'on peut faire avec.

Exemple :

```
class MaClass {}
```

Vocabulaire avec le glossaire

Instance de classe

L'instance de classe est la création d'un objet unique basée sur la définition d'une classe.

Une instance est donc un élément unique créé à partir d'une classe.

Par exemple, à partir d'une classe Maison, on peut créer 2 instances qui seront 2 maisons différentes mais basées sur les mêmes plans de construction.

Exemple :

```
let monInstance = new MaClass();
```

Vocabulaire avec le glossaire

Propriété/attribut de classe

La propriété (dite aussi attribut) de classe est une variable interne à une classe qui pourra évoluer.

Par exemple, un objet maison a un nombre de portes de base, mais cette valeur peut évoluer selon les actions appliquées sur cette maison (des travaux, par exemple).

Vocabulaire avec le glossaire

Propriété/attribut de classe

Exemple :

```
class Maison{
constructor(couleur){
  this.couleur = couleur;
}
}
let maMaison = new Maison('rouge');
console.log(maMaison.couleur) // Donnera "rouge"
```

Vocabulaire avec le glossaire

Méthode de classe

La méthode de classe est une fonction interne à la classe qui permet d'exécuter des actions au sein de la classe instanciée. Par exemple, la méthode “peindre(couleur)” peut changer la propriété “couleur” de l’objet maison.

Vocabulaire avec le glossaire

Méthode de classe

Exemple :

```
class Maison{
  constructor(couleur){
    this.couleur = couleur;
  }
  changerCouleur(nouvelleCouleur){
    this.couleur = nouvelleCouleur;
  }
}
let maMaison = new Maison('rouge');
console.log(maMaison.couleur) // Donnera "rouge"
maMaison.changerCouleur('bleu')
console.log(maMaison.couleur) // Donnera "bleu"
```

Vocabulaire avec le glossaire

Collection

Les collections sont des types de données qui permettent de ranger un ensemble de données dans une seule variable (comme ranger de la donnée dans un tableau).

On trouve principalement deux types de collection : les Array et les Object.

Vocabulaire avec le glossaire

Collection de type Array

Une collection de type Array permet de créer une liste ordonnée d'éléments.

On accède à l'un de ces éléments par son index.

Il faut savoir que le premier index est 0 et non pas 1.

Exemple :

```
let monArray = ['a','b','c'];

console.log(monArray[0]); // donnera 'a' car c'est le premier élément du tableau
```

Vocabulaire avec le glossaire

Collection de type Object

Une collection de type Object est une liste non ordonnée qui ne fonctionne pas à partir d'index mais à partir de clé à laquelle est rattachée une valeur.

Donc pour récupérer une valeur, on doit renseigner la clé de cette valeur.

Exemple :

```
let monObject = { nom: "Jean", prenom: "Dupond", age: 26};

console.log(monObject.age); // donnera 26

console.log(monObject['age']); // donnera aussi 26
```

Vocabulaire avec le glossaire

Bloc de code

Un bloc de code est une partie du code commençant par une accolade ouvrante { et terminant par une accolade fermante }.

Exemple :

```
if (condition) {
  // Dans le bloc if
}
```

Vocabulaire avec le glossaire

Opérateurs logiques

Ce sont des caractères spéciaux qui permettent de faire des conditions plus complexes en liant plusieurs conditions, ou en faisant des comparaisons de valeurs.

On retrouve les caractères suivants :

&&, ||, <, >, >=, <=, ==, ===, != et !.

Exemple :

```
if (condition1 && !condition2) {
  console.log(
    "Cette ligne s’affiche si condition1 est vrai ET si condition2 n’est pas
vrai"
  );
}
```

Vocabulaire avec le glossaire

Exception

Dès qu'il y a une erreur liée à un défaut d'exécution, une erreur est soulevée en JavaScript, on appelle cela une Exception.

On le remarque souvent dans la console en rouge.

Une Exception contient des informations permettant de comprendre les raisons du problème, et on peut facilement récupérer cette Exception au sein du code pour faire un traitement personnalisé grâce au bloc **try{ } catch(e){ }** .

Vocabulaire avec le glossaire

Exception

Exemple :

```
try {
  fonctionQuiRetourneUneException();
} catch(e) {
  console.log("il y a une Exception: "+e.getMessage());
}
```

Vocabulaire avec le glossaire

Paramètre de fonction

Un paramètre est une valeur qu'une fonction attend en entrée.

On lui donne un nom comme une variable qui servira au sein de la fonction pour le traitement.

Il peut bien sûr y avoir plusieurs paramètres dans une fonction.

Exemple :

```
function maFonction(parametre1) {
  console.log(parametre1); // On a affiché la valeur du paramètre

  // Traitement
}
```

Vocabulaire avec le glossaire

Argument de fonction

L’argument d’une fonction est la valeur qu’on va injecter en entrée d’une fonction au moment de l'exécution.

Ce concept est lié directement à la notion de paramètre de fonction, car en général pour chaque paramètre d’une fonction, on devra apporter un argument.

Vocabulaire avec le glossaire

Argument de fonction

Exemple :

```
function maFonction(parametre1){

  console.log(parametre1) // On a affiché la valeur du paramètre

  // Traitement

}
let monArgument = "Bonjour";

maFonction(monArgument); // On obtient un log disant "Bonjour"
```

Vocabulaire avec le glossaire

Valeur de retour

Lors de l'exécution d'une fonction, il y a plusieurs traitements réalisés en son sein, et une valeur finale pourra être retournée avec le mot clé “return”.

Cela permet de récupérer la valeur retournée par la fonction dans une variable, par exemple, et de l'exploiter dans le reste du code.

Vocabulaire avec le glossaire

Valeur de retour

Exemple :

```
function additionner(valeur1, valeur2) {
  return valeur1 + valeur2; // la fonction additionner va retourner la somme de
valeur1 et valeur2
}

let resultat = additionner(12, 13); // resultat vaudra 25
```

Vocabulaire avec le glossaire

Récursivité

La récursivité est un concept dans lequel un élément fait appel à lui-même.

Par exemple, une fonction récursive est une fonction qui s'appelle elle-même d'une façon ou d'une autre.

Exemple :

```
function factorielle(number) {
  if (number <= 1) return 1;
  else return number * factorielle(number - 1);
}
```

Fiche Récap : Apprenez à programmer en Javascript

Développement

1. Définissez par écrit ce que vous voulez faire : ce qui se conçoit bien s'énonce clairement !
2. Intégrez le design de votre objectif en HTML et CSS, sans Javascript. Ne faites pas tout d'un coup : segmenter la difficulté, c'est la clé.
3. Découpez le plus possible la logique de votre code en sous-logiques.
4. Ecrivez chaque bout de code l'un après l'autre en prenant soin de bien les tester à chaque fois.
5. Indentez votre code et commentez-le au fur et à mesure pour qu'il reste facilement maintenable et évolutif.

Code

```javascript
class Person{
    constructor(nom, age = 0){
        //On récupère les informations de la personne en interne
        this.nom = nom;
        this.age = age;
        this.isMajeur = (this.age >= 18);
    }

    salutation() {
        /* On écrit un message de bienvenue
           personnalisé selon qu'il est majeur ou non */
        if(this.isMajeur) {
            console.log("Bonjour " + this.nom + ", tu as " + this.age + " ans donc tu es majeur.");
        }else{
            console.log(`Bonjour ${this.nom}, tu as ${this.age} ans donc tu es mineur.`);
        }
    }
}
const person1 = new Person("Jean", 22);
person1.salutation(); // Affichera dans la console 'Bonjour Jean, tu as 22 ans donc tu es majeur.'
const person2 = new Person("Luc", 12);
person2.salutation(); // Affichera dans la console 'Bonjour Luc, tu as 12 ans donc tu es mineur.'
```

Définitions

Variable
Association d'un nom à une valeur. Elle est déclarée via un mot-clé et peut prendre plusieurs formes : texte, nombre, booléen, etc.

Indentation
Mise en forme qui facilite la lecture des lignes de code.

Opérateurs logiques
Caractères spéciaux qui permettent de lier plusieurs conditions ou de comparer des valeurs.

`&&` `||` `<` `>` `>=` `<=` `==` `===` `!=`

POO (Programmation Orientée Objet)
Modèle qui consiste à définir et faire interagir des éléments que l'on appelle "objets".

Objet
Représentation qui se rattache au monde physique : un livre, une page de livre, une lettre.

Bonnes pratiques

- Nommer les variables et les fonctions de manière explicite.
- Coder en POO pour regrouper les blocs de logique.
- Pratiquer avec des projets personnels, l'expérience est la clé du succès.
- Prendre en compte la portée des variables.
- Analyser les erreurs dans la console.
- Lire les documentations.

Erreurs classiques

- ✗ Faire des fonctions trop longues qui font trop de choses.
- ✗ Utiliser le mauvais type de variable (on ne peut pas faire d'opération mathématique sur des chaînes de caractères).