Les conditions

Les conditions qui permettront de varier la voie que prendra notre programme en fonction d'informations connues se déclinent sous plusieurs versions:

- **simples,**
  Les conditions simples sont une structure composée au moins d'un partie avec le mot-clef **if** suivi d'une condition entre parenthèses et ensuite des instructions contenues entre des accolades.

  => if (condition) { Opérations à faire }

  Ensuite si on veut rajouter une deuxième condition à vérifier dans le cas où la première n'a pas été validée on peut rajouter un partie **else if** suivie elle aussi d'une condition entre parenthèses et de ses instructions entre accolades. On peut rajouter autant de **else if** que l'on désire.

  => else if (condition) { Opérations à faire }

  Finalement si des actions doivent être effectuées à condition qu'aucune des conditions précédentes n'aie été validée on utilisera un **else** simplement suivi des opérations entre accolades.

  => else { Opérations à faire }

  Voici une exemple de structure avec deux **else if**.

  => if (condition) { Opérations }
  else if (condition) { Opérations }
  else if (condition) { Opérations }
  else { Opérations }

  Attention ! Seulement une partie de la structure sera executée au maximum, les autres parties ne seront vérifiées que si les conditions précédentes n'ont pas été validées. (Par exemple si le **if** est executé, la structure de condition est finie)

- **ternaires,**
  Les ternaires sont une manière raccourcie de faire des conditions mais devraient principalement être utilisées quand il y deux options (soit si vrai soit si faux) et quand une valeur est attendue. Les raisons derrière sont que une ternaire est une **expression**, c'est à dire qu'elle retourne une valeur (par opposition à une condition simple qui ne retourne rien), et qu'elle est

structurée de façon à remplacer un **if ... else ...** qui contiennent chacun une instruction. Elle se compose d'une condition suivie d'un point interrogation puis d'une première **expression**, ensuite il y a un ":" ("colon" en Anglais) et une deuxième **expression**.
Si la condition est vraie le résultat de la première **expression** est renvoyé, sinon c'est le résultat de la deuxième **expression** qui est renvoyé.

=> let plusGrand = 8 > 2 ? true : false;

// On assigne à la variable plusGrand le résultat de la ternaire et comme la condition est vraie elle renvoie true et est l'équivalent de

if (8 > 2) { plusGrand = true; } else { plusGrand = false };

- **switch,**
  Un **switch** est une manière compacte de comparer le résultat d'une expression (grâce à une égalité stricte "===") à des valeurs données et d'exécuter des instructions si il y a correspondance.
  Il se compose du mot-clef **switch** suivi d'une expression entre parenthèses et juste après d'accolades qui contiendront tous les cas et leurs instructions.
  Un cas est le mot-clef **case** suivi de la valeur qui sera comparée à l'expression puis d'un ":" et après des instructions à effectuer si la valeurs sont bien égales.

```
=> switch ("hello") {
        case "bye": console.log("au revoir");
        case "hello": console.log("salut");
}
```

// "salut" sera affiché dans la console

Le code fonctionne bien mais cependant il manque quelque chose, dans le cas ou l'expression du switch serait "bye" les deux console.log seraient executés, car une fois qu'un cas est validé toutes les instructions suivantes sont lancées peu importe le cas auquelles elle appartiennent.
C'est là que va intervenir le mot-clef **break** qui va, comme avec les boucles, nous faire sortir de la structure. On l'utilisera donc à chaque fin de case quand on ne veut pas que les instructions suivantes soient lancées.

```
=> switch ("bye") {
        case "bye":
            console.log("au revoir");
            break;
        case "hello":
            console.log("salut");
            break;
}

// Seul le "au revoir" sera affiché avec les break
```

Omettre le **break** peut également avoir des avantages si on veut que plusieurs cas aient la même instruction ou pour avoir des instructions en commun.

```
=> switch ("giraffe") {
        case "elephant":
        case "giraffe":
        case "lion": console.log("C'est un animal");
}

// Affichera "C'est un animal" car l'expression est un des trois cas
```

Une dernière chose importante est qu'on peut définir un cas par défaut qui sera executé si aucun cas ne correspondait à l'expression. On utilisera alors le mot-clef **default** à la place d'un case.

```
=> switch ("lama") {
        case "elephant":
        case "giraffe":
        case "lion":
            console.log("C'est un animal");
            break;
        default:
            console.log("Je ne connais pas cet animal");
}

// Affichera "Je ne connais pas cet animal" car "lama" n'est pas un cas
```

Les conditions contenues dans ces structures ne sont validées que si elles sont au final une valeur true ou truthy (considérée comme vrai) tel que vu dans le cours d'algorithmie. Voici un petit rappel des valeurs qui sont considérées comme fausses, le reste étant toujours considéré comme vrai:

- false (booléen)
- 0 (nombre)
- "" ou '' ou `` (soit toute string vide)
- undefined
- null
- NaN (not a number)

Cependant généralement on n'utilise pas de valeurs directement dans ces conditions, on utilisera plutôt des expressions avec des **opérateurs de comparaison** donc voici un rappel de la liste:

Opérateur	Description
===	Egalité stricte
!==	Différence stricte
>	Plus grand que
<	Plus petit que
>=	Plus grand ou égal que
<=	Plus petit ou égal que

Ces opérateurs de comparaisons se comportent comme les opérateurs de calcul numérique sauf que les résultats seront toujours soit **true** soit **false**.

=> let myBool = 8 > 4; // myBool contiendra la valeur true

Là ou les choses changent c'est pour les **opérateurs logiques**, qui vont permettre de faire des calculs à base de valeurs booléennes ou truthy / false. Voici un mémo:

Opérateur	Description
!	NOT (Inverse booléen)
&&	ET
||	OU

Même principe que le calcul numérique ici sauf que les choses sont plus compliquées que vu lors de l'algorithmie, voici les détails cas par cas:

- ! (NOT), inverse les booléens donc un true devient un false et vice-versa mais quand utilisé avec des non-booléens il les convertis également en booléen.

  => let myBool = !4 // myBool contient false car 4 est truthy (considéré comme vrai) donc il est d'abord converti en booléen (devient true) puis il est inversé et devient false

  D'ailleurs une utilisation "détournée" est de mettre avant une valeur que l'on veut convertir en booléen **deux NOT ("!!")** ce qui va convertir, puis inverser et finalement inverser encore une fois pour arriver au booléen correspondant à la valeur de départ.

  => let myBool = !!4 // myBool contient true car 4 est converti en booléen par le NOT puis inversé en false et de nouveau inversé en true

- && (ET), prend deux valeurs (ne pas oublier que les calculs de comparaison renvoient un booléen) et retourne la *première valeur false ou falsy* ou si il n'y aucune valeur de ce genre retourne la dernière valeur à la place.

  => let myBool = 0 && 4 // myBool contient 0 car c'est la première valeur falsy

  => let myBool = 1 && 4 // myBool contient 4 car il n'y aucune valeur false ou falsy c'est donc la dernière valeur qui est retournée

  Voici donc comment le && (ET) se comporte: dès qu'on tombe sur valeur qui est fausse dans une situation où on a besoin que les deux soient vraies il n'y a pas besoin d'aller voir plus loin.
  Dans nos conditions cela nous donne bien le résultat attendu d'un ET (deux valeurs vraies) car la validation d'une condition n'a lieu que si le résultat final de l'expression entre les parenthèses est true ou truthy.

  => if (1 && 4) { ... } // Sera executé car le résultat est 4 qui est une valeur truthy

- || (OU), prend deux valeurs et retourne la *première valeur <u>true</u> ou <u>truthy</u>* ou dans le cas où aucune n'est présente renvoie la dernière valeur.

  => let myBool = 1 || 4 // myBool contient 1 car c'est la première valeur <u>truthy</u>

  => let myBool = '' || 0 // myBool contient la dernière valeur 0 car aucune valeur n'est <u>true</u> ou <u>truthy</u>

  Notre || (OU) se comporte donc de la façon opposée au ET car: dès qu'on tombe sur une valeur qui est vraie notre OU est validé, on a seulement besoin qu'une des deux valeurs soit vraie il n'y pas besoin de vérifier les autres valeurs.

  => if (1 || 0) { ... } // Sera executé car le résultat est 1 qui est une valeur <u>truthy</u>

On appelle le fait que les opérateurs logiques ET & OU s'arrêtent à la première valeur qui permette de savoir leurs résultats un **court-circuit** (**short-circuit** en Anglais). Cela est fait dans un souci d'optimisation mais de nouvelles possibilités de simplifier ou raccourcir le code voient le jour avec le concept de court-circuit:

- On peut créer une suite d'action qui s'arrêtera à la première valeur **false** ou **falsy**:

  ```
  => if ( nouveauJoueur() ) {
         if ( creerPersonnage() ) {
             lancerJeu();
         }
     }
  ```

  On imagine que dans l'exemple ci-dessus les fonctions renvoient une valeur <u>false</u> ou <u>falsy</u> dans le cas où un échec a lieu. On pourrait alors simplifier et raccourcir cette écriture déjà compacte en utilisant des && (ET)

  => nouveauJoueur() && creerPersonnage() && lancerJeu()

  Si un valeur renvoyée est fausse notre chaîne de && ne va pas voir plus loin et s'arrête donc

- Les ternaires qui servent à raccourcir les **if** peuvent être encore plus raccourcies par exemple dans le cas d'une assignation lorsqu'on veut garder la valeur d'une variable si elle est considérée comme vraie ou alors lui donner une autre valeur:

  => var compteur = compteur ? compteur : 0;

  Ce code regarde si la variable compteur a une valeur true ou truthy et dans ce cas elle garde la valeur sinon elle lui attribue le nombre 0. Mais on peut raccourcir encore plus:

  => var compteur = compteur || 0;

  Ici notre || renverra la première valeur considérée comme vraie et n'atteindra donc le 0 que si compteur est false ou falsy

Voici donc quelques cas qui se marient bien avec le court-circuitage en programmation.

Les opérateurs logiques et les opérateurs de comparaison peuvent s'enchaîner et former des combinaisons et tout comme avec les calculs numériques il y a un ordre des opérations:

- D'abord l'opérateur logique ! (NOT)
- Ensuite les opérateur de comparaisons > (plus grand), < (plus petit), >= (plus grand que ou égal), <= (plus petit que ou égal)
- Puis les opérateurs de comparaisons === (égalité stricte), !== (inégalité stricte)
- Le && (ET)
- Finalement le || (OU)

Il est tout comme en mathématiques très important de suivre le sens des opérations sinon on pourrait s'attendre à un résultat erroné et la logique sera dure à corriger !

=> let myBool = true || false && false // myBool contient true et non false car le && a lieu en premier (false && false => false) puis le OU a lieu (true || false => true)