TP - Création d’un jeu : Puissance 4

1 Le principe du jeu

Édité pour la première fois en 1974, le Puissance 4 est rapidement devenu un jeu de société incontournable. Son principe est simple, suivant une grille de 7x6, deux joueurs déposent tour à tour un jeton qui tombe tout en bas de la grille. Le premier joueur capable d'aligner 4 jetons de sa couleur horizontalement, verticalement ou diagonalement a gagné. Derrière ce principe simple se cache un exercice d'algorithmique plus complexe que nous allons tenter de résoudre ensemble.

2 Objectif du TP

Par groupes de 2 ou 3, recréer un jeu de Puissance 4 jouable par navigateur. Vous devrez utiliser toutes les notions vues jusqu'à présent afin de développer le code du jeu. Nous verrons ensemble les différentes étapes à implémenter afin d'avoir un jeu fonctionnel. S'en suivront quelques idées bonus à implémenter dans le jeu parmi lesquelles vous pourrez piocher.

2.1 Rendu du TP

Le rendu se fera sous forme d'un **projet complet**. Celui-ci contiendra **un fichier README expliquant les démarches à suivre pour le lancer**. Le tout sera distribué sous la forme d'un **dossier compressé .zip**. La date de rendu est le **lundi 30 Novembre à 22h**.

2.2 Notation du TP

Le projet sera évalué selon les modalités suivantes :

- Le projet standard avec les fonctionnalités de base sera noté sur **10 points**.
- Les fonctionnalités bonus seront notées sur **5 points**.
- Un oral individuel viendra compléter la note avec **5 points**.

3 Développement du jeu

Nous allons commencer par définir **le squelette complet du jeu**, c'est à dire les différentes fonctions qui devront intéragir les unes avec les autres afin de donner vie au projet final.

3.1 Le squelette

Énumérons tout d'abord les différents éléments dont nous allons avoir besoin.

- Une grille de 7x6 cases
- Une façon de représenter les jetons dans la grille
- Une représentation graphique de la grille
- Une gestion des tours de jeu
- La possibilité d'insérer un jeton dans une colonne de la grille
- La vérification de la victoire d'un des deux joueurs
- Le reset de la partie en cas de victoire ou de match nul

3.1.1 Initialisation

La première chose à faire au lancement du jeu sera **l'initialisation de la grille**. Il faudra donc créer un tableau à 2 dimensions de 7x6. Il faudra ensuite également **créer une grille identique mais en HTML** afin qu'elle soit visible pour les joueurs. Une fois les éléments de départ créés, nous pourrons lancer le jeu.

Il faudra alors qu'un joueur commence. Nous aurons besoin d'**une variable pour retenir quel joueur est en train de jouer**. Nous ferons commencer le joueur 1.

3.1.2 Attente d'une entrée utilisateur

Le code **a maintenant fini de s'exécuter**. Afin de **reprendre l'exécution**, nous devons attendre que le joueur 1 entreprenne une action. Cette action sera **une entrée au clavier** d'un chiffre entre 1 et 7 qui correspond à **la colonne où le joueur insert un jeton**. Lorsque le joueur appuiera sur une touche du clavier, notre fonction va vérifier que cette touche correspond bien à une entrée valide. Une fois cette vérification faite, la colonne résultante est **transmise à notre prochaine fonction** qui sera en charge de **gérer le tour de jeu**.

3.1.3 Exécution du tour de jeu

Cette fonction sera le **chef d'orchestre du jeu**, elle appellera les autres fonctions qui auront chacune **un rôle plus précis** :

- Tout d'abord, la fonction va **tenter d'insérer un jeton** dans la grille, à la colonne désignée par le joueur.
- Si ce n'est pas possible, car la colonne est déjà pleine, **la fonction s'arrête**, le tour ne change pas, nous attendons à nouveau une entrée valide du joueur.
- Si l'insertion est réussie, elle **vérifiera alors si l'un des deux joueurs a remporté la victoire**.
- Si ce n'est pas le cas, il faudra alors vérifier si **la grille est pleine**, ce qui constituerait une égalité.
- Si aucun des cas précédents ne s'est vérifié, alors le jeu continue, et la dernière action de notre fonction sera de **passer la main à l'adversaire**.

3.1.4 Insérer un jeton

Lors du découpage de notre fonction d'exécution du tour, nous pourrons créer une fonction gérant **uniquement l'insertion d'une pièce dans le tableau**. Cette fonction aura pour rôle de placer un jeton du bon joueur dans la bonne **colonne indiquée en paramètre**. Toute la difficulté ici sera de simuler la gravité du plateau de jeu original, et donc de **placer le jeton dans la bonne rangée**. Il faudra également vérifier s'il est possible d'insérer la pièce, c'est à dire si **la colonne n'est pas pleine**. Nous renverrons donc une valeur afin de **notifier à notre chef d'orchestre si l'insertion s'est bien déroulée**.

Il faudra également **mettre à jour notre vue (la grille HTML)** afin qu'elle reflète l'état interne de notre grille TS. Il serait sinon bien plus compliqué de jouer.

3.1.5 Vérifier la victoire d'un joueur

C'est dans cette fonction que réside la **plus grosse difficulté du projet**. Il faut en effet vérifier si l'un des deux joueurs a gagné horizontalement, verticalement ou diagonalement. Cette fonction va **renvoyer un booléen** permettant de savoir **si un joueur a gagné ou non**.

3.1.6 Vérifier le match nul

Une fonction un peu moins complexe va maintenant vérifier si le **tableau est plein** ou s'il reste des espaces vides. Si le tableau est plein, alors il y a match nul. La fonction va également **renvoyer un booléen** pour **notifier d'un match nul**.

3.1.7 Changer de tour de jeu

Enfin, dernière action à entreprendre, il s'agit de **passer la main à l'adversaire**. Plutôt simple ici, il s'agit de changer la valeur de notre variable qui suit le tour de jeu.

3.1.8 Reset de la grille et nouvelle partie

En cas de victoire ou d'égalité, il faut maintenant **réinitialiser les grilles (tableau et HTML)**, et donner la main à l'un des deux joueurs pour la prochaine partie. Afin de laisser aux joueurs le soin de **relancer la partie**, nous allons **suspendre le jeu** et attendre l'appui sur la touche Entrée avant de tout réinitialiser.

3.2 Initialisation

La première étape est donc d'**initialiser notre grille**, nous devons créer un tableau à 2 dimensions de 7x6. Ce tableau contiendra des entiers afin de retenir le **contenu de chacune des cases**. Nous devons décider de la façon dont nous allons représenter le contenu des cases. **Nous pouvons faire comme bon nous semble**, afin de donner du sens à nos valeurs, nous ferons donc comme suit :

- 0 pour une case vide
- 1 pour un jeton du joueur 1
- 2 pour un jeton du joueur 2

Nous pourrions utiliser **n'importe quelles autres valeurs** bien évidemment, nous décidons simplement d'une **convention pour le projet**.

3.2.1 Quelques constantes pour commencer

Afin d'**éviter d'utiliser des "magic numbers"** (des nombres écrits en dur dans le code, sans aucune justification), nous allons **définir des constantes** au début de notre code. De quelles constantes aurons-nous besoin ? Nous pouvons reprendre le principe du jeu pour les identifier. Nous jouons sur une grille de **7x6** et nous devons aligner **4** jetons. Cela nous fait donc déjà 3 constantes identifiées.

```
const ROWS: number = 6;
const COLUMNS: number = 7;
const IN_A_ROW: number = 4;
```

Une quatrième constante un peu surprenante sera **notre tableau à 2 dimensions**. En effet, constante veut dire que nous ne pouvons pas **réassigner** de valeur à notre variable. Nous pouvons donc créer notre tableau, ajouter des cases, supprimer, modifier le contenu des cases. Ce que nous ne pouvons pas faire, c'est assigner un nouveau tableau à notre constante.

```
const monTableau: number[] = [5, 8, 4, 3];

// Autorisé
monTableau[2] = 7;
monTableau.push(5);
monTableau.splice(2, 1);

// Interdit
monTableau = [];
```

Nos 4 constantes seront donc **globales**, cela veut dire qu'elles sont **accessibles partout dans le code**. Nous essayons en général de **limiter le nombre de variables mutables** (donc non constantes) globales dans notre code. Ici la seule variable mutable est notre tableau, mais nous pouvons difficilement faire autrement sans complexifier le code.

Pourquoi essayer de limiter le nombre de variables globales ? La question est légitime. Les variables globales sont **modifiables partout dans le code**, il est difficile de garder une trace de tous les endroits où elles peuvent être modifiées. **Cela rend plus complexe le debug de code**, et peut mener à des **effets de bord** (du code qui ne se comporte pas toujours comme prévu). C'est donc **un service que l'on se rend à soi-même** d'éviter d'en utiliser trop.

Bref, créons donc notre tableau.

```
const GRID: number[][] = [];
```

Notons bien les **double crochets [][]** signifiant que l'on déclare un tableau à deux dimensions, **un tableau de tableaux d'entiers**.

3.2.2 Générer notre tableau

Contrairement aux tableaux simples à une dimension, créer un tableau à deux dimensions uniquement à la déclaration n'est pas très pratique. Nous pourrions simplement écrire ceci :

```
const GRID: number[][] = [
  [0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0],
  [0, 0, 0, 0, 0, 0, 0]
]
```

Le résultat fonctionne, on se retrouve avec un tableau à deux dimensions de 7x6 cases. Cependant, notre solution n'est **pas très flexible, et surtout pas dynamique**. Si jamais à l'avenir nous voulons que les joueurs décident de la taille de notre grille, **nous ne pourrons pas** avec une telle solution.

Comment faire alors pour créer dynamiquement notre tableau à deux dimensions ? Nous savons utiliser une boucle for pour parcourir un tableau existant :

```
for (let i = 0 ; i < tableau.length ; i++) {
  console.log(tableau[i]);
}
```

Créer un tableau d'une taille dynamique n'est **guère plus compliqué**. Il suffit de remplacer la valeur tableau.length dans la condition par **la longueur que l'on veut pour notre tableau (ici ROWS)**. Ensuite, dans la boucle, on s'occupera d'**ajouter une case au tableau**. Cette opération répétée un certain nombre de fois, notre tableau est généré.

```
let tableau: number[] = [];
for (let i = 0 ; i < ROWS ; i++) {
  tableau.push(0);
}
```

Ici nous ajoutons donc la valeur 0 dans notre tableau un certain nombre de fois. Combien de fois ? ROWS fois (6 fois donc). Superbe, nous avons généré un tableau à une dimension rempli de 0. [0, 0, 0, 0, 0, 0]

L'objectif était cependant de générer un tableau à **deux dimensions**. Chaque ligne ajoutée au tableau n'est pas supposée contenir **0**. Chaque ligne est supposée contenir **un autre tableau**. Et cet autre tableau va, lui, devoir contenir des **0**.

Pour générer nos tableaux, il suffit de **remplacer 0 par []**. Il faut bien penser également à **changer notre type de données**. Nous allons créer un tableau de tableaux.

```
let tableau: number[][] = [];
for (let i = 0 ; i < ROWS ; i++) {
  tableau.push([]);
}
```

Nous avons maintenant un tableau contenant **6 tableaux vides** : [[], [], [], [], [], []] La dernière étape est de **remplir ces 6 tableaux avec des 0**. Et **nous savons le faire à présent**, à chaque tour de boucle, **une autre boucle** va venir peupler nos sous-tableaux.

```
let tableau: number[][] = [];
for (let i = 0 ; i < ROWS ; i++) {
  tableau.push([]);
  for (let j = 0 ; j < COLUMNS ; j++) {
    tableau[i].push(0);
  }
}
```

C'est un peu compliqué quand même. Il y a plusieurs points sur lesquels il faut **être vigilants** ici.

— Nous avons **deux boucles l'une dans l'autre**. La boucle à l'intérieur va s'exécuté **à chaque tour de boucle extérieure**.

— Afin de parcourir nos lignes, nous utilisons l'index **i**. Mais **nous devons aussi parcourir nos colonnes dans la boucle intérieure**. Nous utilisons donc un **deuxième index** nommé **j**. **Il ne faut surtout pas confondre les deux !**

— Nous voulons insérer des **0** dans chaque **sous-tableau** de notre tableau principal. Pour cela, nous accédons à chacun des sous-tableaux avec **tableau[i]** comme s'il s'agissait de valeurs normales d'un tableau. Simplement car c'est ce qu'elles sont.

PFIOU ! Vous avez tout compris ? Super ! Si ce n'est pas le cas, **c'est normal**. Prenez le temps de souffler, d'assimiler tranquillement le code, de **l'exécuter, de jouer avec**. Si vous n'arrivez toujours pas à comprendre, nous reviendrons de toute façon **régulièrement** sur ce genre de structures.

3.2.3 Générer la grille HTML

Nous avons créé notre tableau, celui qui va servir pour le code, pour **la logique du jeu**. Nous allons l'utiliser pour insérer des pièces, vérifier que les tours sont valides, et vérifier la victoire d'un joueur. Néanmoins, nous avons besoin d'**afficher ce tableau**. Pour cela, nous allons utiliser la même logique que précédemment. Nous allons générer un tableau en utilisant des HTML. **Ce tableau ne sera que l'image**, la représentation graphique de notre tableau typescript. Il faudra donc **le mettre à jour nous-mêmes** lorsque l'on modifie notre tableau typescript.

Une première solution est donc de créer "à la main" notre grille, comme nous l'avons fait au début pour notre tableau.

```
<div>
  <div>
    <div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div>
  </div>
  <div>
    <div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div>
  </div>
  <div>
    <div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div>
  </div>
  <div>
    <div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div>
  </div>
  <div>
    <div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div>
  </div>
  <div>
    <div></div><div></div><div></div><div></div><div></div><div></div><div></div><div></div>
  </div>
</div>
```

Pouah, c'est bien pire que le tableau ! Et encore ! Ce n'est même pas le résultat final que l'on veut ! Nous voulons **donner un id unique à chaque case du tableau**, et également **donner des classes à nos divs**. Allez-y, faites le si vous voulez, je vais personnellement préférer **générer dynamiquement cette grille**.

Nous allons donc plutôt **générer du HTML directement dans notre code typescript**. Mais nous ne savons pas encore comment faire... Il est temps d'apprendre à manipuler quelques **fonctions natives du langage**. La première fonction qui nous sera utile est la suivante :

```
document.getElementById(id)
```

Avec un bagage suffisant en anglais, le nom de cette fonction est assez intuitive. Nous allons récupérer un **élément HTML** (une balise) de notre **document** (index.html) **à l'aide de son id**.

Si j'ai le code html suivant :

```
<!DOCTYPE html>
<html>
  <head>
  </head>
  <body>
    <div id="grille"></div>
    <script src="index.js"></script>
  </body>
</html>
```

J'ai une div qui porte comme id "grille". Je peux donc "récupérer" cette div **en tant qu'objet typescript** (une variable un peu spéciale) avec le code suivant.

```
let maGrilleHTML : HTMLElement = document.getElementById("grille");
```

Je déclare une variable et je stocke mon élément html dedans. Ici j'indique le type de données qui est donc un HTMLElement. Ce n'est pas obligatoire de l'indiquer car l'interpréteur est capable de déterminer tout seul le type de données de ce qu'on lui donne.

Ok super, on a donc maintenant récupéré notre div, **qu'est-ce qu'on en fait ?** On va découvrir **une propriété de notre objet nommée innerHTML**.

Le fonctionnement est le même que **monTableau.length**, pour accéder à la propriété, on écrit **maDiv.innerHTML**. La différence est que lorsque l'on utilise monTableau.length, on ne fait que **lire la valeur** de la longueur du tableau. Ici on va pouvoir **lire et écrire** dans cette propriété innerHTML.

Essayez par exemple de taper le code suivant :

```
let maGrilleHTML : HTMLElement = document.getElementById("grille");

maGrilleHTML.innerHTML = "<p>Salut les amis !</p>"
```

Si vous le lancez avec un index.html comme celui de l'exemple du dessus, vous devriez avoir maintenant votre message qui s'affiche sur la page web. Félicitations, **vous venez de modifier votre document HTML dynamiquement grâce à typescript !** Si vous inspectez votre code HTML à présent, vous constaterez que **la balise <p> se trouve à l'intérieur de notre **. En effet les plus habiles d'entre vous auront traduit "innerHTML", cette propriété gère le contenu HTML **à l'intérieur de la balise ciblée**.

Donc tout ce que nous avons à faire, c'est mettre dans notre "grille" le code HTML de notre grille. Nous parlons toujours de cet immonde amas de imbriquées. **innerHTML est de type String**. Nous devons donc lui **fournir une chaîne de caractères**. Nous allons **construire notre chaîne de caractères** à l'aide de boucles imbriquées comme nous l'avons fait pour le tableau.

Le saviez-vous ? Comme 3 + 3 = 6, nous pouvons **"additionner" des chaînes de caractères** : "Jean-" + "Paul" = "Jean-Paul". Ce n'est pas une réelle addition, nous appelons cette opération la **concaténation**. En typescript on utilise le signe + pour la réaliser. Nous allons utiliser cette opération pour concaténer nos morceaux de chaîne de caractère afin d'obtenir notre chaîne finale à fournir à innerHTML.

Reprenons le code de génération de tableau.

```
let tableau : number[][] = [];
for (let i = 0 ; i < ROWS ; i++) {
  tableau.push([]);
  for (let j = 0 ; j < COLUMNS ; j++) {
    tableau[i].push(0);
  }
}
```

Nous avons déjà la structure permettant de générer des lignes et des colonnes. Il suffit maintenant de remplacer notre tableau par la chaîne de caractères à créer et les push de tableaux par des concaténations de div. Allons-y étape par étape :

```
// Nous récupérons notre div "grille"
let maGrilleHTML : HTMLElement = document.getElementById("grille");

// Nous déclarons une chaîne vide
let htmlGrid : string = "";
for (let i = 0 ; i < ROWS ; i++) {
  // Nous ajoutons la balise ouvrante de notre rangée
  htmlGrid += "<div>";
  for (let j = 0 ; j < COLUMNS ; j++) {
    // Nous ajoutons notre cellule
    htmlGrid += "<div></div>";
  }
  // Nous ajoutons la balise fermante de notre rangée
  htmlGrid += "</div>";
}
// Nous donnons notre chaîne construite à notre grille
maGrilleHTML.innerHTML = htmlGrid
```

Nous créons donc une chaîne de caractères vide. À chaque tour de boucle **nous ajoutons une ligne**. Dans cette ligne, à chaque tour de boucle imbriquée, **nous ajoutons une cellule**. Enfin lorsque l'on a mis assez de cellules, nous **fermons notre div de ligne**. Puis nous répétons l'opération. Enfin nous terminons en donnant à notre grille la chaîne de caractères que nous avons généré.

Si vous testez votre code maintenant, **rien ne devrait s'afficher**. En effet, une div n'est qu'un container vide n'affichant rien. Nous allons changer cela. Créez maintenant une feuille de styles CSS et mettez-y le code suivant :

```
.row {
  display: flex;
}

.cell {
  border: 2px solid black;
  height: 50px;
  margin: 2px;
  width: 50px;
}
```

N’oubliez pas d’insérer votre feuille de styles dans votre index.html à l’aide de la balise <link>

```html
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="index.css">
  </head>
  <body>
    <div id="grille"></div>
    <script src="index.js"></script>
  </body>
</html>
```

Nous pouvons maintenant tester et nous rendre compte que... **Toujours pas !** Nous avons créé deux classes, row et cell mais **aucune div n’a ces classes**. Il est temps de reprendre notre double boucle et d’y **ajouter nos classes**.

```ts
// Nous récupérons notre div "grille"
let maGrilleHTML : HTMLElement = document.getElementById("grille");

// Nous déclarons une chaîne vide
let htmlGrid: string = "";
for (let i = 0 ; i < ROWS ; i++) {
  // Nous ajoutons la balise ouvrante de notre rangée
  htmlGrid += '<div class="row">';
  for (let j = 0 ; j < COLUMNS ; j++) {
    // Nous ajoutons notre cellule
    htmlGrid += '<div class="cell"></div>';
  }
  // Nous ajoutons la balise fermante de notre rangée
  htmlGrid += "</div>";
}
// Nous donnons notre chaîne construite à notre grille
maGrilleHTML.innerHTML = htmlGrid
```

Notez les deux changements subtils, chaque rangée possède maintenant **une classe row**, et chaque cellule possède **une classe cell**. Maintenant si on teste. on devrait avoir une magnifique grille. Si vous n’avez toujours rien, cette fois **ce n’est plus normal**, il faut revoir chaque étape.

Bon c’est très bien, on a l’affichage mais il nous manque encore une toute dernière étape. On voulait **donner un id unique à chacune des cases de notre grille**. Pour être sûrs d’avoir un id unique et **qui a du sens pour chaque cellule**, on va leur donner comme nom **la position à laquelle elles se trouvent dans la grille**. Et pour cela, on va utiliser nos deux index **i et j**.

```ts
// Nous récupérons notre div "grille"
let maGrilleHTML : HTMLElement = document.getElementById("grille");

// Nous déclarons une chaîne vide
let htmlGrid : string = "";
for (let i = 0 ; i < ROWS ; i++) {
  // Nous ajoutons la balise ouvrante de notre rangée
  htmlGrid += '<div class="row">';
  for (let j = 0 ; j < COLUMNS ; j++) {
    // Nous créons un id pour la cellule
    let cellId : string = i + "," + j;
    // Nous ajoutons notre cellule
    htmlGrid += '<div class="cell" id="' + cellId + '"></div>';
  }
  // Nous ajoutons la balise fermante de notre rangée
  htmlGrid += "</div>";
}
// Nous donnons notre chaîne construite à notre grille
maGrilleHTML.innerHTML = htmlGrid
```

Et voilà ! Nous avons terminé notre grille d'affichage ! Cette dernière étape n'est pas très visuelle mais elle nous sera utile plus tard lorsque nous voudrons **accéder à une case de la grille en particulier**. Nous avons maintenant initialisé nos éléments de jeu, celui-ci peut donc commencer...

3.3 Attente d'une entrée utilisateur

C'est la première fois que nous allons demander à l'utilisateur d'**intéragir avec notre programme**. Comme nous l'avons vu en cours, un programme informatique s'exécute **de haut en bas sans interruption et se termine lorsqu'il atteint le bas**. C'est bien ce qui se passe ici. Notre programme se termine donc une fois que le bas a été atteint (c'est faux). Comment faire alors pour **attendre une entrée de notre utilisateur ?**

3.3.1 Les événements

Nous allons utiliser ce que l'on appelle des **événements**. Le principe est simple, nous créons une fonction, et **nous l'attachons à un événement en particulier**. Qu'est-ce qu'un événement ? Il y en a de toutes sortes. Par exemple, **le clic de la souris, une touche du clavier qui est appuyée, la fin du chargement de la page web, la fin d'un décompte de chronomètre, etc...** Un événement est comme dans la vraie vie, simplement **un fait qui se produit à un moment donné**. Et donc lorsque cet événement se produit, nous voulons qu'une fonction en particulier s'exécute.

Le principe des événements est assez simple, son fonctionnement interne l'est beaucoup moins. Nous l'étudierons plus précisément en SLAM. Tout ce que nous allons faire ici, c'est **apprendre à nous en servir**. Sans plus tarder, voici la structure que nous allons utiliser :

```
1 function maFonction(event): void {
2   // Ici je fais des trucs lorsque mon utilisateur appuie sur une
        touche
3 }
4
5 document.addEventListener("keydown", maFonction)
```

Nous appelons une fonction **document.addEventListener** (ajouter un écouteur d'événement). Nous lui donnons deux paramètres. Le premier est le nom de l'événement à suivre, ici **"keydown"** pour l'appui sur une touche de clavier. Le second est **la fonction qu'il faudra appeler lorsque cet événement sera déclenché**, ici **maFonction**. Cela indique à la machine "Sois vigilante à cet événement (écoute-le) et s'il se produit, appelle ma fonction".

Les différents événements existants sont répertoriés ici. `https://developer.mozilla.org/fr/docs/Web/Events` Comme vous pouvez le voir, **il y en a vraiment beaucoup**, et la plupart sont très obscurs. Mais dans la liste nous pouvons retrouver "keydown" que nous utilisons ici.

Maintenant nous pouvons voir que nous avons déclaré **un paramètre** que l'on a nommé **"event"** dans notre fonction. C'est un paramètre que l'on doit indiquer et **qui recevra des informations sur l'événement qui s'est produit** lorsque notre fonction sera appelée. Nous appelons ce paramètre "event", mais comme toujours, **c'est une convention**, et nous pourrions l'appeler "pommeDeTerre" que ça ne changerait rien à son contenu. Nous récupérons donc des informations sur l'événement qui s'est déclenché. **Selon l'événement, les informations sont différentes**. Quelles sont donc les informations qui nous intéressent ici ?

3.3.2 Les détails de notre événement

Comme nous écoutons un événement ”appui sur une touche de clavier”, l’information principale que l’on veut, c’est tout simplement **”quelle touche a été pressée ?”** Nous allons utiliser deux propriétés différentes afin de varier les plaisirs. Elles sont très similaires en apparence mais ont **une différence subtile** qu’il est important de comprendre. Ces propriétés sont **event.key** et **event.code**. La première, event.key **prend en compte la langue du clavier**, ainsi que différents paramètres modifiants comme l’appui sur shift ou ctrl. event.code **s’attache uniquement à la position de notre touche sur le clavier, suivant un modèle qwerty**. En résumé, selon qu’on soit en qwerty ou en azerty, event.key renverra une valeur différente pour la lettre Q par exemple. event.code quant à lui renverra toujours ”KeyQ” lorsque l’on appuie sur la lettre A en azerty (qui correspond donc à Q en qwerty).

Afin d’intéragir avec le jeu, nous allons opter pour le comportement suivant : Lorsqu’un joueur veut insérer un jeton dans l’une des 7 colonnes, il devra appuyer sur la touche numérotée correspondante. Nous allons donc utiliser la rangée de touches chiffrées au dessus de la ligne ”azertyuiop” de notre clavier. Les codes de ces touches sont **”Digit1”, ”Digit2”, ”Digit3” etc...** Or, notre fonction va être appelée **à chaque appui** d’une touche sur le clavier, **peu importe laquelle**. Il va donc falloir faire attention à la touche appuyée lors de l’appel de la fonction, faire le tri, et **uniquement exécuter notre code lorsque la touche est un ”DigitX”**.

Nous pourrions donc faire une série de IF afin de vérifier si la touche appuyée est ”Digit1”, ”Digit2”, etc... Une autre solution **plus élégante et moins volumineuse** est simplement de chercher le terme ”Digit” dans notre chaîne de caractères. En effet, si la chaîne contient ”Digit”, alors bingo, c’est l’une des touches qui nous intéresse. Comment vérifier qu’**une chaîne contient une autre chaîne à l’intérieur ?** Nous allons utiliser la méthode **string.includes(substring)** `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/includes`. Le fonctionnement est très simple, un exemple sera plus parlant :

```
let phrase: string = "Il fait beau aujourd'hui"

// Retourne VRAI car "beau" est contenu dans "Il fait beau aujourd'hui"
if (phrase.includes("beau")) {
  console.log("Oui c'est vrai qu'il fait beau");
}
```

La méthode retourne un booléen qui vaut VRAI si le paramètre a été trouvé dans la chaîne, et FAUX sinon. C'est pile ce qu'il nous faut pour vérifier si oui ou non "Digit" fait partie du code de la touche pressée :

```
function maFonction(event): void {
  let keyCode: string = event.code

  if(!keyCode.includes("Digit")) {
    // La touche appuyée n'est pas un digit ! On s'arrête là, cette touche ne nous intéresse pas
    return
  }

  // Ici on sait que la touche nous intéresse, on continue le traitement
}

document.addEventListener("keydown", maFonction)
```

Ici on teste si le code de la touche appuyée contient "Digit", si ce n'est pas le cas, **on quitte immédiatement la fonction** avec l'instruction return. On est donc **assurés par la suite** que l'on a bien appuyé sur une touche qui nous intéresse. Maintenant il faudrait connaître exactement **le numéro de cette touche** afin de le communiquer à nos autres fonctions qui vont s'occuper de placer un jeton dans notre grille. Le chiffre exact de la touche appuyée est **le tout dernier caractère de notre code** (dans "Digit5" c'est le "5" qui nous intéresse).

Comment réussir à **isoler ce dernier chiffre** ? Retournons voir notre documentation du type String. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String`. Nous pouvons lire dans la description que pour extraire une sous-chaîne de notre chaîne, nous pouvons utiliser **maChaine.substring(index)**. Parfait, nous voulons récupérer **le dernier caractère de notre chaîne** (dans "DigitX", nous voulons le "X"). **Les chaînes de caractère possèdent la même propriété length que les tableaux**. Pour récupérer le dernier caractère, nous pouvons donc faire **"chaine.length - 1"**, comme avec notre **"tableau.length - 1"** pour le dernier indice du tableau.

```
function maFonction(event): void {
  let keyCode: string = event.code

  if(!keyCode.includes("Digit")) {
    // La touche appuyée n'est pas un digit ! On s'arrête là, cette touche ne nous intéresse pas
    return
  }
  let characterIndex: number = keyCode.length -1;
  let column: number = keyCode.substring(characterIndex);

}

document.addEventListener("keydown", maFonction)
```

Testons notre code... Aïe, le compilateur me jette parce que je mets un string dans une variable number ! En effet, nous avons extrait notre sous-chaîne de caractères qui correspond à un nombre, mais **qui est tout de même une chaîne**. La valeur **4** est différente de la valeur **"4"** (c'est pourquoi les types de donnée sont importants). Il faut que l'on convertisse notre chaine **"4"**, en un entier **4**. Pour cela, nous allons utiliser la fonction **parseInt(string)**. Cette fonction va tenter de convertir une chaîne de caractères en un nombre. Si elle n'y parvient pas, elle renvoie **NaN (Not a Number, une valeur très particulière)** sinon elle renvoie le nombre correspondant.

```
let columnString: string = keyCode.substring(characterIndex);
let column: number = parseInt(columnString);
```

3.4 Exécution du tour de jeu

Notre prochaine étape sera de **lancer le tour de jeu** suite à l'entrée de l'utilisateur. Nous allons pour cela créer une fonction qui s'occupera de gérer le déroulement du jeu. Cette fonction pourra, grâce à l'entrée utilisateur, essayer d'exécuter le tour de jeu. Toutes les étapes successives ayant été détaillées, nous pouvons donc d'ores et déjà savoir ce que nous allons devoir implémenter dans notre fonction.

```
function play(column: number): void {
  // TODO: Gérer l'insertion de jeton

  // TODO: Gérer la vérification de victoire

  // TODO: Gérer la vérification d'égalité

  // TODO: Gérer le changement de tour
}
```

Nous indiquons **en paramètre** la colonne que l'on a récupéré. Le reste n'en est qu'à l'état de tâches à réaliser. Il vous faudra donc compléter cette fonction **au fur et à mesure** de l'avancement de notre projet.

3.5 Insérer un jeton

Pour insérer un jeton, nous avons besoin de savoir **dans quelle colonne** celui-ci sera inséré. Notre fonction prendra donc **un paramètre** qui sera la colonne en question. Nous savons également que cette fonction doit **notifier si l'insertion s'est bien passée ou non**.

```
function insertCoin(column: number): boolean {
  return true;
}
```

3.5.1 La logique

Voici donc une première base pour notre fonction, déduite uniquement de ce que l'on sait sur **son rôle**. Il nous reste à présent à **parcourir notre tableau** afin de "faire tomber" notre pièce **jusqu'à la dernière case vide**. Comment exploiter cette idée de gravité ? Notre jeton tombe tant qu'il n'a pas un jeton sur lequel s'appuyer. Concrètement, **tant que la case du dessous est vide**, notre jeton tombe. Il y a donc deux cas d'arrêt.

— Le premier est de **trouver un jeton sur la case du dessous**.
— Le second est de **se trouver sur la dernière case**, et là c'est le "sol" du plateau qui nous arrête.

```
function insertCoin(column: number): boolean {
  for (let i = 0 ; i < GRID.length ; i++) {
    // Si nous avons atteint le bas OU si la case du dessous est
       prise
    if (i + 1 >= ROWS || GRID[i + 1][column] != 0) {
      // Poser notre jeton sur la case actuelle
    }
  }
  return true;
}
```

Super, il nous reste à **insérer le bon jeton dans la case courante**. Pour cela, **il nous manque une information**. Celle du tour de jeu. Nous allons devoir créer à nouveau une variable globale pour retenir le tour de jeu. Nous appellerons cette variable **currentPlayerTurn**, ce sera un entier qui vaudra soit 1 soit 2. Il est également possible d'utiliser un booléen, mais un entier sera plus pratique à manipuler. Cet entier sera simplement **la valeur que l'on mettra dans notre grille** à l'endroit de l'insertion de jeton.

```
function insertCoin(column: number): boolean {
  for (let i = 0 ; i < GRID.length ; i++) {
    // Si nous avons atteint le bas OU si la case du dessous est
       prise
    if (i + 1 >= ROWS || GRID[i + 1][column] != 0) {
      GRID[i][column] = currentPlayerTurn
    }
  }
  return true;
}
```

Notre insertion fonctionne bien maintenant, nous pouvons tester l'appel de notre fonction et voir le tableau se remplir (dans le debugger). Il reste néanmoins un cas à gérer. En effet, pour le moment notre fonction insert toujours une valeur, et renvoie toujours vrai. Nous ne gérons pas encore le cas de **la colonne remplie**. Pour cela, il va falloir vérifier **au tout début de la fonction** si notre colonne est pleine. Si c'est le cas, alors nous allons **immédiatement retourner FAUX**.

Une fois cette dernière étape gérée, nous n'avons plus qu'à **appeler notre fonction insertCoin dans la fonction play**. Le flux de la fonction play s'exécute uniquement si l'insertion s'est bien passée. Si l'insertion de jeton renvoie FAUX, **la fonction play doit s'arrêter immédiatement**. En effet, le joueur doit proposer un coup valide avant que l'on puisse passer à la suite...

3.5.2 Et l'affichage alors ?

Nous insérons notre pièce, mais nous ne la voyons pas dans le tableau HTML, que se passe-t-il ? En effet la grille HTML **ne possède pas les informations** de notre tableau 2D. Pour qu'elle se mette à jour, il faut **lui notifier nous-mêmes les changements**. Nous pourrions le faire **directement dans notre fonction insertCoin**, mais souvenons-nous que nous voulons **découper nos tâches à raison d'une par fonction**. L'affichage est donc **le problème d'une autre fonction**.

Pour changer l'affichage, nous avons besoin d'effectuer **la même opération que dans notre tableau**. Mais une fois notre pièce insérée, **nous avons déjà fait nos calculs**, comment retrouver la modification **en dehors de notre fonction ?** La réponse est simple : C'est compliqué. Nous pourrions comparer case par case notre grille HTML et notre tableau 2D, et trouver la différence, ou simplement tout copier à chaque tour. Mais avec **une petite modification** à notre fonction insertCoin, il est possible de faire bien plus simple.

Notre fonction insertCoin renvoie **vrai ou faux** selon si l'insertion s'est bien passée ou non. C'est la seule information qui nous intéresse pour savoir si le tour de jeu s'est bien déroulé. Seulement maintenant, nous aimerions une deuxième information : **où est-ce que la pièce est tombée**. L'instruction return ne nous permet de renvoyer **qu'une seule valeur**, comment faire pour obtenir cette **deuxième information ?** Il faut savoir d'abord que nous n'avons pas besoin de connaître **la colonne** dans laquelle nous insérons la pièce, **nous la connaissons déjà**. Nous nous intéressons donc uniquement à **la ligne**. Notre fonction insertCoin pourrait renvoyer **la valeur de la ligne** simplement. Cependant nous avons maintenant **perdu l'information** de : est-ce que l'insertion s'est bien passée. Il nous faut donc **retrouver cette information binaire**. Heureusement, nous pouvons **combiner** les deux informations.

- Si l'insertion s'est bien passée, alors il nous suffit de renvoyer **le numéro de ligne insérée qui sera entre 0 et ROWS**.
- Si l'insertion s'est mal passée, alors nous pouvons simplement renvoyer **une valeur qui n'est pas entre 0 et ROWS**. Nous avons plusieurs choix à notre disposition, une convention dans plusieurs langages est de renvoyer **-1** pour dire qu'il y a une erreur (lorsque -1 n'est pas un résultat possible). En TS, nous avons **une autre valeur possible**, un mot-clé qui signifie **l'absence de valeur**, il s'agit de **null**. Nous allons utiliser ce mot-clé, cela vous permettra de vous familiariser avec.

Les modifications à prévoir sont donc d'une part, **renvoyer le numéro de ligne à la place de VRAI**, et d'autre part, **renvoyer null à la place de FAUX**. C'était tellement simple, j'aurais pu le dire plus tôt au lieu d'écrire 3 paragraphes entiers juste pour ça.

3.5.3 Oui mais on n'a toujours rien affiché là

C'est vrai qu'il nous faut encore écrire la fonction qui va **modifier la grille HTML**. Cette fonction va donc prendre deux informations en paramètres, **la ligne et la colonne** de la case modifiée. Elle pourrait prendre également **le tour de jeu**, mais comme l'information est globale, ce n'est pas obligatoire.

```
function insertHTMLCoin(row: number, column: number): void {
  document.getElementById()
}
```

Mmh... Mais il nous manque encore quelques informations. Comment je fais référence à ma cellule ? Qu'est-ce que je suis censé en faire de cette cellule ? En regardant à nouveau notre code de génération de grille, nous pouvons voir que nous avions nommé nos cellules avec **un id unique**. Cet id était **"i,j"**. Transposé à chaque ligne et colonne, cela donnait donc "0,0", "0,1", "0,2" etc... Jusqu'à "5,6". C'est donc maintenant que nous allons pouvoir nous servir de ces ids, grâce à eux, nous pouvons faire référence à ces lignes et colonnes simplement.

```
function insertHTMLCoin(row: number, column: number): void {
  document.getElementById(row + "," + column)
}
```

Nous avons maintenant fait référence à **la case que nous avons modifié** dans insertCoin. Il nous reste une étape cependant, c'est **modifier cette cellule**. Comment faire pour modifier **l'affichage d'une div HTML ?** Il faut **changer son style**, il nous faut donc un style à appliquer aux cellules du joueur 1 et aux cellules du joueur 2. Ces styles seront réutilisés pour plusieurs cellules. Ce qui correspond le mieux à nos critère, c'est donc les **classes CSS**. Nous allons créer **une classe CSS pour chaque couleur de jeton**. Reprenons notre feuille de styles CSS, et ajoutons ce dont nous avons besoin.

```
.row {
  display: flex;
}

.cell {
  border: 2px solid black;
  height: 50px;
  margin: 2px;
  width: 50px;
}

.red {
  background-color: red;
}

.yellow {
  background-color: yellow;
}
```

Nous faisons simple, sobre et efficace. Une classe red pour la couleur rouge, une classe yellow pour la jaune. Maintenant il faut que ces couleurs correspondent **chacune à un joueur en particulier**. Nous allons décider que le joueur 1 aura la couleur rouge, et le joueur 2 la jaune. Nous pourrons donc faire référence à "red" et "yellow" en utilisant currentPlayerTurn. Il faut donc **faire correspondre 1 à "red" et 2 à "yellow"**. Comment faire ?

ASTUCE ! Les indices d'un tableau de taille n sont compris entre **0 et n-1**, et un tableau peut contenir n'importe quel type de données, y compris des chaînes de caractères. Nous pourrions nous servir de cela pour faire notre correspondance. Soit le tableau suivant :

```
const PLAYER_COLORS: string[] = ["", "red", "yellow"]
```

Nous avons maintenant un tableau avec :
— PLAYER_COLORS[0] qui vaut **une chaîne vide** (on s'en fiche en fait)
— PLAYER_COLORS[1] qui vaut **"red"**
— PLAYER_COLORS[2] qui vaut **"yellow"**
Nous pouvons donc maintenant utiliser **PLAYER_COLORS[currentPlayerTurn]** pour avoir soit "red" soit "yellow" en fonction du tour du joueur.

Il nous reste une fonction à connaître, celle qui nous permet de **modifier les classes** d'une balise HTML dynamiquement en javascript. Une balise peut avoir **plusieurs classes**, contrairement à l'id qui est unique. La modification des classes est donc un peu plus compliquée que la modification d'un id :

```
// Accéder à la liste des classes
monElement.classList
// Ajouter une classe
monElement.classList.add("maClasse")
// Retirer une classe
monElement.classList.remove("maClasse")
```

Revenons maintenant à notre fonction, **nous avons tous les élements** pour notre modification d'affichage.

```
function insertHTMLCoin(row: number, column: number): void {
  const currentPlayerColor: string = PLAYER_COLORS[currentPlayerTurn]
  document.getElementById(row + "," + column).classList.add(
      currentPlayerColor)
}
```

Il nous reste plus qu'à appeler notre fonction **après l'insertion réussie d'un jeton** dans notre fonction play, et nous avons un affichage de jetons fonctionnel.

3.6 Vérifier la victoire d'un joueur

Et quelle suite ! Nous voici à **l'étape la plus complexe du projet**. Accrochez-vous, nous allons voir ensemble le fonctionnement de cette partie en détail. Avant de foncer dans le code, nous avons besoin de **comprendre** ce que nous devons faire ici.

3.6.1 La théorie

Pour gagner au puissance 4, un joueur doit **aligner 4 jetons de la même couleur**. Soit horizontalement, soit verticalement, soit diagonalement. Cet alignement de jetons peut être présent **partout sur la grille**, **dans tous les sens** et **toutes les directions**. Une chose est sûre : Nous allons devoir **parcourir entièrement notre grille de jeu** afin de trouver un alignement quelque part ! Il faut également garder à l'esprit que l'ordinateur est **une incroyable machine de calcul**, il ne faut pas avoir peur de lui faire réaliser **beaucoup d'opérations**. Afin de se simplifier la vie, nous allons donc nous permettre de parcourir la grille entièrement. Et pour chaque case, nous allons effectuer **4 vérifications** :

- un parcours pour trouver un alignement horizontal
- un parcours pour trouver un alignement vertical
- un parcours pour trouver un alignement diagonal
- un parcours pour trouver un alignement diagonal

Attendez, on a dit deux fois la même chose là non ? En effet, il existe une subtilité concernant le parcours en diagonal. **Deux formations sont possibles** :

- soit un escalier montant vers la **gauche** et descendant vers la **droite**.
- soit un escalier montant vers la **droite** et descendant vers la **gauche**.

Nous effectuerons donc un parcours pour chaque formation diagonale également. Super, nous avons décidé de ce que nous allons faire, un parcours entier de la grille pour trouver des jetons alignés dans chaque configuration possible. Mais... En quoi cela peut-il bien consister de **"trouver des jetons alignés"** ?

3.6.2 Trouver des jetons alignés

Prenons un cas simple, un alignement horizontal. Nous avons représenté les états de nos cellules par **trois entiers différents**. 0 indique une case vide, 1 indique une case occupée par un jeton J1 et 2 indique une case occupée par un jeton J2. La victoire d'un joueur sur un plan horizontal sous-entend une condition. Nous devons trouver **le même nombre** (1 ou 2) **sur 4 cases d'affilée**. Par exemple : [0, 1, 1, 2, 1, 0, 1] n'est pas une ligne gagnante, il y a deux 1 alignés, mais c'est tout. Par contre : [2, 1, 1, 1, 1, 0, 2] est une ligne gagnante, il y a quatre 1 alignés.

Afin de se simplifier la vie, nous pouvons même **déjà savoir à l'avance le nombre que l'on cherche**. En effet, nous faisons la vérification **à la fin du tour d'un joueur**. Cela veut dire que ce joueur en particulier **vient de poser un jeton**. Si personne n'avait gagné jusque là, ce joueur est **le seul à pouvoir gagner à la fin de son tour** (jusqu'ici, c'est logique). La valeur que nous chercherons à trouver plusieurs fois d'affilée est donc la valeur **du joueur dont c'est le tour actuellement**.

En résumé :

- Nous parcourons **entièrement** notre grille case par case.
- **Pour chacune de ces cases**, nous vérifions si cette case contient **la valeur du joueur qui vient de poser un jeton**. Puis nous vérifions si **la case suivante** contient également la valeur du joueur qui vient de poser un jeton. Et nous répétons cette vérification **4 fois** (IN_A_ROW fois en fait).
- Si **à tout moment** cette vérification s'avère fausse, alors **nous passons à l'alignement suivant** et nous répétons le processus.
- Si nous trouvons 4 jetons identiques à la suite, notre joueur a gagné !
- Si aucun alignement n'est gagnant **pour cette case**, alors nous passons à la case suivante.
- Si nous arrivons à la fin du parcours du tableau sans trouver d'alignement, alors notre joueur n'a pas gagné à ce tour.

3.6.3 Parcours de la grille

Nous allons tout d'abord commencer par parcourir entièrement notre grille de jeu. Tableau à deux dimensions, la grille va se parcourir de la même manière qu'elle a été construite : avec deux boucles imbriquées.

```
for (let i = 0 ; i < ROWS ; i++) {
  for (let j = 0 ; j < COLUMNS ; j++) {
    // Faire des trucs avec GRID[i][j]
  }
}
```

Nous avons ici un code qui va parcourir le tableau, case par case. Il nous reste à effectuer nos vérifications. Comme vu précédemment, une vérification va consister à regarder si **la case courante et les cases d'après** contiennent le nombre associé au joueur qui vient de jouer. Les **"cases d'après"** dépendent de l'alignement que l'on est en train de vérifier (horizontal, vertical, diagonal...).

Nous savons que nous cherchons 4 jetons alignés, nous pourrions donc simplement faire 4 if. Cependant, dans un souci de **flexibilité**, nous allons utiliser notre outil préféré, celui qui permet de répéter plusieurs fois un même bout de code : **la boucle !** En effet, avec une boucle, si demain nous décidons de changer le nombre de jetons à aligner, nous aurons simplement notre constante IN_A_ROW à changer, et **toute la vérification continuera à marcher**. Pour notre première vérification horizontale, nous allons donc écrire notre boucle **directement dans nos deux boucles imbriquées**.

```
for (let i = 0 ; i < ROWS ; i++) {
  for (let j = 0 ; j < COLUMNS ; j++) {
    for (let k = 0 ; k < IN_A_ROW ; k++) {
      if (GRID[i][j + k] != currentPlayerTurn) {
        // PERDU, le jeton n'est pas le bon, on s'en va
      }
    }
  }
}
```

Quelle horreur, le code commence à devenir sacrément moche. Et nous avons encore un traitement à faire à la place du commentaire. Et nous n'avons fait que la vérification horizontale. Pas besoin d'écrire plus de code pour se rendre compte qu'il va falloir **découper notre algorithme en fonctions**. Notre parcours de grille est bien comme il est, **il serait difficile de l'extraire dans des fonctions**. Non, ce que nous allons pouvoir extraire, c'est **la vérification**.

Il y a 4 vérifications à faire, nous allons donc créer **4 fonctions différentes**. Elles auront un fonctionnement similaire, mais nous pourrons plus clairement comprendre leur fonctionnement séparément. De plus, créer des fonctions va nous permettre de simplement renvoyer **VRAI ou FAUX** si nous avons trouvé un alignement ou non. Nous aurons donc dans notre parcours de grille uniquement à appeler nos 4 fonctions dans un IF. Les 4 fonctions renvoient **un booléen**, il suffit qu'**un seul d'entre eux soit vrai** pour que l'on sache qu'un alignement victorieux a été trouvé. Si l'on se souvient des opérateurs de combinaison logique, l'opérateur qui nous sera utile ici est le **OU**, || en typescript.

```
for (let i = 0 ; i < ROWS ; i++) {
  for (let j = 0 ; j < COLUMNS ; j++) {
    if (foundHorizontalAlignment(i, j) ||
        foundVerticalAlignment(i, j) ||
        foundDiagonalLeftAlignment(i, j) ||
        foundDiagonalRightAlignment(i, j)) {
      return true // VICTOIRE, on a trouvé !
    }
  }
}
return false // On est arrivés tout au bout sans rien trouver
```

Nous avons ici le code final de notre fonction de vérification de victoire. Résumons-le en quelques phrases. Pour **chaque case de notre grille**, **si** je trouve un alignement horizontal, **ou si** je trouve un alignement vertical, **ou si** je trouve un alignement diagonal. **Alors** je renvoie VRAI, **le joueur a gagné**. **Sinon**, à la fin de mon parcours, je renvoie FAUX car aucun alignement n'a été trouvé, le joueur **n'a pas gagné** à ce tour.

Il nous reste maintenant à (ré)écrire le code de vérification de victoire pour chaque type d'alignement. **Le principe est toujours le même** : On est sur une case, et on vérifie si les suivantes ont la même valeur. Ce que l'on entend par "les suivantes" **dépend de l'alignement que l'on vérifie**. On va donc faire appel à notre **boucle for**, qui va de 0 à IN_A_ROW. Dans cette boucle, on va simplement vérifier que la case que l'on atteint contient bien la valeur du joueur que l'on cherche à valider. Si ce n'est pas le cas, **on s'en va**, on retourne FAUX car le résultat ne convient pas, **l'alignement n'est pas valide**.

```
function foundHorizontalAlignment(row, column): boolean {
  if (column + IN_A_ROW > COLUMNS)
    return false;

  for (let i = 0 ; i < IN_A_ROW ; i++) {
    if (GRID[row][column + i] != currentPlayerTurn) {
      return false;
    }
  }
  return true;
}
```

Ce code représente un parcours classique comme on a déjà pu en voir plusieurs. La complexité réside ici dans la manipulation du tableau à deux dimensions. En effet, nous avons **en paramètre** la ligne et la colonne de la cellule sur laquelle nous faisons notre vérification. Notre case de départ pour la vérification est donc **GRID[row][column]**. Si nous voulons vérifier que **les prochaines cases horizontales** ont bien la valeur attendue, il va falloir faire varier la dimension horizontale dans notre vérification, la deuxième dimension ici représentée par [column]. Commençant à 0, et allant jusqu'à 3, **notre index i est le candidat idéal pour cette variation**. Nous écrivons donc **GRID[row][column + i]** qui va partir de notre cellule de départ, et avancer horizontalement. Si à tout moment, **GRID[row][column + i]** vaut autre chose que **currentPlayerTurn**, alors nous partons sur le champ, en disant "FAUX, cet alignement n'est pas bon !". Si **jusqu'au bout de nos 4 étapes**, GRID[row][column + i] a toujours valu currentPlayerTurn, alors félicitations, **nous avons trouvé un alignement horizontal**, on peut renvoyer "VRAI, l'alignement est bon !".

Cette logique n'est **pas simple à comprendre**, cependant, une fois qu'elle est saisie, le travail est **presque terminé**. En effet, adapter le code ci-dessus à une vérification verticale est trivial :

```
function foundVerticalAlignment(row, column): boolean {
  if (row + IN_A_ROW > ROWS)
    return false;

  for (let i = 0 ; i < IN_A_ROW ; i++) {
    if (GRID[row + i][column] != currentPlayerTurn) {
      return false;
    }
  }
  return true;
}
```

Saurez-vous trouver le changement ? Nous faisons maintenant varier **row** au lieu de **column**. La vérification se fait donc à présent sur **GRID[row + i][column]** plutôt que **GRID[row][column + i]**. Et c'est tout ! Maintenant on descend dans la grille plutôt qu'aller à droite.

Il nous reste maintenant les diagonales, L'une des diagonales reprend juste **le même principe** que les deux fonctions du dessus, **mais en combiné**, la voici :

```
function foundDiagonalRightAlignment(row, column): boolean {
  if (column + IN_A_ROW > COLUMNS || row + IN_A_ROW > ROWS)
    return false;

  for (let i = 0 ; i < IN_A_ROW ; i++) {
    if (GRID[row + i][column + i] != currentPlayerTurn) {
      return false;
    }
  }
  return true;
}
```

Au lieu de partir juste **vers le bas**, ou **juste vers la droite**, à chaque étape nous partons **à la fois vers le bas et vers la droite**. Cela est simplement retranscrit en écrivant **GRID[row + i][column + i]**.

Enfin, presque aussi simple, mais avec **une petite subtilité**, le parcours diagonal gauche demande à ce que l'on retire l'index i aux colonnes plutôt que l'ajouter. Le code final sera donc :

```
function foundDiagonalLeftAlignment(row, column): boolean {
  if (column - (IN_A_ROW - 1) < 0 || row + IN_A_ROW > ROWS)
    return false;

  for (let i = 0 ; i < IN_A_ROW ; i++) {
    if (GRID[row + i][column - i] != currentPlayerTurn) {
      return false;
    }
  }
  return true;
}
```

Et voilà ! Notre vérification est terminé ! C'était simple finalement, non ? Non ? Bon ok, le problème est **dur à formaliser**, mais lorsque l'on parvient à **le décomposer avec rigueur**, chaque morceau est **à notre portée**. Tous ces morceaux **mis bout à bout** forment un algorithme relativement complexe. À partir de là, tout paraîtra simple à présent.

Ah ! Une dernière chose ! Avez-vous remarqué ces if devant chaque boucle dans nos fonctions d'alignement ? Ils permettent d'une part de ne pas aller voir dans **des cases de tableau qui n'existent pas**, et d'autre part de **s'épargner des vérifications inutiles**. En effet, Si nous sommes par exemple à la colonne 6, nous savons que **nous ne trouverons pas 4 jetons alignés vers la droite** vu qu'il n'en reste qu'un après, celui de la colonne 7.

3.7 Vérifier le match nul

Regardez par exemple la vérification du match nul. C'est simple finalement. Encore que... Il faut réussir à **le formaliser**, encore une fois. Qu'est-ce qu'un match nul au puissance 4 ? C'est lorsque **la grille est pleine**, et qu'**aucun joueur n'a encore gagné**. Si nous vérifions le match nul à chaque tour après la vérification de victoire, nous sommes déjà **sûrs qu'aucun joueur n'a encore gagné**. Il nous reste donc à vérifier si **la grille est pleine**. Avec notre formalisme, une grille non pleine est une grille qui contient **au moins une case qui vaut 0**. En effet, 0 représente **une case vide** de jeton.

Vérifier le match nul revient donc à vérifier qu'il n'y a **aucun 0 dans notre grille**. Prenons le problème à l'envers. **Il n'y a pas match nul dès l'instant où je trouve un 0 dans ma grille**. Il faut donc **parcourir toute la grille à la recherche d'un 0**. Plutôt simple maintenant après tout ce que l'on a fait.

```
function isGridComplete(): boolean {
  for (let i = 0 ; i < ROWS ; i++) {
    for (let j = 0 ; j < COLUMNS ; j++) {
      if (GRID[i][j] == 0) {
        return false;
      }
    }
  }
  return true;
}
```

Nous parcourons notre grille à l'aide de la fameuse **double boucle imbriquée**. Chaque case est passée en revue. Si une case **GRID[i][j]** est trouvée contenant la valeur 0, alors **on sort immédiatement en retournant FAUX**. Notre grille n'est pas complète. Si nous parvenons au bout de notre parcours sans jamais trouver de 0, alors **on peut retourner VRAI**, notre grille est complète. Voilà, notre fonction est terminée, pas besoin de nous attarder ici, nous touchons au but !

3.8 Changer de tour de jeu

Encore plus simple, toujours plus simple, nous allons **changer le tour de jeu**. La variable qui décide du joueur en train de jouer est donc currentPlayerTurn. Elle vaut 1 quand le joueur 1 joue, elle vaut 2 quand le joueur 2 joue. À chaque tour, on change.

```
function changeTurn(): void {
  if (currentPlayerTurn == 1) {
    currentPlayerTurn = 2;
  } else {
    currentPlayerTurn = 1;
  }
}
```

Des questions ? Non ça va, on s'ennuierait presque... Allez, divertissons-nous un peu. Il existe un moyen d'écrire ce code en **une seule ligne** d'affectation. Vous connaissez l'opérateur **NOT**, représenté en typescript par " !". Lorsqu'une variable vaut FAUX, elle devient VRAI, et inversement. Nous savons aussi que FAUX vaut 0 et VRAI vaut 1. Si nous pouvions représenter currentPlayerTurn avec **0 et 1** plutôt que **1 et 2**, nous pourrions donc les inverser juste avec un NOT. Pour passer de 0 à 1 et de 1 à 2, il suffit simplement d'ajouter 1, et inversement. Pour inverser notre tour de jeu en une seule ligne, nous pouvons donc écrire :

```
currentPlayerTurn = (!(currentPlayerTurn - 1)) + 1
```

C'était divertissant ? Non ? Tant pis, de toute façon c'est moche et difficile à lire, nous resterons donc sur la première solution. Nous nous sommes égarés, dépêchons-nous de terminer.

3.9 Reset de la grille et nouvelle partie

Nous avons terminé le tour de jeu, nous savons quoi faire lorsque le joueur courant n'a pas gagné, nous changeons de tour. Il reste un cas à gérer, il s'agit de **la victoire d'un joueur**. En effet, lorsqu'un joueur gagne, il ne faut plus que la partie se déroule comme avant. **Les joueurs ne doivent plus pouvoir insérer de jetons**. Ils ne doivent que pouvoir relancer une partie.

3.9.1 Bloquer la partie

Le processus de tour de jeu se déroule dans notre fonction play. Il faut donc empêcher cette fonction d'exécuter tout son code si jamais la partie est terminée. Tout d'abord, nous avons besoin de **retenir cette information "partie terminée"**. L'information étant binaire (vraie ou faux), le plus logique reste d'utiliser **une variable booléenne**. Elle sera également globale, et nous l'appellerons **gameEnded**. Afin d'empêcher la fonction play d'exécuter son code, il suffira de **vérifier au tout début de celle-ci si gameEnded vaut VRAI**. Et si c'est le cas, simplement **exécuter un return** qui nous fera immédiatement sortir de là.

3.9.2 Relancer le jeu

Le jeu va maintenant devoir **se relancer** d'une manière ou d'une autre. Nous allons attendre de l'utilisateur qu'il **appuie sur la touche Entrée** de son clavier afin de relancer une partie. Reprenons le code qui gère les entrées utilisateur.

```
function maFonction(event): void {
  let keyCode: string = event.code

  if(!keyCode.includes("Digit")) {
    // La touche appuyée n'est pas un digit ! On s'arrête là, cette
       touche ne nous intéresse pas
    return
  }
  let characterIndex: number = keyCode.length -1;
  let column: number = keyCode.substring(characterIndex);

  // ...
}
```

Il y a maintenant une nouvelle touche qui nous intéresse, **la touche Entrée**. Mais cette touche nous intéresse uniquement si gameEnded vaut VRAI.

Afin de varier les plaisirs, nous utiliserons **event.key** plutôt que **event.code** pour vérifier que l'utilisateur a appuyé sur Entrée.

```
function maFonction(event): void {
  if (gameEnded && event.key == "Enter") {
    // On reset le jeu
  }

  let keyCode: string = event.code

  if(!keyCode.includes("Digit")) {
    // La touche appuyée n'est pas un digit ! On s'arrête là, cette
       touche ne nous intéresse pas
    return
  }
  let characterIndex: number = keyCode.length -1;
  let column: number = keyCode.substring(characterIndex);

  // ...
}
```

Et voilà, nous avons de quoi relancer notre partie. Il nous reste à effectivement relancer notre partie. Que veut donc dire relancer notre partie ? Il faut **réinitialiser tous nos éléments de jeu**, à savoir :

- Notre tableau 2D
- Notre grille HTML
- Le tour de jeu (on va faire commencer le perdant de la partie)
- La variable gameEnded

3.9.3 Le reset du tableau

Pour remettre à 0 notre tableau, nous allons simplement **le parcourir**, et pour chacune des cases, **remplacer sa valeur par 0**. Vous devriez à ce stade maîtriser cet aspect, le parcours de tableau 2D, et le changement des valeurs qu'il contient.

3.9.4 Le reset de la grille HTML

Peut-être un peu plus délicat, mais pas plus complexe. Il va s'agir ici d'effectuer le même parcours. La différence réside dans le fait de simplement **retirer nos classes** "red" et "yellow" de chaque cellule de notre grille. Nous avons vu plus haut **monElement.classList.remove()** qui va exactement faire ce que l'on veut. Afin de s'assurer de bien **tout retirer à chaque fois**, nous allons appeler cette fonction deux fois, une fois avec "red" et une fois avec "yellow". Cela nous assurera que peu importe la couleur de la case, elle sera retirée, et si la classe n'existe pas, **ce n'est pas un problème** car la fonction ignorera simplement l'appel .

3.9.5 Le reset des variables

Très simple pour gameEnded, il suffit de **le repasser à FAUX**. Pour currentPlayerTurn, il faut s'assurer que **sa valeur a bien changé** par rapport à celle du joueur qui vient de gagner.

3.9.6 Le tout dans une fonction

Toutes ces étapes doivent être appelées **les unes à la suite des autres**, mais nous n'allons pas le faire directement dans notre fonction d'événement. En effet, **ce n'est pas son rôle**. Nous allons plutôt **créer une fonction resetGame()** qui va s'occuper d'appeler les différentes étapes. Et nous appellerons simplement cette fonction dans notre fonction d'événement.

Et voilà qui conclut notre TP ! Nous avons maintenant (normalement) un jeu de puissance 4 fonctionnel. Il n'est pas parfait, surtout au niveau de l'affichage où il est un peu pauvre. Cela nous donne donc **une bonne marge de progression** pour la partie libre de ce projet.

4 Les fonctionnalités bonus

Notre jeu est encore bien austère, et très classique. Il est temps d'y **ajouter de la fantaisie**. Laissez parler votre créativité, et **peaufinez le jeu du mieux que vous pouvez**. Voici quelques pistes de réflexion pour améliorer le jeu.

- Notre jeu manque d'indications, les lignes et colonnes ne sont pas numérotées, on ne sait pas quel joueur est en train de jouer, ni quel joueur a quelle couleur, il n'y a pas de score non plus qui retient les différentes parties. On ne sait pas non plus où se trouve la ligne gagnante.
- Les graphismes sont très simples, on peut imaginer faire des jetons plus jolis, un plateau de jeu mieux travaillé. Pourquoi pas même des animations en javascript ?
- On peut aussi modifier les règles du jeu, faire un puissance 3/4/5/6 sur des grilles de taille variable. Pouvoir jouer à plus de 2 joueurs peut-être.
- Pourquoi pas des power-ups ? effacer un jeton adverse, ou une ligne entière, pouvoir poser un jeton volant, ou toutes autres choses.
- On pourrait imaginer pouvoir enregistrer une partie, pour pouvoir la revoir plus tard.
- Pourquoi ne pas essayer de faire une IA basique qui joue au hasard pour un mode 1 joueur ?
- Mettre du son dans le projet ?
- Les possibilités sont infinies, ne vous laissez pas enfermer par les quelques propositions énoncées ci-dessus. Appropriez-vous le projet, exprimez votre créativité.

Ce ne sont que quelques idées, mais rien n'est imposé, à vous de réfléchir et de faire de ce jeu **votre jeu**. Cette partie est notée sur 5 points, **il ne s'agit pas de blinder le jeu de fonctionnalités**, mais plutôt d'en faire un ensemble cohérent, et **un chouette jeu** surtout ! Bon courage dans cette mission ;)