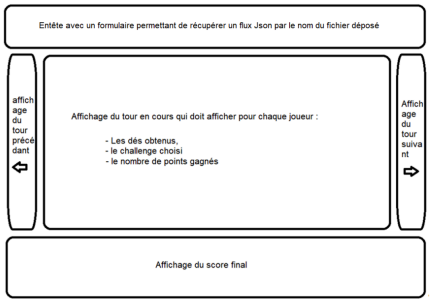
RAPPORT SAE 11 & 12 - Jeu de Yams

Kadir EKICI / Ferencz Roudet / Gauthier CLAUDEL
TP4

Partie W11 (Kadir)

Dans notre groupe, nous nous sommes décidés de nous séparer en 2 groupes: Un groupe de un en W11 et un groupe de deux en P11.
Moi (Kadir), j'ai fait la partie W11.
Pour démarrer mon code, avant tout, j'ai rassemblé tout ce qui était demandé pour ma partie dans le sujet complet donné dans le Moodle. J'ai tout d'abord commencé une page HTML pour faire les 'layouts' et faire en sorte que ça ressemble plus au schéma du Moodle (voir images)



Contrainte : Tous les positionnements se feront avec **Flexbox**.

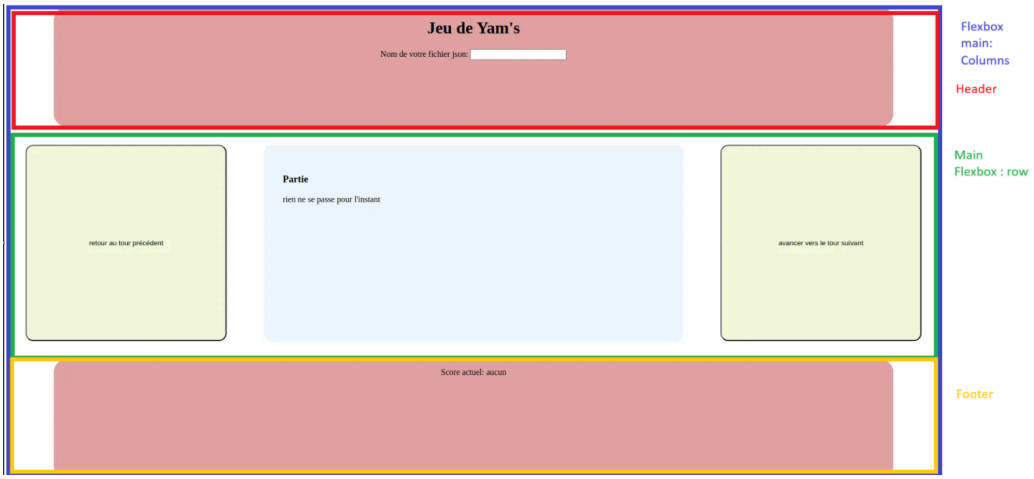
Le jury appréciera :

- L'aspect ergonomique des pages web,
- la possibilité aux pages d'être "responsives" donc de s'adapter à plusieurs tailles d'écran (ordinateur de bureau, tablette, smartphone).

- Schéma du Moodle

- Page HTML/CSS fait par moi

Ceci était la toute première partie et j'avais utilisé les flexbox pour faire les pages évidemment.



Ensuite je m'étais mis à coder la partie JavaScript, pour commencer j'ai essayé de faire seulement afficher les noms du fichier json exemple. Ce que j'avais pas réussi au début.

Lors de la deuxième séance de SAE, j'ai fini par faire afficher les noms dans le header mais puisque nous n'avions toujours pas fait le cours en JavaScript sur les fetch j'ai du chercher sur internet comment faire.

Sans réel succès, juste après le cours, à la maison je reprends et réessaie avec le code donné par le professeur et j'ai réussi à faire mon affichage des joueurs.

Plus tard, dans un cours de SAE encore, j'avais discuté avec un professeur lors du cours pour m'aider à régler un problème sur mon code: le bouton 'recherche' que j'utilise ne fonctionne pas, je ne trouvais pas comment récupérer la valeur écrite dans le champ de texte où il faut écrire l'ID de la partie et comment utiliser le bouton pour récupérer cette valeur et l'utiliser dans le fetch.
Heureusement à l'aide du professeur mon code avait fonctionné mais c'était écrit à sa façon au professeur, et personnellement j'aime pas quand le code que j'ai n'est pas écrit de fonction à ce que je puisse le comprendre moi-même (sur le long terme). Alors j'ai tout réécrit moi-même et j'ai réussi à le faire fonctionner (je n'ai malheureusement pas de screenshot du code du professeur mais dans sa façon, il avait modifié mon code HTML et JS pour faire marcher le bouton, et moi j'ai refait en modifiant seulement mon code JS, ce qui m'a offert une meilleure lisibilité pour moi-même):

```
document.getElementById("search-button").addEventListener('click', search_item)
```

- Tout d'abord on ajoute un EventListener pour détecter le click sur le bouton, et lorsque c'est le cas, on lance la fonction search_item()

```
function search_item()
{
    search = document.getElementById("nomjson").value;
    ShowGame(currentRound, search);
}
```

- Et puis dans la fonction search_item(), on récupère la valeur de l'input de type 'text' et on l'utilise ensuite pour faire la recherche du fetch.

Un peu plus tard, je suis tombé sur un problème. Lorsque nous avons enfin eu l'API yams de disponible, j'ai pensé au début que cela allait être compliqué de remplacer mon code qui affiche le score des joueurs et le nom des joueurs à cause de la structure de l'API.
Mais au final, c'était assez simple.
Voici un exemple de l'utilisation de l'API:

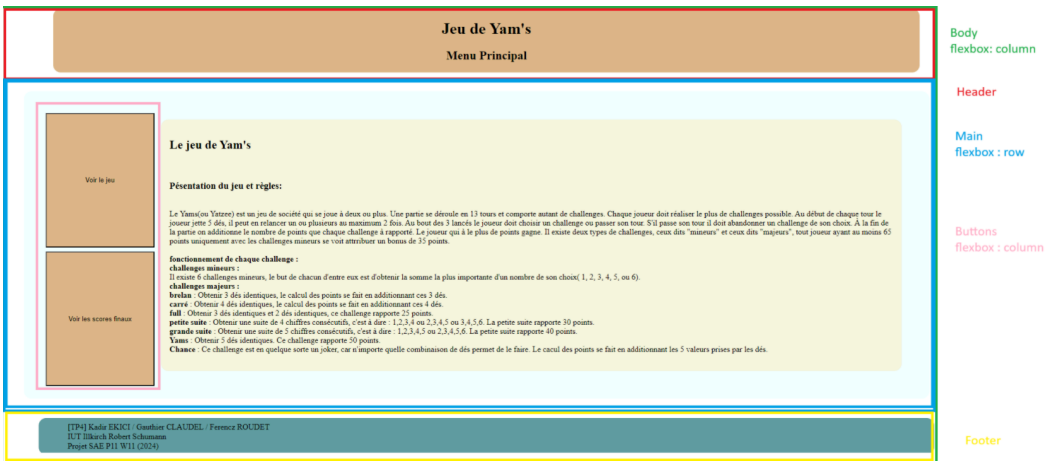
```
function ShowGame(currentRound, search)
{
    // --- DESCRIPTION --- //
    // Cette fonction sert à afficher le jeu en entier avec tous les joueurs individuellement.
    // currentRound est un entier qui définit le round actuel, on prends en compte que le currentRound
    // est une valeur valide (entre 1 et 13) car elle est vérifiée avant de lancer la fonction.
    // ---

    fetch('http://yams.iutrs.unistra.fr:3000/api/games/'+search+'/rounds/'+currentRound)
    .then(response => {
        if (!response.ok) {
            //Code pour gérer le cas où la réponse n'est pas correcte
            throw window.alert(["Veuillez mettre un ID de partie VALABLE et appuyez sur 'Rechercher'!"]
        }
        return response.json(); // Convertir la réponse en JSON
    })
    // AFFICHER LES DONNEES
    .then(data => {
```

Ici, la fonction ShowGame() est appelée lorsqu'on appuie sur le bouton 'recherche' avec le round actuel et la variable search, qu'on a vu avant et qu'on obtient grâce à la fonction search_item().

On peut également noter que j'ai ajouté une ligne en cas d'erreur dans le fetch ('throw window.alert([...]);'), ce qui permet à l'utilisateur de savoir si leur ID de partie est bien valable.

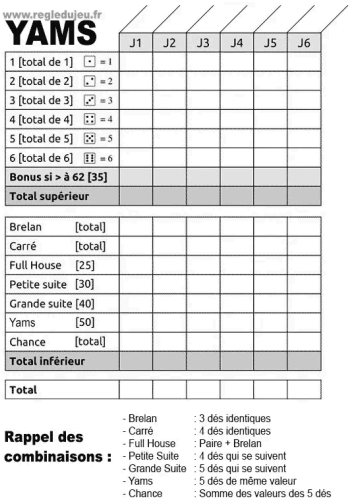
Dans un cours de SAE, un professeur m'avait donné l'idée de créer plusieurs pages pour rendre l'expérience utilisateur meilleure. Et trouvant l'idée bonne, je me suis vite lancé sur la création d'une page d'accueil et une page de scores finaux.

La page d'accueil n'était pas vraiment un problème, seulement deux boutons avec des liens 'href' à mettre et le style (CSS) à bien ajuster:



Cette page ne contient pas de script JS.

Pour la page des scores finaux, j'y ai travaillé pendant les vacances de noël. Je me suis inspiré de l'idée de Ferencz (de mon groupe) qui m'a proposé de faire un tableau qui ressemble à ceux utilisé dans les jeux de yams (voir image pour exemple)



J'ai fini par faire un tableau qui ressemble à ceci :

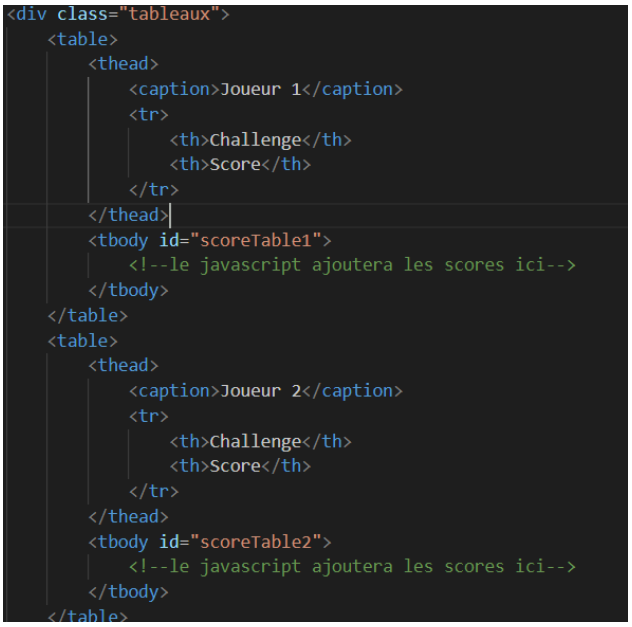
(Je ne peux pas prendre la page entière en screenshot malheureusement)

Sauf que pour faire ceci, j'ai eu de gros problèmes:

- Je ne vais pas écrire un par un chaque élément du tableau et deux fois (je trouve cela pas très beau et très encombrant sur la lecture du HTML)
- Le fetch ne prends pas les données assez vite pour les afficher sur le tableau

Pour y remédier:

- j'ai créé deux tableaux avec rien dedans, seulement le stricte minimum. Puisque dans notre projet, le jeu de yams est joué par seulement 2 joueurs, j'en ai pris l'avantage et fait seulement deux tableaux identiques:



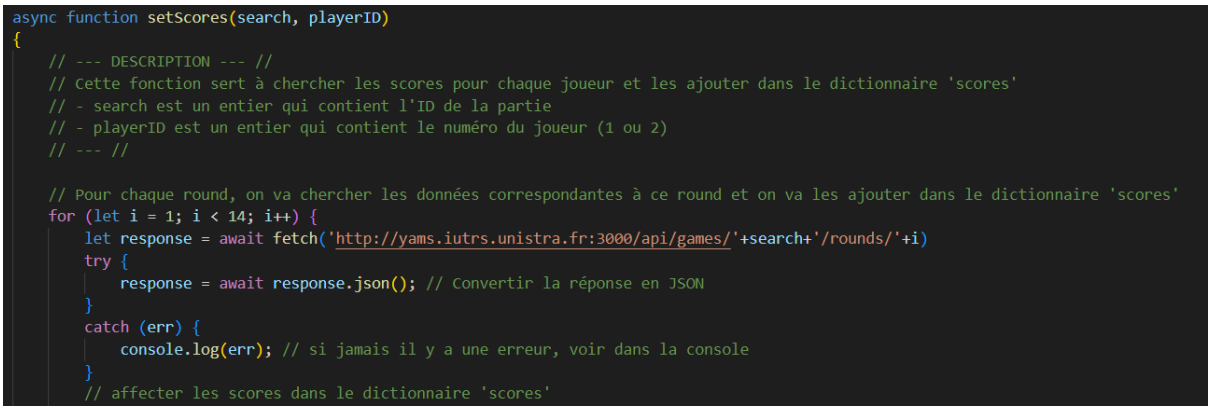
```
<div class="tableaux">
    <table>
        <thead>
            <caption>Joueur 1</caption>
            <tr>
                <th>Challenge</th>
                <th>Score</th>
            </tr>
        </thead>
        <tbody id="scoreTable1">
            <!--le javascript ajoutera les scores ici-->
        </tbody>
    </table>
    <table>
        <thead>
            <caption>Joueur 2</caption>
            <tr>
                <th>Challenge</th>
                <th>Score</th>
            </tr>
        </thead>
        <tbody id="scoreTable2">
            <!--le javascript ajoutera les scores ici-->
        </tbody>
    </table>
```

  Et dans le script JS j'ai ajouté une fonction qui sera exécutée en début de script qui remplira ce tableau de tous les challenges et avec aucune valeur (voir image plus haut dans le rapport pour revoir le tableau).
  J'ai créé une classe "tableaux" pour pouvoir les mettre côte à côte avec un flexbox (au lieu qu'un tableau soit au-dessus de l'autre, à cause de la structure de ma page).

- Le fetch lent:

Lorsque je lançais ma page et que je lançait une recherche, mon tableau se remplissait d'un mélange de différentes valeurs complètement aléatoire de ce qu'on pouvait trouver dans le fichier json. J'ai regardé et réécrit mon code en espérant régler ce problème mais j'avais toujours le même problème. Après avoir utilisé de (très) nombreux console.log() pour pouvoir déboguer mon code, j'ai vu dans la console que mon tableau s'actualisait bien avant que mon fetch finisse de récupérer toutes les données et ait le temps de les mettre dans le dictionnaire des scores (exploité pour créer le tableau).
Le problème est que nous n'avons pas appris comment faire attendre un code pour qu'il récupère les données et **ensuite** les affiche. Mais j'ai fait des recherches, et je suis tombé sur une solution qui peut fonctionner et que j'ai déjà utilisée dans d'autres codes (pas en JS): le **'await'**.

J’ai approfondi mes recherches et j’ai compris que pour utiliser le await dans une fonction, celle-ci doit être ‘asynchrone’ (suffit seulement d’ajouter ‘async’ avant la fonction) et on peut utiliser le await dans la fonction.



```
async function setScores(search, playerID)
{
    // --- DESCRIPTION --- //
    // Cette fonction sert à chercher les scores pour chaque joueur et les ajouter dans le dictionnaire 'scores'
    // - search est un entier qui contient l'ID de la partie
    // - playerID est un entier qui contient le numéro du joueur (1 ou 2)
    // --- //

    // Pour chaque round, on va chercher les données correspondantes à ce round et on va les ajouter dans le dictionnaire 'scores'
    for (let i = 1; i < 14; i++) {
        let response = await fetch('http://yams.iutrs.unistra.fr:3000/api/games/'+search+'/rounds/'+i)
        try {
            response = await response.json(); // Convertir la réponse en JSON
        }
        catch (err) {
            console.log(err); // si jamais il y a une erreur, voir dans la console
        }
        // affecter les scores dans le dictionnaire 'scores'
```

Sur l’image ci-dessus on a un exemple de l’utilisation du await. La fonction est asynchrone et on a changé le .then par un try…catch. Pourquoi? Car on ne fait plus juste un fetch, on met le résultat qu’on a **attendu** du fetch dans la variable ‘response’, ensuite on **essaie** de voir si on peut la convertir en json. Si on ne peut pas, on **attrape** l’erreur et on l’affiche dans la console (dans mon cas, avant setScores() on vérifie déjà si on aura une erreur mais bon, on sait jamais…).

Avec Gauthier nous nous sommes répartis la partie «p11» c’est à dire le c# et la génération du fichier json. J’ai (Ferencz) rédigé les algorithmes ainsi que les fonctions de vérification des challenges et Gauthier s’est chargé de les implémenter dans le main, et a écrit le reste du code concernant la partie de yams et la génération du json.

Écrire les algorithmes et fonctions de vérification de challenges ne m’a pas posé énormément de problèmes, le plus grand étant de ne pas avoir assez testé mes fonctions pour voir si l’algo fonctionnait dans tous les cas de figure. Gauthier et moi nous sommes toutefois toujours vite rendus compte lorsque quelque chose clochait. Cela m’a permit d’à chaque fois corriger la fonction concernée, voire réécrirer l’algorithme si nécessaire.

Pour les challenges mineurs je n’ai écrit qu’une seule fonction, car l’algorithme est le même pour chaque challenge, la seule différence réside dans le chiffre choisi.

```
public static int challenge_mineur(int challenge_choisi, int[] resultats_lancés){
/*
Fonction : challenge_mineur
Description : Calcule le score pour un challenge mineur, qui consiste à obtenir un total de dés d'une certaine valeur.
Cette fonction additionne les valeurs des dés qui correspondent à la valeur spécifiée dans le challenge choisi (par exemple, le total des 1 si "nombre1" est cho

Paramètres :
- challenge_choisi : Entier représentant la valeur du challenge (par exemple, 1 pour "nombre1", 2 pour "nombre2", etc.).
- resultats_lancés : Tableau contenant les résultats des dés obtenus lors du lancer.

Retour :
- int : Le score total obtenu pour ce challenge mineur, c'est-à-dire la somme des valeurs des dés correspondant à la valeur du challenge choisi.
*/

int resultat_challenge = 0;

// Boucle pour parcourir tous les dés et additionner ceux qui correspondent à la valeur du challenge choisi
for (int i = 0; i < resultats_lancés.Length; i++){
    if (challenge_choisi == resultats_lancés[i]){
        resultat_challenge += resultats_lancés[i]; // Ajoute la valeur du dé au total si elle correspond au challenge
    }
}

return resultat_challenge; // Retourne le score total du challenge mineur
```

Concernant les challenges majeurs, je ne détaillerai pas chaque algorithme, car certains sont extrêment simples. Par exmple pour le yams il suffit de stocker la première valeur de la liste de dés dans une variable et regarder si les valeurs qui suivent sont différentes de la première (si tab[i] ≠tab[0] , alors le challenge n'est pas réspecté et on renvoie 0). Pour le challenge «chance» il suffit d'additionner les valeurs de chaque dé.

Concerant les autres challenges, j'ai pensé à utiliser un système de couples. Je m'explique avec pour exemple la grande suite: On commence par trier la liste reçue en utilisant array.sort, puis on parcours cette liste en vérifiant à chaque fois si: tab[i] + 1 == tab[i +1]. Si c'est le cas on rajoute 1 à un compteur, et à la fin de la boucle si le compteur vaut 4 alors tout est bon (c'est comme si on avait quatre couples {1,2}, {2,3},{3,4}{4,5} par exemple). J'ai ensuite adapté cette idée de couples à chacun des autres challenges majeurs.

```
public static int grande_suite(int[] resultats_lancés){
/*
Fonction : grande_suite
Description : Calcule le score pour le challenge "Grande suite", qui consiste à obtenir une séquence de cinq dés consécutifs.
La séquence peut être composée de deux groupes distincts (par exemple 1-2-3-4-5 ou 2-3-4-5-6).
Si la séquence est présente, la fonction retourne un score de 40 points. Sinon, elle retourne 0.

Paramètres :
- resultats_lancés : Tableau contenant les résultats des dés obtenus lors du lancer.

Retour :
- int : Le score de la grande suite, soit 40 points si une grande suite est obtenue, sinon 0.
*/

int compteur_couples = 0; // Compteur pour les dés qui forment une séquence consécutive

// Trie les dés pour faciliter la recherche de séquences consécutives
Array.Sort(resultats_lancés);

// Parcourt les dés triés et vérifie si chaque dé forme une séquence consécutive avec le suivant
for (int i = 0; i < resultats_lancés.Length - 1; i++) {
    if (resultats_lancés[i] + 1 == resultats_lancés[i + 1]) {
        compteur_couples++; // Incrémente le compteur si les dés sont consécutifs
    }
}

// Si cinq dés consécutifs sont trouvés, c'est une grande suite
if (compteur_couples == 4) {
    return 40; // Retourne 40 points pour une grande suite valide
}

// Si moins de cinq paires consécutives sont trouvées, il n'y a pas de grande suite
return 0; // Pas de grande suite, donc score = 0
}
```

Gauthier s'est quant à lui occupé de rédiger le reste du code c#. Il s'est beaucoup servi des structures que ce soit pour générer le fichier json, faire le profil des joueurs ou encore sauvegarder chaque tour d'une partie.

```
public struct json{
    /*
    Structure : json
    Description : Contient les différents champs pour l'écriture dans le fichier JSON
    Champs :
    - parameter : Paramètres de la partie
    - player : Tableau qui contient les deux joueurs
    - round : Tableau qui contient les 13 rounds du Yams
    - final_result : Tableau qui contient les résultats finaux pour les deux joueurs
    */
    public parameter parameters;
    public player[] players;
    public round[] rounds;
    public final_result[] results;
    public json(parameter Parameters, player[] Players, round[] Rounds, final_result[] Final_Result){
        parameters = Parameters;
        players = Players;
        rounds = Rounds;
        results = Final_Result;
    }
}
```

Il a également écrit une fonction de plus de 120 lignes pour les lancés de dés(et les relances) qui mérite qu'on s'attarde dessus. À chaque lancé (tant que les lancés sont strictement inférieurs à 3), le joueur choisit l'indice des dés qu'il veut garder. Pour chaque indice de dé qu'on souhaite relancer on met un 1 à la place d'un 0 dans un autre tableau (mais au même indice, voir en ci-dessous).

```
int[] keeping_dice = new int[5] { 0, 0, 0, 0, 0 }; // Tableau pour marquer les dés à garder (1 = garder, 0 = relancer)
```

Grâce à ce tableau, on sait à quels indices relancer, et à quels indices il n'y a rien à faire. On répète ce procédé 3 fois au total.

Il a également du prendre en compte qu'un joueur peut ne pas vouloir relancer ses des le maximum de fois possible. Pour cela, il a créé un booléen «quitter» et à chaque début de boucle il vérifie s'il contient true ou false. Par défaut ce booléen contient false, si l'entrée de l'utilisateur est «q» alors on lui assigne true et le tour est joué.

```
// Si l'utilisateur entre "q", quitter la boucle
if (keep.ToLower() == "q"){
    quitter = true;
    break;
}
```

Pour savoir quels challenges ont déjà été faits ou pas par chaque joueur, Gauthier s'est servi d'un dictionnaire. Pour la clé il a choisit le code du challenge, pour la valeur une structure qui définit ce challenge (voir ci dessous). Gérer les challenges disponibles a d'ailleurs été la plus grande difficulté que Gauthier a rencontré.

```
challenges = new Dictionary<int, chalenge>{
    { 1, new chalenge("nombre1", "Nombre de 1", "Obtenir le maximum de 1") },
    { 2, new chalenge("nombre2", "Nombre de 2", "Obtenir le maximum de 2") },
    { 3, new chalenge("nombre3", "Nombre de 3", "Obtenir le maximum de 3") },
    { 4, new chalenge("nombre4", "Nombre de 4", "Obtenir le maximum de 4") },
    { 5, new chalenge("nombre5", "Nombre de 5", "Obtenir le maximum de 5") },
    { 6, new chalenge("nombre6", "Nombre de 6", "Obtenir le maximum de 6") },
    { 7, new chalenge("chance", "Chance", "Obtenir le maximum de points (le total des dés obtenus)") },
    { 8, new chalenge("petite_suite", "Petite suite", "Obtenir 1-2-3-4 ou 2-3-4-5 ou 3-4-5-6 (30 points)") },
    { 9, new chalenge("grande_suite", "Grande suite", "Obtenir 1-2-3-4-5 ou 2-3-4-5-6 (40 points)") },
    { 10, new chalenge("yams", "Yams", "Obtenir 5 dés de même valeur (50 points)") },
    { 11, new chalenge("brelan", "Brelan", "Obtenir 3 dés de même valeur (Somme des 3 dés identiques)") },
    { 12, new chalenge("carre", "Carré", "Obtenir 4 dés de même valeur (Somme des 4 dés identiques)") },
    { 13, new chalenge("full", "Full", "Obtenir 3 dés de même valeur + 2 dés de même valeur (25 points)") }};
score_mineur = 0;
score_majeur = 0;
total_score = 0;
```