

P4_Multiplatform_Rapport

P4_ Multiplatform

Prénom : *Aymerick*

Nom : *LAURETTA-PERONNE*

Objectif : Le but de ce projet est de concevoir un Puissance 4 sous l'opérateur système `Android` .

Résultat : Ce programme répond à l'objectif de créer un Puissance 4 proposant la possibilité de jouer en Multijoueur ou en Solo contre l'ordinateur.

[Lien GitHub](#) | [Lien vers le téléchargement de l'application en apk](#) | [Lien vers le PDF](#) | [Version Web](#)

Le code de cette application est partagé entre `Android` et `IOS` (vive kotlin), mais par faute de temps l'avancement de ce dernier n'a pas été commit et push afin de ne pas perturber le projet initial (à la rigueur je ferais une branche afin de travailler en toute sécurité).

Présentation Global

Multijoueur

Le mode **multijoueur** permet de jouer à deux (sur le même appareil). Lors de l'ouverture de l'application, l'utilisateur est convié à choisir son mode de jeux `SinglePlayer` ou `MultiPlayer` .

En choisissant le mode `MultiPlayer` , l'utilisateur est rediriger vers l'activité `LoginActivity` permettant de récupérer les pseudos des deux joueurs afin de lancer le jeux avec les pseudos de nos deux joueurs lors du click sur le bouton start.

On récupère les pseudos des joueurs du `EditText` :

```
val namePlayer1 = player1.text.toString()
val namePlayer2 = player2.text.toString()
```

Une fois que nous avons récupéré nos pseudos, on crée un `intent` et on place les pseudos à l'intérieur cela nous permettra de récupérer nos noms depuis une autre activité :

```
// Création de l'intent
val intent = Intent(this, GameActivity::class.java)

// Placer les noms des joueurs dans l'intent
intent.putExtra("namePlayer1", namePlayer1)
intent.putExtra("namePlayer2", namePlayer2)
```

Bien sur on fait une petite vérification afin d'alerter l'utilisateur si jamais il essaye de cliquer sur le bouton alors que les champs de saisie sont vides :

```
if(namePlayer1.isEmpty() || namePlayer2.isEmpty()){
    val toast = android.widget.Toast.makeText(
        this, "Please enter your nicknames", android.widget.Toast.LENGTH_SHORT
    )
    toast.show()
}
```

Si l'utilisateur a bien rempli tous les champs, on peut lancer l'activité :

```
// On démarre l'activité ciblée
startActivity(intent)

// On termine l'activité en cours
finish()
```

Singleplayer

Pour le mode `SinglePlayer` nous n'avons pas besoin de notre `LoginActivity`, on définit tout simplement :

- le joueur 1 : l'humain
- le joueur 2 : machine

On précise juste à l'utilisateur qu'il joue la pièce rouge.

Affichage du plateau

Pour l'affichage de notre plateau, nous allons utiliser la méthode `init` de `koltin`. Le code à l'intérieur du bloc `init` est le premier à être exécuté lorsque la classe est instanciée.

Un Puissance 4 possède 42 pions, de ce fait il nous faut un tableau de `6` par `7`.

On commence par définir nos variables `cols` et `rows` :

```
private val rows = 6 // Nombre de rangées
protected val cols = 7 // Nombre de colonnes
```

Maintenant il nous faut une double boucle `for` pour la conception de notre plateau :

```
init {
    for (i in 0 until cols) { // Pour chaque colonnes
        // Première Boucle
        for (j in 0 until rows) { // Pour chaque rangées
            // Deuxième Boucle
        }
    }
}
```

`until` de l'anglais `jusqu'à` en français donc de 0 `jusqu'à` nombre définit (par nos variables `cols` et `rows`)

Colonne (Première boucle)

Nous avons nos deux boucles, mais toujours pas de plateau visible. Nous allons créer un objet `LinearLayout` pour contenir les boutons dans la colonne, pour ce faire, on ajoute dans notre première boucle :

```
val linearLayout = LinearLayout(context)
```

On lui ajoute des paramètres, tels que la `largeur` et la `hauteur` :

```
val layoutParams1 = LinearLayout.LayoutParams(
    LinearLayout.LayoutParams.WRAP_CONTENT, // Largeur
    LinearLayout.LayoutParams.MATCH_PARENT // Hauteur
)
```

On définit notre `Largeur` prenant la `Largeur` nécessaire de son contenu

On définit notre `Hauteur` en prenant la `Hauteur` du parent

Les lignes suivantes permettent :

- L'orientation verticale du `LinearLayout`
- De définir les paramètres de mise en page du `LinearLayout` sur les paramètres de mise en page de la colonne
- Changer la gravité de sorte à ce que les boutons soient en bas

```
linearLayout.orientation = LinearLayout.VERTICAL
linearLayout.layoutParams = layoutParams1
linearLayout.gravity = android.view.Gravity.BOTTOM
```

Rangée (Deuxième boucle)

Pour la rangée c'est relativement la même chose, nous ferons un gain de pages en évitant de détailler :

```
// Création d'un LinearLayout pour la pièce
val piece = LinearLayout(context)

// On définit les paramètres de mise en page pour la pièce
val linearParams2 = LinearLayout.LayoutParams(pieceWidth, pieceWidth)

// On définit les marges de la pièce
linearParams2.setMargins(margin, margin, margin, margin)

// On définit les paramètres de mise en page de la pièce sur les paramètres de mise en
page de la pièce
piece.layoutParams = linearParams2

// On définit l'arrière-plan de la pièce sur l'image du cercle
piece.setBackgroundResource(R.drawable.circle) // set the background of the piece to
the circle image

// On ajoute la pièce à la colonne
linearLayout.addView(piece)
```

Vérification Joueur | Function

Pour vérifier si un joueur a gagné ou non, nous allons créer une fonction de type `Boolean` avec comme paramètre les `coordonnées` et le `joueur` (joueur qui est de type `integer` 1 ou 2).

```
private fun checkPlayer(x: Int, y: Int, player: Int): Boolean {}
```

On vérifie si la pièce est hors du plateau, si c'est le cas le joueur ne peut pas gagner :

```
if (x < 0 || x >= cols || y < 0 || y >= rows) return false
```

On vérifie si la pièce n'est pas la pièce du joueur (la pièce de l'adversaire) :

```
if (board[x][y] != player) return false
```

Mise à jour du plateau | Function

Création d'une fonction permettant de mettre à jour le plateau avec la nouvelle pièce dans la ligne et la colonne données (la ligne et la colonne sont basées sur 0).

```
protected fun update(row: Int, col: Int) {
```

On récupère la colonne où la pièce est déposée dans le plateau :

```
val col = mainLayout.getChildAt(col) as LinearLayout
```

On récupère la pièce dans la colonne à la position dans la rangée

```
val piece = col.getChildAt(row) as LinearLayout
```

On définit l'arrière plan de la pièce en fonction de la couleur du joueur :

```
piece.setBackgroundResource(  
    if (player == 1) R.drawable.circle_red  
    else R.drawable.circle_yellow  
)
```

Est gagnant | Function

Pour vérifier si gagnant ou non, on commence par créer une fonction de type `ArrayList<Pair<Int, Int>>` avec pour paramètre notre `player` :

```
protected fun isWon(player: Int): ArrayList<Pair<Int, Int>> {}
```

Ici, nous avons besoin de faire la vérification sur tout notre plateau. On va donc parcourir notre plateau avec une double boucle `for` :

```
for (i in 0 until rows) {  
    for (j in 0 until cols) {}  
}
```

On vérifie ensuite toutes les directions :

- Droite
- Bas
- Diagonale droite
- Diagonale Gauche

Les directions pour vérifier une victoire dans chaque direction :

```
val directions = arrayOf(Pair(0, 1), Pair(1, 0), Pair(1, 1), Pair(1, -1))
```

Ensuite, avec une autre double boucle `for`, nous allons vérifier pour chaque direction si il y a une victoire :

```
for ((direction_i, direction_j) in directions) {}
```

On crée des coordonnées suivant la direction :

```
val coordinates = ArrayList<Pair<Int, Int>>() // Liste des coordonnées
```

On ajoute les coordonnées pour vérifier une victoire dans la direction à la liste des coordonnées :

```
for (k in 0 until 4) coordinates.add(Pair(i + k * direction_i, j + k * direction_j))
```

On vérifie si les 4 coordonnées appartiennent au joueur, si c'est le cas, le joueur en question a gagné :

```
if (coordinates.all { (x, y) -> checkPlayer(x, y, player) }) return coordinates
```

Puis on retourne une liste vide si le player n'a pas gagné > `return ArrayList()`

Match Null | Function

Dans le cas où on remplit tout le plateau sans victoire cela coresspond à un `draw`.

Fonction relativement simple de type `Boolean`

```
protected fun isDraw(): Boolean {
```

On parcours notre plateau à l'aide d'une double boucle `for` :

```
    for (i in 0 until cols)
        for (j in 0 until rows)
```

S'il y a une pièce sur le plateau, alors le jeu n'est pas terminé :

```
        if (board[i][j] == 0) // If there is a piece on the board, the game is
            not finished
```

```
return false
```

Dans le cas contraire, c'est que le jeu s'est terminé par un match nul, plus aucune pièce ne peut être placée :

```
return true  
}
```

Jouer un coup | Function

Jouer un coup dans le plateau et retourner la ligne où la pièce a été lâché, ou -1 si la colonne est pleine.

Fonction ayant pour paramètre la `colonne` et `player` de type `Integer` :

```
protected fun move(col: Int, player: Int): Int {
```

On vérifie si la colonne est pleine, renvoie -1 pour indiquer que le coup était invalide et que la partie doit continuer jusqu'au joueur suivant (le cas échéant)

```
    if (this.board[col][0] != 0) {  
        return -1  
    }
```

On recherche la première ligne vide dans la colonne et on y dépose la pièce dans le cas échéant :

```
    // La rangée où la pièce sera déposée  
    var row = -1  
  
    // Parcourir la colonne de bas en haut  
    for(i in rows - 1 downTo 0) {  
        // Si la ligne est vide  
        if (this.board[col][i] == 0) {  
            // The row where the piece will be placed  
            row = i  
            // Arrêt de la boucle  
            break  
        }  
    }
```

On vérifie si la ligne est -1, la colonne est pleine et le déplacement est invalide :

```
    if (row == -1) {  
        return -1  
    }
```

On positionne la pièce dans le plateau et dans l'interface graphique `GUI` si le coup est valide :

```
    this.board[col][row] = player
    update(row, col)
    return row
}
```

IA | Function

L'implémentation de l'ia est pas terminé, afin d'avoir un mode `Singleplayer` relativement "opérationnel" voici la fonction actuellement dans la version hors prod :

L'IA permet de jouer contre l'ordinateur lors du choix `Singleplayer` .

```
protected fun ai(player: Int): Int {
    var bestScore = -1 // Le meilleur score
    var bestCol = -1 // La meilleure colonne

    // On parcourt les colonnes
    for (col in 0 until cols) {
        // Si la colonne est pleine, on l'ignore
        if (this.board[col][0] != 0) {
            continue
        }

        // Obtient le score de la colonne // au joueur
        val score = getScore(col, player)

        // Si le score est meilleur que le meilleur_score
        if (score > bestScore) {
            // Fixe le meilleur score au score de la colonne
            bestScore = score

            // Définit la meilleure colonne sur le numéro de colonne (0-6)
            bestCol = col
        }
    }

    // Si la meilleure colonne est -1, l'IA fait un mauvais coup
    if (bestCol == -1) {
        return -1
    }

    // Play
    return move(bestCol, player)
}
```

On récupère le score d'une colonne et d'un joueur donnés (AI) :

```

private fun getScore(col: Int, player: Int): Int {
    var score = 0 // Le Score

    // On parcourt les colonnes
    for (i in 0 until rows) {

        // Si la ligne est vide
        if (this.board[col][i] == 0) {
            score += 1 // Ajoute 1 au score
        } else {
            // Si la ligne est occupée par le joueur
            if (this.board[col][i] == player) {
                score += 2 // Ajout 2 au score
            } else {
                score -= 1 // Soustrait 1 au score
            }
        }
    }
    return score
}

```

TODO : Si le temps, faire en sorte que l'IA bloque le joueur si il aligne 3 pièce | rajouter le choix de colonne de façon aléatoire afin que l'IA ne fasse pas le même pattern | Si l'AI align 3 Pièces, faire en sorte qu'elle joue sa 4ème pièce si le jouer ne la pas bloqué.

Clear | Function

Fonction permettant de nettoyer le plateau en retirant toutes les pièces et de remettre la couleur du plateau à la normale (cette fonction est appelée lorsque l'utilisateur appuie sur le bouton recommencer) :

```

protected fun clear() {
    // Parcours le tableau et efface les pièces
    for (i in 0 until cols)
        for (j in 0 until rows){
            // Récupère les pièces
            var p: LinearLayout = mainLayout.getChildAt(i) as LinearLayout
            p = p.getChildAt(j) as LinearLayout

            // Régle l'alpha sur 1 (100% de transparence)
            p.alpha = 1f

            // Reset du tableau
            board[i][j] = 0

            val col = mainLayout.getChildAt(i) as LinearLayout

            // Get the piece
            val piece = col.getChildAt(j) as LinearLayout

            // On remet la pièce à sa couleur par défaut (vide)

```

```
        piece.setBackgroundResource(R.drawable.circle)
    }
}
```

Game | Function

Cette fonction est présente dans une méthode `init` dans un `OnClickListener` :

```
column.setOnClickListener {
    game(col)
}
```

Fonction qui à pour paramètre `col` la colonne cliquée par l'utilisateur :

```
override fun game(col: Int) { // col is the column clicked by the user
```

Si le jeu en cours est différent de `ended` (terminé) :

```
if (!ended){} // Si le jeu n'est pas terminé (Si différent de terminé)
```

On ajoute une pièce à la colonne en récupérant la ligne ou la pièce à été ajouté :

```
val row = play(col, player)
```

Si la pièce a été ajouté on met à jour le `tableau` :

```
if (row != -1) {
    update(row, col)
}
```

On fait appel à notre fonction `isWon` qui nous renvoie une liste de position gagnante :

```
val wonList = isWon(player)
```

Puis on vérifie si un joueur a gagné avec une condition :

```
when {
```

Si notre `wonList` est différent de vide c'est qu'il y a un gagnant :

```
wonList.isNotEmpty() -> {

    // On met fin au jeux
    ended = true

    // On met à jour le text en indiquant 'victoire'
    messageTextView.setText(R.string.won)

    // On désactive la possibiliter de cliquer sur le plateau
    mainLayout.isEnabled = false

    // On met en surbrillance les pièce victoire
    highlightWinning(wonList)
}
```

Si le jeu est un match nul (plus de coups) :

```
isDraw() -> {

    // On met fin au jeux
    ended = true

    // On désactive la possibiliter de cliquer sur le plateau
    mainLayout.isEnabled = false

    // On retire le nom des joueurs
    currentPlayerEditText.text = ""

    // On met en place le message " Match Null " à l'écran
    messageTextView.setText(R.string.draw)
}
```

Si la partie n'est pas encore terminée (pas de coups gagnants et pas de match nul) et que le joueur est joueur 1

```
else -> {
```

On passe du joueur 1 au joueur 2 et on met à jour le message pour indiquer que le joueur actuel est le joueur 2 :

```
player = if (player == 1) 2 else 1
```

```
(1 -> 2, 2 -> 1) (player1 -> player2, player2 -> player1)
```

On met à jour le nom du joueur :

```
currentPlayerEditText.text = if (player == 1) player1 else player2
```

Et on met à jour la couleur de la pièce du joueur qui joue :

```
        currentPieceColor.setBackgroundResource(  
            if (player == 1) R.drawable.circle_red  
            else R.drawable.circle_yellow  
        )  
    }  
}
```

Si la pièce n'a pas été ajouté (la colonne est pleine) on affiche un message d'erreur :

```
else{  
    val toast = android.widget.Toast.makeText(  
        context, "Can't play here", android.widget.Toast.LENGTH_SHORT  
    )  
    toast.show()  
}  
}
```