# Programmation de composants mobiles
# Examen le 14 décembre 2020 , durée 2h, 14h30-16h30

## Documents autorisés

Documents et matériel autorisés :

- tous les documents papier sauf livres,
- documentation sur le site d'android `https://developer.android.com/index.html`, `https://developer.android.com/reference/packages`
- documentation kotlin `https://kotlinlang.org/docs/reference/`
- tous les documents sur le moodle du cours,
- vos propres programmes de TP et votre projet,
- vos appareils android pour tester les programmes.

Il est strictement **interdit** d'utiliser d'autres sites web que mentionnés ci-dessus, en particulier :

- il est interdit d'utiliser google ou n'importe quel autre moteur de recherche (vous devez aller directement sur la page de la documentation d'android ou kotlin et faire les recherches depuis cette page, sans passer par google),
- d'utiliser les forums de développeurs,
- utiliser l'ordinateur, tablettes, smartphones à d'autres fins que le développement et le test de vos programmes.

**Le sujet comporte 5 pages.**

## Consignes

Pour chaque application vous devez créer le fichier zip depuis AndroidStudio : `File -> Export to ZIP File ...` et déposer ce fichier sur le moodle du cours dans le dépôt dans la section Examen. Si moodle ne marche pas, ce qui est fréquent, vous pouvez envoyer le fichier zip par mail à `zielonka@irif.fr` avec `exam android` comme sujet. Votre mail vous devez donner votre nom, prénom, numéro d'étudiant (je vais ignorer les mails dont l'auteur(e) n'est pas identifiable).
Dans l'examen vous devez développer une seule application mais cela est divisé en plusieurs exercices. Il est prudent de sauvegarder le fichier zip après chaque exercice réussi sinon vous risquez de vous retrouver à la fin de l'examen avec un programme qui ne marche plus.

Avant de commencer les exercices récupérez sur moodle (section Examen) les fichiers layout de l'unique activité de l'application et le fichier aux.kt.
**Vous pouvez, et parfois vous devez, modifier à votre guise le fichier layout fourni. (Vous pouvez aussi fabriquer votre propre layout, même si je n'y vois pas d'intérêt.**

## Démineur

Le but des exercices qui suivent est d'implémenter le jeu bien connu, le démineur.
Le jeu se joue sur un tableau rectangulaire couvert de rectangles. Soit `largeur` la largeur de tableau (le nombre de colonnes) et `hauteur` le nombre de lignes.

Dans notre implémentation chaque rectangle est initialement vert et affiche les coordonnées : le numéro de ligne et de colonne. Les lignes sont numérotées de haut vers le bas et les colonnes sont numérotées de gauche à droite. La numérotation commence à 0. Bien sûr,
Derrière certains rectangles se cachent les mines.
Un clic sur un rectangle permet de découvrir si le rectangle cache une mine ou non.

Si le rectangle cache une mine elle explose et le joueur perd la partie (il pourra commencer une nouvelle partie). Dans votre implémentation le rectangle devient rouge et on remplace les coordonnées affichées dans le rectangle par un String vide.

Si le rectangle ne cache pas de mine,
- il devient jaune,
- à la place de coordonnées on affichera dans le rectangle le nombre de rectangles adjacents qui cachent les mines. Les rectangles adjacents au rectangle dont les coordonnées sont $(i, j)$ sont les rectangles $(i-1, j)$, $(i+1, j)$, $(i, j-1)$, $(i, j+1)$ (à condition que ces coordonnées sont valides).

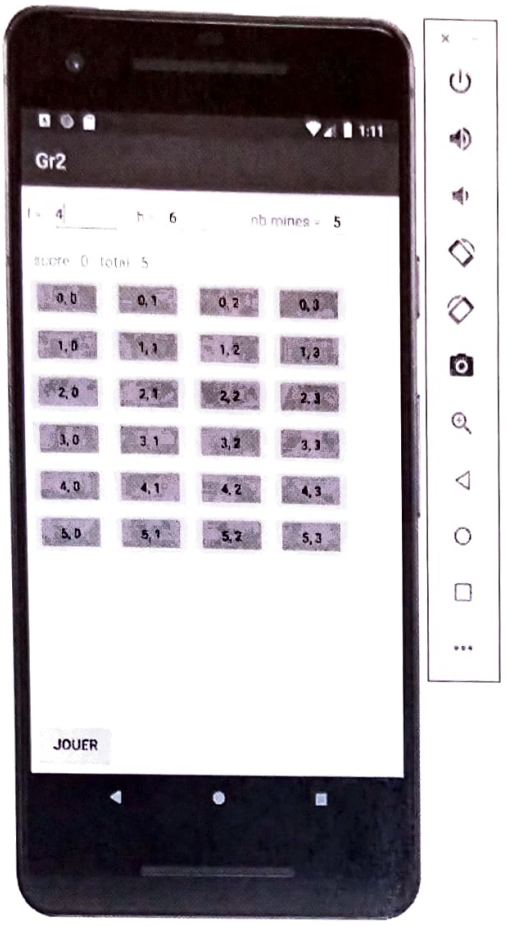
Dans votre implémentation les rectangles seront des `Buttons` qu'on placera dans un `GridLayout`
.

On vous fournit le fichier layout `activity_main.xml`. Les trois `EditTexts` sont utilisés par le joueur pour indiquer respectivement le nombre de colonnes (id `larg`), le nombre de lignes (id `hauteur`) et le nombre de mines (id `nbmines`).
Le `GridLayout` défini dans le fichier layout est initialement vide. Notez que les `Views` qu'on place dans le `GridLayout` sont numérotés par un seul entier, de 0 à `largeur*hauteur - 1`.

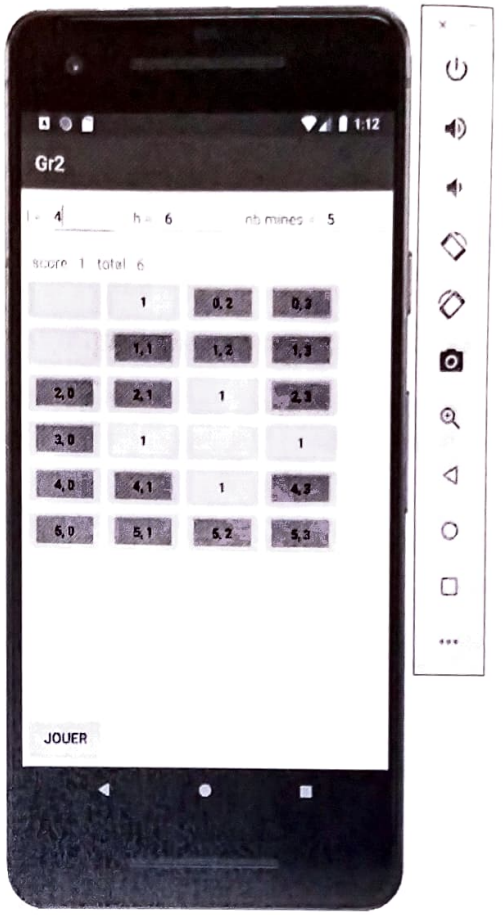
Comme ce n'est pas un examen d'algorithmique, par accélérer le travail on vous fournit le fichier `aux.tk` qui contient quelques fonctions auxiliaires :
- `pos` prend le coordonnées d'un rectangle : le numéro de ligne (codeligne), le numéro de colonne (`col`), et la largeur de grille (`largeur`) et retourne le position de rectangle dans `GridLayout`.
- la fonction `colonne` prend la position `pos` de rectangle dans `GridLayout`, et la largeur de la grille, et retourne le numéro de la colonne du rectangle.
- la fonction `ligne` prend la position `pos` de rectangle dans `GridLayout`, et la largeur de la grille, et retourne le numéro de la ligne du rectangle.
- La fonction `nbMinesDansVoisinage()` prend la position `pos` de rectangle dans le `GridLayout`, la largeur (le nombre de colonnes) et la hauteur (le nombre de lignes) de la grille, et la liste de positions de mines. Pour le rectangle à la position `pos` la fonction retourne le nombre de ses voisins qui sont minés.
- Pour avoir l'interface un peu plus intéressant, on vous propose la fonction `View.changeColor( color: Int )` qui change la couleur de background d'un View. L'utilisation de la fonction est expliquée dans le commentaire dans le fichier. Si vous changez la couleur en utilisant cette fonction, au lieu de changement instantané, la couleur change progressivement pendant 2 secondes (vous pouvez modifier la durée).

Toutes ses quatre fonctions doivent être définies à **l'extérieur** de définitions de classes. Le plus simple c'est les copier dans un fichier séparé, différent de celui qui contient l'activité.

Dans la suite on suppose que les différents widgets de l'interface graphique sont disponibles comme les propriétés de l'activité.




(a) L'écran du jeu juste après la création de boutons. Tous les boutons sont verts.




(b) L'écran du jeu après quelques coups. Les boutons sans nombre affiché ou avec un seul nombre sont jaunes.

**Exercice 1 :**

Écrire une fonction `verification()`, membre de l'activité, qui prend les valeurs de trois `EditTexts` : la largeur de la grille, la hauteur et le nombre de mines et vérifie que les trois valeurs sont positives et que le nombre de mines est inférieur à `largeur*hauteur`. Si c'est le cas alors on stocke les trois valeurs comme des propriétés de l'activité. Si une de valeurs ne satisfait pas ses conditions on informe le joueur par un message dans un dialogue et on efface les `EditTexts` qui contiennent une valeur interdite.
Vérifier que la fonction marche correctement en l'activant quand le joueur clique sur le bouton `jouer`.
Si vous n'arrivez pas à faire l'affichage avec un dialogue utilisez un `Toast` (mais cela donne moins de points).
La fonction peut être sans paramètres si les trois `EditTexts` sont des propriétés de l'activité (ou vous pouvez ajouter les paramètres qui vous conviennent).

**Exercice 2 :**

Écrire la fonction

```
fun creerMines(nbCases: Int, nbMines: Int): ArrayList<Int>
```

qui crée de façon aléatoire la liste de positions pour placer les mines. `nbCases` c'est le nombre total de rectangles (`nbCases == largeur*hauteur`), `nbMines` le nombre de mines à placer. La fonctions retourne une liste de `nbMines` valeurs `Int` entre 0 et `nbCases-1`. Tous les éléments de la liste doivent être différents (donc il faut générer les `Int` tant qu'on n'obtient pas `nbMines` valeurs différentes).
Pour générer vous pouvez utiliser soit le générateur de nombres aléatoires de `java` soit le générateur `kotlin.random.Random` du kotlin.
Ce dernier peut être initialisé par

```
val random = Random( System.currentTimeMillis())
```

et ensuite chaque appel `random.nextInt(0, nbCases)` retourne un `Int` aléatoire dans l'intervalle demandé.

**Exercice 3 :**

Ajouter dans l'activité la fonction membre

```
fun createGrid(l: Int, h: Int, mines: ArrayList<Int>)
```

qui crée `l*h` `Buttons`, les peint en vert, sur chaque bouton met ses coordonnées dans la grille, et les ajoute dans `GridLayout` (disponible comme une propriété de l'activité).
**Indications.** Si `grid` est une référence vers le `GridLayout` alors

```
grid.rowCount = hauteur
grid.columnCount = largeur
```

initialise la taille de la grille, et chaque appel à

```
grid.addView(b, pos)
```

ajoute une `View b` à la position `pos` dans le `GridLayout`.
Vérifier que votre fonction est correcte en implémentant l'affichage de la grille avec les boutons quand l'utilisateur clique sur le bouton `jouer`.

**Exercice 4 :**

Compléter la fonction de l'exercice précédent (soit en complétant le code soit à l'aide d'une nouvelle fonction) pour l'implémenter la réponse à un clic de joueur sur un bouton.
Soit `pos` la position du bouton dans `GridLayout`.

Si `pos` se trouve sur la liste de mines, alors le bouton devient rouge et le texte dans le bouton est effacé. On affiche dans un `Toast` : `saute sur une mine`. Cela termine la partie.

Si `pos` ne se trouve sur la liste de mines, alors le bouton devient jaune et le texte dans le bouton affiche le nombre de voisins minés (ou n'affiche rien si tous les voisins sont libres de mines). Notez que cela peut terminer la partie si le nombre de rectangles non-visités est égal au nombre de mines (on a visités tous les rectangles sans mines).
Pour détecter cette situation on peut maintenir comme une propriété la liste `visites` de positions de toutes les boutons déjà visités.
Si

```
mines.size + visites.size == nbCases
```

où `mines` la listes de positions de mines, `visites` la liste de positions visitées et `nbCases` le nombre total de boutons alors la partie est terminée et on affiche dans un `Toast` ou dans un dialogue le message approprié.

**Exercice 5 :**

Faire en sorte que depuis la création de la grille de jeu jusqu'à la fin de la partie (gagnante ou perdante) le bouton `jouer` soit inactif (on ne peut pas récréer la grille tant que la partie n'est pas terminée).
Pour commencer une nouvelle partie, peut être avec une nouvelle taille de la grille, il faut d'abord vider le `GridLayout`. Ce n'est pas commode de le faire à la fin de la partie, juste après avoir gagné ou perdu, parce que cela empêchera d'analyser la configuration finale du jeu.
C'est plutôt au début de la fonction qui crée remplit le `GridLayout` avec le boutons qu'on enlève d'abord tous les boutons.

Les deux deux exercices suivants sont indépendants l'un de l'autre et vous pouvez le faire dans n'importe quel ordre.

**Exercice 6 :**

On ajoute deux propriétés `score` : le nombre de parties gagnées par le joueur, et `total` : le nombre de tous parties jouées. C'est deux valeurs seront affichés pendant le jeux (`TextView score` et `total` du layout).
Ces deux valeurs doivent être mises à jour à la fin de chaque partie et être stockées de façon permanente dans `SharedPreferences`.

**Exercice 7 :**

Quand on tourne l'appareil la grille de jeu disparaît. Faire en sorte après avoir tourner l'appareil on retrouve le jeu dans le même état, avec la même grille, les mêmes cases jaunes, la case rouge (si on a tourné l'appareil à la fin d'un partie perdante), les mines sur les mêmes cases.

**Indication.** `ArrayList<Int>` peut être stocké dans un `Bundle`.