TP **

*Séance du 06/02/2023*

# Table des matières:

[Table des matières:](#_hyynuijs0af0)

[Configuration du TP](#_7fpith4slzz)

[Rappel Programmation Orientée Objet (et un peu de fonctionnel)](#_3zj5a6t8rh9e)

[Jeu de la vie avec LibKTX](#_iyrga4bmozqx)

[Algorithme cellulaire lambda !](#_wdjgthv4i2bj)

# 

# Configuration du TP

Ce TP est séparé en trois parties, qui sont chacune indépendante[[1]](#footnote-0):

* Dans la première partie, nous reverrons rapidement des bases de la Programmation Orientée Objet, et sa syntaxe en Kotlin. Je recommande de la faire pour se familiariser avec les bases du langage, mais si vous pensez être fort, c’est possible de ne pas le faire du tout.
* Dans la seconde partie, on va coder un [jeu de la vie](https://www.wikiwand.com/en/Conway%27s_Game_of_Life) from scratch. Vous allez voir, c’est rigolo
* Dans la 3e partie, on va continuer le jeu de la vie, et l’étendre pour des cas plus généraux, en utilisant quelques concepts très Kotlin.

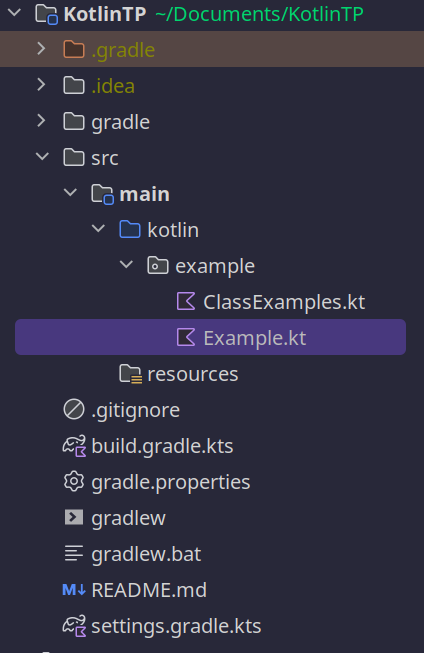
Pour commencer, il vous faudra cloner le repo Git. Pour ça, si vous avez git, vous pouvez simplement faire:

| git clone https://github.com/club-code/TP-Kotlin.git |
| --- |

Sinon, vous pouvez télécharger une archive en .zip en cliquant [ici](https://github.com/club-code/TP-Kotlin/archive/refs/heads/main.zip).

**Il est cependant fortement recommandé d’avoir Git pour ce TP**, même si ce n’est pas nécessaire. Si vous n’avez pas Git, vous pouvez changer de branches en allant sur la page GitHub et en téléchargeant les branches intéressantes en .zip.

Une fois cela téléchargé, vous pouvez ouvrir le dossier avec votre IDE préféré. Votre dossier devrait ressembler à cela, modulo le dossier *.idea* et un dossier *.eclipse* ou *.vscode* en fonction de votre IDE:



Parce que je déteste les TP qui disent la phrase précédente sans expliquer ce que chaque fichier fait:

* Les fichiers *build.gradle.kts*, *settings.gradle.kts*, *gradle.properties*, *gradlew* et *gradlew.bat* sont des fichiers relatifs à Gradle, un outil pour gérer les packages et la compilation sur tout ce qui touche à Java (et à Kotlin). Vous n’aurez à priori pas besoin de les utiliser. Le dossier *.gradle* correspond à des fichiers temporaires utilisés par Gradle. Le dossier *gradle* contient des .jars utilisés par gradle.
* Le fichier *.gitignore*, pour ceux qui connaissent un peu Git, sert à éviter que certains dossiers soient sauvegardés dans Git (notamment les fichiers compilés).
* C’est dans le dossier *src* qu’on va retrouver tous nos fichiers de code source. Le sous-dossier *main* est seul, mais il pourrait y avoir un autre sous-dossier *test* pour les tests. Ensuite, on indique notre langage (on pourrait mélanger du Java et du Kotlin), et on indique qu’on travaille dans le package *example*.

Normalement, à partir de ce moment, vous devriez pouvoir ouvrir les fichiers Example.kt ou ClassExample.kt, et exécuter leur main() avec votre IDE.

**Si vous n’arrivez pas à exécuter les main de ces classes, appelez-moi pour que je vienne aider. De même si vous avez un problème à n’importe quel moment.**

Je vous invite à lire un peu le code présent dans les deux fichiers d’exemples, ils sont là pour essayer d’illustrer un max de code déjà présenté pendant la formation. N’hésitez pas à aller les consulter quand vous avez des questions, ou à me poser des questions dessus ! Ça me permettra de les améliorer pour les générations futures.

# Rappel Programmation Orientée Objet (et un peu de fonctionnel)

Commencez par vous positionner sur la branche *first\_part*, soit avec votre IDE[[2]](#footnote-1), soit avec la commande suivante:

| git checkout first\_part |
| --- |

Le but de cette partie est de vous familiariser avec les classes et la manière de coder Kotlin. Pour ce faire, je vous propose de reprendre le concept d’un TP de Java de ma 1A, faire une armée de Monstre. Le package Answer contient des fichiers Question#n°, qui sont les réponses aux différentes questions. Dans le fichier monster.Main, il y a une fonction main qui vous guidera à travers de TP. Pour des raisons de compilation, les noms des différentes classes sont légèrement différents dans ce package que celui que nous leur aurions donné normalement..

1. Tout d’abord, nous allons définir une classe **Monster**. Pour le moment, on ne va pas faire trop compliqué: on va simplement dire qu’un Monstre, ça a un nom, une taille (Double), et un endroit où il vit (String). Comme il y a du code dans le main() de Monster, je vous invite à respecter la convention suivante: le nom est *name*, la taille est *size*, et l’endroit où il vit est *location*. **Écrivez la classe Monster en une seule ligne.**

Dé-commentez le premier bloc de main et vérifier qu’il tourne sans problème. Vous pouvez remarquer un “problème” lors de l'exécution. Réglez ce problème de la manière que vous voulez, puis vérifiez que cela fonctionne. Re-commentez-le bloc de code.

1. Il apparaît que même si c’est pratique de faire une classe Monster aussi libre, ce n’est peut être pas si bien que ça. Nous allons donc plutôt faire une interface, puis implémenter cette interfaces quelques fois. Commencez par passer la classe Monster en une interface (mot-clef ***interface***) (vous devrez éventuellement supprimer la fonction toString que vous avez définie plus tôt), puis implémentez cette interface en trois monstres: **Vampire**, **Gnome**, et **JavaDev**, avec chacun ayant des valeurs par défauts différentes pour les différents attributs de Monster. **Rappel**: Pour faire de l’héritage en Kotlin, on utilise la syntaxe **class** **MyClass**(args): OtherClass { /\* do stuff \*/}, et dans le corps de la classe, vous devrez définir les valeurs présentes dans l’interface. Dé-commentez le bloc de code dans main, et testez l'exécution.
2. Comme dans la question 1, on aimerait en fait avoir un **toString**() qui fonctionne pour tous les monstres, sans avoir besoin de le redéfinir. Si vous essayez de le faire sur votre interface, ça ne fonctionnera pas: c’est logique, on ne peut rien implémenter sur une interface. On va donc plutôt passer Monster en classe abstraite (mot-clef ***abstract***), c'est-à-dire une classe qu’on ne peut pas instancier. Vous pouvez ensuite changer la manière dont l’héritage est fait, si vous le souhaitez: plutôt que de mettre les valeurs des attributs dans le corps de la classe, vous pouvez les donner dans le constructeur, en utilisant la syntaxe **class** **MyClass**(args): OtherClass(otherArgs)
3. ***OPTIONNEL*** - En fait, on a pas besoin d’être aussi précis sur la taille. Créez une énumération (mot-clef ***enum***) Size, qui contient trois niveaux Tall, Medium, Small. Modifier votre code pour refléter ça. Une énumération, puisque vous ne l’avez pas vu en Java apparemment, c’est un type qui contient plusieurs instances prédéfinies (par exemple, dans notre cas, il n’y aurait que 3 tailles: Tall, Medium, Small), et qui sont initialement comparables entres elles, dans l’ordre où vous les avez définies.. Si vous n’avez pas l’habitude d’en créer, n’hésitez pas à regarder dans la réponse.
4. Décommentez le dernier bloc de code. Remarquez que, même si on a fait techniquement vérifié toutes les possibilités, on est obligé de rajouter un else. Trouvez un moyen d’enlever cela (i.e donnez au compilateur le moyen de connaître tous les Monster à la compilation).
5. On veut pouvoir facilement comparer deux Monstres: nous allons donc faire en sorte que la classe Monster implémente **Comparable<Monster>**. Cela veut dire qu’on peut comparer n'importe quel Monster avec n’importe quel Monster. Si votre IDE ne vous le dit pas, cela veut dire que n’importe quelle classe héritant de Monster doit posséder une méthode ***compareTo(other: Monster)***. En pratique, cette méthode sera la même pour tous les monstres (et au pire on peut l’override), donc définissez là dans la classe Monster. On décide arbitrairement qu’un monstre est supérieur à un autre si et seulement si il est plus grand que l’autre. En cas d’égalité, on compare les noms des deux monstres (la méthode compareTo est déjà définie pour les String). Filtrez liste ainsi triée pour ne garder que les monstres qui sont cools.

**Félicitations ! Vous avez révisé les bases de la POO en Kotlin.**

Avant de passer à la partie suivante, vous devez sauvegarder votre progrès. Je vous invite donc à faire les commandes suivantes:

| git add . git commit -m "Première partie." |
| --- |

# 

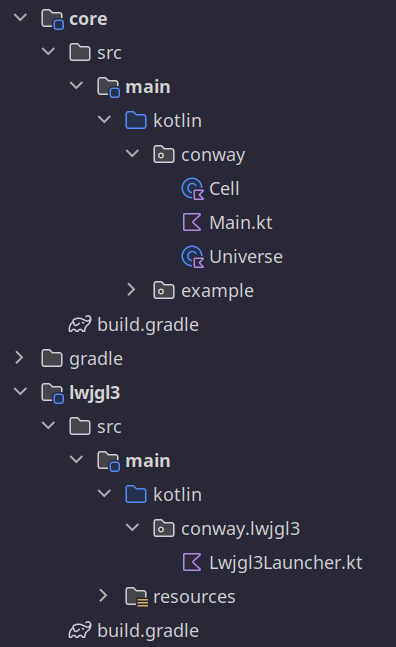
# Jeu de la vie avec LibKTX

Dans cette deuxième partie, nous allons faire un jeu de la vie, en utilisant la librairie graphique LibGDX, et son extension pour Kotlin LibKTX. Est ce que c’est overkill ? Totalement. Est ce que je me suis souvenu de l’existence de Processing uniquement après avoir fini d’écrire ce TP ? Oui. Mais en vrai c’est pas si lourd, et ça peut être intéressant de l’utiliser. Surtout, j’avais la flemme de tout réécrire, oupsi.

Commencez par vous positionner sur la bonne branche:

| git checkout second\_part |
| --- |

Vous remarquerez que votre IDE panique un peu: c’est normal, la structure du projet a changé. Le plus important à savoir, c’est: mais où qu’il est passé mon Main ? En fait, **il faut exécuter la méthode Main du Lwjgl3Launcher présent dans le dossier lwjgl3**. Du reste, il faut juste coder dans le dossier core/src/main/kotlin, comme avant (sauf qu’il y a le dossier *core* en plus).



Normalement, vous n’avez pas besoin de toucher à quoi que ce soit en dehors du package *conway*. Les deux fichiers principaux ont déjà été créés, il faut encore les compléter. Cette partie du TP est un peu plus libre, il n’y a donc pas de réponses écrites. Il y a cependant une branche fonctionnelle, *second\_part\_corrected*, pour ceux que ça intéresse.

1. Commencez par regarder rapidement le code du launcher, et surtout celui de Main.kt. La méthode *render* est appelée à chaque frame, est ce que vous comprenez bien ce qu’elle fait ? Dé-commentez la ligne qui dessine l’univers, et lancez le code. D’où vient l’erreur ? Félicitations, vous venez de découvrir la fonction **TODO** ! Elle renvoie une erreur à l'exécution et un Warning à la compilation.
2. Commencez par compléter la classe **Univers**: elle doit avoir une **List** immutable de Cell. Cette List est une **variable**, qui sera changée à chaque appel à *update*. Initialisez cette List, en le peuplant de Cell aux coordonnées aléatoires comprises entre les bornes de l’Univers (Vous pouvez utiliser la méthode .random des IntRange pour obtenir des indices aléatoires). Complétez la méthode *draw* de la classe Univers en utilisant cette List et la méthode **.*forEach***. Exécutez.

D’où vient le problème maintenant ?

1. Complétez la classe **Cell**: notamment, rajoutez un booléen indiquant si la cellule est vivante ou morte. Une cellule est créée lorsqu’elle prend vie, donc ce booléen vaut ***true* par défaut** (cela évite de devoir modifier votre code précédent). Complétez la méthode *draw*, en utilisant la méthode *rect(x, y, width, height)* de shapeRenderer, en pensant à utiliser Cell.cellSize, et en pensant à changer la couleur du shapeRenderer en fonction de l’état de la cellule[[3]](#footnote-2). Rajoutez une méthode *.equals* qui vérifie l’égalité des coordonnées à Cell.
2. Maintenant, il faut faire la méthode *update* de l’Univers. Nous allons changer le paradigme de la Liste, car nous avons désormais besoin d’accéder aux cellules à partir de leurs coordonnées, rapidement. Remplacez la liste par une **Map<Pair<Int, Int>, <Cell>>**.

**NB:** En Kotlin, une Pair<A, B> est un tuple, dont le premier élément est de type A, et le 2e de type B. On peut créer une Pair avec le constructeur (**val pair = Pair(1, 2)**), ou bien avec le mot-clef ***to*** (**val pair = 1 to 2**)

Vous pouvez utiliser la méthode *.toMap*, qui transforme une List<Pair<K,V>> en Map<K,V>. Créez une méthode *.getNeighbours* à Cell qui renvoie la liste des coordonnées de ses voisins (une List<Pair<Int, Int>>).

La méthode update fonctionne de la manière suivante:

* On crée une MutableMap<Point, Int>, qui nous permettra de compter le nombre de voisins de chaque cellule.
* Pour chaque cellule vivante dans notre cellMap (utilisez *.filter*), comptez que ses voisins ont un voisin de plus. Autrement dit, parcourez la liste de ses voisins, et incrémentez de 1 leur valeur dans la mutableMap précédemment créée (ou bien mettez sa valeur à 1 si elle n’était pas dans la map).
* Créez une nouvelle Map, selon les règles suivantes:
  + Si une Cellule a exactement trois voisines vivantes, alors elle est vivante à l’étape suivante;
  + Si une Cellule a exactement deux voisines vivantes, elle garde son état actuel;
  + Sinon, la Cellule meurt.
* Remplacer la map de l’Univers par la map nouvellement créée, en supprimant les cellules qui sont en dehors de ses limites.

Décommentez la ligne correspondante dans Main.kt, et lancez le programme. Normalement, lorsque vous appuyez sur Espace, le jeu de la vie se met à jour !

**Félicitation, vous venez de créer la vie !**

Comme avant, sauvegardez votre avancée dans git avec

| git add . git commit -m "Deuxième partie." |
| --- |

# Algorithme cellulaire lambda !

Dans cette partie, nous allons reprendre le code précédent, et le modifier pour faire un algorithme cellulaire quelconque, avec les règles que l’on veut ! Commencez par vous positionner sur la branche correspondante au début de cette partie:

| git checkout third\_part |
| --- |

Il y a très peu de différence avec votre code à la fin de la partie 2. La principale différence est que vous pouvez désormais cliquer pour changer l’état des cellules. L’idée est de changer la manière dont nous gérons les Cellules, afin de faire un algorithme cellulaire à trois cellules: une bleue (Water), une marron (Dirt), et une Marron Clair (Mud). Il y aura techniquement un quatrième type de cellules, correspondant à “rien”.

1. Pour commencer, nous allons généraliser le code de Cell. Passez Cell en *sealed*, et rajoutez lui un attribut abstrait color: Color. Changez la méthode *draw* en conséquence. Écrivez les classes Dirt, Water et Mud. Modifiez la fonction *switch* pour qu’elle fasse cycler entre ces trois classes, et la “cellule vide” (qui n’est pas une vraie cellule, mais une l’absence de cellule aux coordonnées données). Utilisez un *when*.
2. Ensuite, implémentez, dans la méthode *update* de l’Universe, les règles suivantes, en utilisant des *when*:
   * Une case vide reste vide sauf si la case située au dessus d’elle est Water ou Mud, auquel cas elle devient respectivement Water et Dirt
   * Une case Dirt reste un case Dirt sauf s’il y a exactement un bloc d’eau autour d’elle, auquel cas elle devient Mud.
   * Une case Water reste une case Water sauf s’il y a au moins une case Dirt autour d’elle, ou si la case en dessous d’elle est vide, auquel cas elle devient null
   * Une case Mud devient une case Dirt, sauf s’il y a au moins une case Water autour d’elle.

N’hésitez pas à utiliser les méthodes *filter*, *count* et *mapNotNull* des Maps/Sets/Lists, c’est très pratique !

**Félicitations, vous avez utilisé plusieurs Kotlineries dans votre code !**

1. J’ai toujours rêvé d’écrire ça, ça fait tellement professionnel. [↑](#footnote-ref-0)
2. Allez dans l’onglet Git, cliquez-droit sur la branche qui vous intéresse, et faites “Checkout” [↑](#footnote-ref-1)
3. Cette méthode ne prend que des Float en argument, donc vous devrez utiliser la méthode *toFloat* pour les convertir. [↑](#footnote-ref-2)