Module 3 – introduction à l'OO

A. Réflexion

1. Citez 2 grandes différences entre la programmation orientée objet et la programmation procédurale.

2. Que signifie « couplage fort » ?
   1. le fait qu'un objet regroupe à la fois des données et du code
   2. le fait que deux bouts de code dépendent fortement l'un de l'autre
   3. le fait que deux objets puissent s'envoyer des messages mutuellement
   4. le fait que deux fonctions s'appellent l'une l'autre

3. Parmi les propositions suivantes, laquelle n'est pas une conséquence de l'encapsulation ?
   1. une sécurité accrue
   2. un couplage plus faible
   3. un code plus facile à modifier
   4. un code plus efficace

4. Qu'est ce qui peut distinguer des objets différents ?

5. Que sont les sélecteurs et les modificateurs ?

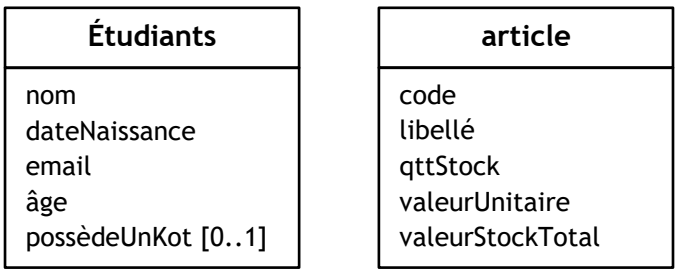
6. Qu'est ce qu'une association en UML ?

7. Dans une description de classe en python, que représente le « self » ?

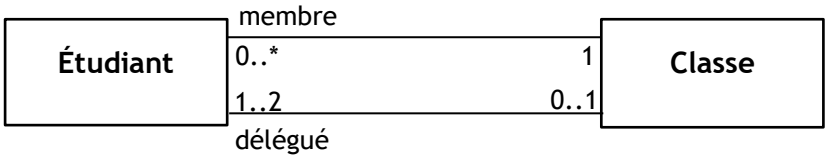
B. Un peu de modélisation

Par **convention** entre nous, lors des exercices et sauf avis contraire, quand vous modélisez une classe, il vous est demandé de spécifier :
- Le nom de la classe
- Les attributs de la classe
- Le type des attributs booléens
- Si un attribut est facultatif ou composite
- Si un attribut demandé est dérivable

Quelles critiques/remarques pouvez-vous faire sur ces classes ?



Comment doit-on lire les associations suivantes :



Modélisez sous la forme de diagrammes UML chacun des points.

- Un électeur vote pour au moins un candidat
- Un électeur vote pour au plus un candidat
- Une droite est déterminée par deux points. La droite porte un nom et chaque point est déterminé par ses coordonnées en X et en Y.
- Un cours, caractérisé par un titre et un nombre d'heures, aborde un ou plusieurs sujets. Chaque sujet est décrit par un nom et un niveau d'importance (entre 1 et 5). À chaque sujet est associé des questions (d'interrogation) décrites par un énoncé et un niveau de difficulté. Une question peut couvrir entre 1 et 3 sujets.
  Une interrogation, caractérisée par une date, comporte entre 5 et 10 questions principales auxquelles viennent s'ajouter au maximum 3 questions facultatives.

- Un plat est décrit par son titre, son concepteur et le fait qu'il est végétarien ou pas. Tout plat est associé à une catégorie caractérisée par un libellé (entrée froide, entrée chaude, dessert…).

Université

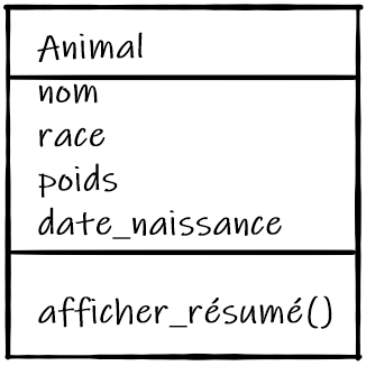
Modélisez le système suivant :

- Une classe **Section** avec un nom.
- une classe **Bloc**, chaque bloc (d'étude à l'Henallux) étant décrit par un numéro (1, 2 ou 3) et une section ;
- une classe **Étudiant**, chaque étudiant étant décrit par ses nom et prénom, sa date de naissance, son bloc d'étude et le fait qu'il est finançable ou pas ;
- une classe **Professeur**, chaque professeur étant décrit par son nom, son prénom, sa spécialité et éventuellement la section à laquelle il est attaché.
- une classe **Cours**, chaque cours étant décrit par un nom, un nombre d'heures, le bloc d'étude où il est donné et le(s) professeur(s) qui s'en charge(nt) ;
- une classe **Voiture**, chaque voiture (d'un professeur) étant décrite par sa plaque et sa marque (pour le contrôle du parking).

C. Traduire des classes en python

Voici des diagrammes de classes à traduire en Python.

1. Animal



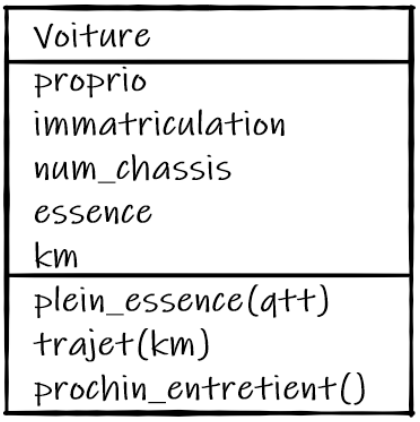
La classe Animal a 4 attributs d'instances. Ils sont tous passés en paramètre au constructeur pour être initialisés.

La méthode `afficher_résumé` permet d'afficher un petit résumé de l'animal. Par exemple sous la forme :
<nom> (<race>) né le <date> pèse <poids>kg.

1. Traduisez cette classe en Python en tenant compte de ces remarques.
2. Lancez votre code.
3. En console, créez 2 animaux de votre choix et affichez leur résumé.

2. Voiture

Traduisez cette classe en Python en suivant les étapes suivantes.



1. Commencez par écrire le constructeur. Il prendra en paramètre le nom du propriétaire, le numéro d'immatriculation et le numéro de châssis de la voiture. La quantité d'essence et le nombre de kilomètre ne sont pas passé en paramètre. En effet, lorsqu'une nouvelle voiture est créée, ces deux informations doivent être automatiquement mises à 0.

2. Ecrivez la méthode `plein_essence` qui prend en paramètre une quantité et l'ajoute à l'essence déjà présente.

3. Ecrivez la méthode `trajet` qui prend en paramètre la distance du trajet (en km). Cette méthode effectue le trajet (ajoute le nombre de km et retire la quantité d'essence) uniquement s'il y a assez d'essence dans le réservoir (sinon, affichez un message d'erreur). Supposez que la voiture consomme du 6 litres aux 100 km.

4. Ecrivez la méthode `prochain_entretient` qui retourne le nombre de kilomètre restants avant le prochain entretient de la voiture en sachant que l'entretient doit être fait tous les 15000 km.

5. Réécrivez la fonction prédéfinie `__str__` pour que lorsque vous faites un `print` d'un objet de la classe Voiture, l'affichage ressemble à ceci :
« Voiture <immat> de <proprio> [<km>] – essence : <essence> - prochain entretient dans <prochain_entretetient> ».

6. Exécutez votre code puis faites quelques tests en console.
   - Créez un objet Voiture et faites un `print` de celui-ci.
   - Faites-lui faire un trajet de 10 km
   - Faites un plein d'essence de 50 litres
   - Refaites lui faire un trajet de 10 km
   - Réaffichez la voiture

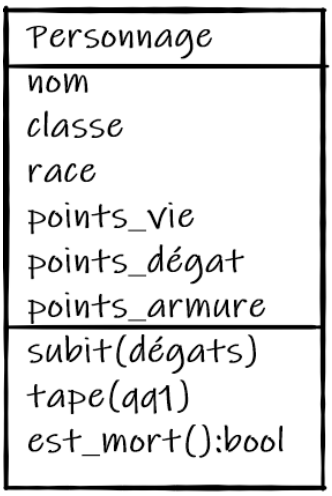
Si vous créez la voiture immatriculée « 1-BOB-600 » appartenant à « Moi » dont le numéro de châssis est « JECPAS12345678XYZ », voici ce qui devrait s'afficher dans l'ordre.

```
Voiture 1-BOB-600 de Moi [0km] - essence : 0 - prochain entretient dans 15000km
Pas assez d'essence
Voiture 1-BOB-600 de Moi [10km] - essence : 49.4 - prochain entretient dans 14990km
```

7. Adaptez votre code pour que la consommation aux 100 km ne soit pas hardcodée mais soit contenue dans un **attribut de classe**.

3. Personnage

1. Voici la classe Personnage. Implémentez là en suivant les remarques ci-dessous.



- Lors de la création d'un nouveau personnage, s'ils ne sont pas précisés en paramètre, les points de vie sont à 20, les points de dégât à 2 et les points d'armure à 1.
Il s'agit donc de paramètres facultatifs dans le constructeur.

- Lorsqu'un personnage subit des dégâts, il ne prend pas la totalité des dégâts passés en paramètre. En effet, son armure encaisse une partie. Les dégâts réellement subits (et donc retirés à ses points de vie) correspondent aux dégâts infligés moins les points d'armure.
Attention, en aucun cas le personnage ne peut récupérer des points de vie au cas où les dégâts sont inférieurs aux points d'armure !

- Quand un personnage tape un autre, il lui fait subir des dégâts équivalents à ses points de dégât.

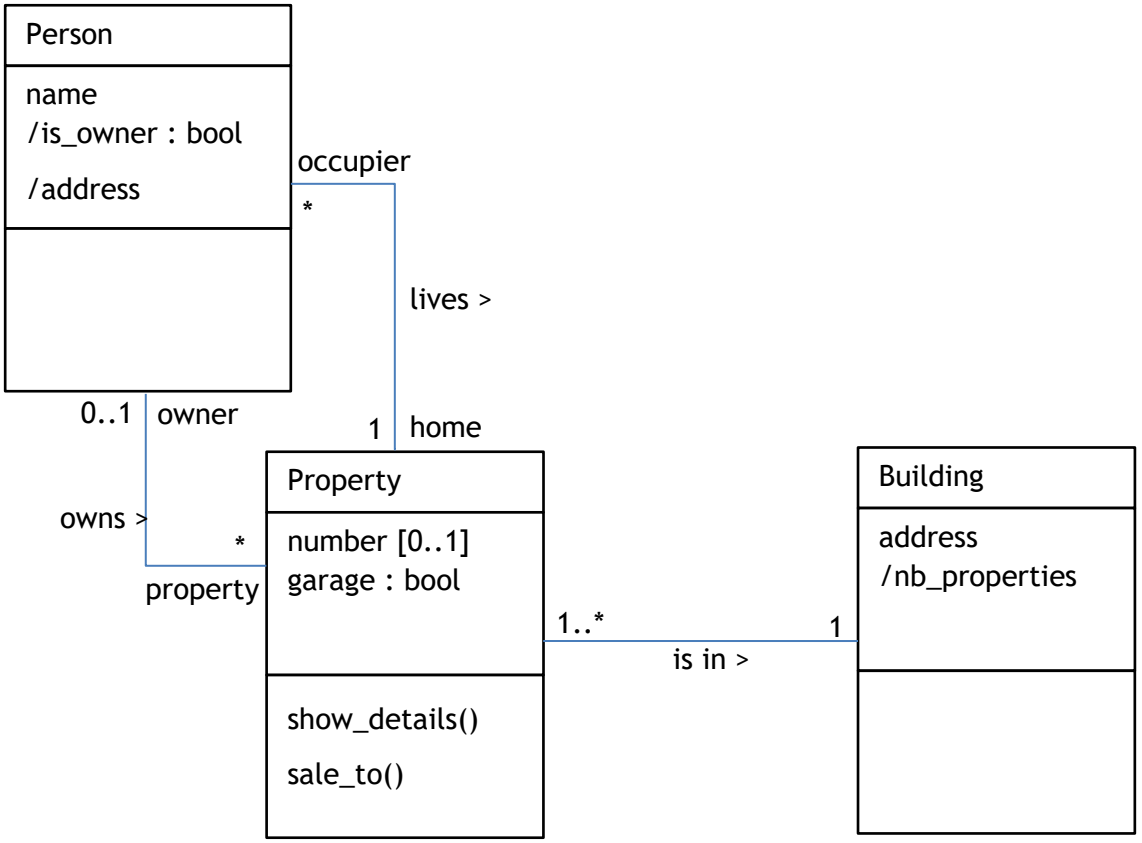
- Un Personnage est considéré comme mort s'il n'a plus de point de vie (0 ou moins).

2. Redéfinissez la fonction `__str__` pour que l'affichage d'un personnage soit plus lisible pour vous.

3. Pour tester votre classe, vous allez faire un petit script qui fait s'affronter deux personnages. Pour cela, dans un bloc main (en dehors de votre classe et après la définition de celle-ci), créez deux personnages de votre choix et affichez-les. Ensuite, tant qu'aucun des deux n'est mort, faites-les se taper l'un l'autre et réaffichez-les.

4. Lancez votre script et corrigez si besoin.

D. Système de gestion de location de propriétés



Voici le diagramme UML correspondant à l'énoncé suivant :

Une personne a un nom, est propriétaire ou non et a une adresse (celle de l'endroit où il habite). Un immeuble a une adresse et contient un certain nombre de propriétés (au moins 1). Une propriété se trouve dans un immeuble et a éventuellement un numéro (s'il s'agit d'un appartement). On retient également si une propriété possède un garage ou non.

Une personne peut posséder plusieurs propriétés mais il arrive que certaines propriétés n'appartiennent pas à une personne (mais plutôt à une organisation ou à la commune). Chaque personne habite dans une propriété et une propriété peut abriter plusieurs occupants.

On veut également pouvoir afficher les détails d'une propriété et la vendre.

Transformation du schéma

1. Transformez le diagramme UML correspondant à l'énoncé pour retirer les relations et pouvoir le traduire en classes python.
2. Vous noterez que le lien entre Building et Property et de type min1-min1. Quel problème va en découlez ?

Pour régler ce problème, changez la multiplicité du côté de Property en mettant un 0..*.

Traduire en classes Python

Traduisez le schéma intermédiaire en classes python en suivant ces étapes. Pensez à tester régulièrement votre code.

1. Créez 3 classes et mettez-y le constructeur correspondant.
   Pour `Property`, si la présence du garage n'est pas passé en paramètre, on suppose qu'il n'y en a pas.

2. Implémentez les propriétés correspondant aux attributs dérivables.

3. Implémentez une méthode `add_property` dans `Building` qui prendra en paramètre une propriété et l'ajoutera à la liste des propriétés de l'immeuble.

4. Réécrivez la méthode `__str__` de `Building` pour que quand on souhaite afficher un immeuble, il s'affiche sous la forme : **Building** *address* **(***nb properties***)**. Pensez à réutiliser ce que vous avez déjà écrit.

5. Réécrivez la méthode `__str__` de `Person` pour que quand on souhaite afficher une personne, elle s'affiche sous la forme : *name* **(***home address***)**.

6. Réécrivez la méthode `__str__` de `Property`. S'il s'agit d'un appartement (possède un numéro), il faudra qu'il s'affiche : **Apartment** *number*, *building address***.** S'il s'agit d'une maison (ne possède pas de numéro), il faudra qu'il s'affiche : **House** *building address***.**

7. Testez ce que vous avez fait. Pour cela, dans le bloc main :
   a. Créez un building `b1` se trouvant à l'adresse « Commonstreet nb 5 » et un building `b2` se trouvant à l'adresse « Magicstreet nb 18 ».
   b. Créez un appartement `ap1` qui se trouve dans l'immeuble `b1` au numéro 1 et qui dispose d'un garage. Ajoutez cet appartement à `b1`.
   c. Créez 3 autres appartements `ap2`, `ap3` et `ap4` se trouvant également dans l'immeuble `b1`, respectivement aux numéros 2, 3 et 4. N'oubliez pas de les rajouter dans `b1`.
   d. Créez une maison `h` qui se trouve dans `b2` et dispose d'un garage. Pensez à ajouter la maison ainsi créée dans `b2`.
   e. Créez 3 personnes : Bob, Alice et Bobby Jr qui vivent ensemble dans la maison sur Magicstreet. Bob est propriétaire de la maison et des appartements `ap1` et `ap2`.
   f. Créez Chris qui habite dans l'appartement numéro 1.
   g. Créez David qui habite et possède l'appartement numéro 3.
   h. Créez Elen qui habite et possède l'appartement numéro 4.
   i. Faites les changements nécessaires dans les propriétés.
   j. Grâce au `print()`, affichez les infos des 2 immeubles, de la maison et de l'appartement numéro 2. Affichez également Bob, Alice et Elen.

Si vous avez correctement programmé jusqu'ici, voici ce qui devrait s'afficher :

```
Building Commonstreet nb 5 (4)
Building Magicstreet nb 18 (1)

House Magicstreet nb 18
Apartment 2, Commonstreet nb 5

Bob (Magicstreet nb 18)
Alice (Magicstreet nb 18)
Elen (Commonstreet nb 5/4)
```

8. Implémentez une méthode `show_details` pour une propriété qui affiche tous les détails de cette dernière comme ceci :

```
>>> h.show_details()
--- House Magicstreet nb 18 ---
Owner : Bob (Magicstreet nb 18)
Occupiers :
- Bob
- Alice
- Bobby Jr
Garage : YES

>>> ap2.show_details()
--- Apartment 2, Commonstreet nb 5 ---
Owner : Bob (Magicstreet nb 18)
Occupiers :
Garage : NO
Nb Apart in Building : 4

>>> ap3.show_details()
--- Apartment 3, Commonstreet nb 5 ---
Owner : David (Commonstreet nb 5/3)
Occupiers :
- David
Garage : NO
Nb Apart in Building : 4
```

9. Dans `Person`, définissez une méthode qui affiche le détail de toutes les propriétés que la personne possède.

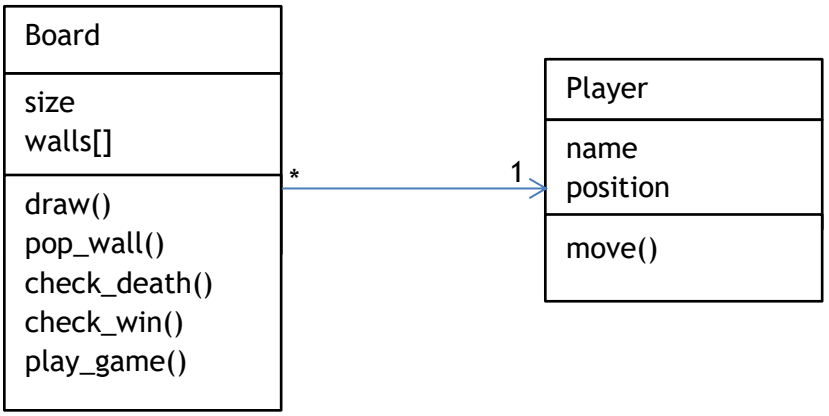
10. A la fin du bloc main, appelez la méthode que vous venez de définir pour voir le détail des propriétés de Bob.

11. Écrivez la méthode `sale_to` dans la classe `Property`. Elle prend en argument la personne à qui la propriété est vendue et s'occupe de gérer la vente. C'est-à-dire changer le propriétaire de la propriété, enlever la propriété de celles de l'ancien propriétaire et l'ajouter à celles du nouveau propriétaire.

12. À la fin du bloc `main`, vendez l'appartement 2 à Elen. Affichez ensuite le détail des propriétés de Bob et d'Elen pour vérifier que la vente s'est bien passée.

E. Exercice supplémentaire : Jeu du labyrinthe magique

Le jeu du labyrinthe magique est un petit jeu pour 1 joueur qui se joue sur un plateau carré. La taille de ce plateau peut être variable mais elle est en général de 6x6. Le joueur commence au coin supérieur gauche et doit arriver dans le coin inférieur droit. Pourquoi ce labyrinthe est-il magique ? Parce-qu'à chaque mouvement que le joueur fait, un mur apparait sur une des cases du plateau. Si le joueur sort des limites du plateau ou se retrouve dans un mur, il perd.



1. Squelette

Pour commencer, écrivez le squelette des deux classes avec leur constructeur en tenant compte des remarques suivantes.

Une case sera représentée par un tuple (ligne,colonne) sachant que le (0,0) se trouve en haut à gauche.

	0	1	2	3	4	5
0						
1						
2						
3						
4						
5						

Le constructeur de `Player` prend uniquement le nom en paramètre. La position du joueur sera automatiquement (0,0).

Le constructeur de `Board` prend un joueur et une taille (qui est de 6 par défaut) en paramètre. `walls`, qui retient la liste des murs présents sur le plateau (et sera donc une liste de tuple), est initialement vide.

2. Dessiner le plateau

Implémentez la méthode `draw` qui affiche le plateau de jeu. Par exemple :

```
. . . X . .
. . . . . .
. . . X . X
. . . X X .
X X . X X .
. . . . O .
```

Un point représente une case vide, une croix représente un mur et le cercle représente le joueur.

Aide : si vous avez du mal à vous lancer, commencez simplement par afficher un plateau avec uniquement des points mais qui dépend de la taille de *`Board`*. Une fois ceci fait, au moment d'afficher le point, regardez s'il n'y a pas un mur à cette place ou si ce n'est pas la case du joueur.

3. Faire bouger le joueur

Etape 1 : Déterminer les touches du joueur

Commencez par réfléchir aux touches (caractères) que le joueur utilisera pour se déplacer. Par exemple, zqsd, 5123, etc.

Définissez alors un dictionnaire **`keyboard_key`** qui associe une touche au tuple correspondant au mouvement.

Par exemple :

- monter signifie faire -1 à la ligne ; le tuple correspondant est (-1,0)
- aller à droite correspond à faire +1 à la colonne, donc le tuple sera (0,1)

Donc, si j'utilise le 'z' pour monter, il y aura l'élément « 'z' :(-1,0) » dans le dictionnaire.

`keyboard_key` sera un **attribut de la classe** Player car tous les joueurs auront les mêmes touches.

Etape 2 : Ecrire la méthode **`move`**

La méthode `move` ne prend pas d'arguments. Dans un premier temps, elle demande au joueur d'entrer une touche et tant que la touche n'est pas un mouvement possible, elle redemande. Ensuite, la position du joueur est modifiée.

Pour cela, le dictionnaire *`keyboard_key`* va vous être utile.

4. Petit test

Dans le bloc `main`, créez un joueur et ensuite un plateau que vous affichez. Appelez la méthode move sur le joueur puis réaffichez le plateau.

Est-ce que tout fonctionne bien ?

5. Faire apparaitre des murs au hasard

Ecrivez la méthode `pop_wall` de la classe `Board` qui fait apparaitre un mur au hasard. Autrement dit, elle ajoute un tuple généré aléatoirement (dans les limites du terrain) à la liste des murs (`walls`).

Attention : si le mur existe déjà, aucun mur n'est ajouté.

Testez la méthode. Par exemple, après avoir créé le plateau, faites apparaitre 3 murs puis affichez le plateau.

6. Vérifier la victoire ou la défaite

Implémentez les méthodes `check_win` et `check_death`, chacune renvoie un booléen qui indique respectivement s'il y a une victoire ou une mort.

Rappel : il y a une victoire si le joueur se trouve dans le coin inférieur droit. Il y a une défaite si le joueur est hors limite ou si un mur se trouve sur la même case que lui.

7. Jouer (enfin presque)

Il ne vous manque plus que la méthode `play_game` qui permet de jouer une partie complète :

- Une partie commence avec l'affichage du plateau initial.
- Ensuite, tant que le joueur n'a pas gagné et n'est pas mort, celui-ci se déplace, puis, un mur apparait. Le plateau est alors réaffiché avant de passer au tour suivant.
- Quand la partie est finie, indiquez si le joueur a gagné ou perdu…

8. Jouez (vraiment ce coup-ci)

Pour pouvoir jouer, il vous suffit dès lors de créer un joueur et un plateau dans votre bloc `main` (en précisant les paramètres souhaités) et d'appeler la méthode `play_game` sur votre plateau.

Vérifiez que votre jeu fonctionne. Sortez du plateau, rentrez dans un mur, gagnez la part… Essayez avec des plateaux de différentes tailles…

Quelques questions

Voici quelques questions de réflexion. Vous pouvez simplement y répondre ou essayer d'implémentez ce qu'elles proposent.

1. Imaginons que l'on compte les points d'un joueur (par exemple, le joueur commence à 30 et perd un point par mouvement et s'il meurt, retombe à 0). Que faudrait-il changer ?

2. Imaginons, ensuite, que l'on veuille garder en mémoire le meilleur joueur (celui qui a eu le meilleur score). Comment faire ?

3. Si vous deviez implémenter une méthode pour que l'utilisateur choisisse ses touches, quel type de méthode utiliseriez-vous ?

Pour aller plus loin…

Maintenant que votre jeu est fait et fonctionne, vous pouvez l'améliorer de plein de manières différentes. Par exemple, empêcher qu'un mur n'apparaisse sur le joueur ou sur un autre mur…

Vous pouvez également faire des variantes du jeu : pouvoir aller en diagonale, mettre 2 joueurs, faire apparaitre des bonus…