

Série G : OOP - Programmation orientée objet

Table des matières

Exercice G.1 :	Dessin de lignes et de rectangles	2
Exercice G.2 :	Animation : Différents objets en mouvement	2
Exercice G.3 :	Javapede.....	3
Exercice G.4 :	Angry balls.....	9
Exercice G.5 :	Space Invaders	15
Exercice G.6 :	Space Invaders II	19

Exercice G.1 : Dessin de lignes et de rectangles

Reprenez l'exercice F.5 – Traceur de figures en couleur (ou F.4 - Traceur de lignes en couleur) et transformez-le en un programme qui profite au mieux de l'héritage et du polymorphisme pour dessiner, sauvegarder et lire des lignes et des rectangles.

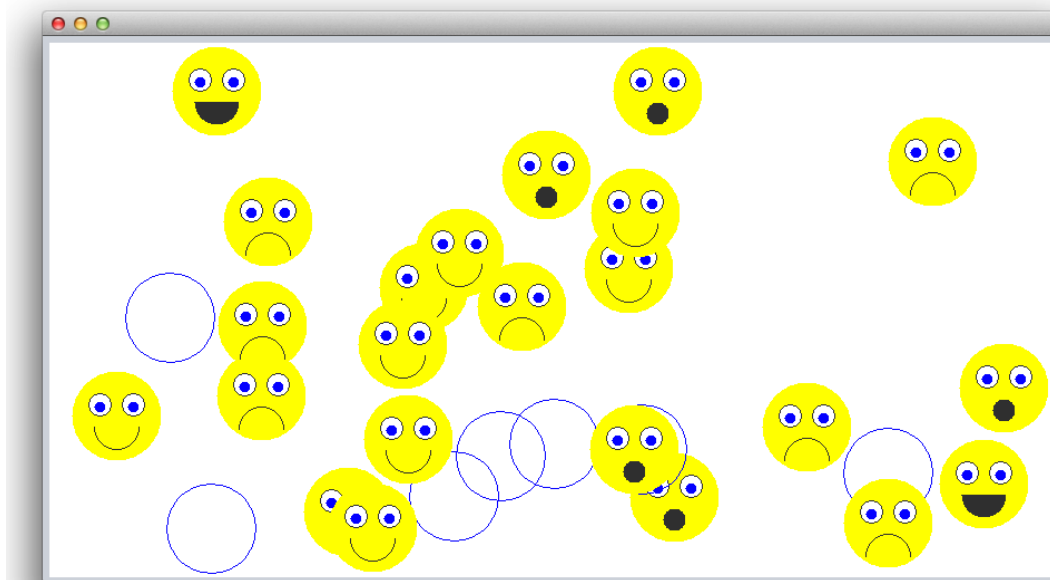
Exercice G.2 : Animation : Différents objets en mouvement

Reprenez l'exercice E.8 (*Animation : Boules en mouvement*), puis modifiez-le en suivant les étapes que voici :

Dérivez de **MovingBall** les classes **EmoSmile**, **EmoSad**, **EmoBigSmile**, **EmoSurprised** qui représentent des *émoticônes* (Smileys) avec différentes expressions. Tous les émoticônes ont la même couleur de fond (jaune) et les mêmes yeux. (Soyez créatifs ! ;-)

Remarque : Consultez l'aide sur les méthodes **drawArc** et **fillArc**.

Testez votre programme en produisant aléatoirement des balles et différents types d'émoticônes. Profitez au mieux des connaissances que vous avez de l'OOP.



Extensions possibles :

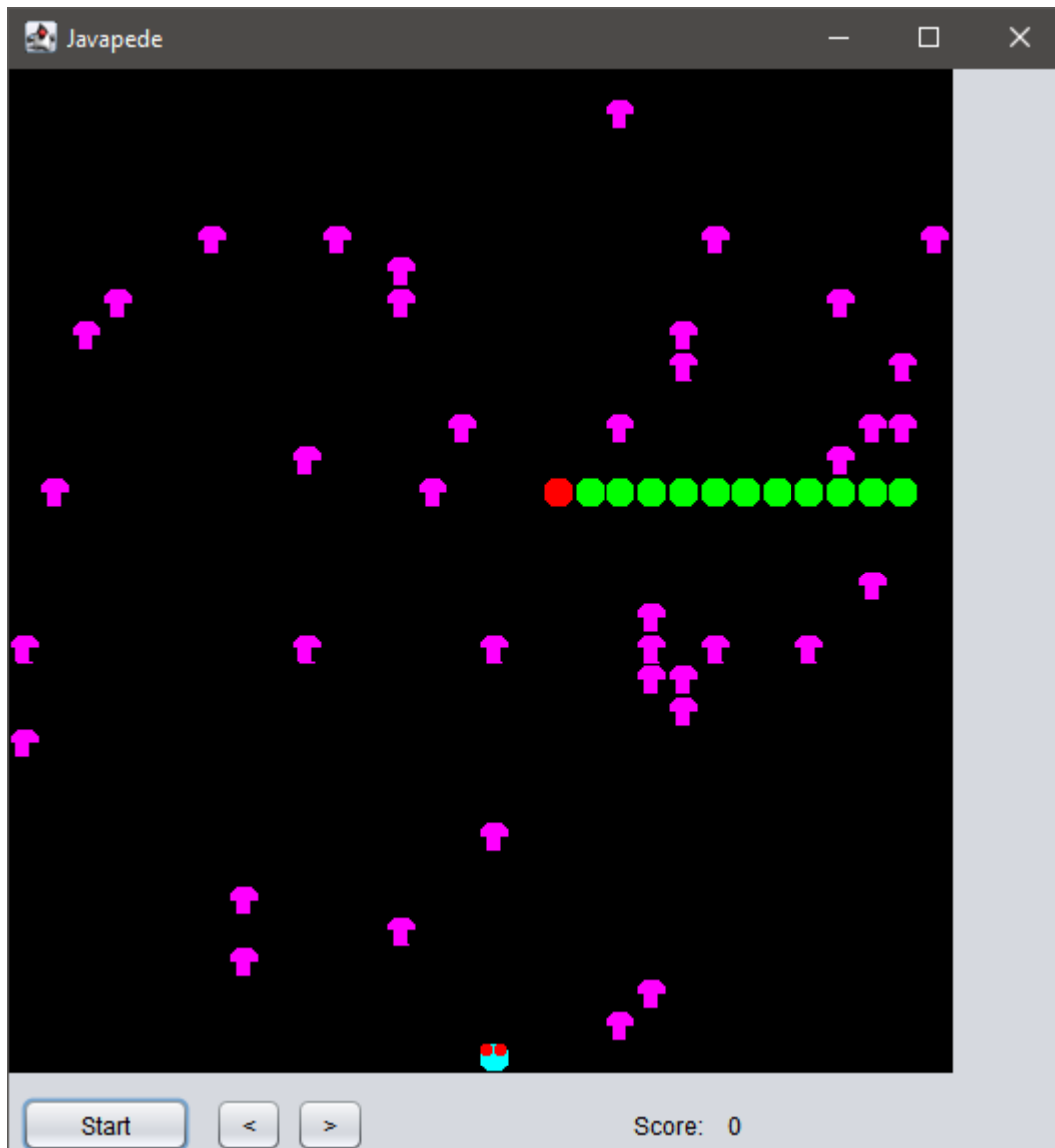
- Ajoutez **EmoNaughty** qui tire sa langue de temps en temps
- Modifiez **EmoSad** pour qu'ils disparaissent après un temps fixé lors de la création.

Notions requises : héritage, chronomètre, surcharge d'une méthode, encapsulation

Exercice G.3 : Javapede

Dans la suite, vous allez développer une variante très simplifiée dénommée « **Javapede** » du jeu *Centipede d'Atari*. Le joueur se trouve en bas de l'écran. Il doit éliminer un mille-pattes (en : *centipede*), qui se déplace lentement vers le bas, en lui tirant sur les différentes parties de son corps jusqu'à ce qu'il soit détruit. Les nombreux champignons disséminés sur l'aire de jeu compliquent la tâche du joueur. Le jeu se termine si le joueur a éliminé complètement le mille-pattes ou si celui-ci touche le joueur.

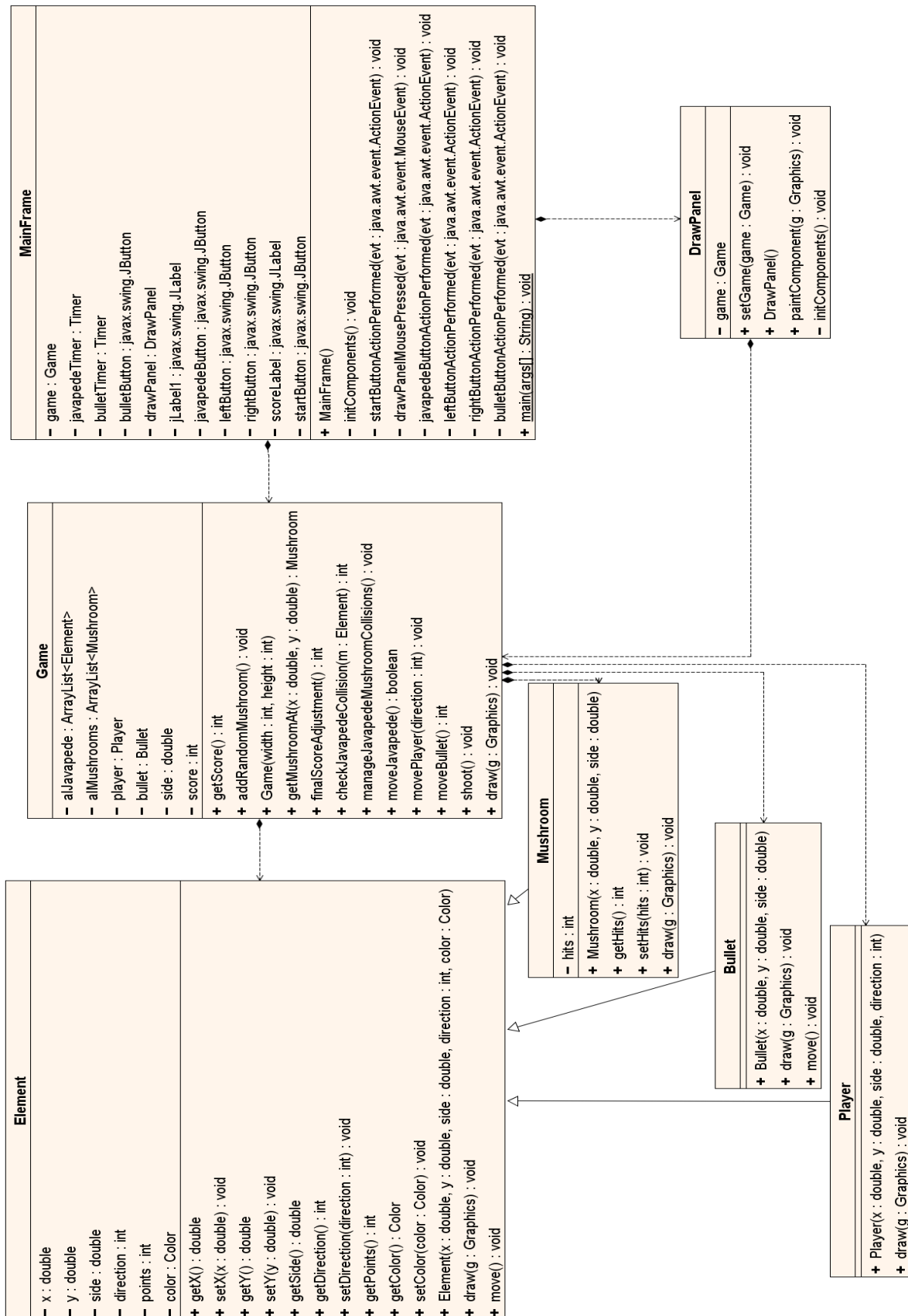
Consultez le modèle **Javapede.jar** sur eduMoodle.



Créez un nouveau projet nommé **Javapede**.

Réalisez ce programme en vous basant sur la version exécutable, ainsi que sur le diagramme UML fourni tout en respectant les instructions et précisions données dans la suite. Pour autant que possible, chaque classe fille doit se servir des méthodes de sa classe mère.

Diagramme UML



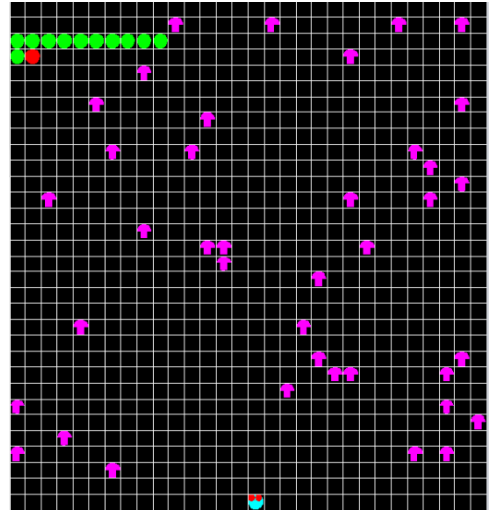
Attributs et méthodes standard

Element

Implémentez la classe **Element** – avec ses attributs, accesseurs et manipulateurs requis – en vous basant sur la description UML donnée et les indications supplémentaires ci-dessous.

La classe **Element** représente un élément de base du jeu. Les coordonnées **x** et **y** correspondent au coin supérieur gauche du carré de côté **side** à l'intérieur duquel l'élément est dessiné. L'attribut **direction** correspond au sens de déplacement horizontal de l'élément (-1 → gauche, +1 → droite, 0 → pas de déplacement horizontal). L'attribut **points** correspond au nombre de points gagnés, l'attribut **color** correspond à la couleur de dessin de l'élément.

- Le constructeur initialise l'attribut **points** à une valeur aléatoire dans l'intervalle [1,10] et les autres attributs aux valeurs passées par paramètre.
- La méthode **draw** dessine l'élément sous forme d'un disque de diamètre **side**. Les attributs **x** et **y** correspondent au coin supérieur gauche du carré dans lequel le disque est inscrit.
- L'aire de jeu est subdivisée en 30 x 32 cellules carrées de côté **side**. Sur le dessin ci-contre, ces cellules sont indiquées à l'aide d'un grillage blanc (ce grillage n'est pas à dessiner). La méthode **move** déplace l'élément horizontalement d'une cellule dans la direction **direction** pour autant qu'il reste dans les limites de l'aire de jeu. Si ce n'est pas le cas, l'attribut **direction** change de signe et l'élément se déplace d'une cellule vers le bas, sauf si le bord inférieur est atteint. Dans ce cas, l'élément se déplace d'une cellule vers le haut.

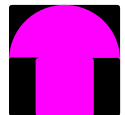


Mushroom

Implémentez la classe **Mushroom** – avec ses attribut, accesseur et manipulateur requis - en vous basant sur la description UML donnée et les indications supplémentaires ci-dessous.

La classe **Mushroom** représente un champignon. L'attribut **hits** correspond au nombre de fois que le champignon a été touché. Le constructeur initialise sa couleur à la couleur magenta et la direction à 0.

La méthode **draw** dessine le champignon. Sa couleur est modifiée en fonction du nombre de coups **hits** (0 → magenta, 1 → bleu, 2 → gris). La méthode ajoute à l'élément dessiné un rectangle noir de dimensions **side** x **side**/2 qui cache la partie inférieure du champignon. Ensuite, la tige (de la même couleur que le chapeau) est dessinée sous forme d'un carré de côté **side**/2 au milieu de la partie inférieure du dessin.



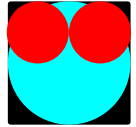
Player

Implémentez la classe **Player** en vous basant sur la description UML donnée et les indications supplémentaires ci-dessous.

La classe **Player** représente le joueur. Le constructeur initialise sa couleur à la couleur cyan.

- La méthode **move** déplace le joueur d'une cellule dans la direction **direction** pour autant qu'il reste entièrement dans les limites de l'aire de jeu.

- La méthode **draw** dessine le joueur. La méthode ajoute deux disques adjacents de couleur rouge, de diamètre **side/2** dans la moitié supérieure de l'élément dessiné.



Bullet

Implémentez la classe **Bullet** en vous basant sur la description UML donnée et les indications supplémentaires ci-dessous.

La classe **Bullet** représente un projectile. Le constructeur initialise sa couleur à la couleur blanche et la direction à 0.

- La méthode **draw** dessine le projectile sous forme d'un trait vertical de hauteur **side** et de largeur 1 px passant par le centre de l'élément.
- La méthode **move** déplace le projectile d'une cellule vers le haut.



Game

Implémentez la classe **Game** – avec ses attributs et son accesseur requis – en vous basant sur la description UML donnée et les indications supplémentaires ci-dessous.

Cette classe est responsable de la gestion du jeu. L'attribut **score** correspond au nombre de points marqués.

- La méthode **addRandomMushroom** ajoute à la liste **alMushrooms** un nouveau champignon placé dans une cellule aléatoire sur l'aire de jeu (pour rappel, l'aire de jeu se compose de 30 x 32 carrés de côté **side**) sachant que la première et la dernière ligne ne contiennent jamais de champignon. Il n'y a pas besoin de vérifier s'il y a déjà un champignon à la position choisie (il pourrait donc y avoir deux champignons dans la même cellule).
- Le constructeur :
 - calcule la valeur de **side** de manière à maximiser la surface de jeu en fonction des dimensions du canevas transmises par paramètre ;
 - initialise le score à 0 ;
 - ajoute de manière aléatoire 40 champignons sur l'aire de jeu ;
 - initialise le joueur **player** : il se situe en bas au milieu de l'aire de jeu et sa direction vaut 0 ;
 - initialise le mille-pattes **alJavapede**, dont tous les éléments sont situés en haut de l'aire de jeu et dirigés vers la droite. Il se compose d'une tête de couleur rouge située au milieu de la première ligne et d'un corps formé par 11 éléments de couleur verte situés à gauche de la tête (voir schéma ci-dessous) ;

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
				●	●	●	●	●	●	●	●	●	●	●	●	●													

- La méthode **getMushroomAt** retourne le premier champignon qui se trouve à la position indiquée par les paramètres. S'il n'y a pas de champignon à la position indiquée, la méthode retourne **null**.
- La méthode **finalScoreAdjustment** retourne le nombre maximal de points disponibles parmi les éléments qui composent le mille-pattes ou 0 s'il n'a pas d'éléments.

- La méthode **checkJavapedeCollision** retourne l'indice de l'élément du mille-pattes qui se trouve à la même position que l'élément passé en paramètre ou -1 si un tel élément n'existe pas.
 - La méthode **manageJavapedeMushroomCollisions** gère les collisions entre les éléments du mille-pattes et les champignons. Pour cela, elle parcourt la liste de champignons **alMushrooms** et vérifie pour chaque champignon si une collision a lieu entre le champignon et le mille-pattes. Si une telle collision a lieu, on effectue les opérations suivantes avec l'élément du mille-pattes concerné :
 - sa direction est inversée ;
 - les deux déplacements suivants sont effectués :
 - on le déplace d'une cellule dans la nouvelle direction pour autant qu'on reste dans les limites de l'aire de jeu ;
 - on le déplace d'une cellule vers le bas.
 - La méthode **moveJavapede** effectue les opérations suivantes pour chacun des éléments du mille-pattes :
 - l'élément est déplacé ;
 - les collisions entre les éléments du mille-pattes et les champignons sont gérées ;
 - s'il y a collision entre un des éléments du mille-pattes et le joueur, le score est adapté en retranchant le nombre maximal de points (ou la moyenne, selon l'alternative choisie) des éléments qui composent le mille-pattes. Dans ce cas, la méthode retourne **false**, sinon elle retourne **true**.
 - La méthode **movePlayer** modifie la direction du joueur selon la direction passée en paramètre. Ensuite, le joueur est déplacé.
 - La méthode **moveBullet** est responsable du déplacement du projectile **bullet**, pour autant que celui-ci existe. Plusieurs cas peuvent se présenter :
 - il y a un champignon à l'endroit où se trouve le projectile. Dans ce cas, le nombre de fois que le champignon concerné a été touché est incrémenté et les points du champignon sont ajoutés au score. Si le champignon a été touché 3 fois, il est effacé de la liste des champignons. Le projectile est détruit (valeur **null**) et le score est incrémenté ;
 - il y a une collision entre le projectile et un des éléments du mille-pattes. Dans ce cas, le score est augmenté du nombre de points du dernier élément du mille-pattes et celui-ci est effacé. Le projectile est détruit ;
 - le projectile a atteint le bord supérieur de l'aire de jeu. Le projectile est détruit ;
 - dans tous les autres cas, le projectile est déplacé d'une cellule vers le haut.
- La méthode retourne une des trois valeurs 0, 1 ou 2 (0 → le projectile n'a pas été déplacé mais détruit sans que le mille-pattes ne soit éliminé, 1 → le projectile a été déplacé, 2 → le mille-pattes a été éliminé).
- La méthode **shoot** crée un nouveau projectile juste au-dessus du joueur, pour autant qu'il n'y ait pas déjà de projectile.
 - La méthode **draw** :
 - dessine un rectangle noir avec les dimensions de l'aire de jeu ;
 - dessine le mille-pattes et les champignons ;
 - dessine le joueur et, si possible, le projectile.

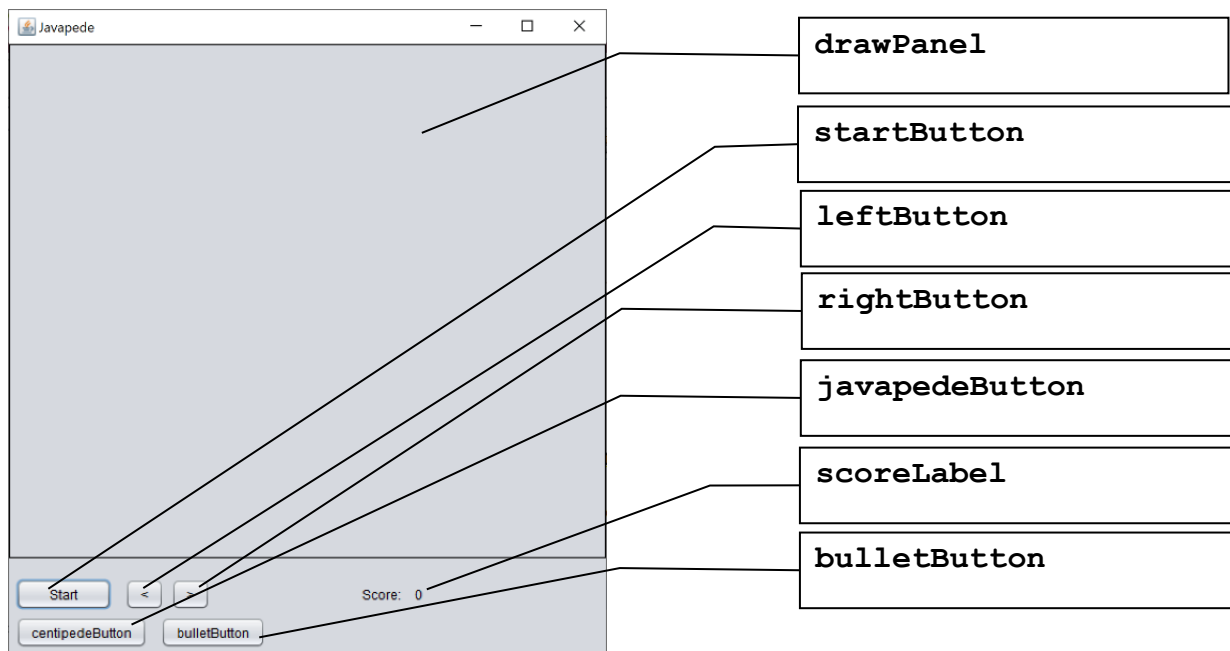
DrawPanel

Implémentez la classe **DrawPanel** – avec son attribut et manipulateur requis – en vous basant sur la description UML donnée. La méthode **paintComponent** dessine, si possible, le jeu sur le canevas.

MainFrame

Implémentez la classe **MainFrame** – avec ses attributs requis – en vous basant sur la description UML donnée et les indications supplémentaires ci-dessous.

- Réalisez l'interface graphique ci-dessous. Pensez à inscrire « Javapede » dans l'entête de la fenêtre.

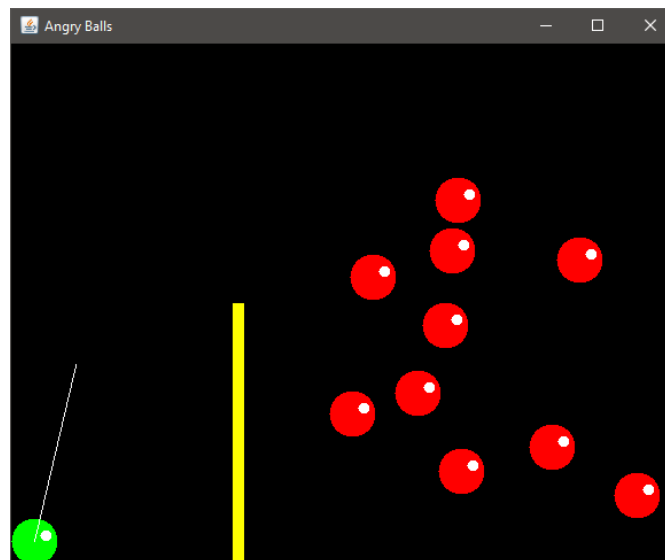


- Le constructeur initialise le chronomètre **javapedeTimer** avec une périodicité de 150 ms, associé au bouton caché **javapedeButton** et le chronomètre **bulletTimer** de périodicité 70 ms, associé au bouton caché **bulletButton**.
- La méthode **startButtonActionPerformed** :
 - crée un nouveau jeu basé sur les dimensions du canevas ;
 - démarre le chronomètre **javapedeTimer**.
- Une pression sur un bouton de la souris dans le canevas crée un projectile et démarre le chronomètre **bulletTimer**.
- La méthode **javapedeButtonActionPerformed** déplace le mille-pattes. S'il n'est pas possible de déplacer le mille-pattes, les chronomètres sont arrêtés et le texte « YOU LOST » est affiché dans l'en-tête de la fenêtre.
- Les méthodes **leftButtonActionPerformed** et **rightButtonActionPerformed** déplacent le joueur dans la direction adéquate.
- La méthode **bulletButtonActionPerformed** :
 - déplace le projectile ;
 - met à jour l'affichage du score ;
 - si le projectile n'a pas été déplacé, le chronomètre **bulletTimer** est arrêté ;

- si le mille-pattes a été détruit, le chronomètre **javapedeTimer** est arrêté et le texte « YOU WON » est affiché dans l'en-tête de la fenêtre.

Exercice G.4 : Angry balls

Dans la suite vous allez développer l'application « Angry Balls ». Vous trouvez une version exécutable du programme, nommé **AngryBalls.jar**, sur eduMoodle. Avant de continuer, il est recommandé de lancer et de tester ce programme. Le but du jeu est d'éliminer toutes les balles rouges à l'aide de la balle verte du joueur. À cette fin, appuyez sur un bouton de la souris à un endroit quelconque dans la fenêtre et bougez-la pour définir le vecteur force à appliquer. Ensuite, lâchez le bouton pour catapultuler la balle verte.



Créez dans **NetBeans** un nouveau projet nommé **AngryBalls**.

Réalisez cette application en vous basant sur la version exécutable fournie ainsi que sur le diagramme UML de la dernière page de l'exercice et tout en respectant les instructions et précisions données dans la suite.

La classe Ball

Cette classe sert à modéliser une balle.

Attributs :

x et **y** représentent les coordonnées du centre de la balle.

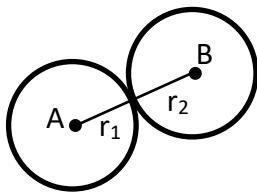
r représente le rayon de la balle.

color représente la couleur de remplissage de la balle.

Méthodes :

- Implémentez le **constructeur**, les **accesseurs** (*getters*) et les **manipulateurs** (*setters*) selon le schéma UML.
- La méthode **draw** sert à dessiner une balle sans bordure de couleur **color**. Un petit disque blanc de rayon $r/4$ est dessiné quelque part dans la partie supérieure droite de la balle.
- La méthode **isTouching** retourne *true* si la balle touche la balle **pBall** passée en paramètre. Sinon, la méthode retourne la valeur *false*.

Remarque : Deux balles se touchent lorsque la distance entre leurs centres respectifs est inférieure ou égale à la somme de leurs rayons respectifs.



Voici la formule pour calculer la distance entre deux points A et B : $\sqrt{(x_B - x_A)^2 + (y_B - y_A)^2}$

La classe MovingBall

Cette classe est dérivée de sa classe mère **Ball** et sert à modéliser une balle en mouvement.

Attributs :

dX et **dY** représentent le pas de déplacement horizontal respectivement vertical. Tous les deux sont initialisés à 0.

Méthodes :

- Implémentez le **constructeur**, les **accesseurs** et les **manipulateurs** selon le schéma UML.
- La méthode **move** effectue le déplacement d'une balle en faisant une mise à jour des coordonnées **x** et **y** en fonction des pas de déplacement **dX** et **dY**. De plus, **dY** est incrémenté de 0.15.

La classe Game

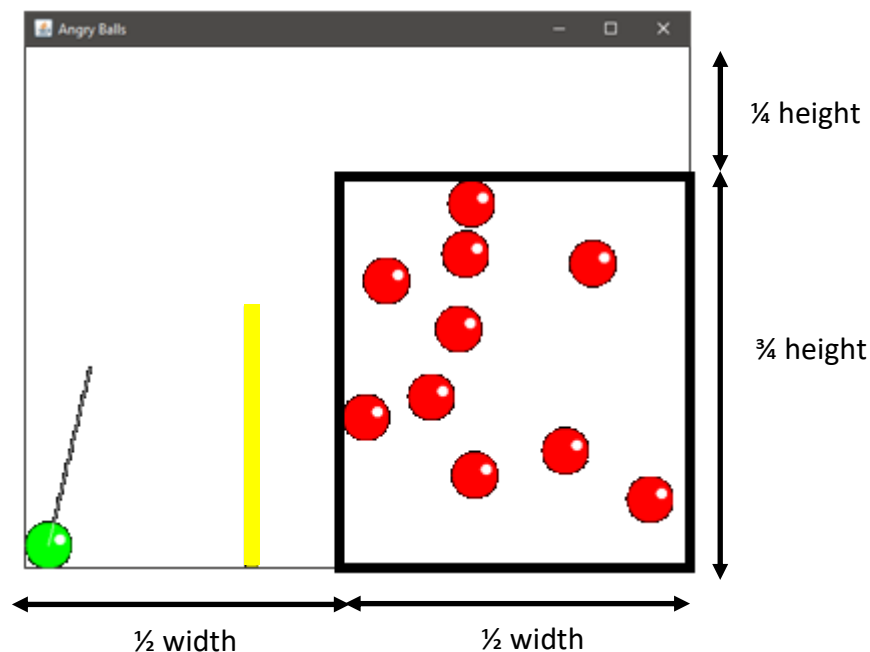
La classe **Game** est responsable de la gestion du jeu.

Attributs :

width	représente la largeur de la surface du jeu.
height	représente la hauteur de la surface de jeu.
playerBall	représente la balle du joueur (balle verte).
alBalls	représente la liste avec les balles rouges.
mousePosition	représente la position de la souris sur la surface de jeu et est initialisé à null .

Méthodes :

- Le constructeur **Game** :
 - Initialise les attributs **width** et **height** avec les paramètres respectifs.
 - Remplit la liste **alBalls** avec 10 balles rouges de type **Ball**. Le rayon d'une balle est de 20 pixels. La position d'une balle est déterminée au hasard de manière à ce que la balle se trouve complètement dans la surface encadrée du dessin ci-dessous.

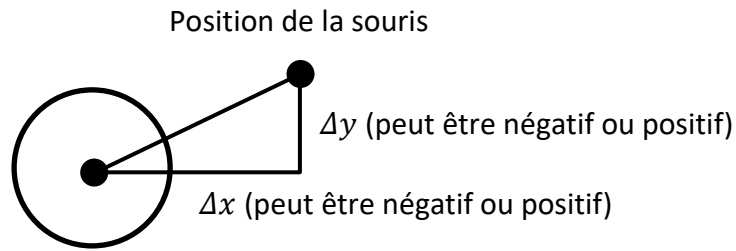


- Crée la balle verte du joueur (**playerBall**) de type **MovingBall** (position initiale dans le coin gauche en bas de la surface de dessin ; rayon de 20 pixels).
- Implémentez le manipulateur (*setter*) pour l'attribut **mousePosition**.

- La méthode **setPlayerBallSteps** sert à mettre à jour les attributs **dX** et **dY** de **playerBall** si la valeur de **mousePosition** est différente de **null**. Utilisez les formules suivantes :

$$dX = \Delta x / 20$$

$$dY = \Delta y / 20$$



Avec :

- Δx étant la différence entre les abscisses x de la position de la souris et celle du centre de la balle.
- Δy étant la différence entre les ordonnées y de la position de la souris et celle du centre de la balle.
- La méthode **draw** dessine
 - toutes les balles rouges.
 - la balle verte du joueur.
 - le mur jaune, sans bordure, d'une largeur de 10 pixels, placé à un tiers de la largeur totale de la surface de jeu. Sa hauteur occupe la moitié de la surface de jeu.
 - le « vecteur force », représenté par un trait blanc partant du centre de la balle verte jusqu'à la position actuelle de la souris. Ce trait est uniquement dessiné si l'attribut **mousePosition** est différent de **null**.
- La méthode **move**
 - déplace la balle du joueur.
 - vérifie si la balle du joueur touche le mur. Vous trouvez le code qui réalise ce test dans le fichier **code_a_copier.txt** dans votre dossier. Copiez ce code dans votre méthode **move**.
 - vérifie si la balle du joueur touche des balles rouges et enlève celles-ci le cas échéant.
 - retourne la valeur 1 si la balle verte a quitté complètement la surface de jeu en bas, à droite ou à gauche. Sinon la valeur 0 est retournée.
- La méthode **playerBallReset** crée une nouvelle balle verte (**playerBall**) dans le coin en bas à gauche de la surface de jeu.
- La méthode **isOver** retourne la valeur booléenne *true* si la liste des balles rouges est vide. Sinon, la valeur *false* est retournée.

La classe DrawPanel

La classe **DrawPanel** est responsable de la visualisation du jeu.

- Cette classe possède un attribut **game** et un manipulateur **setGame**.
- La méthode **paintComponent** dessine d'abord l'arrière-plan noir, puis, si possible, le jeu.

La classe MainFrame

Attributs :

game sert à représenter le jeu.

timer chronomètre, responsable pour le mouvement de la balle verte.

- Dans le constructeur,
 - créez une instance **game** dans le constructeur et veillez à ce que l'attribut **game** du **drawPanel** pointe sur cette même instance.
 - créez une instance du chronomètre **timer** de façon à exécuter le bouton **stepButton** toutes les 10 millisecondes.
 - configurez le titre de l'application à « Angry Balls ». Vous pouvez aussi configurer le titre directement dans l'interface graphique de NetBeans.
 - rendez le bouton **stepButton** invisible.
- Lorsque l'utilisateur appuie sur un bouton de la souris et si le chronomètre **timer** n'est pas actif, alors :
 - l'attribut **mousePosition** de l'objet **game** est mis à jour.
 - le canevas est redessiné.

Ces mêmes actions sont à programmer pour le cas où l'utilisateur bouge la souris avec un bouton enfoncé.

- Si le bouton de la souris est relâché et si le chronomètre est arrêté, alors les attributs **dX** et **dY** de la balle verte sont mis à jour, **mousePosition** dans l'objet **game** est remis à **null** et le chronomètre est déclenché.
- À chaque tic du chronomètre, la balle verte est déplacée. Si la balle a quitté la surface de jeu en bas, à droite ou à gauche, le chronomètre est arrêté et une nouvelle balle verte est affichée. Si toutes les balles rouges ont été éliminées, le chronomètre est arrêté et un nouveau jeu est créé. Le dessin est actualisé.

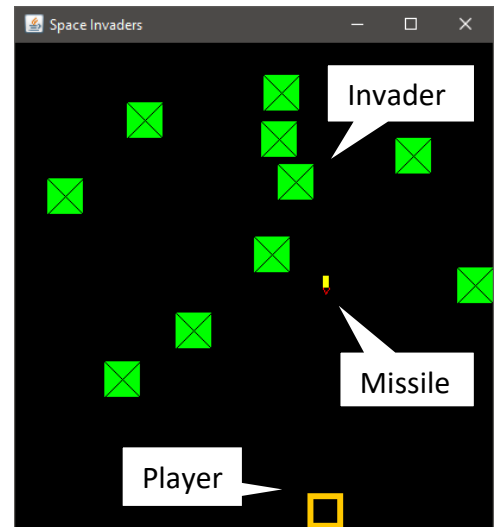
Schéma UML :



Exercice G.5 : Space Invaders

Il s'agit de réaliser le jeu « Space Invaders » où le joueur doit combattre des envahisseurs extraterrestres. Consultez le modèle **SpaceInvaders.jar** pour vous familiariser avec le fonctionnement de l'application. Bougez la souris dans la surface de jeu pour déplacer votre joueur. Appuyez le bouton gauche pour lancer un missile.

Créez le projet « **SpaceInvaders** » et respectez les instructions et précisions données ci-dessous ainsi que le diagramme UML que vous trouvez en dernière page.



Classe MovingObject

Cette classe est la classe de base pour les différents objets du jeu (Player, Invader, Missile).

Dérivez la classe **MovingObject** de la classe standard de Java **Rectangle** ! La classe Rectangle (contenue dans le paquet java.awt) représente un espace rectangulaire et dispose des attributs suivants :

- x coordonnées x du coin supérieur-gauche.
- y coordonnées y du coin supérieur-gauche.
- width largeur du rectangle
- height hauteur du rectangle

De plus vous pouvez profiter de la méthode **intersects(Rectangle r)** qui permet de vérifier si 2 rectangles se touchent.

Ajoutez les attributs et méthodes suivants :

color représente la couleur de l'objet.

dX représente le pas de déplacement horizontal. **dX** est initialisé à 0.

dY représente le pas de déplacement vertical. **dY** est initialisé à 0.

MovingObject(...) constructeur pour initialiser les attributs hérités ainsi que la couleur **color**.

getters/setters laissez NetBeans générer les accesseurs et manipulateurs qui figurent dans le schéma UML.

draw(...) dessine aux coordonnées (x,y) un rectangle plein, aux dimensions **width** × **height** et de couleur **color**. Les coordonnées (x,y) désignent le point supérieur gauche du rectangle.

move() déplace l'objet de **dX** unités en horizontale et de **dY** unités en verticale.

Classe Player

Cette classe est une spécialisation de la classe **MovingObject** et modélise le joueur.

Player(...) constructeur permettant de créer un joueur de couleur orange, aux dimensions 30 × 30 pixels et positionné aux coordonnées (x,y).

draw(...) dessine un rectangle plein, de couleur noire et aux dimensions 20 × 20 au centre du rectangle dessiné par la méthode **draw** héritée (voir modèle).

Classe Invader

Cette classe est une spécialisation de la classe **MovingObject** et modélise un envahisseur.

Invader(...)	constructeur permettant de créer à la position (x,y) un envahisseur de taille 30 × 30 pixels et de couleur verte. Chacun des attributs dX et dY est à initialiser à une valeur aléatoire entière de l'intervalle [-2 , 2].
draw(...)	méthode qui dessine en plus du rectangle dessiné par la méthode héritée, 2 lignes diagonales noires à l'intérieur du rectangle (voir modèle).
move(...)	méthode qui déplace l'objet de dX respectivement dY unités. L'envahisseur rebondit sur les bords à gauche, en haut et à droite. En hauteur, l'envahisseur rebondit sur la ligne imaginaire qui se trouve à ¾ de la hauteur totale.

Classe Missile

Cette classe est une spécialisation de la classe **MovingObject** et modélise un missile.

Missile(...)	constructeur permettant de créer à la position (x,y) un missile de taille 5 × 10 pixels et de couleur jaune. L'attribut dX est initialisé à 0 ; dY est initialisé à moins trois (-3).
draw(...)	méthode qui dessine en plus du rectangle dessiné par la méthode héritée, un jet de feu réalisé par 2 lignes rouges en forme de V en dessous du missile (voir la figure sur la première page du questionnaire). La hauteur du V est de 5 pixels.

Classe Game

Cette classe modélise le jeu en lui-même.

player	représente le joueur.
missile	représente le missile (initialisé à null).
allInvaders	liste pour gérer tous les envahisseurs en jeu.
Game(...)	le constructeur crée 10 envahisseurs à des positions aléatoires dans la moitié supérieure de la surface de jeu aux dimensions width × height . Veillez à ce que les envahisseurs se trouvent complètement à l'intérieur de cette surface. Le joueur est créé au coin inférieur gauche de la surface de jeu.
draw(...)	dessine les envahisseurs, le joueur et le missile s'il n'est plus null .
move(...)	déplace d'abord les envahisseurs. Ensuite s'il n'est pas null , le missile est déplacé. Si le missile sort complètement de la surface de jeu, il est remis à null . Pour terminer, la méthode vérifie si un éventuel missile entre en collision avec un envahisseur. Profitez de la méthode intersects héritée de la classe Rectangle pour vous faciliter le test. Si c'est le cas, l'envahisseur est détruit et le missile remis à null .
setPlayer(...)	met le joueur à la position x transmis par paramètre. Veillez à ce que le joueur ne déborde pas la surface de jeu.
isOver()	détermine si le jeu est terminé, donc s'il n'y a plus d'envahisseurs.
launchMissile()	crée un nouveau missile qui sortira du milieu du joueur. Un nouveau missile est seulement créé s'il n'y en a pas en jeu actuellement.

Classe DrawPanel

game représente le jeu. L'attribut possède un manipulateur **setGame(...)**.

paintComponent(...) efface le contenu du canevas (couleur noire) et dessine, si possible, le jeu.

Classe MainFrame

game représente le jeu. Il faut effectuer toutes les initialisations nécessaires pour que le jeu puisse être affiché.

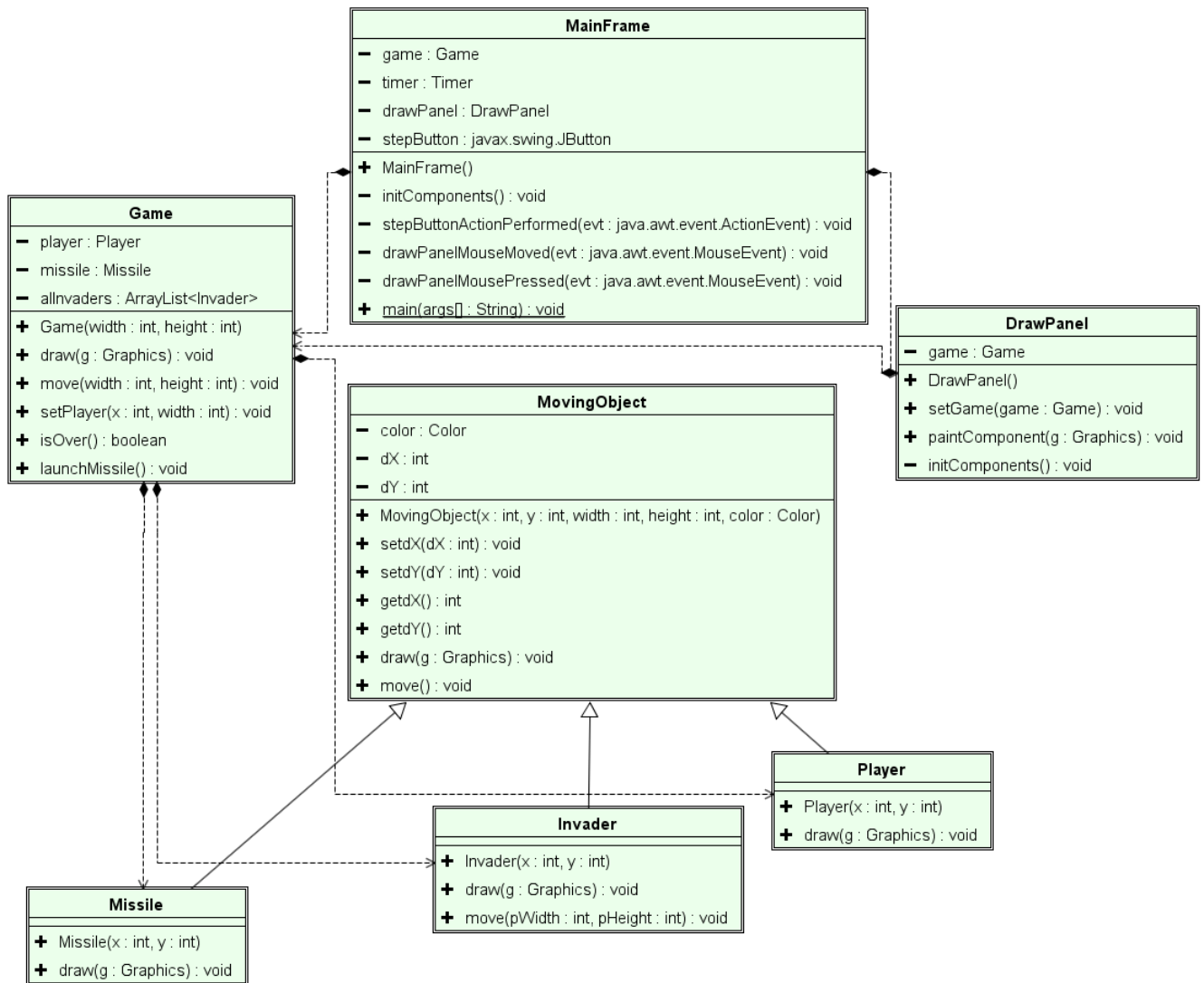
timer chronomètre utilisé pour effectuer le déplacement des envahisseurs et du missile. Il est démarré dès le départ du programme et déclenche toutes les 10 millisecondes un clic du bouton caché **stepButton**.

stepButtonActionPerformed déplace les envahisseurs et le missile. Si tous les envahisseurs sont détruits, un nouveau jeu est créé.

- Lorsque la souris est déplacée à l'intérieur de la surface de jeu, le joueur suit la position de la souris sur le bord inférieur de la fenêtre.
- Un clic sur le bouton gauche de la souris lance un missile s'il n'y existe pas déjà un.
- Le titre de la fenêtre de l'application est « Space Invaders »

Extensions

- Au lieu de profiter de la classe prédéfinie **Rectangle** de Java, créez votre propre méthode intersects.
- Ajoutez la gestion d'un score
- Ajoutez des niveaux de jeu (level)
- Ajoutez des bombes que les envahisseurs laissent tomber.

Diagramme UML :

Exercice G.6 : Space Invaders II

Dans la suite vous allez développer le jeu Space Invaders II. Dans ce jeu il s'agit d'éliminer autant d'envahisseurs que possible avec des missiles. Le joueur peut bouger à gauche et à droite avec la souris et tirer avec le bouton gauche. Tant qu'un missile est en cours de route, il n'est pas possible d'en tirer un nouveau.

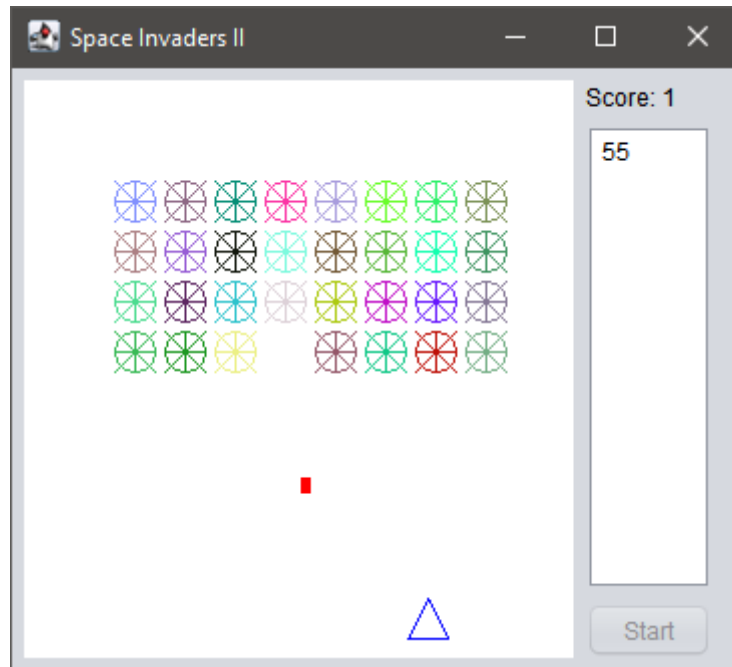
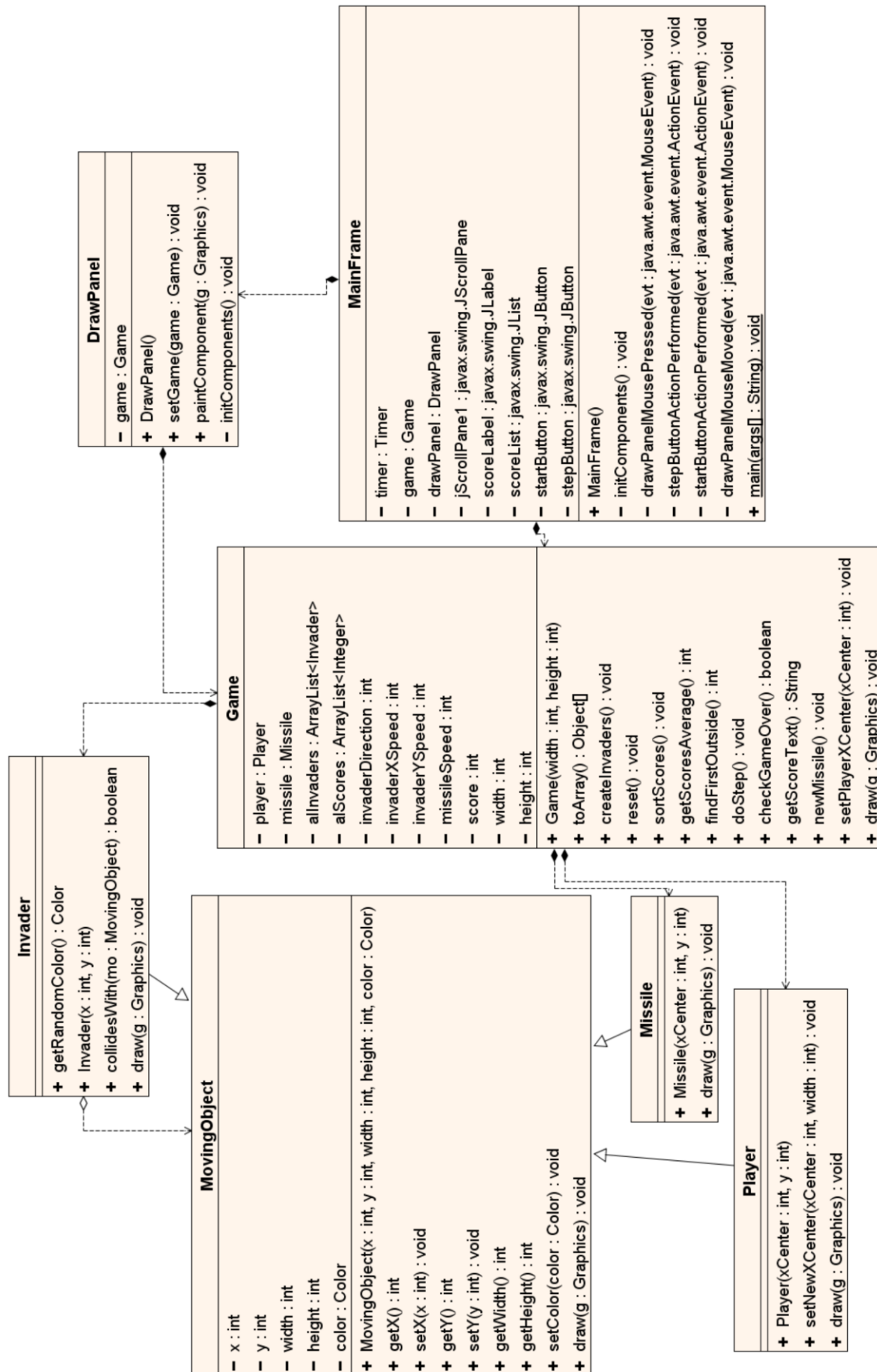


Diagramme UML du jeu :



La classe **MovingObject**

Implémentez la classe **MovingObject** en vous basant sur le diagramme UML donné et les indications supplémentaires ci-dessous.

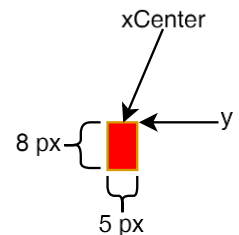
Les coordonnées **x** et **y** représentent le coin supérieur gauche de l'objet, **width** et **height** sa largeur et hauteur, **color** sa couleur.

- Implémentez les attributs, le constructeur ainsi que les accesseurs et manipulateurs.
- La méthode **draw** ne fixe que la couleur de dessin.

La classe **Missile**

La classe **Missile** représente le missile tiré par le joueur. Implémentez la classe **Missile** en vous basant sur le diagramme UML donné et les indications supplémentaires ci-dessous.

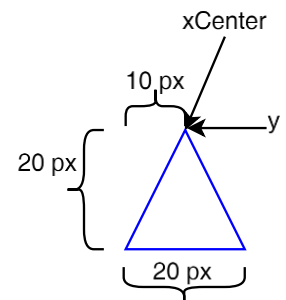
- Le constructeur initialise le missile avec les valeurs indiquées sur l'image ci-contre. Le paramètre **xCenter** représente le centre horizontal du missile. Le missile est de couleur rouge.
- La méthode **draw** dessine le missile.



La classe **Player**

La classe **Player** représente le joueur. Implémentez la classe **Player** en vous basant sur le diagramme UML donné et les indications supplémentaires ci-dessous.

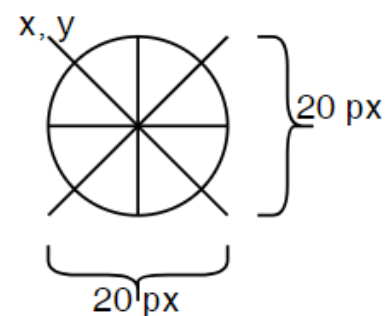
- Le constructeur initialise le joueur avec les valeurs indiquées sur l'image ci-contre. Sa couleur est bleue.
- La méthode **setNewXCenter** place le joueur à la nouvelle position, mais seulement si celle-ci maintient le joueur entièrement à l'intérieur de l'aire de jeu dont la largeur est passée en paramètre.
- La méthode **draw** dessine le joueur.



La classe **Invader**


La classe **Invader** représente un envahisseur. Implémentez la classe **Invader** en vous basant sur le diagramme UML donné et les indications supplémentaires ci-dessous.

- La méthode **getRandomColor** retourne une couleur aléatoire.
- Le constructeur initialise un envahisseur avec les valeurs indiquées sur l'image ci-contre et de couleur noire. Ensuite il change la couleur en couleur aléatoire.
- La méthode **collidesWith** détermine si l'envahisseur touche un objet donné, c.-à-d. si les rectangles englobants se touchent. Dans ce cas elle retourne **true**, sinon **false**.
- La méthode **draw** dessine l'envahisseur comme sur l'image ci-dessus.

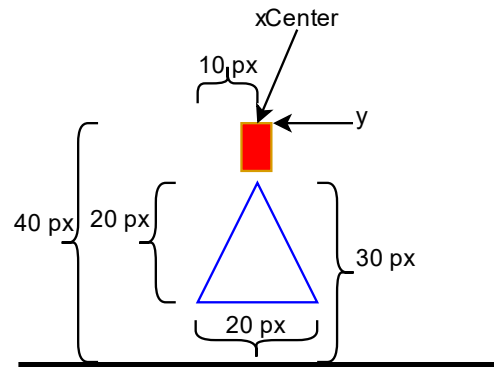


La classe Game

La classe **Game** gère le jeu. Implémentez la classe **Game** en vous basant sur le diagramme UML donné et les indications supplémentaires ci-dessous.

- Implémentez les attributs et le constructeur sachant que :
 1. **player** représente le joueur ;
 2. **missile** représente un missile, s'il y en a un ;
 3. **alInvaders** contient les envahisseurs qui existent encore ;
 4. **alScores** contient les scores ;
 5. **invaderDirection** prend initialement la valeur 1 pour indiquer que les envahisseurs se déplacent de la gauche vers la droite. Cet attribut prend la valeur -1 quand les envahisseurs se déplacent de la droite vers la gauche ;
 6. **invaderXSpeed** prend initialement la valeur 5 et représente la vitesse horizontale des envahisseurs, c'est-à-dire le nombre de pixels que chaque envahisseur bouge vers la droite ou la gauche ;
 7. **invaderYSpeed** a une valeur de 10 et représente le nombre de pixels que tous les envahisseurs se déplacent vers le bas lorsqu'au moins un a atteint le bord droit ou gauche ;
 8. **missileSpeed** a une valeur de 10 et représente le nombre de pixels qu'un missile se déplace vers le haut lorsqu'il est en cours de route ;
 9. **score** est initialisé à 0 et représente le score actuel du joueur ;
 10. **width** et **height** représentent les dimensions de l'aire de jeu.
- La méthode **createInvaders** vide la liste et ajoute 32 envahisseurs. Ils sont placés en 4 rangées de 8 avec 5 pixels de distance horizontale entre les envahisseurs et 5 pixels de distance verticale entre les 4 rangées (voir le graphique à droite). Le premier se trouve à 10 pixels du bord gauche et à 10 pixels du bord supérieur de l'aire de jeu.
 
- La méthode **reset** redonne les valeurs initiales aux attributs **invaderXSpeed**, **score** et **missile**. Elle crée un nouveau joueur **player** situé au milieu de l'aire de jeu et à 30 pixels du bas (voir image de la page suivante), ainsi que les envahisseurs.
- La méthode **sortScores** trie les scores par ordre croissant par la méthode de sélection directe.
- La méthode **getScoresAverage** retourne la moyenne des scores ou 0 si la liste est vide.
- La méthode **findFirstOutside** retourne l'index du premier envahisseur qui, lors de son prochain mouvement, toucherait le bord droit ou gauche. S'il n'y en a pas elle retourne -1.
- La méthode **doStep** est responsable de la gestion d'une étape du jeu.
 1. Si un missile est en cours de route, la méthode déplace le missile. Ensuite la méthode vérifie si le missile a quitté complètement l'aire de jeu. Si c'est le cas, le missile est enlevé, sinon la méthode efface le premier envahisseur qui a été touché. Dans ce cas, le score est incrémenté et le missile enlevé. S'il ne reste plus d'envahisseurs, la vitesse horizontale des envahisseurs est augmentée de 50 % (la partie décimale est à ignorer) et de nouveaux envahisseurs sont créés.
 2. Si un envahisseur toucherait le bord droit ou gauche lors de son prochain mouvement, la direction de déplacement est inversée.
 3. Tous les envahisseurs sont déplacés horizontalement. Si la direction de déplacement a changé, ils sont aussi déplacés verticalement. Chaque envahisseur qui quitte ainsi entièrement le canevas vers le bas est enlevé.
- La méthode **checkGameOver** vérifie si un envahisseur touche le joueur. Si c'est le cas, la liste des scores est actualisée et triée et la méthode retourne **true**. Sinon la méthode retourne **false**.

- La méthode **getScoreText** retourne le score précédé du texte « Score: ». Si le score actuel est supérieur à la moyenne des scores existants, elle retourne le score précédé du texte « ! Score: ».
- La méthode **newMissile** vérifie si un joueur existe et s'il n'y a pas de missile en cours de route. Si tel est le cas elle crée un nouveau missile situé au-dessus du joueur (voir image ci-contre).
- La méthode **setPlayerXCenter** place le joueur horizontalement à la position passée en paramètre.
- La méthode **draw** dessine le joueur, le missile et les envahisseurs s'ils existent.



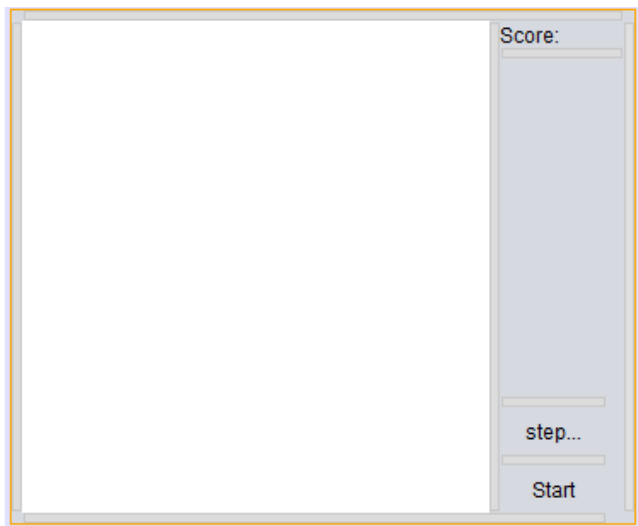
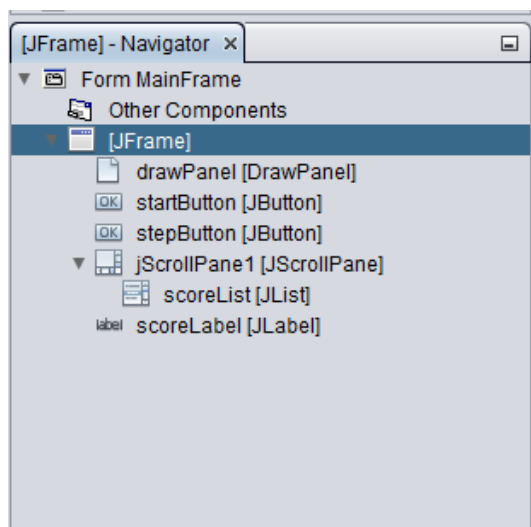
La classe DrawPanel

- Implémentez la classe **DrawPanel** – avec son attribut et manipulateur requis – en vous basant sur le diagramme UML donné.
- La méthode **paintComponent** dessine un rectangle blanc sur toute la surface du canevas et, si possible, le jeu.

La classe MainFrame

Implémentez la classe **MainFrame** – avec ses attributs requis – en vous basant sur le diagramme UML donné et les indications supplémentaires ci-dessous.

- Implémentez l'interface graphique (voir le graphique ci-dessous). L'attribut **timer** représente le chronomètre qui a une périodicité de 50 millisecondes et est lié au bouton caché **stepButton**. L'attribut **game** représente le jeu. Le titre de l'application est « Space Invaders II ».



- Complétez le constructeur qui crée un nouveau chronomètre et un nouveau jeu.
- La méthode **startButtonActionPerformed** désactive le bouton de démarrage, fait une mise à zéro du jeu et lance le chronomètre.
- La méthode **stepButtonActionPerformed** exécute une étape du jeu. Elle affiche le score actuel et vérifie si le jeu est terminé. Si c'est le cas elle arrête le chronomètre, actualise l'affichage des scores et rend le bouton de démarrage disponible.

- La méthode **drawPanelMouseMoved** bouge le joueur si un jeu est en cours. La position horizontale de la souris correspond au milieu du joueur.
- La méthode **drawPanelMousePressed** crée un nouveau missile.