EXAMEN DE FIN D'ÉTUDES SECONDAIRES – Sessions 2024

QUESTIONNAIRE

Date :	18.09.24		*Horaire :*	08:15 - 12:15	*Durée :*	240 minutes
Discipline :	PROGR	*Type :*	écrit	*Section(s) :*	GIN	
				Numéro du candidat :		

Indication importante :

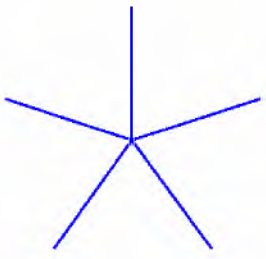
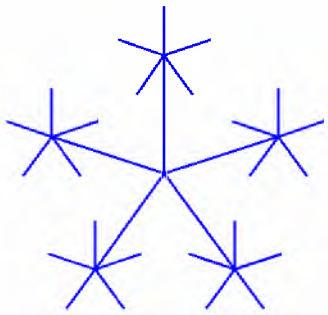
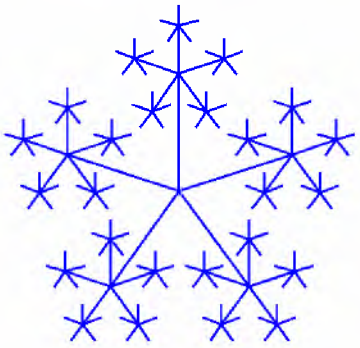
- Renommez d'abord le dossier EEEE_GIN_x-xx en remplaçant EEEE par le code de votre établissement et x-xx par votre numéro d'examen !
- Les projets *NetBeans* à compléter se trouvent dans ce dossier
- Des modèles de démonstration des programmes finis se trouvent dans le sous-dossier **dist**.
- *Attention : après avoir ouvert le projet en NetBeans et avant d'ouvrir **MainFrame**, effectuez un « Clean and Build » pour éviter des difficultés avec l'intégration du **DrawPanel**.*

Question 1 : « StarFractal » [6,5p.]

Un modèle de démonstration du projet fini se trouve dans votre dossier sous **dist/StarFractal-demo.jar**. Démarrez ce programme maintenant pour vous familiariser avec son fonctionnement !
Le code dans **MainFrame** et dans **DrawPanel** restera inchangé. Toutes vos modifications se feront dans la classe **Star**.
La fractale est construite de façon récursive :
Sur une étoile de base sont placées 5 étoiles dont les centres se trouvent chaque fois au bout de chaque bras de l'étoile de base. Les angles de rotation des cinq étoiles sont déphasés par rapport à l'étoile de base. Cet angle de déphasage est fixé par l'attribut **angleShift** de **Star**, qui a une valeur entre 0 et 360 degrés. Ensuite, on fait de même récursivement pour les 5 étoiles qu'on vient de placer (qui serviront alors de bases aux prochaines étoiles) etc. (voir modèle de démonstration).

Les étoiles qui sont placées sur les branches de l'étoile de base ont un rayon **2.5** fois plus petit que celui de l'étoile de base.
L'attribut **depth** de **Star** indique la profondeur de la récursion.

Exemples (pour angleShift = 0) :

pour *depth*=1:	pour *depth*=2:	pour *depth*=3:
		

Remarques :

- Tous les calculs sont effectués avec des nombres réels. Les valeurs sont arrondies uniquement lors du dessin à l'écran.
- Rappel : les méthodes trigonométriques de Java calculent avec des angles en radians.

Travail à réaliser : Complétez les méthodes dans la classe **Star**.

Dans la classe **Star**, développez la méthode privée ***getEndPoint(…)*** qui sert à déterminer le point final d'un segment d'une étoile, pour un point de départ, un angle et une longueur donnés. Cette méthode vous aidera à trouver facilement les coordonnées cartésiennes (point(x,y)) pour des coordonnées polaires (point, longueur, angle) données. Paramètres de ***getEndPoint(…)*** :

Point2D p	Les coordonnées du point de départ du segment (type double)
double length, double angle	La longueur du segment et son angle (en degrés)

Valeur retour de la méthode :

Point2D	Les coordonnées de la fin (du dernier point) du segment

[1,5p]

Définissez la méthode privée récursive ***drawFractal*** qui sait dessiner la fractale de l'étoile ! Servez-vous de la méthode ***getEndPoint(…)***!

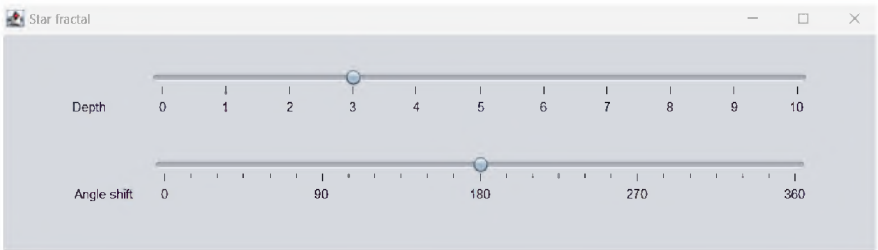
Paramètres de la méthode ***drawFractal*** :

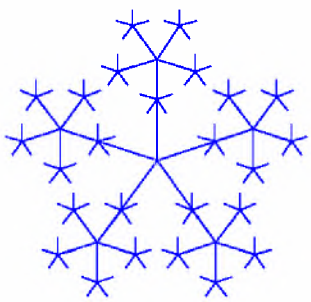
Graphics g	Le canevas cible
Point2D center	Les coordonnées du centre de la fractale
double length	Le rayon de l'étoile de base de la fractale à dessiner
double angle	L'angle d'inclinaison de l'étoile de base de la fractale à dessiner (en degrés)
int depth	La profondeur de la récursion

[4p]

Complétez la méthode **draw(Graphics g, int width, int height)** dans **StarFractal** pour lancer la représentation. Les cinq bras de l'étoile de base ont un rayon quatre fois plus petit que la longueur maximale possible sur le canevas. Le premier bras de l'étoile de base est parallèle à l'axe des y.

[1p]



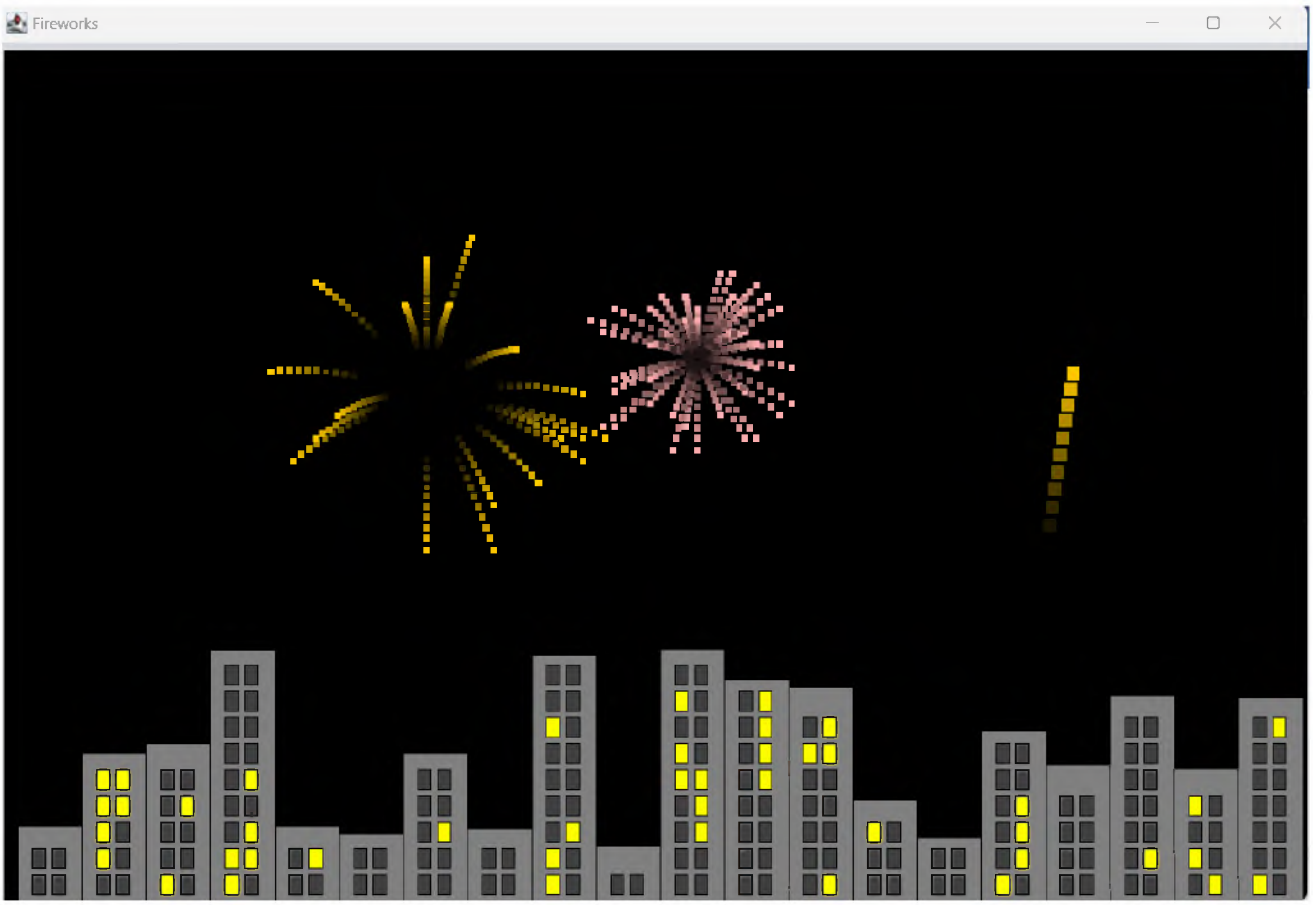


Question 2 : « Fireworks » [53,5p.]

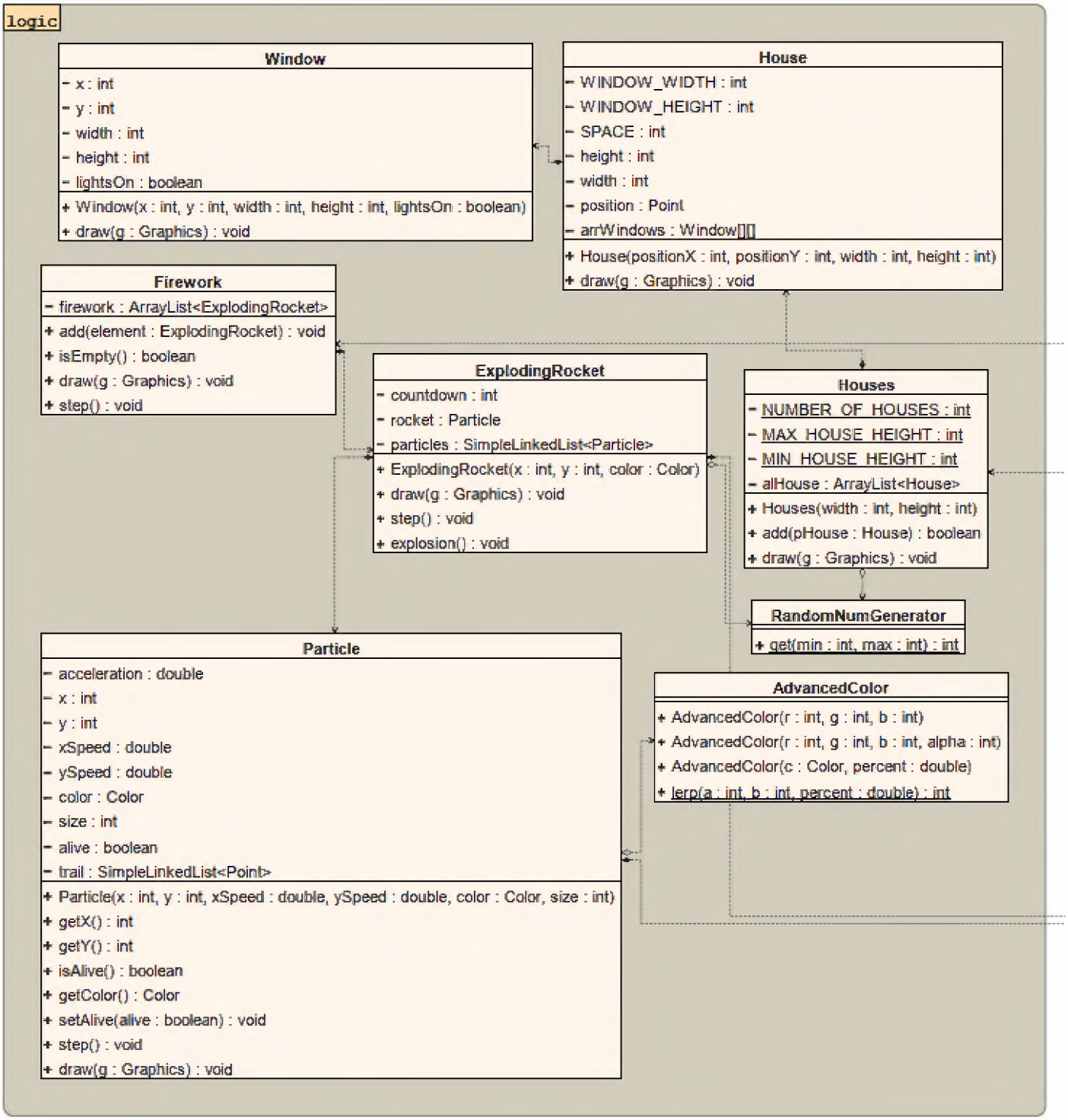
Le projet **Fireworks** est à compléter pour simuler des feux d'artifice autour d'une ville. L'utilisateur peut cliquer dans le ciel (à l'intérieur du panneau) pour générer une explosion de particules d'un feu d'artifice colorées à l'endroit cliqué.

Un modèle de démonstration du projet fini se trouve dans votre dossier sous **dist/Fireworks.jar**. Démarrez ce programme maintenant pour vous familiariser avec son fonctionnement !

Ouvrez le project « **Fireworks** » et faites tout d'abord un **CLEAN AND BUILD**. Les classes **AdvancedColor** et **DrawPanel** sont déjà entièrement définies, tandis que les autres classes doivent être complétées ou créées.



Sur la page suivante se trouve le modèle UML du paquet **logic.**


logic
Window
- x : int
- y : int
- width : int
- height : int
- lightsOn : boolean
+ Window(x : int, y : int, width : int, height : int, lightsOn : boolean)
+ draw(g : Graphics) : void
House
- WINDOW_WIDTH : int
- WINDOW_HEIGHT : int
- SPACE : int
- height : int
- width : int
- position : Point
- arrWindows : Window[][]
+ House(positionX : int, positionY : int, width : int, height : int)
+ draw(g : Graphics) : void
Firework
- firework : ArrayList<ExplodingRocket>
+ add(element : ExplodingRocket) : void
+ isEmpty() : boolean
+ draw(g : Graphics) : void
+ step() : void
ExplodingRocket
- countdown : int
- rocket : Particle
- particles : SimpleLinkedList<Particle>
+ ExplodingRocket(x : int, y : int, color : Color)
+ draw(g : Graphics) : void
+ step() : void
+ explosion() : void
Houses
- NUMBER_OF_HOUSES : int
- MAX_HOUSE_HEIGHT : int
- MIN_HOUSE_HEIGHT : int
- allHouse : ArrayList<House>
+ Houses(width : int, height : int)
+ add(pHouse : House) : boolean
+ draw(g : Graphics) : void
RandomNumGenerator
+ get(min : int, max : int) : int
Particle
- acceleration : double
- x : int
- y : int
- xSpeed : double
- ySpeed : double
- color : Color
- size : int
- alive : boolean
- trail : SimpleLinkedList<Point>
+ Particle(x : int, y : int, xSpeed : double, ySpeed : double, color : Color, size : int)
+ getX() : int
+ getY() : int
+ isAlive() : boolean
+ getColor() : Color
+ setAlive(alive : boolean) : void
+ step() : void
+ draw(g : Graphics) : void
AdvancedColor
+ AdvancedColor(r : int, g : int, b : int)
+ AdvancedColor(r : int, g : int, b : int, alpha : int)
+ AdvancedColor(c : Color, percent : double)
+ lerp(a : int, b : int, percent : double) : int

1ère partie : Silhouette de la ville *[13,5p.]*

La classe RandomNumGenerator *[1p.]*

Créez une classe **RandomNumGenerator** et développez une méthode statique ***get(int min, int max)*** pour générer des valeurs entières aléatoires dans l'intervalle [min,max].

La classe Window *[1,5p.]*

La classe **Window** représente une fenêtre d'une maison et possède les attributs suivants :

int x, y	Les coordonnées de la fenêtre (coin supérieur gauche)
int width, height	La largeur et la hauteur de la fenêtre
boolean lightsOn	Etat de la fenêtre. « true » signifie que la lumière est allumée, « false » signifie que la lumière est éteinte

Travail à réaliser :

Complétez dans la classe **Window** la méthode ***draw(Graphics g)*** qui dessine la fenêtre en respectant les indications suivantes :

- Les contours de la fenêtre sont dessinés en noir (**BLACK**).
- L'intérieur de la fenêtre est de couleur gris foncé (**DARK_GRAY**) si la lumière est éteint, sinon de couleur jaune (**YELLOW**). Utilisez l'attribut **state** pour déterminer l'état de la fenêtre !

La classe House *[8,5p.]*

La classe **House** représente une maison de la silhouette de la ville et possède les attributs suivants :

int height, width	La hauteur et largeur de la maison.
Point position	La position du coin inférieur gauche de la maison. La position est un point dont les coordonnées sont x et y.
int WINDOW_HEIGHT, WINDOW_WIDTH	La hauteur et la largeur de la fenêtre qui sont fixées à des valeurs constantes de 15 respectivement 10 points.
int SPACE	Espace vertical et horizontal entre les fenêtres qui est fixé à une valeur constante de 5 points. SPACE indique aussi l'espace minimum à l'intérieur de la maison entre les bords de la maison et les fenêtres (voir Figure 1, lignes vertes en pointillés).
boolean [ ][ ] arrWindows	Le tableau (EN : Array) contient toutes les fenêtres de la maison. Les fenêtres sont gérées dans une grille à deux dimensions. La position d'une fenêtre est définie par une ligne et une colonne.

Tenez également compte des informations supplémentaires indiquées sur la Figure 1.

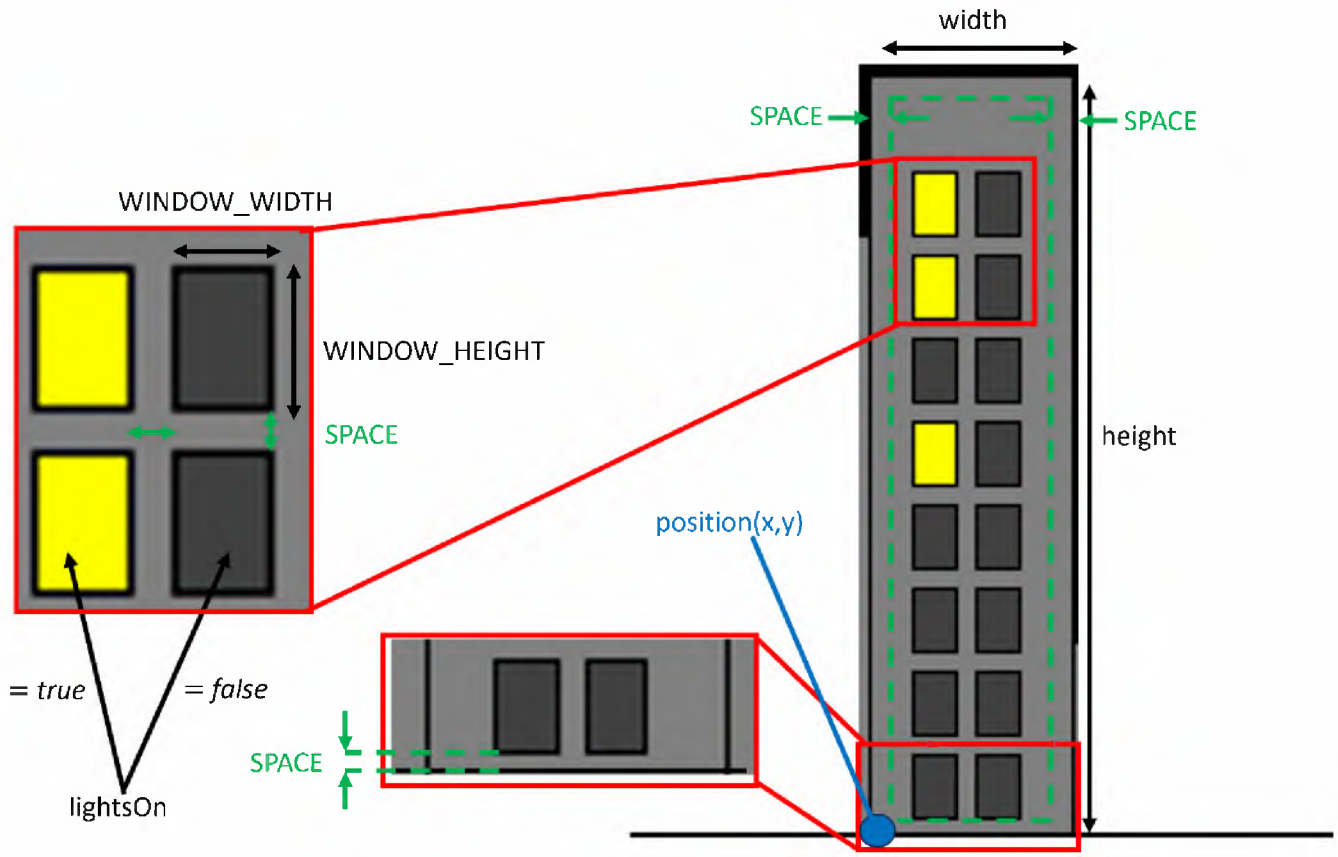


*Figure 1 Indications sur les attributs de la classe **House***

Travail à réaliser :

1. Complétez le constructeur de la classe qui
   - Initialise les attributs :
     i. **height**, **width**, **position** (rien à faire !)
     ii. **arrWindows** est initialisé en déterminant le nombre maximum de fenêtres en fonction des espaces requis, de la hauteur et de la largeur des fenêtres.
   - Crée autant de fenêtres que possible pour chaque maison :
     i. Veillez à ce que les espaces entre les fenêtres et l'espace à gauche et à droite du bâtiment soient pris en considération.
     ii. Placez les fenêtres en les centrant horizontalement, c'est-à-dire en les plaçant à égale distance des côtés gauche et droit du mur.
     iii. Une fenêtre a **25%** de chance d'être dans un état « allumé » (=*true*).

   [6,5p.]
2. Implémentez la méthode ***public void draw(Graphics g)*** qui dessine la maison entière avec toutes ses fênetres avec les indications suivantes :
   - Les contours de la maison sont dessinés en noir (**BLACK**).
   - L'intérieur de la maison est de couleur grise (**GRAY**).

   [2p.]

La classe Houses *[2,5p.]*

La classe **Houses** représente toutes les maisons de la silhouette de la ville et possède les attributs suivants :

int NUMBER_OF_HOUSES	Le nombre de maisons
int MAX_HOUSE_HEIGHT	La hauteur maximale d'une maison
int MIN_HOUSE_HEIGHT	La hauteur minimale d'une maison
ArrayList<House> alHouse	Une liste qui contient toutes les maisons

Travail à réaliser :

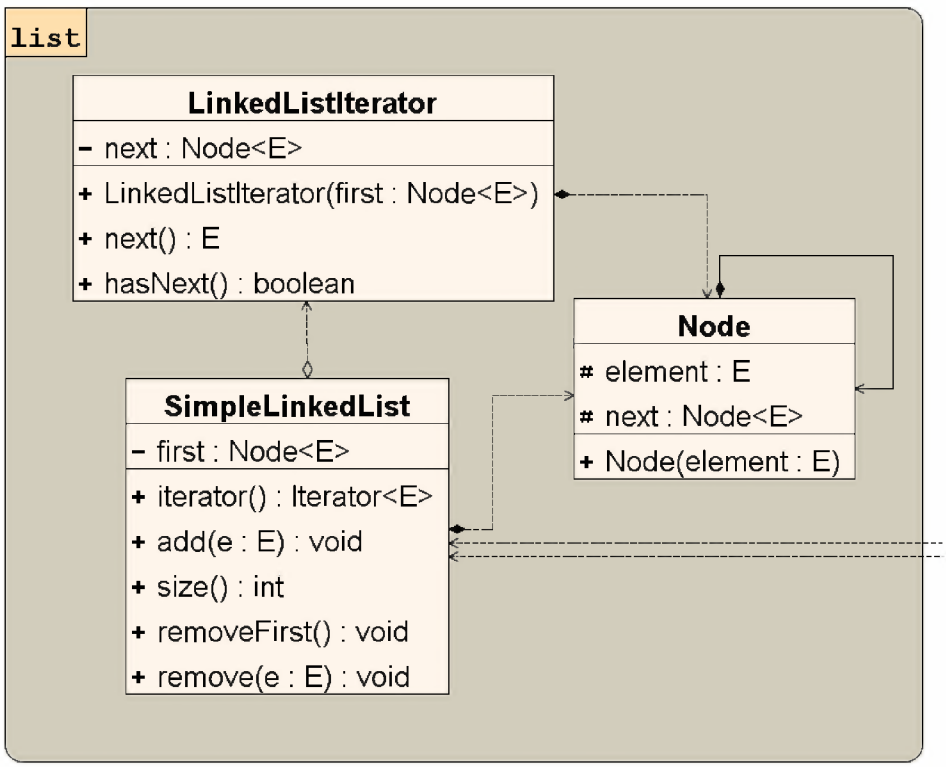
Complétez le constructeur de la classe **Houses** qui initialise la silhouette de la ville :

- Toutes les maisons ont la même largeur. Calculez la largeur des maisons en fonction du nombre de maisons défini par l'attribut **NUMBER_OF_HOUSES**, en veillant à ce qu'il y ait un espace de 10 pixels avant la première maison.
- Ensuite, créez de nouvelles maisons avec une hauteur aléatoire dans l'intervalle [**MIN_HOUSE_HEIGHT**, **MAX_HOUSE_HEIGHT**]. Les coordonnées doivent être définies de manière à ce que les maisons soient alignées les unes à côté des autres (consultez la Figure 1 Indications sur les attributs de la classe **House**).
- Vous pouvez maintenant tester la création et le dessin des maisons avec les fenêtres.

2ème partie : Le feu d'artifice *[30,5p.]*

Dans le paquet **list** développez la classe **SimpleLinkedList** (une liste chaînée) et la structure **Node** associée.

La structure à développer est **générique**, c'est-à-dire, les types des clés et valeurs sont choisis par le programmeur qui utilise la structure !



La classe SimpleLinkedList *[12,5p.]*

Travail à réaliser :

Développez la classe **Node** avec les attributs et le constructeur nécessaire. [1p.]

Développez la classe **SimpleLinkedList** (voir UML) en implémentant les méthodes suivantes :

1. Implémentez la méthode ***public void add(E e)*** qui ajoute un élément à la fin de la liste. [2p.]
2. Implémentez la méthode ***public int size()*** qui retourne le nombre d'éléments de la liste. [1,5p.]
3. Implémentez la méthode ***public void removeFirst()*** qui supprime le premier élément dans la liste. [1p.]
4. Implémentez la méthode ***public void remove(E e)*** qui supprime l' élément donné comme paramètre. [3p.]
5. Réalisez enfin un itérateur **LinkedListIterator** pour la classe **SimpleLinkedList**. Il n'est pas nécessaire de réaliser la méthode optionnelle **remove**. Apporter les modifications nécessaires à la classe **SimpleLinkedList** afin d'utiliser l'itérateur. [4p.]

Remarque : Si vous n'avez pas réussi à créer la classe **SimpleLinkedList**, utilisez plutôt une **ArrayList** dans ce qui suit !

La classe Particle *[7p.]*

La classe **Particle** représente une particule (un carré) d'un feu d'artifice et sert à stocker le chemin d'une particule sous forme de points (**Point**) dans une liste chaînée. Ces points constituent la traînée (EN ; trail, DE : Schweif), donc la trace laissée dans le ciel. La classe **Particle** possède les attributs suivants :

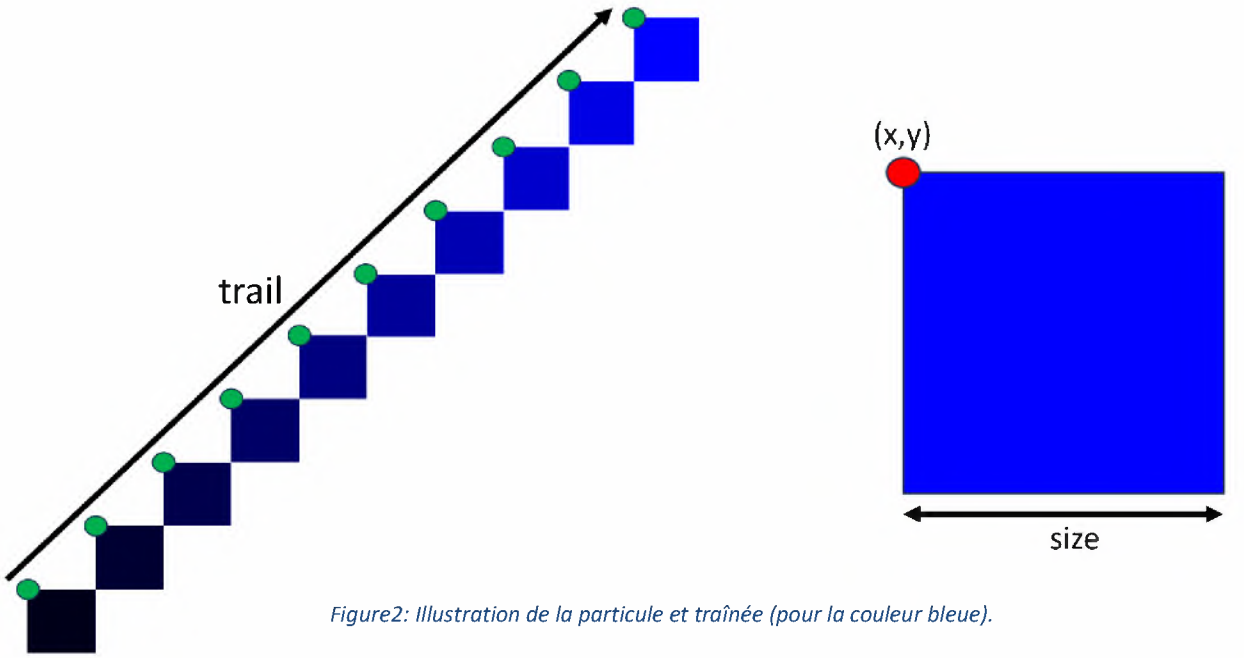


Figure2: Illustration de la particule et traînée (pour la couleur bleue).

int x,y	La position horizontale et verticale-de la particule
int xSpeed, ySpeed	La vitesse horizontale (valeur aléatoire dans l'intervalle [-2.0, 2.0]) et verticale (valeur aléatoire dans l'intervalle [-10.0, -20.0]) de la particule
double acceleration	L'accélération affectant le mouvement vertical de la particule avec une valeur de 0,25
int size	La taille de la particule
Color color	La couleur de la particule
boolean alive	Indique si la particule est vivante/active (*true*) ou non (*false*).
SimpleLinkedList<Point> trail	Le chemin de la particule stocké dans une liste d'objets **Point** (marqués en vert dans la Figure 2)

Les attributs (sauf la liste chaînée **trail**), le constructeur et l'accesseur ***isAlive()*** sont déjà implémentés.

Travail à réaliser :

1. Implémentez la méthode ***public void step()*** qui met à jour les coordonnées x et y de la particule en fonction de sa vitesse (**xSpeed** et **ySpeed**). L'accélération est ajoutée à la vitesse verticale **ySpeed**. A chaque appel de **step()**, un nouveau point (=les coordonnées **x** et **y** adaptées) de la particule sont ajoutées à la fin de la liste (**trail**). Après avoir ajouté 10 coordonnées de la traînée de la particule, le premier point est toujours supprimé de la liste est un nouveau point est ajouté à la fin, basé sur le principe FIFO (First In First Out). Enfin si la vitesse verticale (**ySpeed**) est supérieure à 10, la particule n'est plus vivante/active.

   [3p.]

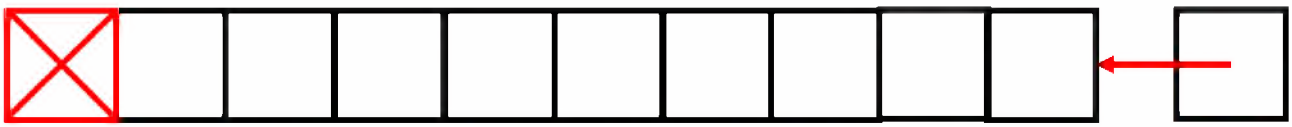


*Figure 3 Un exemple d'ajout et de suppression de points dans la liste chaînée (**trail**) s'il y en a plus de 10.*

2. Implémentez la méthode ***public void draw(Graphics g)*** qui dessine la particule sous forme d'un carré dans la couleur donnée comme attribut et une traînée (EN : trail) de la particule. Dans la liste chaînée **trail**, la première coordonnée (**first**) est la plus récente de la particule et est donc dessiné en couleur claire. Chaque trace/ point de la liste sera dessiné sous forme de carré dans une couleur légèrement plus foncée. Pour ce faire, créez un nouvel objet du type **AdvancedColor** qui prend comme paramètres la couleur de la particule, et un pourcentage (notation décimale) indiquant le rapport de mélange des couleurs. **AdvancedColor** calcule l'interpolation linéaire entre les couleurs avec le pourcentage donné et crée une nouvelle couleur sur cette base. Les points en début de liste seront foncés et la couleur s'éclaircit linéairement pour les points qui se trouvent à la fin de la liste (voir Figure 4, page suivante). Le dernier point aura alors la couleur de l'attribut **color**. Le pourcentage de **AdvancedColor** sera à ce moment 0%.

   [4p.]

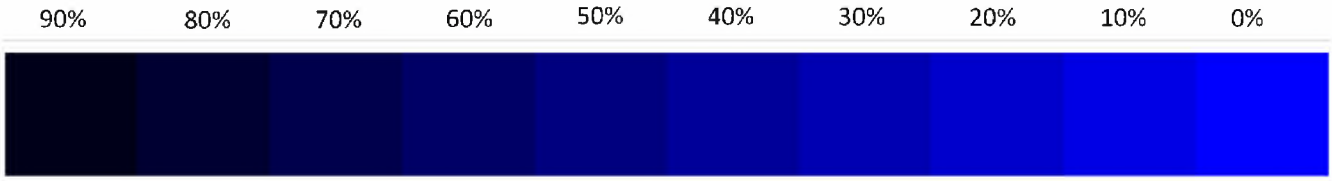


Figure 4 Nuançage du tracé pour l'exemple de la particule de couleur bleue.

La classe ExplodingRocket *[10p.]*

La classe **ExplodingRocket** dans le paquet **logic** représente une fusée (du type **Particle**) qui monte puis, après un certain temps, explose dans une multitude de particules (gérés dans une liste de **Particle**). Un tel dispositif possède les attributs suivants :

int countdown	Un compte à rebours pour l'explosion
Particle rocket	La fusée qui va exploser en une multitude de petites particules
SimpleLinkedList<Particle> particles	Une liste contenant toutes les particules après l'explosion (étincelles qui partent dans les différentes directions)

Travail à réaliser :

Complétez la classe **ExplodingRocket**

1. Implémentez le constructeur de la classe prenant comme paramètres les coordonnées de départ **x** et **y**, et la couleur (**color**) de la particule.
   - Initialisez la particule **rocket** en prenant :
     i. une valeur entière aléatoire dans l'intervalle [-2, 2] pour la vitesse x (**xSpeed**),
     ii. une valeur entière aléatoire dans l'intervalle [-10, -20] pour la vitesse y (**ySpeed**),
     iii. une taille par défaut de 10 (**size**)
   - Le compte à rebours (**countdown**) est initialisé par une valeur entière aléatoire dans l'intervalle [30, 60].
   - Initialise la liste **particles** par une liste vide.

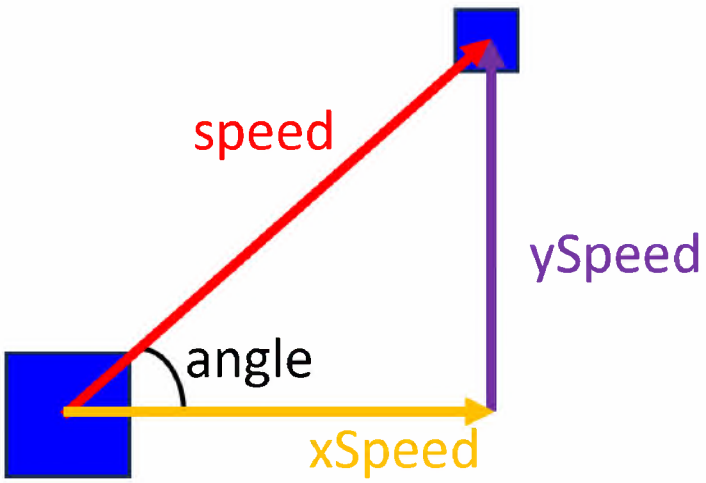
   [1,25p.]
2. Implémentez la méthode ***public void draw(Graphics g)*** qui dessine la fusée (**rocket**) si elle est vivante, ainsi que toutes les particules de la liste **particles** si la liste n'est pas vide et que les particules sont vivantes.

   [1,5p.]
3. Implémentez la méthode ***public void step()*** qui effectue les opérations suivantes :
   - si la fusée est vivante : appelle la méthode ***step()*** de la particule **rocket**, décrémente le compte à rebours de un et appelle la méthode ***explosion()*** si le compte à rebours atteint 0.
   - sinon elle appelle la méthode **step()** de chaque particule vivante dans la liste **particles**. Si la particule n'est plus vivante, elle est supprimée de la liste.

   [3p.]
4. Implémentez la méthode ***public void explosion()*** qui déclenche une explosion de la fusée du feu d'artifice (**rocket**). L'explosion crée une multitude de particules qui partent dans différentes directions. Le nombre de particules créées lors de l'explosion est un nombre aléatoire de l'intervalle [20, 50] .

- Chacune des particule créées (étincelles) a :
  i. une taille de 5 points
  ii. une vitesse aléatoire entière de l'intervalle [5, 10]
  iii. une trajectoire qui est déterminée par un angle aléatoires en radian de l'intervalle
  [0.0, 2π].
  iv. la même couleur, et la même position (x,y) que la fusée explosée.

  Indice : Calculer la vitesse **xSpeed** et la vitesse **ySpeed** en fonction de la vitesse et de l'angle aléatoire.



- Les particules sont créées et ajoutées à la liste **particles**.
- Après l'explosion, la particule **rocket** est marquée comme non vivante.

[4,25p.]

La classe Firework *[1p.]*

La classe **Firework** représente le feu d'artifice.

ArrayList<ExplodingRocket> firework	Une liste qui contient toutes les fusée du feu d'artifice.

Les méthodes ***add()*** et ***isEmpty()*** sont déjà implémentées.

Travail à réaliser :

1. Ajoutez une méthode ***draw(Graphics g)*** qui dessine toutes les fusée de la liste **firework**.
   [0,5p.]

2. Ajoutez une méthode ***step()*** qui permet d'avancer d'un pas toutes les fusées de la liste **firework**.
   [0,5p.]

3ème partie : L'interface graphique *[9,5p.]*

Dans le paquet **gui** se trouve la classe **MainFrame** qui représente la vue et le contrôleur du MVC (Model View Controller).

La classe MainFrame *[9,5p.]*

Travail à réaliser :

1. Complétez le constructeur de la **MainFrame.** Pour que toutes les fusées de la liste **firework** et ses particules se déplacent, la méthode **step()** doit être appelée toutes les **80 ms.**

   [1,75p.]
2. Ajoutez un événement **MousePressed** qui différencie entre un clic gauche et un clic droit :
   a. Pour un clic du bouton gauche (MouseEvent.BUTTON1), une nouvelle fusée explosive est créée en bas du **DrawPanel** à la position x de la souris. Sa couleur est choisie couleur au hasard de la liste **fireworkColors**. Ensuite, ajoutez la nouvelle fusée à la liste **firework**.
   b. Pour un clic du bouton droit (MouseEvent.BUTTON3) implémentez et lancez une simulation qui crée toutes les 250 millisecondes une nouvelle fusée comme pour le clic gauche, mais avec un départ aléatoire pour la coordonnée **x**. Si la simulation est en cours, un clic droit l'arrête, sinon une nouvelle simulation est lancée.

   [4,25p.]
3. Jusqu'à présent, toutes les fusées brûlées restent dans la liste **firework** sans être supprimées. Rendez votre code plus efficace en ajoutant la suppression automatique des fusées brûlées (**ExplodingRockets**) de la liste **firework**.

   [3,5p.]