

Travaux Dirigés

Langage Orienté Objet / Java

Partie 9 – Fight !

The Learning Souls Game

Ce TD doit être réalisé à partir du code créé dans la partie précédente.

Maintenant que tous les mécanismes sont en place, il nous reste à permettre au joueur de réellement jouer, c'est à dire d'utiliser les compétences de son héro (ex. lancer une attaque contre un monstre).

Le jeu se déroulera au tour par tour : à chaque tour le héro pourra déclencher une (et une seule) action, puis ce sera au tour du (des) monstre(s), puis au tour du héro, etc.

Les actions du héro seront rendues disponibles au travers d'une barre de compétences implémentée dans la classe **SkillBar**. Cette barre sera utilisable lorsqu'il s'agira du tour du héro (jeu en attente de l'action du joueur), et désactivée (**disabled**) pendant le tour du monstre. Elle regroupera les diverses actions possibles du héro (instances de **SkillAction**), qui pourront être déclenchées au travers de boutons d'action implémentés sous la forme d'instances de **SkillTrigger**.

1. SkillAction

Nous allons tout d'abord définir la notion d'action pouvant être exécutée au travers d'une **SkillBar**.

1.1. Définissez l'interface **lsg.graphics.widgets.skills.SkillAction**

- elle obligera ses instances à fournir une implémentation de la méthode **public void execute()** ;
NB : cette méthode sera plus tard utilisée comme une sorte de *handler* sans paramètre

2. SkillTrigger

Un **SkillTrigger** (qui viendra peupler la **SkillBar**) est un composant associant une image, un code clavier, un texte et une **SkillAction**. La **SkillAction** associée sera exécutée lors du déclenchement du trigger.

2.1. Créez la classe **lsg.graphics.widgets.skills.SkillTrigger** :

- Sous-classe de **javafx.scene.layout.AnchorPane**
- Attributs :
 - **private ImageView view** : image qui sera affichée
 - **private Label text** : texte qui sera placé au dessus de l'image (correspondant à la représentation d'un raccourcis clavier)
 - **private KeyCode keyCode** : le code du raccourcis clavier
 - **private SkillAction action** : l'action exécutée lors du déclenchement du trigger
- Les *getters* et *setters* pour tous les attributs sauf pour **view**
- Les *getter* et *setter* pour l'image de **view** :
 - **public Image getImage()**
 - **public void setImage(Image image)**
- Un constructeur public :
 - **SkillTrigger(KeyCode keyCode, String text, Image image, SkillAction action)**
- Une méthode **private void buildUI()** <- à appeler dans le constructeur
 - (NB : Intégrez la feuille de style *SkillTrigger.css* au projet)
 - Applique la classe css « **skill** » au trigger
 - Fixe la **fitWidth** et **fitHeight** de **view** à 50 pixels
 - Ajoute **view** au trigger et lui fait prendre tout l'espace
 - Ajouter **text** au trigger et lui fait prendre tout l'espace (donc par dessus **view**)
- Une méthode **public void trigger()** qui :
 - ne fait rien si le composant est **disabled**
 - exécute **action** si elle n'est pas nulle
- Une méthode **private void addListeners()** <- à appeler dans le constructeur
 - met un *listener* sur le **mouseClicked** du trigger et appelle **trigger()** quand l'événement est déclenché

.../...

3. SkillBar

Une **SkillBar** est un regroupement d'instances de **SkillTrigger**.

3.1. Créez la classe **lsg.graphics.widgets.skills.SkillBar** :

- Sous-classe de **javafx.scene.layout.HBox**
- Définir un *binding* des touches par défaut à grâce au code :

```
private static LinkedHashMap<KeyCode, String> DEFAULT_BINDING = new LinkedHashMap<>() ;
static {
    DEFAULT_BINDING.put(KeyCode.DIGIT1, "&" );
    DEFAULT_BINDING.put(KeyCode.DIGIT2, "é" );
    DEFAULT_BINDING.put(KeyCode.DIGIT3, "\"" );
    DEFAULT_BINDING.put(KeyCode.DIGIT4, "'" );
    DEFAULT_BINDING.put(KeyCode.DIGIT5, "(" );
}
```

NB :

- On utilise une **LinkedHashMap** du fait qu'elle garantisse un parcours ordonné de la liste (suivant l'ordre d'ajout des éléments)
- La clause **static** fait s'exécuter le code contenu dans le bloc lors de la création de la classe dans la JVM (donc avant la création d'instance de **SkillBar**)
- Attribut **private SkillTrigger triggers[]** (les triggers de la barre)
- Constructeur **public SkillBar()** qui :
 - Fixe à 10 pixels l'espacement des composants
 - Fixe la **prefHeight** de la barre à 110px
- Méthode **private void init()** qui :
 - instancier triggers avec la taille de **DEFAULT_BINDING** (un trigger par *binding*)
 - remplir triggers avec des instances de **SkillTrigger** : utiliser le code et le texte de chaque élément de **DEFAULT_BINDING** ; laisser l'image et l'action à **null**.
 - ajouter chaque *trigger* instancié aux *children* de la barre
- Méthode **public SkillTrigger getTrigger(int slot)**
Permet de récupérer un trigger à partir de sa place dans la barre
- Méthode **public void process(KeyCode code)** qui :
 - Ne fait rien si la barre est désactivée (**disabled**)
 - Déclenche le *trigger* qui est lié au **code** passé en paramètre

4. Ajout de la barre de compétences du héros

Nous allons faire apparaître la barre de compétences en l'ajoutant à **hudPane**.

4.1. Dans **HUDPane** :

- Créez un attribut privé **skillBar** de type **SkillBar**
- Implémentez un *getter* pour cet attribut
- Ajoutez la méthode **private void buildBottom()** <- appelée dans le constructeur
 - instancier **skillBar**
 - l'ajouter au *bottom*

4.2. Tester et peaufiner le constructeur de **SkillBar** pour que l'affichage corresponde à :



5. Remplissage de la barre

5.1. Dans LearningSoulsGameApplication

- Ajoutez un attribut privé **skillBar** de type **SkillBar**
- Créez la méthode **private void createSkills()** qui :
 - initialise **skillBar** en récupérant celle de **hudPane**
 - place l'image **ImageFactory.SPRITES_ID.ATTACK_SKILL** dans le 1er trigger
 - ajoute un **handler** au 1er **trigger** : pour l'instant on affichera simplement « ATTACK » dans la console
 - ajoute un **setOnKeyReleased** sur la scène pour re-router le code de l'évènement vers **skillBar** (faire appel à **process**)
- Appelez **createSkills** dans **play** juste après l'appel à **createHero**

5.2. Testez.

- Le visuel doit correspondre à :



- Un click ou l'appui sur la touche « & » doit déclencher l'affichage de « ATTACK » dans la console

6. Animer les *triggers*

Pour fournir un premier retour visuel aux actions du joueur, nous allons animer les triggers lors de leur déclenchement.

6.1. Dans **SkillTrigger**

- Créez la méthode **private void animate()** qui déclenche une animation de *zoomIn* (100ms, **scale** 1.3) et retour à la normale (utiliser **setAutoReverse**)
- Appelez **animate()** dans **trigger()** juste avant l'appel à **execute()** sur **action**

6.2. Testez

- Le trigger doit s'animer lorsqu'il est déclenché

7. Chacun son tour

Ca y est : le joueur a une barre de compétences avec laquelle il peut interagir (même si elle ne fait pas grand chose d'effectif pour l'instant...).

Cependant on peut noter que la barre toujours active. Par exemple, il est possible que le joueur déclenche les triggers (et donc les actions liées aux compétences) pendant l'arrivée des personnages, ce qui n'est pas notre but.

Pour pallier ce problème, nous allons désactiver la barre de compétences pendant les moments où le joueur n'est pas censé jouer.

7.1. Activation / désactivation de **skillBar**

Dans **LearningSoulsGameApplication** :

- Créez un attribut **heroCanPlay** : on va le définir en tant que propriété (booléenne) pour pouvoir enregistrer un écouteur sur son état

```
private BooleanProperty heroCanPlay = new SimpleBooleanProperty(false) ;
```
- Modifiez **createSkills** pour :
 - initialiser la propriété **disable** de **skillBar** en fonction **heroCanPlay**
 - ajouter un *listener* sur **heroCanPlay** pour en répercuter les changements d'états sur la propriété **disable** de **skillBar**
- Dans **play()**, remplacez le *handler* de **createMonster** (où nous avons mis un test d'attaque) pour indiquer que le héros peut jouer (**heroCanPlay** passe à **true**) dès que le monstre est arrivé à sa place

7.2. Testez.

- La barre de compétences ne doit pas répondre (par click ou touche « & ») pendant l'entrée en scène des personnages, puis devenir active une fois le monstre en place.

7.3. Un meilleur rendu pour l'état désactivé

Le retour visuel de la désactivation de la barre n'est pas assez significatif. Nous allons donc rendre cet état plus visible en dé-saturant l'image des triggers lorsque la barre est désactivée.

Pour ce faire, nous allons profiter du fait qu'en JavaFX, l'état *désactivé* d'un nœud est automatiquement transmis à ses enfants. Ainsi, lorsqu'on désactive une **SkillBar**, l'état *disable* est automatiquement transmis (changement d'état de leur **disabledProperty**) aux instances de **SkillTrigger** affichés dans la barre.

Dans **SkillTrigger** :

- Créez un attribut **private ColorAdjust desaturate** : correspond l'effet qui sera appliqué à **view** lors d'une désactivation
- Dans le constructeur
 - instancier **desaturate**
 - fixer sa *saturation* à -1
 - fixer son *brightness* à 0.6
- Dans **addListeners**, ajouter un *listener* sur la **disabledProperty** du trigger qui :
 - ajoute l'effet **desaturate** quand **disabledProperty** vaut **true**
 - retire cet effet quand **disabledProperty** vaut **false**

7.4. Testez.

- L'image doit maintenant apparaître dé-saturée quand **skillBar** est *disabled*



8. LearningSoulsGameApplication : à l'attaque !

Pour toutes les actions correspondantes aux compétences que l'on va implémenter, le déroulement sera à peu près le même :

- Désactiver la barre de compétences du héros (pour éviter d'autres actions)
- Exécuter l'action choisie.
- Si le monstre est encore vivant : le faire attaquer ; Si le monstre est mort, faire entrer un nouveau monstre, et le faire attaquer.
- Réactiver la barre de compétences (pour remettre le jeu en attente de la prochaine action du héros)

La première compétence que nous allons implémenter est l'attaque du héros. Cependant, le déroulement d'une attaque étant similaire quel que soit le personnage (héros ou monstre), les attaques seront gérées par une méthode commune paramétrée nommée **characterAttack**.

8.1. Créez la méthode **characterAttack**

Signature:

```
/**
 * Methode qui gère l'attaque et le coup porté par un agresseur sur sa cible,
 * aussi bien du point de vue du modèle (Character),
 * que du point de vue de l'animation (CharacterRenderer).
 * @param agressor : le modèle de l'attaquant
 * @param agressorR : la représentation de l'attaquant (pour animation attack)
 * @param target : le modèle de la cible
 * @param targetR : la représentation de la cible (pour animation hurt ou die)
 * @param finishedHandler : appelé lorsque les calculs et animations sont terminés
 */
private void characterAttack(Character agressor, CharacterRenderer agressorR,
                             Character target, CharacterRenderer targetR,
                             EventHandler<ActionEvent> finishedHandler){...}
```

Déroulement :

- Calcul de l'attaque de l'agresseur (**attack** de **Character**)
- Animation **attack** de l'agresseur (**CharacterRenderer**)
- Lorsque l'animation d'attaque est terminée :
 - Calcul du coup sur la cible (**getHitWith** de **Character**)
 - Si la cible est encore vivante : animation **hurt** de **CharacterRenderer**
 - Si la cible est morte : animation **die** de **CharacterRenderer**
 - Lorsque l'animation est terminée, déclenchement du **finishedHandler**
- En cas de levée d'exception lors de l'attaque :
 - Afficher un message adéquat dans **hudPane.messagePane**
 - Lancer (quand même) l'animation d'attaque (même s'il n'y a aucun dégât, pour simuler un coup dans le vide)
 - Lorsque l'animation est terminée, déclencher **finishedHandler**

8.2. Créez la méthode **private void heroAttack()** qui :

- Fait passer **heroCanplay** à *false* (ce qui désactivera **skillBar** du fait du *listener* installé précédemment)
- Appelle **characterAttack** avec **hero** comme agresseur, et **zombie** comme cible. Dans un premier temps, on affichera juste un message dans la console lorsque **characterAttack** est terminée (c'est à dire dans son *handler*)

8.3. Dans **createSkills**, appelez **heroAttack** dans le *handler* lié au *trigger*

- Testez : vous devez pouvoir lancer une attaque (avec séquence d'animations) et voir ses effets dans les barres d'état

8.4. Créez la méthode **private void monsterAttack()** qui :

- Appelle **characterAttack** avec **zombie** en agresseur et **hero** en cible
- Lorsque **characterAttack** est terminée :
 - si le héros est vivant : faire (re)passer **heroCanPlay** à *true* (pour réactiver **skillBar** et attendre la prochaine action du joueur)
 - si le héros est mort, appeler une méthode nommée **gameOver()** qui ne fera pour l'instant qu'afficher le message « GAME OVER » dans **hudPane.messagePane**.

8.5. Créez la méthode **private void finishTurn()** qui :

- Si le monstre est vivant : le fait attaquer-> appelle **monsterAttack**
- Si le monstre est mort :
 - retire le **zombieRender** (mort) des *children* d'**animationPane**
 - crée un nouveau monstre (appelle **createMonster**)
 - lorsqu'il est en place, le fait attaquer -> appelle **monsterAttack**

8.6. Dans **heroAttack** :

- Appelez **finishTurn** lorsque l'attaque du héros est terminée (donc dans le *handler* de **characterAttack**)
- Testez : vous devez pouvoir enchaîner les coups et les combats !

9. Joli score !

Notre héros peut tuer des monstres mais nous n'avons pas encore mis en place la gestion de son score. Nous allons donc ajouter cette information au **HUDPane**, puis lier cet affichage à un calcul de score géré au niveau de **LearningSoulsGameApplication**.

9.1. Dans **HUDPane**

- Ajoutez l'attribut **score** tel que :

```
private IntegerProperty score = new SimpleIntegerProperty();
```
- Créez le getter **public IntegerProperty scoreProperty()** pour cet attribut
- Ajoutez l'attribut **scoreLabel** de type **GameLabel**
- Dans **buildTop()**, instanciez et ajoutez **scoreLabel** au centre avec :
 - texte de départ : « 0 »
 - un *scaling* de 1.3
 - une translation sur l'axe Y de 40px



- À la fin du constructeur, ajoutez un *listener* sur **score** pour faire en sorte que les modifications de score soient répercutées sur le texte affiché dans **scoreLabel**.

9.2. Dans `LearningSoulsGameApplication`

- Ajoutez l'attribut **score** tel que :

```
private IntegerProperty score = new SimpleIntegerProperty();
```

(oui, il ressemble fort à celui de `HUDPane`...)

- À la fin de `play()`, liez la `scoreProperty` de `hudPane` à `score` :

```
hudPane.scoreProperty().bind(score);
```

NB : ainsi, un changement de **score** sera répercuté sur la propriété **score** de **HUDPane** qui, grâce au *listener* installé dans le point précédent, mettra automatiquement à jour **scoreLabel**.

- Dans `finishTurn()`, si le monstre est mort, incrémentez `score`
- Testez : à la mort de chaque monstre, le score doit augmenter :



10. Récupération

Du fait du manque de stamina, il est difficile de faire un bon score. Nous allons donc fournir à notre héros une nouvelle compétence lui permettant de ne rien faire pendant 1 tour, si ce n'est récupérer un peu de force.

10.1. Dans `lsg.characters.Hero`

- Créez un attribut **private int stamRegen** qui représentera la capacité de régénération en stamina du héros lorsqu'il se repose, et initialisez-la par défaut à 10.
- Créez les getter et setter correspondants
- Créez la méthode **public void recuperate()** qui régénère la stamina du héros à hauteur de **stamRegen**, sans bien entendu dépasser **maxStamina**

10.2. Dans **LearningSoulsGameApplication**

- Créez la méthode **private void heroRecuperate()** qui :
 - désactive la barre de compétences (**heroCanPlay**)
 - fait récupérer le héros
 - termine le tour (**finishTurn**)
- Complétez **createSkills** en ajoutant cette nouvelle compétence.
Utilisez l'image correspondant à **ImageFactory.SPRITES_ID.RECUPERATE_SKILL**
- Testez :
 - vous devez avoir une nouvelle compétence de récupération dans la barre
 - la barre de stamina doit remonter juste avant l'attaque du monstre



- 10.3. Si vous le souhaitez, vous pouvez modifier **recuperate()** pour aussi rajouter un peu de vie au héros pendant la récupération 😊.

.../...

11. Ajout du consommable

La prochaine étape est de permettre à notre héros d'utiliser le consommable qu'il a équipé. Nous allons donc étendre **SkillBar** en y ajoutant un trigger spécial et dédié, instance de la classe **ConsumableTrigger** sous classe de **SkillTrigger** : dans les faits, un **ConsumableTrigger** sera un **SkillTrigger** lié à un **Consumable**.

De manière à faciliter l'affichage d'une image adaptée à un type de consommable, nous allons créer un *helper* nommé **CollectibleFactory**, au fonctionnement similaire à **ImageFactory**, et qui servira à récupérer le visuel adapté à chaque collectible. (Pour mémoire : **Consumable** est une sous-classe de **Collectible**).

Enfin, pour faciliter nos tests, nous allons commencer par créer 2 nouveaux types de consommables un peu plus adaptés à notre RPG.

11.1. Créez les 2 nouvelles classes de consommable :

- **lsg.consumables.drinks.SmallStamPotion**
 - sous-classe de **Drink**
 - qui a pour nom « Small stamina potion »
 - qui a pour capacité 20
- **lsg.consumables.food.SuperBerry**
 - sous-classe de **Food**
 - qui a pour nom « Super berry »
 - qui a pour capacité 1000

11.2. CollectibleFactory

Comme indiqué plus haut, **CollectibleFactory** est un *helper* qui servira à récupérer une image associée à un type de collectible, et donc de consommable.

- Ajoutez le dossier « **Basic RPG Item Free** » qui vous a été fourni aux images de votre projet
- Editez **lsg.graphics.ImageFactory** : ajouter 2 chemins (dans **SPRITES_ID**) :

SUPER_BERRY	("images/Basic RPG Item Free/Berry_03.png"),
SMALL_STAM_POTION	("images/Basic RPG Item Free/Small Potion_01.png"),

- Créez la classe **lsg.graphics.CollectibleFactory** avec la méthode :

```
/**
 * Méthode qui renvoie l'image associée à un Collectible
 * @param collectible : le collectible pour lequel l'image est demandée
 * @return : l'image représentant collectible
 */
public static Image getImageFor(Collectible collectible){...}
```

- Implémentez la méthode en ne prenant (pour l'instant) en compte que les **Collectible** du type **SmallStamPotion** et **SuperBerry**.
(Retourner **null** si c'est un autre type)
(Utilisez **ImageFactory**)

11.3. ConsumableTrigger

ConsumableTrigger permettra d'instancier un trigger lié à un consommable. Du fait de son type particulier, nous allons modifier quelque peu son visuel (par rapport aux autres triggers) grâce à une feuille de style supplémentaire.

- Créez le fichier **ConsumableTrigger.css** dans le dossier **lsg.graphics.css**

Définissez-y la classe css `.consumable` avec :

- épaisseur des bords : 4 pixels
- couleur des bords : blanc

- Créez la classe **lsg.graphics.widgets.skills.ConsumableTrigger** :

- sous-classe de **lsg.graphics.widgets.skills.SkillTrigger**
- avec l'attribut privé **consumable** de type **Consumable**
- avec le *setter* :

```
/**
 * Permet d'associer un consommable au trigger
 * @param consumable
 */
public void setConsumable(Consumable consumable) {...}
```

→ Ne pas oublier de mettre la bonne image grâce à **CollectibleFactory**

- avec le constructeur :

```
public ConsumableTrigger(KeyCode keyCode, String text, Consumable consumable, SkillAction action) {...}
```

→ Chargez la feuille de style et appliquez la classe css « `consumable` »

→ Faites appel à **super** et utilisez **setConsumable**

11.4. SkillBar étendue

Il n'y aura qu'un seul **ConsumableTrigger** dans une **SkillBar**, et celui-ci sera géré un peu à part des autres *triggers*. Pour ces raisons, nous n'allons pas l'ajouter à l'attribut **triggers** (le tableau) de **SkillBar**, mais lui fournir un attribut dédié nommé **consumableTrigger**.

Dans **SkillBar** :

- Ajoutez l'attribut :

```
private ConsumableTrigger consumableTrigger = new ConsumableTrigger(KeyCode.C, "c", null, null) ;
```

→ la touche de raccourcis sera donc le 'c'

- Créez un *getter* pour cet attribut
- Modifiez la méthode **process** pour qu'elle prenne en compte **consumableTrigger**
- Modifiez la méthode **init** pour ajouter **consumableTrigger** aux children après les autres **SkillTrigger**

NB : vous pouvez aussi ajouter un **javafx.scene.shape.Rectangle** de 30 pixels de large avant **consumableTrigger** de manière à insérer un espace entre ce trigger et les autres...

11.5. Dans `LearningSoulsGameApplication`

- Modifiez **createHero** (juste après l'appel à **setWeapon**) en plaçant une instance de **SuperBerry** dans le consommable du héro (**setConsumable**).
- Créez la méthode **heroConsume** (sur le modèle de **heroRecuperate**)
Affichez les messages adéquats dans **hudPane.messagePane** en cas d'exception
- Modifiez **createSkills** :
 - liez le consommable du héro au trigger dédié :

```
skillBar.getConsumableTrigger().setConsumable(hero.getConsumable());
```
 - complétez l'action du *trigger* en faisant appel à **heroConsume**
- Testez. Le nouveau visuel :



12. Consommable vide

Il n'y a pour l'instant aucun retour visuel permettant de savoir si le consommable est encore utilisable ou vide. Nous allons donc faire en sorte que lorsque la capacité du consommable est nulle, le trigger reste désactivé.

12.1. Dans `lsg.consumables.Consumable`

- Ajoutez une propriété qui permettra de suivre l'état de la capacité du consommable :

```
private SimpleBooleanProperty isEmpty ;
```
- Créez le *getter* **public SimpleBooleanProperty isEmptyProperty()**
- Instanciez **isEmpty** dans le constructeur, sa valeur étant fonction de **capacity**
- Modifiez **setCapacity** pour mettre à jour **isEmpty** à chaque changement de valeur de la capacité
- **ATTENTION :**
 Nous avons fait une erreur dans la méthode **use** en affectant directement la valeur 0 à **capacity** (**capacity = 0**) au lieu d'utiliser la méthode dédiée **setCapacity** (avec **setCapacity(0)**)
 Corrigez cette erreur pour que **setCapacity** soit bien appelée par **use**, et en conséquence, pour que **isEmpty** soit bien tenue à jour lors d'un appel à **use**.

12.2. Dans `lsg.graphics.widgets.skills.ConsumableTrigger`

- Étendez `setConsumable` en ajoutant un *listener* à la propriété `isEmpty` du consommable passé en paramètre afin de désactiver/réactiver le *trigger* (`setDisable`) en adéquation avec l'état de la capacité du consommable.
- Testez.

Le consommable doit rester désactivé (image dé-saturée) après son utilisation :



13. À vous de jouer

Il est maintenant possible d'ajouter l'affichage (à la demande) de l'armure dans la partie gauche de **HUDPane**, l'affichage du sac dans sa partie droite, d'ajouter un générateur de *loot* pour que chaque mort de monstre permette de ramasser un nouveau collectible à équiper, d'inventer de nouvelles compétences, et un peu plus peupler la **SkillBar**...

Nous n'aurons pas le temps d'implémenter ces autres fonctionnalités dans le cadre de ce TP. Néanmoins, l'ensemble des notions que nous avons étudiées et mises en œuvre vous ont fourni les bases qui vous permettront de développer vos propres fonctionnalités, que ce soit dans *Learning Souls Game*, ou dans les autres projets auxquels vous participerez : ces mécanismes (encapsulation, héritage, polymorphisme, interfaces, événements, écouteurs, observables, etc.) sont au cœur de la très grande majorité des logiciels produits aujourd'hui.

- 13.1. Inspirez-vous de ce que nous avons fait, créez de nouvelles fonctionnalités ou de nouveaux programmes, amusez vous, et développez plus avant vos propres *skills* dans cette belle approche qu'est la Programmation Orientée Objet.