

Preuve de connaissance

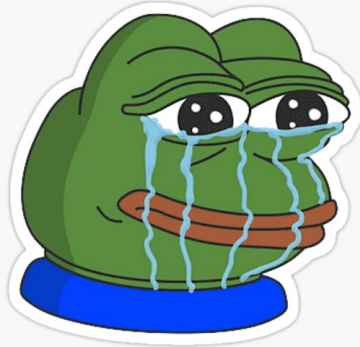
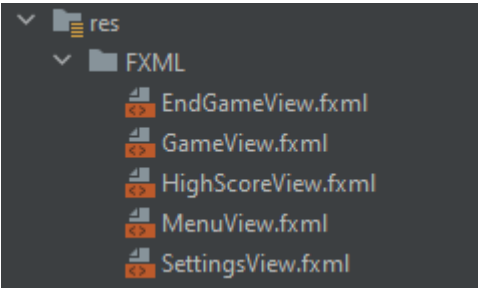
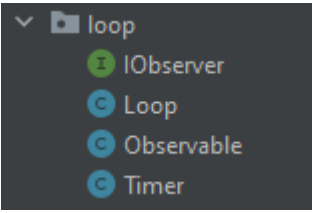
Connaissances /compétences	Exemple de code	Explications
Je maîtrise les règles de nommage Java.	 <pre data-bbox="443 763 1050 1115">package launch; import ... public class Launcher extends Application { private static Stage stage; private static ViewManager viewManager; private static PersistenceManager persistenceManager; public void start(Stage stage) throws Exception { Launcher.stage = stage; viewManager = new ViewManager(title: "TheGameShip", height: 720, width: 1280, persistenceManager = new PersistenceManager(); stage.show(); } public static ViewManager getViewManager() { return viewManager; } }</pre>	Les noms de package et variable sont en minuscule et les noms de classe commencent par une majuscule.
Je sais binder bidirectionnellement deux propriétés JavaFX.	<pre data-bbox="443 1151 1050 1435">@FXML private Slider difficultySlider; @FXML private Slider volumeSlider; difficultySlider.valueProperty().bindBidirectional(settings.difficultyProperty()); volumeSlider.valueProperty().bindBidirectional(settings.volumeProperty());</pre>	Les paramètres du jeu sont bind bidirectionnelle, c'est-à-dire que si l'on change la vue ça se répercute sur l'objet et si on modifie l'objet, cela se répercute sur la vue.
Je sais binder unidirectionnellement deux propriétés JavaFX.	<pre data-bbox="443 1487 1050 1720">Rectangle r = new Rectangle(); r.setFill(color); r.heightProperty().bind(h.heightProperty()); r.widthProperty().bind(h.widthProperty()); r.xProperty().bind(h.xProperty()); r.yProperty().bind(h.yProperty()); r.visibleProperty().bind(e.getVisibleBooleanProperty()); pane.getChildren().add(r);</pre>	Lorsqu'une des propriétés de l'objet joueur change, cela se répercute sur la vue est uniquement dans ce sens

<p>Je sais coder une classe Java en respectant des contraintes de qualité de lecture de code</p>	<pre>public class Location extends Component { private final DoubleProperty x = new SimpleDoubleProperty(); public double getX() { return x.get(); } public void setX(double x) { this.x.set(x); } public DoubleProperty xProperty() { return x; } private final DoubleProperty y = new SimpleDoubleProperty(); public double getY() { return y.get(); } public void setY(double y) { this.y.set(y); } public DoubleProperty yProperty() { return y; } private final DoubleProperty width = new SimpleDoubleProperty(); public double getWidth() { return width.get(); } public void setWidth(double width) { this.width.set(width); } public DoubleProperty widthProperty() { return width; } }</pre>	<p>Comme on peut le voir le code est lisible, les noms des classes sont clairs.</p> <p>Si ce n'est pas ça je ne vois pas à quoi ça fait référence.</p>
<p>Je sais contraindre les éléments de ma vue, avec du binding FXML.</p>	<pre>Rectangle r = new Rectangle(); r.setFill(color); r.heightProperty().bind(h.heightProperty()); r.widthProperty().bind(h.widthProperty()); r.xProperty().bind(h.xProperty()); r.yProperty().bind(h.yProperty()); r.visibleProperty().bind(e.getVisibleBooleanProperty()); pane.getChildren().add(r);</pre>	<p>Comme on peut le voir la propriété est belle est bien binder sur un élément du modèle.</p>
<p>Je sais définir une CellFactory fabriquant des cellules qui se mettent à jour au changement du modèle.</p>		<p>Il existe de nombreuses classes dans lesquelles on a délégué la création d'entités, de settings, highscore.</p>
<p>Je sais éviter la duplication de code.</p>		<p>Comme on peut le voir (il faut un peu plisser les yeux ^^), chaque "composant" d'une entité est séparé dans une classe et ensuite l'entityFactory compose les entités avec les différents composants</p>
<p>Je sais hiérarchiser mes classes pour spécialiser leur comportement.</p>		<p>L'interface IMove permet de généraliser l'accès aux méthodes de déplacement. MoveEnemy et Move "spécialise" son comportement.</p> <p>IMove move = new Move();</p>

Je sais intercepter des événements en provenance de la fenêtre JavaFX.	<pre>public void menu(ActionEvent actionEvent) { Launcher.getViewManager().setView("Menu"); PersistenceManager.saveSettings(settings); }</pre>	On récupère l'action d'appui sur un bouton et ensuite on agit en conséquence.
Je sais maintenir, dans un projet, une responsabilité unique pour chacune de mes classes.	<pre>//Et si ce n'est pas en collision, sa déplace l'entité ColliderInfo ci = c.isCollision(nextx, nexty, l.getHeight(), l.getWidth(), id); if (!ci.isCollision()) { l.setX(nextx); l.setY(nexty); } return ci;</pre>	Dans Move, la collision est gérée par le collisionneur et non par Move directement. Les responsabilités sont donc bien réparties.
Je sais gérer la persistance de mon modèle.		On a des classes Serializable qui permettent de rendre l'objet serializable (enregistrable dans un fichier XML par exemple). Puis, il est géré par le PersistenceManager qui fait office de façade, aux méthodes de persistance des objets.
Je sais utiliser à mon avantage le polymorphisme .		On a une entité qui possède une liste de Composant. Pour avoir un composant (et donc les propriété et méthode spécifique au composant), il faut caster le composant voulu en la classe que l'on veut (Location,Sprite...).
Je sais utiliser GIT pour travailler avec mon binôme sur le projet.		Le projet étant sur gitlab, il a bien fallu apprendre à s'en servir. Push,Pull,Branch... Lorsqu'il y a eu conflit, on a su fusionner les

	 <pre> Migration_entré - Suppression des ancienne classes - Merge remote-tracking branch 'origin/main' - Ajout des fichiers manquant - Migration Fini ! - Ajout d'une persistance pour les différents - Migration en cours ... - La persistance fonctionne ENFIN, merci l'err - Migration en cours ... - fuck la persistance des objets - Migration en cours ... - Migration en cours ... - Migration en cours ... </pre>	<p>codes.</p> <p>Pour l'ajout d'une fonctionnalité avec un impact assez conséquent sur le code, on a créé une branche qui dérive de main, ajouter la fonctionnalité et fusionner les deux branches.</p>
<p>Je sais utiliser le type statique adéquat pour mes attributs ou variables.</p>	<pre> public static ViewManager getViewManager() { return viewManager; } public static Stage getStage() { return stage; } public static PersistenceManager getPersistenceManager() { return persistenceManager; } </pre> <pre> private final File settingsFile = new File(pathname: "../res/Settings/settings.xml"); </pre> <pre> public abstract class Compolement { private ECompolementType type; public ECompolementType getType(){ return type; } public Compolement(ECompolementType type) { this.type=type; } } </pre>	<p>Les propriétés static présente dans Launcher permettent un accès aux managers partout où l'on a besoin. Seul le strict minimum a été mis en static.</p> <p>Le variable settingsFile est final, car on ne souhaite plus pouvoir modifier le contenu de celle-ci. C'est notamment pratique pour les constantes</p> <p>La classe Compolement a été mis abstract pour permettre l'héritage, mais pas l'instanciation de Compolement. De cette manière seuls les objets qui héritent de Compolement sont instanciables.</p>
<p>Je sais utiliser les différents composants complexes</p>	<pre> @FXML private ListView<String> highScoreList; </pre>	<p>Afin d'avoir une liste de Score sur la vue, il a été nécessaire d'utiliser une listView.</p>

<p>(listes, combo...) que me propose JavaFX.</p>	<pre>public void initialize() { highScore = Launcher.getPersistenceManager().getHighScore(); highScoreList.setItems(highScore.getListScore()); } @Override public ObservableSet<IEntity> getEntityCollection() { world.getEntityCollection().addListener((SetChangeListener<IEntity>) e -> { if (e.wasAdded()) { addEntity(e.getElementAdded()); } else if (e.wasRemoved()) { pane.getChildren().remove(e.getElementRemoved()); } }); }</pre>	<p>L'utilisation d'un observable set a permis d'actualiser la vue plus simplement.</p>
<p>Je sais utiliser les lambda-expressions.</p>	<pre>Launcher.getStage().setOnCloseRequest(e -> { gameManager.exit(); }); world.getEntityCollection().addListener((SetChangeListener<IEntity>) e -> { if (e.wasAdded()) { addEntity(e.getElementAdded()); } else if (e.wasRemoved()) { pane.getChildren().remove(e.getElementRemoved()); } });</pre>	<p>Il y a vraiment besoin d'un commentaire ? XD</p>
<p>Je sais utiliser les listes observables de JavaFX.</p>	<pre>@Override public ObservableSet<IEntity> getEntityCollection() { world.getEntityCollection().addListener((SetChangeListener<IEntity>) e -> { if (e.wasAdded()) { addEntity(e.getElementAdded()); } else if (e.wasRemoved()) { pane.getChildren().remove(e.getElementRemoved()); } }); }</pre>	<p>L'ajout d'un listener qui, quand un élément est ajouté ou retiré, appelle des méthodes pour actualiser la vue.</p>
<p>Je sais utiliser un convertisseur lors d'un bind entre deux propriétés JavaFX.</p>	<pre>life.textProperty().bind(Life.cast(world.getPlayer()).hpProperty().asString()); score.textProperty().bind(world.getCurrentLevel().scoreProperty().asString());</pre>	<p>Il n'a pas été nécessaire d'utiliser un convertisseur pour le binding.</p>
<p>Je sais utiliser un fichier CSS pour styler mon application JavaFX.</p>	<pre>#background{ -fx-background-image: url("/Sprites/Background.jpg"); -fx-background-size: stretch; -fx-background-position: center; -fx-pref-width: 1280; -fx-pref-height: 720; } public ViewManager(String pathView, String defaultView, String cssPath) throws Exception { this(pathView, defaultView); main.getStylesheets().add(cssPath); }</pre>	<p>On a utilisé un fichier CSS pour simplifier le balisage FXML. Ce CSS est passé en paramètre.</p>

<p>Je sais utiliser un formateur lors d'un bind entre deux propriétés JavaFX.</p>		<p>Il n'y a pas eu le besoin d'utiliser un formateur dans le code.</p>
<p>Je sais développer un jeu en JavaFX en utilisant FXML.</p>		<p>Toute la partie graphique du jeu est faite en FXML. Que ça soit dans un fichier .fxml ou bien, dans le code behind à de plus rares occasions.</p>
<p>Je sais intégrer, à bon escient, dans mon jeu, une boucle temporelle observable.</p>	 <pre data-bbox="443 1227 1050 1939"> public class Loop extends Observable implements Runnable { private final long millis; public long getMillis(){ return millis; } private boolean isRunning = true; public Loop(long millis) { this.millis = millis; } @Override public void run() { while (isRunning) { try { sleep(millis); Platform.runLater(() -> notifier()); } catch (InterruptedException e) { e.printStackTrace(); } } } public void StopLoop() { isRunning = false; } public void RestartLoop() { isRunning = true; } public void destroyLoop() { unsubscribeAll(); } } </pre>	<p>Tout le jeu est basé sur la boucle, chaque méthode qui demande une actualisation (sprite, déplacement...), sont abonnées à la boucle.</p>

```
@Override
public void start() {
    loop.subscribe(timer1);
    loop.subscribe(timer2);
    loop.subscribe(timer3);
    loop.subscribe( listener: this);
}

@Override
public void exit() { loop.unsubscribeAll(); }
```