Jérémy TREMBLAY B1
Guillaume TOUMAZET

Preuves maîtrise des compétences

DOCUMENTATION (dossier **`documentation`** à la racine) :

- Je sais décrire le contexte de mon application, pour que n'importe qui soit capable de comprendre à quoi elle sert (voir le fichier **`contexte.pdf`**). 👍
- Je sais concevoir et décrire un diagramme de cas d'utilisation pour mettre en avant les différentes fonctionnalités de mon application (voir les fichiers **`contexte.pdf`** et **`description_diagramme_cas_utilisation.pdf`**). 👍
- Je sais concevoir un diagramme UML de qualité représentant mon application (voir les fichiers SVG dans le dossier **`diagramme_de_classes`**). 👍
- Je sais décrire mon diagramme UML en mettant en valeur et en justifiant les éléments essentiels (voir le fichier **`description_diagramme_de_classes.pdf`**). 👍

CODE (dossier **`code`** à la racine) :

✔ Je sais utiliser les *Intent* pour faire communiquer deux activités. 👍

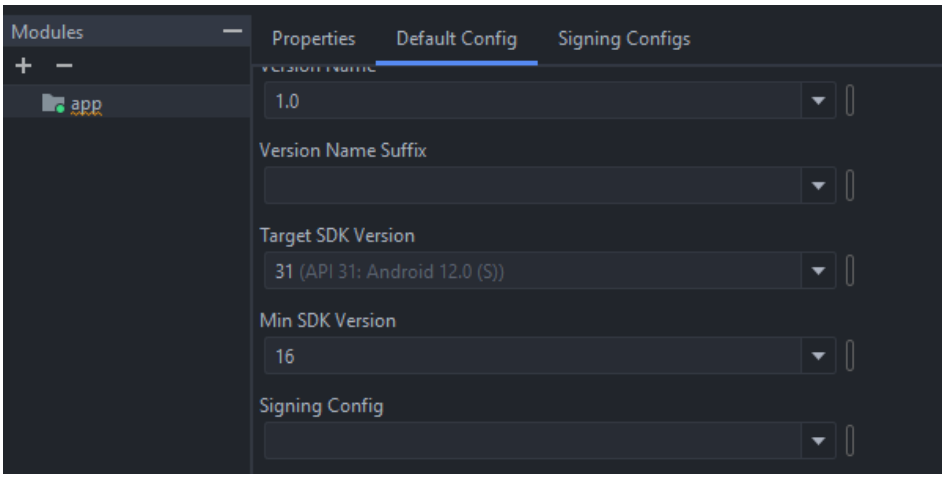
=> Nous avons utilisé les *Intent* pour faire communiquer nos activités, en particulier l'activité du menu principal, avec le menu du jeu :

```
public void goToJouer(View view) {
    Intent intent = new Intent( packageContext: this, ActiviteJeu.class);
    int numeroNiveau = Integer.parseInt(niveauChoisi.substring(niveauChoisi.length() - 1));
    intent.putExtra(NUMERO_NIVEAU, numeroNiveau);
    startActivity(intent);
}
```

Classe ActiviteMenuPrincipal. *On réalise un intent dans lequel on passe des données, puis on lance la nouvelle activité.*

✔ Je sais développer en utilisant le SDK le plus bas possible. 👍

=> Depuis Android Studio, on peut voir le SDK minimum visé (le plus bas) dans *File → Project Structure → Modules* :



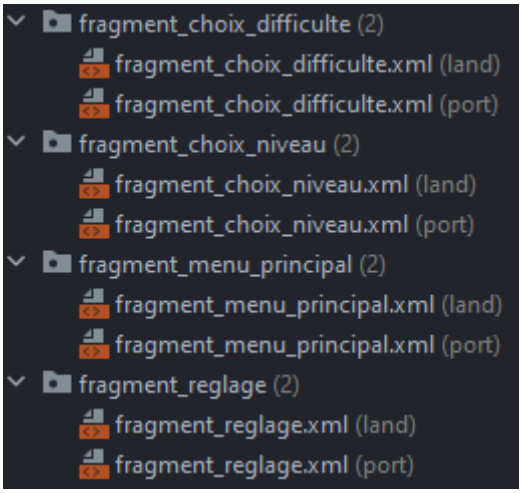
SDK minimum : 16, pour toucher tous les appareils.

Aussi visible depuis le fichier *build.gradle* :

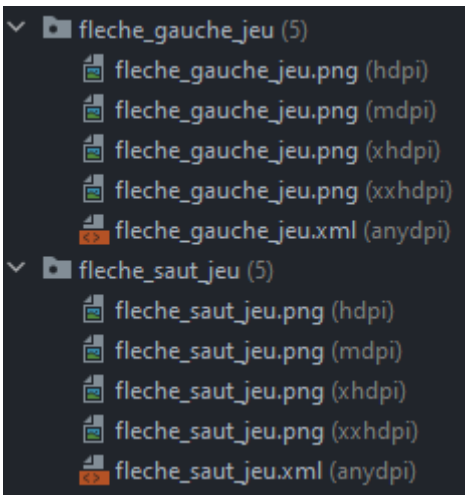
```
defaultConfig {
    applicationId "fr.iut.speedjumper"
    minSdk 16
    targetSdk 31
```

✔ Je sais distinguer mes ressources en utilisant les qualifier 👍

=> Réalisation de fichiers XML de *layout* avec des *qualifiers* qui vont venir changer l'affichage en fonction de si le téléphone est en mode portrait ou paysage :



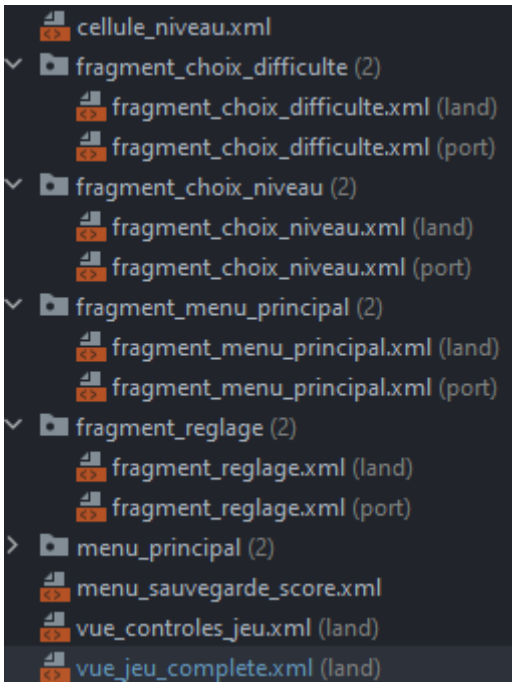
De même pour les images et vecteurs qui sont sélectionnés en fonction de la densité de pixels du téléphone :



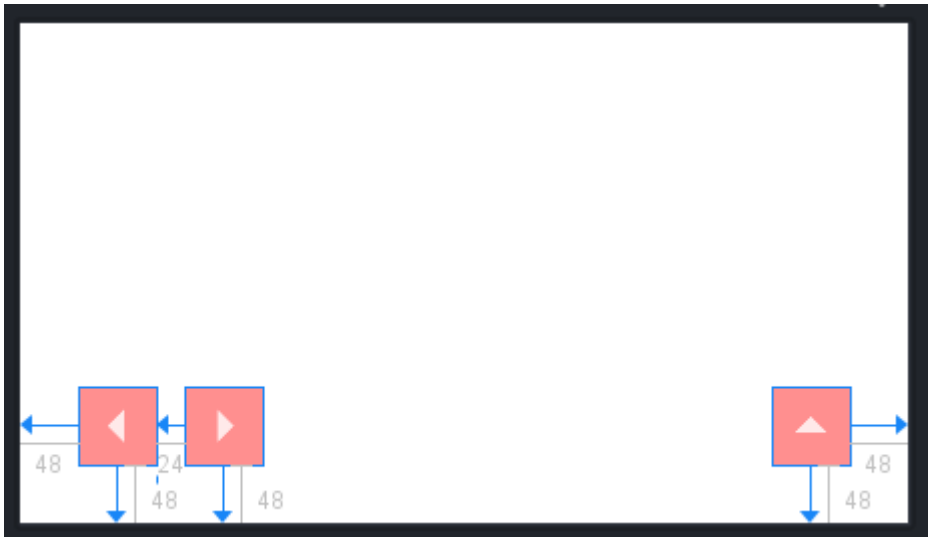
👍

- ✔ Je sais faire des vues xml en utilisant layouts et composants adéquats

=> Nous avons réalisé beaucoup de vues en XML pour nos contrôles, nos menus :



Nous avons utilisé principal le *ConstraintLayout*. Nous nous sommes également servis du *LinearLayout* et du *FrameLayout*. Le *ConstraintLayout* a par exemple été utilisé pour contraindre la position des boutons dans la fenêtre de notre jeu :



Capture d'écran depuis le designer *pour mettre en évidence les contraintes sur les boutons du jeu.*

Voir les fichiers de la capture d'écrans pour plus d'informations (dossier **`layout`**).

- ✔ Je sais coder proprement mes activités, en m'assurant qu'elles ne font que relayer les évènements 👍

=> Nos activités ne font que relayer les évènements, elles n'ont pas plus de responsabilité. Elles ne font rien que ferait le modèle (c'est lui qui gère tout le jeu, le système de déplacements, de collisions, etc.). Par exemple, lors de la détection d'une touche, on vient simplement ajouter cette touche dans une liste, et le modèle s'occupera ensuite de la traiter :

```java
private void genererEcouteur(ImageButton bouton, Touche typeTouche) {
    bouton.setOnTouchListener((view, motionEvent) -> {
        if (motionEvent.getAction() == MotionEvent.ACTION_DOWN) {
            if (!lesTouchesPressees.contains(typeTouche)) {
                lesTouchesPressees.add(typeTouche);
            }
        }
        else if (motionEvent.getAction() == MotionEvent.ACTION_UP) {
            lesTouchesPressees.remove(typeTouche);
        }
        return true;
    });
}
```

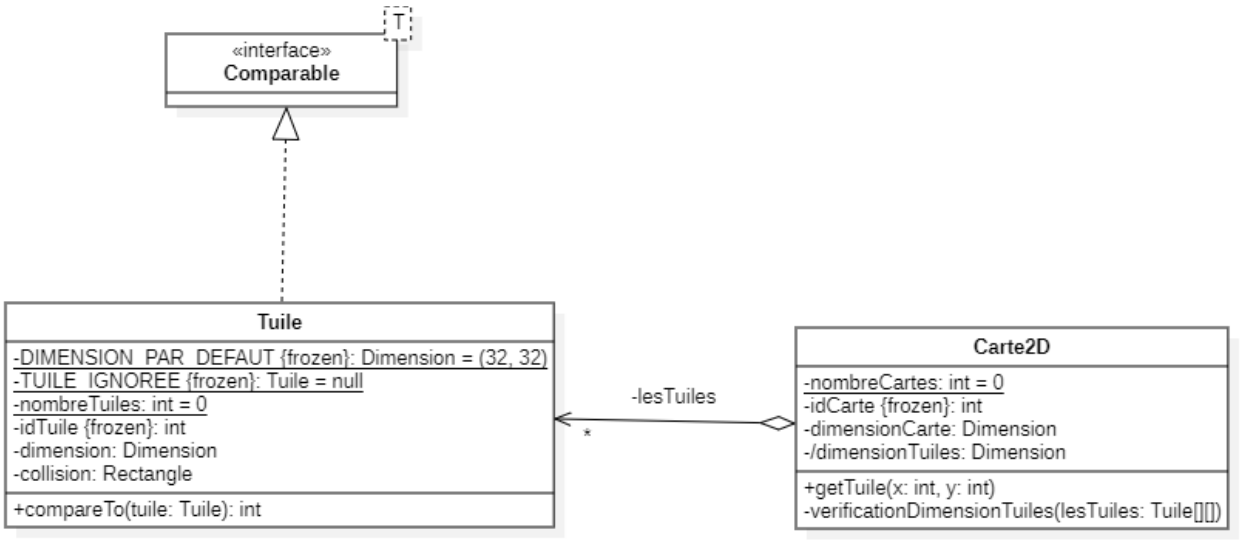
Classe *RecuperateurDeTouchesAndroid*. Il s'occupe simplement d'ajouter la touche pressée dans une collection qui est ensuite traitée par le modèle :

```java
@Override
public List<Commande> attribuerAction() {
    lesTouches = recuperateurDeTouches.detecte();
    lesCommandes.clear();

    if (lesTouches.contains(Touche.ESPACE)) {
        lesCommandes.add(espace);
    }
    if (lesTouches.contains(Touche.FLECHE_DROITE)) {
        lesCommandes.add(flecheDroite);
    }
    if (lesTouches.contains(Touche.FLECHE_GAUCHE)) {
        lesCommandes.add(flecheGauche);
    }
    if (lesTouches.contains(Touche.ECHAP)) {
        pause = true;
    }
    return lesCommandes;
}
```

Le modèle gère ensuite la logique derrière la pression de ces touches (classe GestionnaireActionUtilisateurJeu*).*

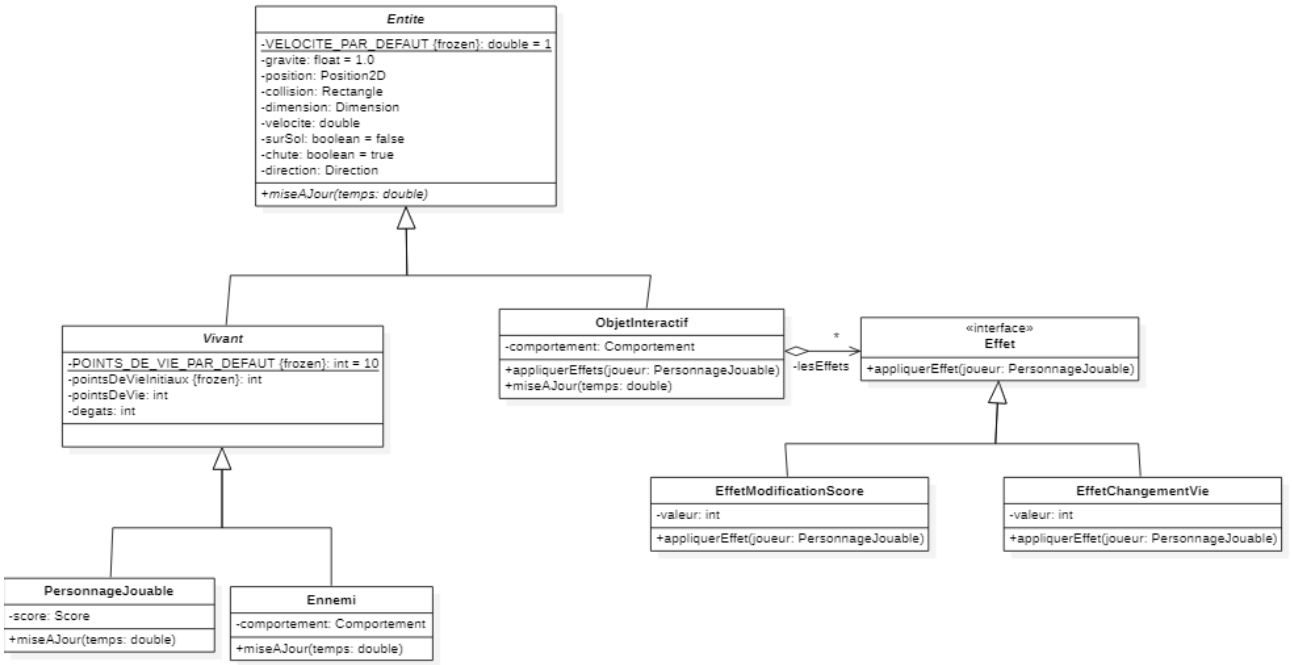
- ✔ Je sais coder une application en ayant un véritable métier 👍

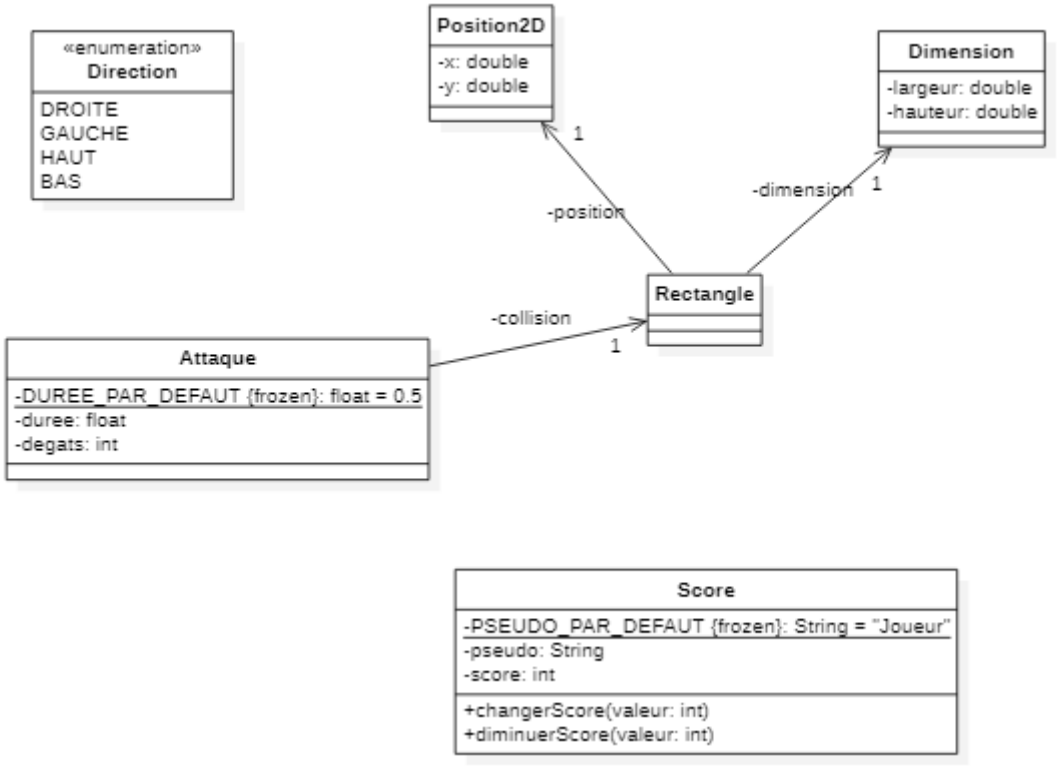
=> Notre modèle est encapsulé de manière à ce que chaque classe n'est qu'une, et une seule responsabilité. Notre modèle contient de nombreuses classes qui ne sont que des données métier (qui vont contenir nos données). Aucune donnée métier n'est traitée et manipulée dans la vue, le modèle les conserve dans des collections et les traite de la manière dont il a été conçu. Voici par exemple notre classe *Carte2D* qui encapsule toutes les données contenues dans une carte :



Cette classe gère les données contenues à l'intérieur d'une et une seule carte. On y retrouve par exemple la carte sous forme d'un tableau à deux dimensions de tuiles, ainsi que ses dimensions et un identifiant unique.

On peut aussi citer les classes liées au maintient des scores de notre jeu, ou encore les classes liées à nos entités (qui représentent des être vivants déplaçables).





✔ Je sais parfaitement séparer vue et modèle 👍

=> Notre vue possède notre *GestionnaireDeJeu* qui est une façade et qui vient faciliter l'utilisation de notre jeu. A aucun moment nos vues viennent modifier les données de notre modèle et vice-versa. Le modèle possède sa boucle de jeu qui le met à jour (se référer à la question de la boucle *threadée*), et la vue est simplement notifiée par ce modèle. Tout ce que fais la vue, c'est créer le modèle et s'y abonner de manière à mettre à jour l'affichage :

```
// Création du modèle.
gestionnaireDeJeu = new GestionnaireDeJeu(new RecuperateurDeTouchesAndroid(vueBoutons),
        gestionnaireDeRessources);
gestionnaireDeJeu.changerNiveau(numeroNiveau % 3);
tableauJeu = gestionnaireDeJeu.getTableauJeu();
vueJeu = new VueJeu( context: this, tableauJeu);
parent.addView(vueJeu);
parent.addView(vueBoutons);
// Abonnement pour notification et mise à jour de la vue.
gestionnaireDeJeu.attacher( observateur: this);
if (!gestionnaireDeJeu.isLance()) {
    gestionnaireDeJeu.lancerJeu();
}
```

Classe ActiviteJeu. *Création du modèle et abonnement. Aucune autre action n'est réalisée par la vue hormis rafraîchir de nouveau l'affichage de manière périodique.*

✔ Je maîtrise le cycle de vie de mon application 👍

=> Nous avons essayé de contrôler au maximum le cycle de vie de notre application. Pour ce faire, nous gérons différemment les actions que nous réalisons en fonction de si l'application est ouverte, fermée ou mise en pause.

Dans la classe *ActiviteJeu* et *ActiviteMenuPrincipal*, nous affichons un Toast sur le téléphone de l'utilisateur lorsqu'il ferme le jeu.

```
@Override
protected void onDestroy() {
    super.onDestroy();
    Toast t = Toast.makeText(getApplicationContext(), text: "A bientôt !",
            Toast.LENGTH_LONG);
    t.setGravity( gravity: Gravity.BOTTOM | Gravity.RIGHT, xOffset: 0, yOffset: 0);
    t.show();
}
```

Affichage d'un toast lors de la fermeture du jeu dans la classe ActiviteMenuPrincipal.

Lorsque l'activité est mise en pause, on met le jeu en pause depuis *l'ActiviteJeu* et on affiche un petit toast également :

```
@Override
protected void onPause() {
    super.onPause();
    gestionnaireDeJeu.fermerJeu();
    messageToast = Toast.makeText(getApplicationContext(), text: "Jeu en pause !",
            Toast.LENGTH_LONG);
    messageToast.setGravity( gravity: Gravity.BOTTOM | Gravity.RIGHT, xOffset: 0, yOffset: 0);
    messageToast.show();
}
```

Affichage d'un toast pour rassurer l'utilisateur en lui précisant que le jeu est en pause.

De même, on vient gérer différemment les façons dont nos objets sont instanciés en fonction de l'état dans lequel est l'activité. On s'assure toujours que notre activité reste dans un état cohérent. Voir le code pour plus de détails.

✔ Je sais utiliser le *findViewById* à bon escient 👍
=> Nous avons limité au maximum les *findViewById* qui viennent parcourir toute la vue pour retrouver l'élément qui possède l'id qu'on lui précise. Nous n'avons aucun *findViewById* qui est appelé à plusieurs fois d'affilé, et si cela devait arriver, nous stockons le résultat de la première recherche pour éviter une recherche inutile.

Si vous voulez vous assurer de la non-duplication des *findViewById*, vous pouvez aller dans *Navigate → Search Everywhere →* taper *findViewById →* aller dans *Files → Find in Files*. Vous verrez alors tous les appels, et vous pourrez vous assurer qu'aucun n'a été dupliqué.

✔ Je sais gérer les permissions dynamiques de mon application 👍
=> Notre application ne demande aucune autorisation.

✔ Je sais gérer la persistance légère de mon application 👍
=> Nous utilisons la persistance légère pour faire passer le numéro de notre niveau d'une activité à une autre. Ce numéro est récupéré lorsque l'utilisateur clique sur un bouton pour sélectionner un niveau. Nous faisons alors passer ce numéro dans notre *intent* avec un *putExtra* pour pouvoir le récupérer dans notre nouvelle activité :

```
public void goToJouer(View view) {
    Intent intent = new Intent( packageContext: this, ActiviteJeu.class);
    int numeroNiveau = Integer.parseInt(niveauChoisi.substring(niveauChoisi.length() - 1));
    intent.putExtra(NUMERO_NIVEAU, numeroNiveau);
    startActivity(intent);
}
```

Dans l'activité du menu principal, on fait passer le niveau sélectionné par notre intent.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    numeroNiveau = getIntent().getExtras().getInt(ActiviteMenuPrincipal.NUMERO_NIVEAU);
```

On peut ensuite la récupérer depuis la nouvelle activité.

```
@Override
protected void onSaveInstanceState(@NonNull Bundle outState) {
    super.onSaveInstanceState(outState);
    outState.putInt(ActiviteMenuPrincipal.NUMERO_NIVEAU, numeroNiveau);
}

@Override
protected void onRestoreInstanceState(@NonNull Bundle savedInstanceState) {
    super.onRestoreInstanceState(savedInstanceState);
    numeroNiveau = (Integer) savedInstanceState.get(ActiviteMenuPrincipal.NUMERO_NIVEAU);
}
```

On sauvegarde ne numéro du niveau lorsque l'activité est sauvegardé en la mettant dans un bundle, *et on la récupère de ce* bundle *lorsqu'elle est restaurée.*

✔ Je sais gérer la persistance profonde de mon application 👎

=> Nous n'avons réalisé aucune persistance profonde de notre application.

✔ Je sais afficher une collection de données 👍

=> Nous avons réalisé l'affichage d'une collection de données, les noms des différents niveaux de notre jeu.

```
RecyclerView laListView = view.findViewById(R.id.recycleView);
laListView.setLayoutManager(new LinearLayoutManager(activiteParente));
laListView.setAdapter(new CustomAdapter(listNiveau));
```

Classe FragmentChoixNiveau. *Nous instancions notre* RecyclerView *en lui donnant un Adaptateur que nous avons codé nous-même (voir preuve à la question suivante).*

Nous avons ainsi créé notre *ViewHolder* :

```
public class MonViewHolder extends RecyclerView.ViewHolder {
    private final Button leBouton;
    public MonViewHolder(LinearLayout leLayout) {
        super(leLayout);
        leBouton = itemView.findViewById(R.id.buttonNiveau);
    }
    public Button getLeBouton() { return leBouton; }

    public void setNiveauCourant(String niveauCourant) {
        leBouton.setOnClickListener(v -> {
            ((ActiviteMenuPrincipal)leBouton.getContext()).setNiveauChoisi(niveauCourant);
            ((ActiviteMenuPrincipal)leBouton.getContext()).getSupportFragmentManager().beginTransaction()
                    .setReorderingAllowed(true)
                    .replace(R.id.fragmentMenu, FragmentChoixDifficulte.class,  args: null)
                    .commit();
        });
    }
}
```

Classe MonViewHolder.

Cette preuve est grandement liée à celle expliqué ci-dessous.

- ✔ Je sais coder mon propre adaptateur 👍

=> Nous avons créé notre adaptateur pour afficher notre collection de données comme expliqué précédemment. Nous avons ainsi redéfini les 3 méthodes qu'il faut redéfinir pour que cela fonctionne, à savoir, *getItemCount* qui retourne simplement le nombre d'éléments de la collection, *onBindViewHolder* qui précise comment va être *bindé* et affiché nos données (le nom du niveau est affiché dans le bouton), et *onCreateViewHolder* qui créé un *ViewHolder* à partir d'un *LinearLayout* (où les données seront affichées).

```
public CustomAdapter(List<String> dataSet) { this.listNiveau = dataSet; }

@NonNull
@Override
public RecyclerView.ViewHolder onCreateViewHolder(ViewGroup viewGroup, int viewType) {
    LinearLayout leLayout = (LinearLayout) LayoutInflater.from(viewGroup.getContext())
            .inflate(R.layout.cellule_niveau, viewGroup, attachToRoot: false);
    return new MonViewHolder(leLayout);
}

@Override
public void onBindViewHolder(@NonNull RecyclerView.ViewHolder holder, final int position) {
    String niveauCourant = listNiveau.get(position);
    ((MonViewHolder)holder).setNiveauCourant(niveauCourant);
    ((MonViewHolder)holder).getLeBouton().setText(niveauCourant);
}

@Override
public int getItemCount() {
    return listNiveau.size();
}
```

Classe CustomAdapter.

- ✔ Je maîtrise l'usage des fragments 👍

=> Nous avons réalisé des fragments dans l'optique de nous en resservir plus tard. Nous en avons 4 au total, mais le plus intéressant est certainement celui qui est lié aux options. Il précise ainsi comment est affiché le menu des options (contient le volume du son et de la musique) :

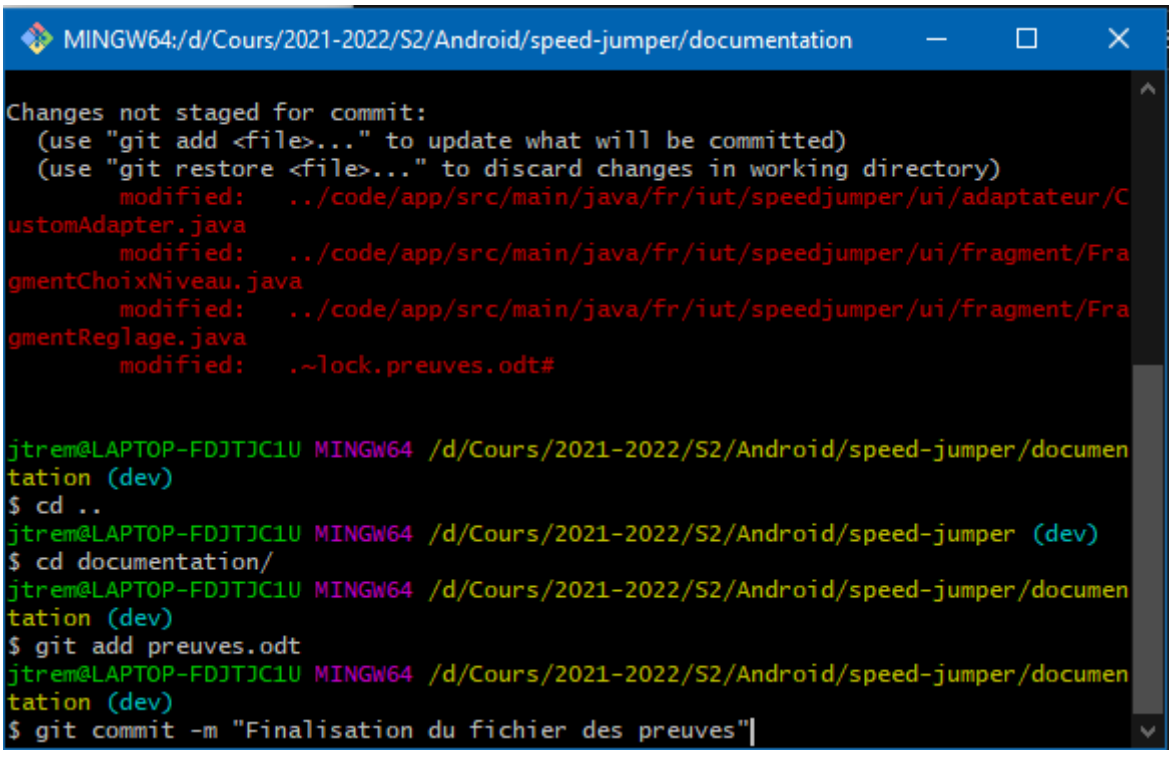
```java
@Override
public void onViewCreated(@NonNull View view, @Nullable Bundle savedInstanceState) {
    super.onViewCreated(view, savedInstanceState);
    activiteParente = (ActiviteMenuPrincipal) getContext();

    view.findViewById(R.id.bouton_retour).setOnClickListener(view1
            -> activiteParente.getSupportFragmentManager().beginTransaction()
            .setReorderingAllowed(true)
            .replace(R.id.fragmentMenu, FragmentMenu.class, args: null)
            .commit());

    SeekBar volumeMusique = view.findViewById(R.id.reglageMusique);
    TextView affichageMusique = view.findViewById(R.id.valeurMusique);
    volumeMusique.setProgress(activiteParente.getVolumeMusique());
    affichageMusique.setText(String.valueOf(activiteParente.getVolumeMusique()));
    // Même chose pour la seconde seekbar, et ajout des listeners sur les seekbar.
```

Classe FragmentReglage *qui contient les* seekbar *utilisées pour régler le volume.*

✔ Je maîtrise l'utilisation de Git 👍

=> Nous avons utilisé le Gitlab de l'IUT pour versionner notre projet. Nous avons créé différentes branches pour nous y retrouver dans l'avancée du projet (main comme branche principale, dev pour l'avancement général et master pour le dépôt final).

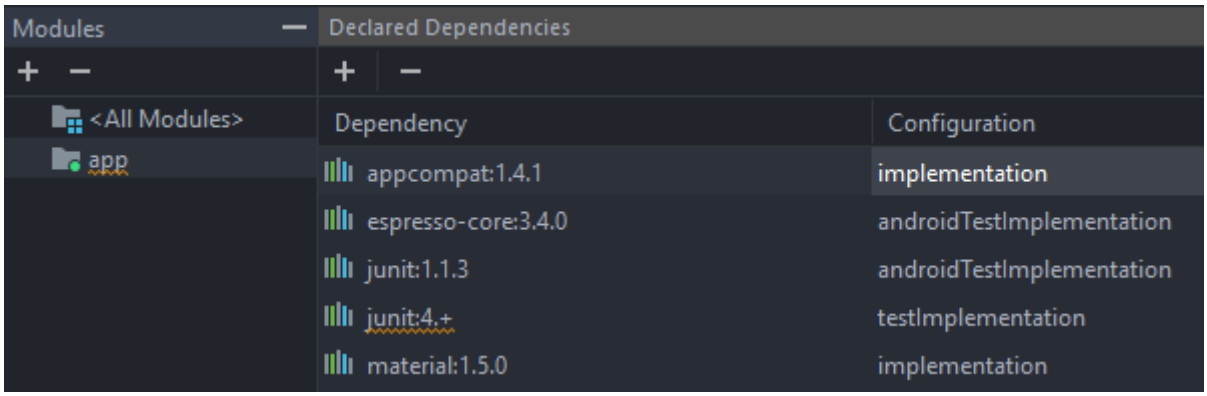
Tous mes commits personnels (Jérémy) étaient réalisés sous Git Bash



APPLICATION :

- Je sais développer une application sans utiliser de librairies externes. 👍

=> Depuis Android Studio, on peut s'assurer qu'aucune bibliothèque externe n'a été ajoutée dans *File → Project Structure → Dependencies* :



Ce qui correspond aux bibliothèque standard.

- Je sais développer une application publiable sur le store. 👎

=> Nous n'avons pas publié notre application sur le store.

- Je sais développer un jeu intégrant une boucle de jeu *threadée* observable.

=> Nous avons bien une boucle de jeu *threadée* observable.

```
@Override
public void run() {
    actif = true;
    dernierTemps = System.nanoTime();
    long tempsAttente;

    while(actif) {
        tempsCourant = System.nanoTime();
        tempsEcoule = tempsCourant - dernierTemps;
        if (tempsEcoule >= TEMPS_AVANT_NOTIFICATION) {
            ticker(tempsEcoule);
            tempsEcouleTotal += tempsEcoule;
            dernierTemps = tempsCourant;
        }
        else {
            tempsAttente = TEMPS_AVANT_NOTIFICATION - tempsEcoule;
            try {
                sleep( millis: tempsAttente / TEMPS_MILLISECONDE,
                        (int) (tempsAttente % TEMPS_MILLISECONDE));
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
    }
}

public void ticker(double timer) { notifier(timer); }
```

Classe BoucleDeJeu. *Elle implémente* Runnable *de manière à pouvoir être lancée sous forme d'un autre* Thread. *Elle s'endort un temps spécifique et se réveille à intervalle réguliers pour envoyer une notification afin de mettre à jour le jeu.*

A la création, le jeu s'abonne en tant qu'observateur :

```
public Jeu(RecuperateurDeTouches recuperateur, GestionnaireDeRessources gestionnaireDeRessources)
        throws IllegalArgumentException {
    tableauJeu = new TableauJeu(gestionnaireDeRessources);
    gestionnaireActions = new GestionnaireActionUtilisateurJeu(recuperateur, tableauJeu);
    VisiteurCollisions visiteur = new VisiteurCollisionsBasique(tableauJeu);
    gestionnaireDeCollisions = new GestionnaireDeCollisions(tableauJeu, visiteur);
    collisionneurPointRectangle = new CollisionneurPointRectangle();
    chuteur = new Chuteur(tableauJeu);
    boucleDeJeu = new BoucleDeJeu();
    boucleDeJeu.attacher( o: this);
    pause = true;
}
```

On peut la lancer de cette manière :

```
public void lancerJeu() throws IllegalStateException {
    if (processus != null && processus.isAlive()) {
        throw new IllegalStateException("Le thread du jeu est déjà lancé et doit d'abord être interrompu.");
    }
    boucleDeJeu.setActif(true);
    pause = false;
    processus = new Thread(boucleDeJeu,  name: "Speed Jumper Thread");
    processus.start();
}
```

On peut l'interrompre :

```
public void arreterJeu() {
    boucleDeJeu.setActif(false);
    try {
        if (isLance()) {
            processus.join();
        }
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    pause = true;
}
```

- Je sais développer un jeu graphique sans utiliser de SurfaceView 👍

=> Nous n'avons pas utilisé de *SurfaceView,* nous avons créés nos propres classes *View* et *ViewGroup* que nous avons affichées. Voir les classes du paquetage **`fr.iut.speedjumper.ui.vues.`**

Voici quand même quelques détails. Nos tuiles sont gérées par une classe *VueTuile* qui étend *View* :

```
public class VueTuile extends View {
    private Paint paint;
    private Bitmap image;
    private Tuile tuile;
```

Classe VueTuile. *Elle a pour responsabilité de faire correspondre une tuile de notre modèle avec une image.*

De plus, cette classe redéfinie la méthode *onDraw* qui est appelée lorsque la vue doit être redessinée :

```
@Override
protected void onDraw(Canvas canvas) {
    super.onDraw(canvas);
    canvas.drawBitmap(image,  left: 0,  top: 0, paint);
}
```

Classe VueTuile, *méthode* onDraw. *Une* vueTuile *s'affiche simplement en affichant son image à la position 0, 0.*

Ce n'est pas sa responsabilité de bien se placer sur l'écran, c'est une autre classe qui gère cela. Cette autre classe est la classe *VueCarte*, qui contient plusieurs *VueTuile*. Or cette *VueCarte* est aussi une *View*, on a donc affaire à une composition. Par conséquent, la *VueCarte* étend la classe *ViewGroup* car elle contient un regroupement de vues (celles de chaque tuile de la carte) :

```
public class VueCarte extends ViewGroup {
```

Composition de la classe VueCarte qui contient plusieurs tuiles.

Lors de la création de cette *VueCarte*, on vient simplement ajouter un certain nombre de *VueTuile* afin de pouvoir recréer graphiquement notre carte :

```
Tuile[][] vueCarte = carteCourante.getLesTuiles();
generationImage(vueCarte);
for (int y = 0; y < vueCarte.length; y++) {
    for (int x = 0; x < vueCarte[y].length; x++) {
        addView(new VueTuile(getContext(), vueCarte[y][x],
                lesImages.get(vueCarte[y][x])));
    }
}
```

Création des VueTuile *à partir des données du modèles et des images, et ajout dans la liste de fils maintenus dans la classe* ViewGroup *(la classe* ViewGroup *gère ses vues filles).*

Enfin, notre *VueCarte* redéfinie des méthodes pour mesurer ses fils (*onMeasure*) et pour spécifier comment les afficher (*onLayout*). Les méthodes sont assez grosses, voir le code pour plus de détail (classe *VueCarte*).

De même, une *VueCarte* est contenue dans une *VueNiveau*. Une *VueNiveau* est une *ViewGroup*, car elle contient également les boutons (pour les contrôles), et le personnage (une *VueEntite*). Nous avons ainsi réussis à générer un affichage avec nos propres vues personnalisées, et sans utiliser de *SurfaceView*.