

Pokemon Géo

Développement Mobile Android

Anthony Chomienne (CPE Lyon)

Pokemon Géo

Cette application a pour but de jouer à pokemon en nous baladant dehors en se basant sur la géolocalisation de l'appareil que celle-ci soit au premier plan ou non. Un certain nombre de données seront issues d'un webservice ou de données géographiques.

Objectifs

- Prendre en main Android
- Développer et Déployer une application native (java)
- Voir les différentes étapes du développement Android
- Utiliser un webservice REST
- Connaître un peu mieux le cœur d'Android

Note:

Cette application réalisée sur les 7 séances de TP aura pour but de vous faire voir un certain nombre d'aspects de la programmation pour Android en Java. On utilisera pas de bibliothèque externe en dehors de l'aspect cartographie afin de vous montrer ce qui se passe au cœur de votre système. L'objectif est de vous montrer comment faire vous-même des opérations qui n'ont pas de bibliothèque adaptée ou dont vous ne pouvez pas vous servir pour que vous ne soyez pas complètement démunis. Les séances ne sont pas découpées dans le sujet. Plus vous avancez, plus vous pourrez ajouter des fonctionnalités intéressantes. L'évaluation tiendra compte de l'avancement, de la rigueur que vous avez appliquée et de la qualité du travail fourni.

Premier Pas

Lors de la création d'un nouveau projet sous Android Studio, choisissez comme base de départ, le template pour smartphone et tablette: Empty Activity. Ceci vous donnera déjà une base de code pour commencer minimaliste mais suffisamment claire pour voir

comment s'articulent les différents éléments d'un projet. Il vous faudra ensuite déterminer le nom de votre application et définir le package name. Le package name de votre application est un élément important, c'est l'un des moyens qu'ont les smartphones d'identifier les différentes applications. Il doit donc être unique, deux applications ne peuvent avoir le même package name (ni sur le store, ni sur un même téléphone). Pour le choix de la version minimum d'Android, vous avez le choix. Android Studio vous donne un pourcentage approximatif de terminaux actifs que vous pouvez supporter à partir de cette version. (Les données sont basées sur les accès aux playstore)

Actuellement selon les besoins, je choisis une version minimale 21 correspondant à Lollipop. La quantité de personnes ne pouvant pas utiliser l'application étant de toute façon minimal (~2%)

L'arborescence de votre projet sera alors la suivante:

- app
 - manifests
 - java
 - res
- Gradle Scripts
 - build.gradle (Project: PokemonGeo)
 - build.gradle (Module: app)

La partie **manifests** contient le fichier AndroidManifest.xml. La partie **java** contient les packages de votre projet et vos sources java. Par défaut vous avez un package *package_name* (choisi lors de la création du projet) qui contient votre MainActivity. Apparaissent également deux autres packages avec le package name que vous avez choisi, avec la mention *androidTest* et *test*, on évoquera pas ces aspects ici. La partie **res** contient les différentes ressources de votre projet réparties dans des dossiers appropriés.

Il y a un dossier **drawable**, **layout**, **mipmap** et **values** au sein du dossier res. Ces dossiers peuvent être suffixés par divers élément comme vue en cours.

drawable contient les images de votre projet. **layout** contient les fichiers xml permettant de décrire les Vues au sens MVP du terme. **mipmap** contient l'icône de l'application dans différentes résolutions. **values** contient les fichiers de style, les fichiers de textes...

Lorsque vous créez le projet par défaut, Android Studio vous propose de créer une activity plus ou moins vide en fonction de ce que vous avez choisi. Dans notre cas, si on regarde le fichier *res/layout/activity_main.xml* (ou autre si vous lui avez donné un autre nom), elle contient déjà beaucoup de choses : Un ConstraintLayout et une TextView contenant «Hello World ! »

Les deux fichiers build.gradle servent à l'outil de compilation Gradle. Le second contient en particulier différentes informations concernant votre application (voir cours).

Build

Le build se fait automatiquement lorsque vous lancez votre application, mais vous pouvez le lancer manuellement depuis le menu Build. Pour déployer sur un téléphone, vous avez deux cas:

- Utilisez un Android Virtual Device (AVD).
- Utilisez un device Android.

Android Virtual Device (AVD)

Dans la barre d'outils d'Android Studio vous pouvez voir à droite un certain nombre



d'icônes dont la suivante:

Une fenêtre s'ouvre contenant vos devices virtuels présents sur votre système (Normalement aucun) et vous offre la possibilité d'en créer d'autres en partant d'une base existante (Nexus 5, Nexus 4, Pixel...), vous pouvez également choisir la version de votre device afin de tester sur différentes versions du système.

Android Device

Deux éléments sont nécessaires pour développer en utilisant son téléphone:



- Autoriser l'installation depuis des sources inconnues

Rendez-vous dans les paramètres du téléphone, dans le menu sécurité et activer l'autorisation d'installer des applications depuis une source inconnue.

- Activer le mode développeur et notamment le debug USB

Pour activer le mode développeur, dans les paramètres du téléphone, allez dans «À propos du téléphone» puis «Information sur le logiciel» et appuyer plusieurs fois sur le «numéro de build», au bout d'un moment il va vous dire qu'il a débloqué les options développeur. Vous pourrez alors vous rendre dans les options développeur pour activer le debug USB.

Run

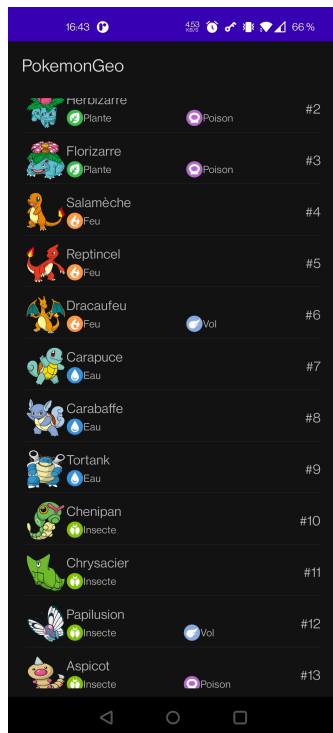
Un clic sur Run dans l'IDE  ou sur Debug  installe l'application sur le device que vous sélectionnerez dans la fenêtre qui s'ouvrira alors. Après l'installation, il lancera votre application sur le téléphone.

Première Activity, Liste et DataBinding

Nous allons commencer cette application par une liste des pokemons, visible dans un premier temps sur l'écran de démarrage de l'application. Cet écran permettra

également à terme d'accéder à la carte, à l'inventaire, nos pokemons et au pokedex. Dans un premier temps, nous nous contenterons d'afficher une liste statique. Nous utiliserons le databinding et le ViewModel pour ça.

L'objectif est d'obtenir quelque chose comme suit :



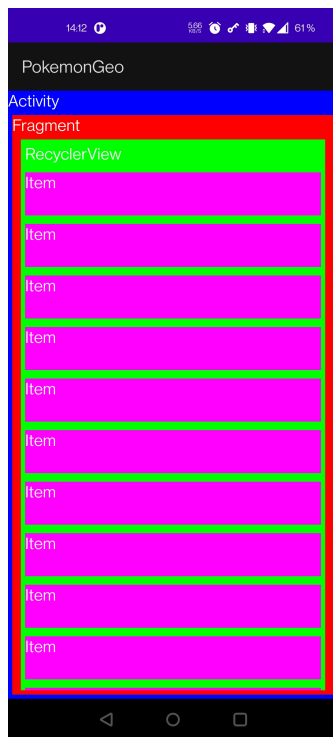
Pour commencer dans le fichier *build.gradle* du dossier *app*. Ajouter dans la balise *android*, la balise **buildFeatures** avec **dataBinding true**, comme suit:

```
android {
    ...

    buildTypes {
        ...
    }
    buildFeatures {
        dataBinding true
    }
}
```

Synchronisez le projet après avoir fait cette modification

Le schéma suivant vous présente les différentes strates de l'interface que nous allons mettre en place sur cette partie, elle sera amenée à évoluer par la suite.



N'hésitez pas à reproduire ce schéma et à le compléter d'annotation pour vous aider à comprendre l'architecture des différents éléments par rapport à votre code.

Nous allons maintenant modifier le fichier **activity_main.xml** afin de mettre en place le databinding pour celui-ci ainsi que de préparer l'utilisation de fragment.

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    >
    <data/>
    <LinearLayout
        android:background="#0000FF"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:orientation="vertical">
        <FrameLayout
            android:layout_margin="5dp"
            android:id="@+id/fragment_container"
            android:layout_width="match_parent"
            android:layout_height="match_parent"
            />
    </LinearLayout>
</layout>
```

Le `FrameLayout` ici nous servira de point d'ancrage pour nos différents `Fragments`. Il est ici, pour être remplacé par les fragments successifs que nous utiliserons à l'aide de transactions.

MainActivity.java du fait de l'utilisation du `DataBinding`, va devoir évoluer et ressembler à ce qui suit.

```
public class MainActivity extends AppCompatActivity {

    private ActivityMainBinding binding;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        binding = DataBindingUtil setContentView(this, R.layout.activity_main);
    }
}
```

Nous voulons manipuler des pokemons. Un pokemon correspond à un nom, un poids, une taille, un ou deux types, une image de face, une position dans le pokedex. Nous aurons donc comme classe modèle la classe `Pokemon` qui suit.

```
package fr.mobdev.pokemongo;

public class Pokemon {
    private int order;
    private String name;
    private int height;
    private int weight;
    private int frontResource;
    private POKEMON_TYPE type1;
    private POKEMON_TYPE type2;

    public Pokemon() {
        order = 1;
        name = "Unknown";
        frontResource = R.drawable.p1;
        type1 = POKEMON_TYPE.Plante;
    }

    public Pokemon(int order, String name, int frontResource,
        POKEMON_TYPE type1, POKEMON_TYPE type2) {

        this.order = order;
        this.name = name;
        this.frontResource = frontResource;
        this.type1 = type1;
        this.type2 = type2;
    }
}
```

```

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public int getOrder() {
    return order;
}

public void setOrder(int order) {
    this.order = order;
}

public int getHeight() {
    return height;
}

public void setHeight(int height) {
    this.height = height;
}

public int getWeight() {
    return weight;
}

public void setWeight(int weight) {
    this.weight = weight;
}

public int getFrontResource() {
    return frontResource;
}

public void setFrontResource(int frontResource) {
    this.frontResource = frontResource;
}

public POKEMON_TYPE getType1() {
    return type1;
}

public void setType1(POKEMON_TYPE type1) {

```

```

        this.type1 = type1;
    }

    public POKEMON_TYPE getType2() {
        return type2;
    }

    public void setType2(POKEMON_TYPE type2) {
        this.type2 = type2;
    }

    public String getType1String() {
        return type1.name();
    }

    public String getType2String() {
        return type2.name();
    }
}

```

Pour la gestion des types, nous pourrions passer par une énumération comme suit:

```

enum POKEMON_TYPE {
    Acier,
    Combat,
    Dragon,
    Eau,
    Electrique,
    Fee,
    Feu,
    Glace,
    Insecte,
    Normal,
    Plante,
    Poison,
    Psy,
    Roche,
    Sol,
    Spectre,
    Tenebre,
    Vol
}

```

Créez un second layout nommé **pokedex_fragment.xml** il contiendra ce qui suit:

```

<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"

```



```

xmlns:app="http://schemas.android.com/apk/res-auto"
xmlns:tools="http://schemas.android.com/tools">
<data/>
<androidx.constraintlayout.widget.ConstraintLayout
    android:background="#FF0000"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
    <androidx.recyclerview.widget.RecyclerView
        android:background="#00FF00"
        android:id="@+id/pokemon_list"
        android:layout_width="0dp"
        android:layout_height="0dp"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintTop_toTopOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        android:layout_marginStart="10dp"
        android:layout_marginEnd="10dp"
        android:layout_marginTop="5dp"
        android:layout_marginBottom="5dp"
        />
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>

```

Ce layout sera affiché sur l'écran d'accueil de l'application dans un premier temps. Une liste des pokemons existant.

La RecyclerView ajouté il y a quelques années est mieux que la ListView en terme de gestion de la mémoire mais son fonctionnement est similaire. Il automatise certaine chose qu'il fallait faire manuellement avant.

De la même façon que pour l'Activity, nous avons besoin d'un fichier Java associé. Celui-ci étendra le type Fragment. Créez le fichier **PokedexFragment.java**. Il contiendra dans un premier temps ce qui suit:

```

public class PokedexFragment extends Fragment {
    @Nullable
    @Override
    public View onCreateView(@NonNull LayoutInflater inflater,
        @Nullable ViewGroup container,
        @Nullable Bundle savedInstanceState) {

        PokedexFragmentBinding binding = DataBindingUtil.inflate(inflater,
            R.layout.pokedex_fragment, container, false);
        binding.pokemonList.setLayoutManager(new LinearLayoutManager(
            binding.getRoot().getContext()));
        return binding.getRoot();
    }
}

```

```

    }
}

```

La fonction `getRoot` de `PokedexBinding` nous renvoie la View Java après avoir été créé à partir du fichier xml par la fonction `inflate`. Cette vue et ses enfants contiennent tout ce que l'on a défini dans le layout afin de gérer le rendu graphique.

Pour afficher un fragment dans votre activity comme vu en cours, nous devons passer par le `FragmentManager`, créer une transaction, réaliser un certain nombre d'opération. À partir de là vous pouvez lancer et voir certain résultat sur votre téléphone ou émulateur. Pensez à appeler cette fonction !

```

public void showStartup() {
    FragmentManager manager = getSupportFragmentManager();
    FragmentTransaction transaction = manager.beginTransaction();
    PokedexFragment fragment = new PokedexFragment();
    transaction.replace(R.id.fragment_container, fragment);
    transaction.commit();
}

```

Que vous utilisiez une `ListView` ou une `RecyclerView`, il faudra un `Adapter`. Un `Adapter` est le morceau de code qui va servir à indiquer à la `RecyclerView` les éléments présents dans celle-ci, leur nombre mais aussi la façon dont ils seront affichés. Il existe des `Adapter` basiques, mais nous voulons un affichage un peu plus compliqué et personnalisé que ce que permette les `Adapter` de base. L'`Adapter` s'occupe en fonction de l'affichage de créer le bon nombre d'élément affichable et de les recycler si ceux-ci ne sont plus visibles. Si notre liste contient 50 titres et que la vue ne peut en afficher que 10 à la fois. L'`Adapter` créera seulement 11 ou 12 vue élément. Quand une vue sera hors écran suite à un défilement il la réutilisera pour le nouvel élément qui apparaît.

Pour gérer la disposition des éléments d'un titre à afficher, nous allons devoir définir un nouveau layout **`pokemon_item.xml`**

```

<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    >
    <data/>
    <androidx.constraintlayout.widget.ConstraintLayout
        android:background="#FF00FF"
        android:layout_margin="5dp"
        android:layout_width="match_parent"
        android:layout_height="50dp">

        <ImageView
            android:id="@+id/front"
            android:layout_width="wrap_content"
            android:layout_height="0dp"
            android:src="@drawable/p3"

```

```

        app:layout_constraintBottom_toTopOf="@id/separator"
        android:scaleType="centerInside"

        app:layout_constraintEnd_toStartOf="@id/name"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toTopOf="parent" />

<TextView
    android:textAppearance="@style/TextAppearance.AppCompat.Medium"
    android:id="@+id/name"
    app:layout_constraintStart_toEndOf="@id/front"
    app:layout_constraintEnd_toStartOf="@id/number"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toTopOf="@id/type1_text"
    android:layout_width="0dp"
    android:layout_height="wrap_content"/>
<TextView
    android:id="@+id/type1_text"
    app:layout_constraintStart_toEndOf="@id/front"
    app:layout_constraintEnd_toStartOf="@id/type2_text"
    app:layout_constraintBottom_toTopOf="@id/separator"
    app:layout_constraintTop_toBottomOf="@id/name"
    android:layout_width="0dp"
    android:layout_height="wrap_content"/>
<TextView
    android:id="@+id/type2_text"
    app:layout_constraintStart_toEndOf="@id/type1_text"
    app:layout_constraintEnd_toStartOf="@id/number"
    app:layout_constraintBottom_toTopOf="@id/separator"
    app:layout_constraintTop_toBottomOf="@id/name"
    android:layout_width="0dp"
    android:layout_height="wrap_content"/>
<TextView
    android:id="@+id/number"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toTopOf="@id/separator"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:textAppearance="@style/TextAppearance.AppCompat.Medium"/>
<TextView
    android:id="@+id/separator"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintBottom_toBottomOf="parent"
    style="?android:attr/listSeparatorTextViewStyle"

```

```

        android:layout_width="match_parent"
        android:layout_height="5dp"/>
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>

```

Passons à l'adapter. Ce dernier va recevoir en entrant la liste des données à afficher. La fonction `onCreateViewHolder` sera appelée pour chaque vue à créer (nombre limité évoqué plus haut). On ne passera que peu de fois dans la fonction. A contrario, la fonction `onBindViewHolder` est quant à elle appelée pour remplir la vue associée au `ViewHolder` avec les données d'un élément de notre liste. Si on fait défiler la vue, les `ViewHolder` plus visible à l'écran seront recyclés pour prendre la place de ceux s'affichant en dessous des précédents. On reçoit ici la position de l'élément à afficher. La fonction `getItemCount` permet de donner l'information à la `RecyclerView` sur le nombre maximum d'élément et du coup savoir quand on est à la fin de la liste ou non.

```

public class PokemonListAdapter extends
    RecyclerView.Adapter<PokemonListAdapter.ViewHolder> {

    List<Pokemon> pokemonList;

    public PokemonListAdapter(List<Pokemon> pokemonList) {
        assert pokemonList != null;
        this.pokemonList =pokemonList;
    }

    @NonNull
    @Override
    public ViewHolder onCreateViewHolder(@NonNull ViewGroup parent, int viewType) {
        PokemonItemBinding binding = DataBindingUtil.inflate(
            LayoutInflater.from(parent.getContext()),
            R.layout.pokemon_item, parent, false);
        return new ViewHolder(binding);
    }

    @Override
    public void onBindViewHolder(@NonNull ViewHolder holder, int position) {
        Pokemon pokemon = pokemonList.get(position);
        holder.binding.front.setImageResource(pokemon.getFrontResource());
        holder.binding.name.setText(pokemon.getName());
        holder.binding.type1Text.setText(pokemon.getType1String());
        holder.binding.number.setText("#"+pokemon.getOrder());
        if (pokemon.getType2() != null) {
            holder.binding.type2Text.setText(pokemon.getType2String());
        }
    }

    @Override

```

```

    public int getItemCount() {
        return pokemonList.size();
    }

    static class ViewHolder extends RecyclerView.ViewHolder {

        private PokemonItemBinding binding;

        ViewHolder(PokemonItemBinding binding) {
            super(binding.getRoot());
            this.binding = binding;
        }
    }
}

```

Pour affecter notre adapter à notre liste nous aurons besoin d'ajouter dans le fragment les lignes suivantes. Remplacez les ... par la création d'élément servant à alimenter la liste avec des données statiques. Nous verrons plus tard pour récupérer des données depuis une base de données ou une api.

```

List<Pokemon> pokemonList = new ArrayList<>();
...
PokemonListAdapter adapter = new PokemonListAdapter(pokemonList)
binding.pokemonList.setAdapter(adapter);

```

Astuce:

Vous pouvez passer par un fichier json pour charger l'ensemble des données, plutôt que d'avoir une longue liste de code. Ce dernier devra se trouver dans le dossier *res/raw* ou *assets*

Pour lire un fichier json depuis les ressources:

Avec un JSON comme suit:

```

[
{"name":"Bulbizarre", "image":"p1", "type1":"plante", "type2":"poison"},
{"name":"Herbizarre", "image":"p2", "type1":"plante", "type2":"poison"},
{"name":"Florizarre", "image":"p3", "type1":"plante", "type2":"poison"},
{"name":"Salamèche", "image":"p4", "type1":"feu"}
]

```

On peut facilement lire le fichier à l'aide du code suivant:

```

//Ouverture du fichier res/raw
InputStreamReader isr = new InputStreamReader(getResources().openRawResource(R.raw.poke));
// Ouverture du fichier dans assets
// InputStreamReader isr =
//         new InputStreamReader(getResources().getAssets().open("poke.json"));

```

```

BufferedReader reader = new BufferedReader(isr);
StringBuilder builder = new StringBuilder();
String data = "";

//lecture du fichier. data == null => EOF
while(data != null) {
    try {
        data = reader.readLine();
        builder.append(data);
    } catch (IOException e) {
        e.printStackTrace();
    }
}

//Traitement du fichier
try {
    JSONArray array = new JSONArray(builder.toString());
    for (int i = 0; i < array.length(); i++) {
        JSONObject object = array.getJSONObject(i);
        String name = object.getString("name");
        String image = object.getString("image");
        String type1 = object.getString("type1");
        String type2 = null;
        if (object.has("type2"))
            type2 = object.getString("type2");
    }
} catch (JSONException e) {
    e.printStackTrace();
}

```

Pour accéder à une ressource (drawable par exemple) à partir de son nom et récupérer son id:

```

int id = getResources().getIdentifier("nomDuDrawableSansExtension", "drawable",
                                     binding.getRoot().getContext().getPackageName());

```

À faire:

- Mettez en place cette liste, n'hésitez pas à personnaliser l'aspect et/ou la disposition des éléments pour correspondre à votre envie/besoin
- Essayer de bien comprendre tout le code fourni et comment fonctionnent les interactions entre les différents éléments, comment ils sont liés entre eux. Vous aurez besoin de réappliquer ces éléments pour d'autres parties de l'application
- Construisez un jeu d'essai qui vous servira à tester l'affichage des éléments et si votre liste s'affiche correctement, se met à jour correctement lors d'un défilement...

Model - View - ViewModel

Comme vu en cours, le modèle c'est Pokemon. Notre view c'est pokemon_item.xml. Il nous reste à définir notre View-Model. Pour avoir notre ensemble complet. Le ViewModel communique à la fois avec la view et à la fois avec le model.

Notre ViewModel est assez simple, nous n'avons que des champs à afficher. Le View-Model étend BaseObservable. Ceci afin de rafraîchir la vue, si des données étaient amenées à changer. Ce sera notre cas. C'est l'annotation @Bindable qui indique qu'un champ peut être mis à jour. Il peut être mis à jour dans une fonction setXXX et il existe deux méthodes pour notifier ce changement notifyChange() qui indique que tout a été changé et qu'il faut donc mettre à jour tous les champs Bindable ou notifyPropertyChanged(BR.propertyName) pour indiquer qu'une propriété en particulier à changer. (BR => BindableResource, fichier généré comme le fichier R). Notre cas sera le premier, si on change de fichier dans le ViewModel, il faut obligatoirement que tous les champs s'y rapportant soit mis à jour.

```
public class PokemonViewModel extends BaseObservable {
    private Pokemon pokemon = new Pokemon();

    public void setPokemon(Pokemon pokemon) {
        this.pokemon = pokemon;
        notifyChange();
    }

    @Bindable
    public int getFront() {
        return pokemon.getFrontResource();
    }

    @Bindable
    public String getName() {
        return pokemon.getName();
    }

    @Bindable
    public String getType1() {
        return pokemon.getType1String();
    }

    @Bindable
    public String getType2() {
        if (pokemon.getType2() != null)
            return pokemon.getType2String();
        return "";
    }
}
```

```

@Bindable
public String getNumber() {
    return "#" + pokemon.getOrder();
}

public Drawable getImage(Context context, int res) {
    if(res != -1)
        return ResourcesCompat.getDrawable(context.getResources(),
                                            res, context.getTheme());
    else
        return null;
}
}

```

Pour créer le lien entre la vue et le viewmodel nous avons la balise data que nous avons laissé vide dans nos layout. Ce ne sera plus le cas pour **pokmon_item.xml**. `pokemonViewModel` est le nom de la variable que l'on va utiliser (vous pouvez l'appeler comme vous voulez) et son type est la classe `PokemonViewModel` que nous avons défini au-dessus.

```

<data>
    <variable
        name="pokemonViewModel"
        type="fr.mobdev.pokemongeo.PokemonViewModel" />
</data>

```

Pour l'utiliser dans un layout, il n'y a rien de bien compliqué. Là où vous avez besoin de cette variable. Ci-dessous pour mettre le titre dans la textView vous écrivez `@nomVariable.donnee`. La fonction `getDonnee()` du ViewModel sera donc appelé pour remplir ce champ.

```

<TextView
    android:textAppearance="@style/TextAppearance.AppCompat.Medium"
    android:id="@+id/name"
    android:text="@{pokemonViewModel.name}"
    app:layout_constraintStart_toEndOf="@id/front"
    app:layout_constraintEnd_toStartOf="@id/number"
    app:layout_constraintTop_toTopOf="parent"
    app:layout_constraintBottom_toTopOf="@id/type1_text"
    android:layout_width="0dp"
    android:layout_height="wrap_content"/>

```

Noté que pour les images c'est un peu plus compliqué, il ne suffit pas de retourner l'entier `R.drawable.resource` via le viewModel mais de fournir un `Drawable` à l'`ImageView`. Le souci étant que pour fournir cette image, il faut un context comme on peut le voir dans le ViewModel avec la fonction `getImage`. Vous pouvez voir ci-dessous un exemple de cette utilisation.


```

<ImageView
    android:id="@+id/front"
    android:layout_width="wrap_content"
    android:layout_height="0dp"
    android:src="@{pokemonViewModel.getImage(context,
        pokemonViewModel.front)}"
    app:layout_constraintBottom_toTopOf="@id/separator"
    android:scaleType="centerInside"

    app:layout_constraintEnd_toStartOf="@id/name"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

```

Dernière étape pour finir c'est donner au databinding le ViewModel afin qu'il puisse l'utiliser avec la vue. Dans la fonction onBindViewHolder nous avons un certain nombre de fonction setText(). Nous allons les remplacer par

```
holder.viewModel.setPokemon(pokemon);
```

Dans le viewHolder nous allons ajouter la création du ViewModel qui sera une variable membre et l'affecter au binding.

```

static class ViewHolder extends RecyclerView.ViewHolder {

    private PokemonItemBinding binding;
    private PokemonViewModel viewModel = new PokemonViewModel();

    ViewHolder(PokemonItemBinding binding) {
        super(binding.getRoot());
        this.binding = binding;
        this.binding.setPokemonViewModel(viewModel);
    }
}

```

À faire

- Utiliser MVVM dans votre projet en suivant ce qui est indiqué au-dessus et continuer à l'utiliser par la suite pour les autres vues qui en auraient besoin

Communication Fragment vers Activity

Il arrive fréquemment que l'on ait besoin de faire communiquer le fragment avec l'activity qui le contient ou entre deux fragments de la même activity. Pour cela, il y a plusieurs étapes:

- Définition d'une interface (le Listener)
- Implémentation de l'interface au sein de l'activity

- Transmission de l'objet qui implémente l'interface au fragment qui doit transmettre une information
- Déclenchement de la transmission d'information

Un listener c'est une interface définissant une ou plusieurs méthodes pouvant être déclenché afin de remonter une ou plusieurs informations. Dans l'exemple qui suit nous avons un listener avec une méthode `onClickOnNote` qui sera déclenché sur un clic sur une vue.

```
public interface OnClickOnNoteListener{
    public void onClickOnNote(long noteId);
}
```

À l'endroit où vous allez vouloir remonter l'information vous allez devoir implémenter le listener. Ce sera souvent dans une Activity ou dans un fragment (dans le cas où vous avez des fragments au sein d'un fragment). Ici lorsque le listener sera déclenché, la fonction `showNoteDetail` sera exécuté. Ici, il s'agit d'une implémentation anonyme (pas d'implémentation dans une classe avec un nom et un fichier associé). listener est une instance de cette classe.

```
OnClickOnNoteListener listener = new OnClickOnNoteListener(){
    @Override
    public void onClickOnNote(long noteId){
        showNoteDetail(noteId);
    }
};
```

La transmission de ce listener au fragment qui va émettre l'information.

```
fragment.setOnClickOnNoteListener(listener);
```

Dans le fragment qui doit transmettre une information, nous aurons donc quelque chose qui ressemble à ce qui suit.

```
private OnClickOnNoteListener listener;
public void setOnClickOnNoteListener(OnClickOnNoteListener listener)
{
    this.listener = listener;
}
```

Dans ce même fragment nous avons par exemple une fonction `onEventFunction` qui réagit lors d'un certain évènement (clic sur une vue, traitement en arrière plan qui se finit...) et à la fin de l'exécution de celle-ci nous voulons remonter l'information que tout s'est bien déroulé à notre activity, pour cela nous utilisons donc le listener que nous avons reçu. À la fin de `onEventFunction`, on appelle `onClickOnNote` tel qu'il a été implémenté. C'est-à-dire qu'il appellera la méthode de l'activity qui s'appelle `showNoteDetail`.

```
public void onEventFunction(long noteId) {
    if (listener != null)
        listener.onClickOnNote(noteId);
}
```

}

Si vous avez plusieurs couches à traverser. Activity <- Fragment <- Fragment <- Adapter rien ne vous empêche de transmettre le listener jusqu'à là où il est nécessaire.

L'avantage du listener c'est qu'il ne crée pas un couplage fort entre Activity et Fragment. On peut tout à fait réutiliser le fragment avec une autre activity sans que cette dernière ne s'intéresse à l'évènement produit à la fin de onEventFunction auquel cas elle ne crée pas de listener et ne le transmet pas au fragment. Le fragment n'a aucune connaissance du fait qu'il est dans une activity ou dans un fragment quelconque, il ne sait pas qu'il a des échanges avec quelqu'un d'autre. Il émet une information éventuellement si on lui donne un canal de communication: le listener, mais il ne sait pas qui est au bout.

À faire

- Créer un second fragment qui affichera les infos détaillées d'un Pokemon (poids, taille, nom, image en grand, type, capacité, description...). Ce fragment sera appelé lorsque l'on clique sur un élément de notre liste.

Navigation

Notre application va s'étoffer, au fur et à mesure, de plusieurs écrans:

- La carte
- le pokedex
- les pokemons capturés
- l'inventaire

Le nombre relativement restreint de page nous permet à priori d'utiliser une BottomBar afin d'avoir une navigation adéquate entre les différents écrans. C'est un outil relativement simple à mettre en place.

À faire

- En vous aidant de la documentation suivante: <https://material.io/components/bottom-navigation/android#using-bottom-navigation> mettez en place la BottomBar permettant de naviguer entre nos différents Fragment. Vous ajouterez les fragments au fur et à mesure que vous les aurez réalisés. Le code exemple fourni est cependant en Kotlin. when correspond à un switch, le reste ne devrait pas trop vous poser de soucis. La documentation référence néanmoins les différentes classes java au besoin: <https://developer.android.com/reference/com/google/android/material/bottomnavigation/BottomNavigationView>

Carte et Geolocalisation

Le jeu nécessite que nous ayons une carte de là où nous situons et des pokemons que nous pouvons capturer à proximité. En cours, nous avons vu la géolocalisation, la

possibilité de gérer une carte Open Street Map ainsi que d'ajouter des markers sur la carte. Pour cela, vous aurez à demander la permission de géolocalisation et de stockage (pour le cache). Ensuite de gérer la carte en elle-même. La carte sera gérée également au sein d'un Fragment.

À faire

- Demandez la permission d'accéder aux stockages externes et à la position gps
- Afficher une carte à l'aide d'OSMDroid, centré sur la position du joueur et testé la possibilité de mettre des pokemons sur la carte.
- Pensez à ajouter la navigation entre les différents fragments au sein de votre Navigation Bar

Base de données

L'essentielle des données de jeu seront issues d'une base de donnée locale. Le pokedex à l'origine est vide et fonction des pokemons que l'on rencontre ce dernier se remplit, du moins d'un point de vue joueur, rien ne vous empêche d'avoir l'ensemble des pokemons dans votre base avec un attribut indiquant s'il a été découvert ou non. En plus des données du pokedex, nous aurons à stocker les pokemons que nous aurons capturé de même qu'il faudra que vous définissiez les caractéristiques qui vous intéressent pour votre jeu (niveau, attaque, défense, point de vie...) Si vous envisagez la possibilité d'avoir un inventaire, réfléchissez à ce dont vous avez besoin. Vous pouvez utiliser un script pour remplir votre base de donnée, ce dernier sera une ressource de votre projet et sera lu et exécuté lors du démarrage de votre application la première fois.

À faire

- Établissez le schéma de la base de donnée pour votre application
- Enregistrez les informations dans votre base de données
- Affichez les données depuis votre base de donnée
- Pensez à protéger l'accès à la base de donnée pour ne pas la rendre accessible à toute l'application. Uniquement quelques fonctions doivent être accessibles.

Jeu

Il est temps de transformer notre application en jeu. Lors du premier lancement, le joueur a le choix parmi trois pokemon initiaux. Ce pokemon lui servira à affronter les pokemons qu'il rencontrera sur sa route et d'éventuellement les capturer. Lorsqu'il rencontre un pokemon inconnu, il sera ajouté au pokedex et ses informations seront donc accessibles depuis ce dernier. On ne peut affronter un pokemon que si on est à proximité de celui-ci.

Pour cette partie, c'est à vous de gérer vos règles du jeu. Une version basique peut consister à avoir tout les pokemons avec la même force et même défense à un niveau

équivalent et d'augmenter celle-ci en fonction du niveau. Pour les combats pour donner l'avantage ou non à un pokemon, le type de ce dernier peut avoir un impact sur celui de son opposant (exemple: électrique contre eau). Dans un premier temps, vous pouvez considérer que vos pokemon récupère l'intégralité de leur santé après le combat et que vous avez un nombre illimité de pokeball, soin. En terme d'action, lors d'un combat: attaque, défense, soin, capture sont les actions qui pourront être mises en place simplement. Ne tentez pas, de faire des animations dans cette partie, du moins pas dans un premier temps.

Pour cette partie, il est fortement conseillé de faire les choses de manière itérative afin d'avoir un résultat fonctionnel à un instant T même si ce n'est pas forcément ce que vous avez imaginé pour la suite.

La gestion du jeu peut nécessiter l'usage d'un Thread pour notamment gérer l'apparition/disparition de pokémon sur la carte en fonction de la position du joueur qui évolue mais aussi en fonction du temps qui passe. Indépendant de la carte, il permettrait de conserver les pokémons proches même si on navigue dans les différentes vues de notre application (pokedex, inventaire...)

À faire

- Ajouter la gestion du combat entre deux pokemons
- Ajouter la possibilité de capturer un pokemon (la difficulté de capture dépend de sa santé restante)
- Ajouter la gestion de l'inventaire (les objets peuvent être gagnés lors des combats dans un premier temps)

API

Il existe des api pour trouver facilement des points d'intérêt sur OpenStreetMap. L'api utilisé est Overpass et est relativement simple à utiliser. Vous pouvez essayer en ligne ici: <https://overpass-turbo.eu/>

```
//Pour avoir les données au format json:
[out:json];
//pour avoir les points d'intérêt de type boulangerie/pâtisserie en utilisant
//les tag OpenStreetMap et entre parenthèse les coordonnées de votre carte.
//(Coin Sud-Ouest et Coin Nord-Est)
node [shop=bakery] (latitude sud, longitude ouest, latitude nord, longitude est);
out;
```

Cette chaîne encodée sera utilisée alors dans la requête API en tant que paramètre data.

Exemple: <https://overpass-api.de/api/interpreter?data=%5Bout%3Ajson%5D%3Bnode%20%5Bshop%3Dbakery%5D%20%2845.76000839079471%2C4.856901168823242%2C45.790659088204684%2C4.902176856994628%29%3B%20out%3B>

Pour retrouver les informations de tag OpenStreetMap:

- <https://wiki.openstreetmap.org/wiki/Key:shop>
- <https://wiki.openstreetmap.org/wiki/Key:amenity>
- https://wiki.openstreetmap.org/wiki/Category:Key_descriptions

Il est très facile d'accéder aux coordonnées des coins de la carte à l'aide de la MapView.

À faire

- Pokecenter pour soigner vos pokemon en se basant sur la position des pharmacies
- Magasin en se basant sur la position de ce qui vous semble pertinent

Fonctionnalités en plus

- Possibilité de combattre un autre joueur sur le même réseau
- Possibilité d'échanger un pokemon avec un autre joueur
- Ajouter des quêtes
- Ajouter des arènes
- ...

Intégration Continue

Pour que l'intégration continue dans votre projet il faut créer dans votre dépôt le fichier `.woodpecker.yml` avec le contenu suivant à la racine du dépôt:

```
pipeline:
  build:
    image: eclipse-temurin:11
    commands:
      - bash ./gradlew assembleDebug
    environment:
      - ANDROID_HOME=/mnt/sdk
    volumes:
      - /home/woodpecker/sdk:/mnt/sdk
```